Logical Specification of Verifiable Robotic Systems

Paper #XXX

***Abstract*—Modern robotic systems rely on autonomous software control, node-based architectures, and safety requirements. Verifying that distinct nodes behave as expected often requires using different verification techniques. This paper describes an approach to formally specify nodes so that they can be verified using the most appropriate technique. We use First-Order Logic (FOL) to specify each node's assumptions and guarantees. Then, we introduce inference rules that enable the combination of these FOL specifications. The specifications guide the verification of the nodes, which can involve an array of formal or non-formal techniques, as required. Further, this approach enables us to swap one node for another, which may be verified using a different technique, because it can also be guided by the FOL specification. To illustrate our contributions, we demonstrate the specification and verification of an autonomous rover in a search and rescue scenario. Finally, we present a discussion of how our approach fits into the process of developing a robotic system, and how to relate differing levels of confidence we might have in the range of verification techniques used on one system.**

***Index Terms*—integrated formal methods, modular robotic systems, heterogeneous specification and verification**

I. Introduction

Robotic systems are increasingly deployed in industrial, often safety-critical, scenarios. For example, monitoring offshore structures [1], nuclear inspection/decommissioning [2], [3], and space exploration [4], [5]. These applications often require certification and so the software controlling the robot must be robustly verified. Software verification encompasses a range of techniques including practical testing, simulation and formal methods [6]. In particular, formal verification can be applied to the implemented system; or to an abstract specification, which can then be refined to program code.

Formal methods include model-checkers [7], which exhaustively explore the search space to show that a property holds; and theorem provers [8], which employ mathematical proof to ensure that the system behaves as expected. These formal verification approaches allow us to prove that the software is correct and their results can be used as certification evidence.

Formal verification generally follows a simple paradigm: if the program is executed in a state satisfying a given property, then it will terminate in a state satisfying another property [9]. The *Assume-Guarantee* paradigm [10] is used to decompose a large verification task into smaller, simpler ones where components are verified individually and then the separate assessments can be combined, potentially under some assumption about the underlying concurrency. To combine descriptions, the *pre-* and *post*-condition specifications for a component are required; this is similar to the notion of Hoare logic that underlies these techniques [11]. Still, it is up to the software engineer to specify the Assume-Guarantee properties to be verified which is a non-trivial task.

Complex robotic systems tend to be difficult to verify because they are not normally designed with verification in mind from the outset. For developers, the emphasis is often on "getting the system working", which is hard in itself, but there is currently no clear, and general, approach to producing reliable software for these systems. Thus, by augmenting the software development process so that these systems are built with verification in mind offers a huge range of benefits. Furthermore, these systems are frequently composed of heterogeneous subsystems that may be implemented using different languages and toolsets. Recent work has argued that the heterogeneous nature of these systems necessitates the use of a combination of formal and non-formal methods to verify the complete system with individual subsystems verified using the approach most suitable for them [12]. For example, a vision system might be verified by software testing, an autonomous agent by program model-checking, a planner by theorem proving, and so on.

In this paper, we encourage the developer to specify the functionality of system nodes using First-Order Logic (FOL), thus, augmenting the software development process so that these systems are built with verification in mind from the beginning. These FOL specifications are used to provide guidance for both the verification and implementation development phases, and can be viewed as lightweight specification documentation for individual nodes. Once specified, individual components are verified using testing and/or formal methods with the high-level FOL specifications providing a base guideline. Furthermore, we provide a calculus for reasoning about how these component specifications combine to provide whole system verification with a range of heterogeneous formal verification approaches applied to individual nodes.

To validate our work, we specify the assumptions and guarantees of a robotic system in a search and rescue scenario. Here, an autonomous rover navigates an area looking for victims in a post-disaster environment. We construct the FOL specification of each node which then guides our use of various verification techniques for the system's nodes.

§II discusses related work. §III describes our approach and calculus for verifying robotic systems using Assume-Guarantee specifications written in FOL. In §IV, we illustrate our approach with a search and rescue scenario, discuss the individual verification techniques applied to the nodes, and describe how this links to the high-level specification. In §V, we discuss how our approach fits into the development process and leveraging existing tool support for verification. Finally, §VI presents our concluding remarks and future work.

II. Related Work

Our work follows from Broy's approach to systems engineering [13] 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 is specialised to the software engineering of robotic systems.

The work in [14] gives a formal FOL framework for describing the capabilities of robotic system nodes. Our work generalises and expands on it by enabling the integration of different formalisms that are unified and guided by a FOL specification. Conversely, their framework provides a richer treatment of data streams, whereas we consider the connections between nodes to be the simple communication channels commonly found in robotic systems. Related to this, the Focus method [15], and its associated toolset, AutoFocus3 [16], enable the specification and verification of interactive systems using stream processing functions over message histories.

In [17], a workflow is proposed for systematically verifying the design of cyber-physical system models using formal refinement and model-checking. Our work also deals with several levels of abstraction, but we facilitate the use of heterogeneous verification methods.

Recently, [18] analysed a portion of the literature and identified common patterns appearing in robotic missions (e.g. patrolling/obstacle avoidance). These patterns are specified in linear temporal/computation tree logic and can be reused in future developments. Their work promotes the reuse of these verified patterns whereas we present a logical methodology for developing verifiable robotic systems. While the specifications presented in their work are domain-specific, our use of Assume-Guarantees in FOL with inference rules is more general and can be applied to a range of robotic systems.

Elkader et al. propose compositional verification using Assume-Guarantee reasoning as a solution to the state explosion problem that limits the efficacy of model-checkers [19]. Their work focuses on circular Assume-Guarantee reasoning where components are connected in such a way that the verification of individual components mutually depends on each other. Furthermore, their work uses counter-example-guided-abstraction-refinement (CEGAR) to attempt to identify assumptions automatically. We view this work as complementary to ours and we seek to integrate their results as future work. Related to this, Graf et al. [20] propose a rule for unifying circular and non-circular Assume-Guarantee reasoning.

The Integration Property Language (IPL) aims to combine FOL reasoning across many system views [21]. Here, views correspond to behaviourless component models that are annotated with types and properties. This work facilitates the specification and verification of modular robotic systems. It does not, however, address the use of heterogeneous verification techniques for individual components.

III. Specifying Verifiable Robotic Systems

No single verification approach suits every node in a robotic system [12]. For example, the verification of a learning-based vision system differs to that of a deductive planner or rational agent. Our approach facilitates the combination of heterogeneous verification methods for robotic system nodes by expressing Assume-Guarantee properties in *typed* FOL to define high-level node specifications and using first-order *temporal logic* to reason about how these specifications relate. Fig. 1 illustrates some of the distinct formalisms and verification techniques that can be used for individual nodes. Each of these offers its own range of benefits, and suits the verification of particular types of behaviour.

Specifically, we consider the usual definition of first-order logic with quantifiers ($\forall$, $\exists$) and the logical connectives ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$) over logical propositions including basic set theory [22]. We use the Assume-Guarantee specifications to guide the modelling and verification of individual nodes. Thus, these specifications must be written in a generic formalism that shares commonalities with other verification approaches so that the Assume-Guarantees are understandable and useful. When reasoning about these properties, we adopt first-order temporal logic since it is more expressive than FOL.

A. Assume-Guarantees in First-Order Logic

For a given node, C, we specify $\mathcal{A}_C(\overline{i_C})$ and $\mathcal{G}_C(\overline{o_C})$ where $\overline{i_C}$ is a vector of variables representing the input to the node, $\overline{o_C}$ represents the output from the node, and $\mathcal{A}_C(\overline{i_C})$ and $\mathcal{G}_C(\overline{o_C})$ are FOL formulae describing the assumptions and guarantees, respectively, of C. In particular, C, obeys the following:

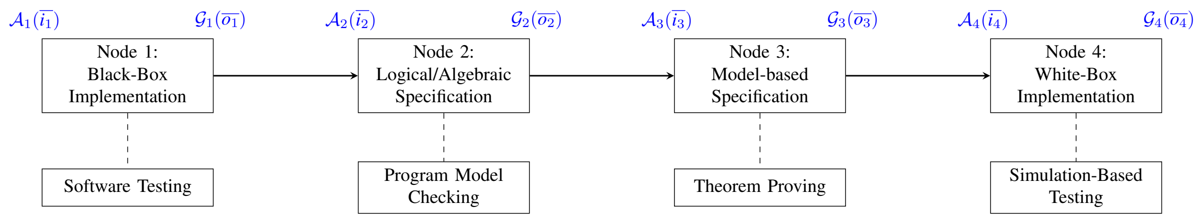
$$\forall\, \overline{i_C}, \overline{o_C} \cdot \mathcal{A}_C(\overline{i_C}) \Rightarrow\ \Diamond \mathcal{G}_C(\overline{o_C})$$

where '$\Diamond$' is Linear-time Temporal Logic (LTL)'s [23] "eventually" operator. Intuitively, if the assumptions $\mathcal{A}_C(\overline{i_C})$ hold in the specification or program code of C, then *eventually* the guarantee $\mathcal{G}_C(\overline{o_C})$ will hold. Note that our use of temporal operators here is motivated by the real-time nature of robotic systems and will be of use in later extensions of this calculus for larger, more complex systems.

We envisage that a top-down approach to software engineering of robotic systems would begin with FOL Assume-Guarantee node specifications which would be further refined [24], ideally to more detailed formal specifications and, eventually to concrete implementations.

B. Calculus for Combining Node Specifications

Nodes in a (modular) robotic system can be linked as long as their input types/requirements match. The basic way to describe such structures is to first have the specification capture all of the input and output streams and then describe how they are combined using the appropriate inference rules. We compose the Assume-Guarantee specifications of individual



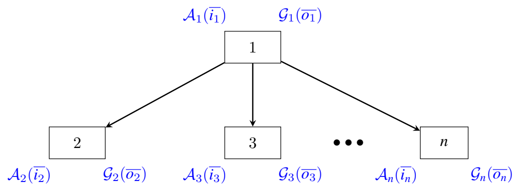
$\mathcal{A}_1(\overline{i_1})$
$\mathcal{G}_1(\overline{o_1})$
$\mathcal{A}_2(\overline{i_2})$
$\mathcal{G}_2(\overline{o_2})$
$\mathcal{A}_3(\overline{i_3})$
$\mathcal{G}_3(\overline{o_3})$
$\mathcal{A}_4(\overline{i_4})$
$\mathcal{G}_4(\overline{o_4})$
Node 1: Black-Box Implementation
Node 2: Logical/Algebraic Specification
Node 3: Model-based Specification
Node 4: White-Box Implementation
Software Testing
Program Model Checking
Theorem Proving
Simulation-Based Testing

Fig. 1. Assume-Guarantee specifications for each node ($\mathcal{A}(\overline{i})$ and $\mathcal{G}(\overline{o})$ respectively) guide the individual verification approach applied (dashed lines). Solid arrows denote data flow between nodes and the assumption of the next follows from the guarantee of the previous.

nodes in a number of ways, the simplest being via sequential composition. The inference rule for such linkage is:

$$\frac{\begin{array}{l} \forall \overline{i_1}, \overline{o_1} \cdot \mathcal{A}_1(\overline{i_1}) \;\Rightarrow\; \Diamond\mathcal{G}_1(\overline{o_1}) \\ \forall \overline{i_2}, \overline{o_2} \cdot \mathcal{A}_2(\overline{i_2}) \;\Rightarrow\; \Diamond\mathcal{G}_2(\overline{o_2}) \\ \overline{o_1} = \overline{i_2} \\ \vdash\; \forall \overline{o_1}, \overline{i_2} \cdot \mathcal{G}_1(\overline{o_1}) \;\Rightarrow\; \mathcal{A}_2(\overline{i_2}) \end{array}}{\forall \overline{i_1}, \overline{o_2} \cdot \mathcal{A}_1(\overline{i_1}) \;\Rightarrow\; \Diamond\mathcal{G}_2(\overline{o_2})} \tag{R1}$$

This rule states that given two nodes that are connected such that the output of the first is equal to the input of the second and the guarantee of the first implies the assumption of the second then if the assumptions of the first node hold, we can conclude that the guarantees of the second node will eventually hold. Fig. 1 contains a simple, linear chain of nodes. But a robotic system is generally more complex than this, potentially including multiple, branching output streams as shown:



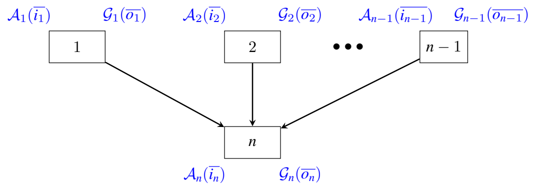
$\mathcal{A}_1(\overline{i_1})$
$\mathcal{G}_1(\overline{o_1})$
1
2
3
•••
n
$\mathcal{A}_2(\overline{i_2})$
$\mathcal{G}_2(\overline{o_2})$
$\mathcal{A}_3(\overline{i_3})$
$\mathcal{G}_3(\overline{o_3})$
$\mathcal{A}_n(\overline{i_n})$
$\mathcal{G}_n(\overline{o_n})$

We use the following inference rule to account for branching.

$$\frac{\begin{array}{l} \forall \overline{i_1}, \overline{o_1} \cdot \mathcal{A}_1(\overline{i_1}) \;\Rightarrow\; \Diamond\mathcal{G}_1(\overline{o_1}) \\ \qquad\vdots \\ \forall \overline{i_n}, \overline{o_n} \cdot \mathcal{A}_n(\overline{i_n}) \;\Rightarrow\; \Diamond\mathcal{G}_n(\overline{o_n}) \\ \overline{o}_1 = \overline{i_2} \cup \ldots \cup \overline{i_n} \\ \vdash\; \forall \overline{o_1}, \overline{i_2}, \ldots, \overline{i_n} \cdot \mathcal{G}_1(\overline{o_1}) \;\Rightarrow\; \mathcal{A}_2(\overline{i_2}) \wedge \ldots \wedge \mathcal{A}_n(\overline{i_n}) \end{array}}{\forall \overline{i_1}, \overline{o_2}, \ldots, \overline{o_n} \cdot \mathcal{A}_1(\overline{i_1}) \;\Rightarrow\; \Diamond\mathcal{G}_2(\overline{o_2}) \wedge \ldots \wedge \Diamond\mathcal{G}_n(\overline{o_n})} \tag{R2}$$

In this case, the combined inputs of all 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.

Alternatively, a node's input might comprise the union of several outputs as illustrated below.

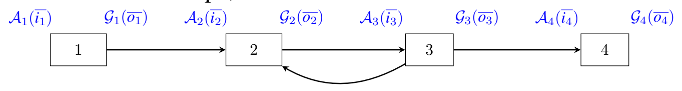


$\mathcal{A}_1(\overline{i_1})$
$\mathcal{G}_1(\overline{o_1})$
$\mathcal{A}_2(\overline{i_2})$
$\mathcal{G}_2(\overline{o_2})$
$\mathcal{A}_{n-1}(\overline{i_{n-1}})$
$\mathcal{G}_{n-1}(\overline{o_{n-1}})$
1
2
•••
$n-1$
n
$\mathcal{A}_n(\overline{i_n})$
$\mathcal{G}_n(\overline{o_n})$

In this case, we propose the following inference rule.

$$\frac{\begin{array}{l} \forall \overline{i_1}, \overline{o_1} \cdot \mathcal{A}_1(\overline{i_1}) \;\Rightarrow\; \Diamond\mathcal{G}_1(\overline{o_1}) \\ \qquad\vdots \\ \forall \overline{i_n}, \overline{o_n} \cdot \mathcal{A}_n(\overline{i_n}) \;\Rightarrow\; \Diamond\mathcal{G}_n(\overline{o_n}) \\ \overline{i_n} = \overline{o_1} \cup \ldots \cup \overline{o_{n-1}} \\ \vdash\; \forall \overline{i_n}, \overline{o_1}, \ldots, \overline{o_{n-1}} \cdot \mathcal{G}_1(\overline{o_1}) \wedge \ldots \wedge \mathcal{G}_{n-1}(\overline{o_{n-1}}) \\ \qquad\qquad \Rightarrow \mathcal{A}_n(\overline{i_n}) \end{array}}{\begin{array}{l} \forall \overline{i_1}, \ldots\; \overline{i_{n-1}}, \overline{o_n} \cdot \mathcal{A}_1(\overline{i_1}) \wedge \ldots \wedge \mathcal{A}_{n-1}(\overline{i_{n-1}}) \\ \qquad\qquad \Rightarrow \Diamond\mathcal{G}_n(\overline{o_n}) \end{array}} \tag{R3}$$

As expected, this rule is essentially the dual of R2. These three simple inference rules (R1, R2 and R3) constitute our basic calculus for reasoning about high-level FOL Assume-Guarantee specifications of nodes in a robotic system. Note that we are *not* specifying the fine-grained concurrency/streaming of processes/data but are just specifying interface expectations of nodes in a robotic system. Next we consider how to tackle loops, as illustrated below.



$\mathcal{A}_1(\overline{i_1})$
$\mathcal{G}_1(\overline{o_1})$
$\mathcal{A}_2(\overline{i_2})$
$\mathcal{G}_2(\overline{o_2})$
$\mathcal{A}_3(\overline{i_3})$
$\mathcal{G}_3(\overline{o_3})$
$\mathcal{A}_4(\overline{i_4})$
$\mathcal{G}_4(\overline{o_4})$
1
2
3
4

We propose the following rule to account for such loops.

$$\frac{\begin{array}{l} \forall \overline{i_1}, \overline{o_1} \cdot \mathcal{A}_1(\overline{i_1}) \;\Rightarrow\; \Diamond\mathcal{G}_1(\overline{o_1}) \\ \forall \overline{i_2}, \overline{o_2} \cdot \mathcal{A}_2(\overline{i_2}) \;\Rightarrow\; \Diamond\mathcal{G}_2(\overline{o_2}) \\ \forall \overline{i_3}, \overline{o_3} \cdot \mathcal{A}_3(\overline{i_3}) \;\Rightarrow\; \Diamond\mathcal{G}_3(\overline{o_3}) \\ \forall \overline{i_4}, \overline{o_4} \cdot \mathcal{A}_4(\overline{i_4}) \;\Rightarrow\; \Diamond\mathcal{G}_4(\overline{o_4}) \\ \overline{i_2} = \overline{o_1} \cup \overline{o_3} \wedge \overline{i_3} = \overline{o_2} \wedge \overline{o_3} \subseteq \overline{i_4} \\ \vdash \forall \overline{i_2}, \overline{o_3} \cdot \mathcal{A}_2(\overline{i_2}) \Rightarrow \Diamond\mathcal{G}_3(\overline{o_3}) \\ \vdash \forall \overline{i_3}, \overline{o_2}, \overline{o_4} \cdot \mathcal{A}_3(\overline{i_3}) \;\Rightarrow\; \Diamond\mathcal{G}_2(\overline{o_2}) \wedge \Diamond\mathcal{G}_4(\overline{o_4}) \\ \vdash \forall \overline{i_1}, \overline{i_3}, \overline{o_2} \cdot \mathcal{A}_1(\overline{i_1}) \wedge \mathcal{A}_3(\overline{i_3}) \;\Rightarrow\; \Diamond\mathcal{G}_2(\overline{o_2}) \end{array}}{\forall \overline{i_1}, \overline{o_4} \cdot \mathcal{A}_1(\overline{i_1}) \;\Rightarrow\; \Diamond\mathcal{G}_4(\overline{o_4})} \tag{R4}$$

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. Then the conclusion below the line is drawn from their combination and simplification. Thus, R4 is not part of our core calculus (R1, R2 and R3), as it is derived from our previous rules, but is included to explicitly capture the behaviour of loops and it is easy to see how this is extended to incorporate an arbitrary number of nodes between nodes 2 and 3 in the loop illustration above.

Note that we assume that assumptions and guarantees are not mutually independent and that circular reasoning does not occur. However, we intend to investigate this in future work and potentially leverage existing techniques for addressing circular Assume-Guarantee reasoning [19]. Alternatively, we could potentially augment R4 to include fixed point reasoning in first-order temporal logic.

In this section, we have described how to specify FOL Assume-Guarantee properties of the nodes in a robotic system. We have also presented a calculus for reasoning about how these FOL specifications are propagated throughout the system. Next, we illustrate our approach via an example.

IV. A Search and Rescue Case Study

Autonomous robots, such as *rovers*, enable the safe exploration of unknown and/or hazardous environments e.g. areas of extreme radiation, temperature, or that lack a breathable atmosphere. These robots are especially well-suited for search and rescue missions. One of the first known uses of robots in search and rescue was during the World Trade Center disaster [25]. Recently, several unmanned aerial vehicles were used in conjunction with social media to aid in search and rescue operations during floods in Texas [26].

Our case study focuses on the navigation system for an autonomous rover as illustrated in Fig. 2. The rover's goal is to navigate to all heat positions on a 2D grid map of size n. The **Vision** system is used by the rover while traversing the map to detect obstacles. The **Infrared** node keeps track of which grid locations are hotter than expected. Based on these heat locations, the autonomous **Goal Reasoning Agent** selects the hottest location as the goal, unless the **Battery Monitor** (via the **Plan Reasoning Agent**) indicates that it must recharge. The **Planner** returns a set of obstacle-free plans for navigating from the current position to the goal. The autonomous **Plan Reasoning Agent** selects the shortest plan, provided that it has enough battery to do so, otherwise it alerts the **Goal Reasoning Agent**. Finally, the **Interface** translates the navigation actions of the plan into the instructions required by the hardware components (e.g. wheels/motors) and alerts the **Goal Reasoning Agent** when it reaches the goal.

A. FOL Specification

In this section we describe in more detail the FOL specification of each individual node in the system. Fig. 2 illustrates our rover navigation system and contains the FOL specifications that guide our verification of individual nodes. The arrows in Fig. 2 indicate data flow between the nodes. In particular, **Vision** and **Infrared** send their output only once, while all other nodes are continuous. Even the **Battery Monitor**, which does not have a loop, sends its output to the **Plan Reasoning Agent** once it starts. We assume to the following functions: $\mathit{position}()$, $\mathit{obstacle}()$, $\mathit{heat}()$, $\mathit{adjacent}()$, $\mathit{length}(), \mathit{on}()$ and $\mathit{batteryLow}()$, which represent the rover's ground truth. For example, $\mathit{position}()$ returns the rover's actual location. Below, we describe the FOL Assume-Guarantee specifications of the individual nodes to show how they were constructed.

Vision: The inputs to this node, $\overline{i_V}$, are taken from the camera and it knows the length, n, of a side of the square area to be navigated. It assumes that $n > 0$ and this n is equal to that used by the **Infrared** node. The outputs, $\overline{o_V}$, are the rover's initial position, s_0, and a set of grid locations, *Obs*, where obstacles (e.g. walls) have been detected. It assumes, $\mathcal{A}_V(\overline{i_V})$,that $n > 0$. It guarantees, $\mathcal{G}_V(\overline{o_V})$, that s_0 is not an obstacle location and it is the rover's current physical position. Furthermore, every element of *Obs* is within the navigated square area and there is a physical obstacle at each of these locations.

Infrared: The inputs, $\overline{i_H}$, are from a heat camera/sensor and it knows the length of the grid. It assumes that $n > 0$ and this n is equal to that used by the **Vision** node. It outputs, $\overline{o_H}$, a set H of heat locations and temperatures observed. Elements of H are guaranteed, $\mathcal{G}_A(\overline{o_H})$, to lie within the navigated area and greater than normal heat is physically recorded at each indicating that a human may be present.

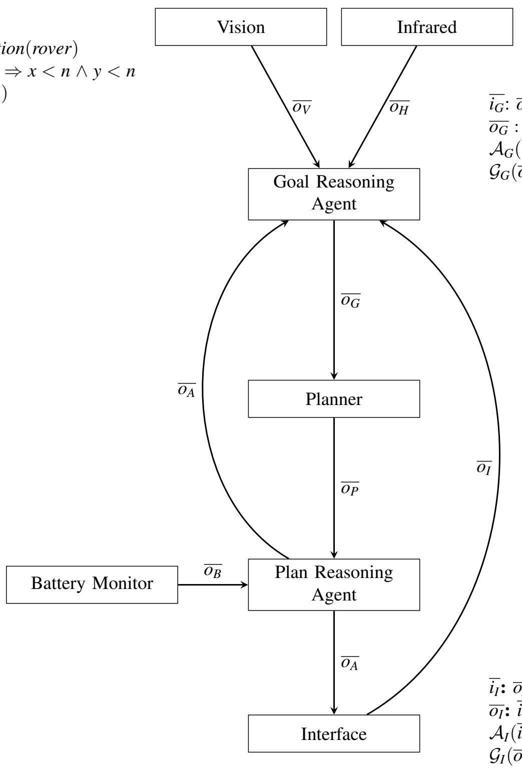
Goal Reasoning Agent: The inputs to this node are the outputs ($\overline{o_V}$, $\overline{o_H}$, $\overline{o_A}$ and $\overline{o_I}$) from those above and the **Plan Reasoning Agent** and **Interface**. We propagate all information between the nodes in our system, and so the outputs of this node, $\overline{o_G}$, are its inputs and the goal that it has selected. It assumes that the guarantees of the nodes that it takes input from are preserved (for nodes that have not yet been executed this devolves to *true*). It then constructs a *GoalSet* by adding the location of the charger, *chargepos*, to H from which it chooses the warmest goal location, g, with $g \notin \mathit{Obs}$. This is the guarantee, unless the rover has to recharge (as indicated by the **Plan Reasoning Agent**) in which case it sets $g = \mathit{chargepos}$. Once the **Interface** has reached the goal location and updated s_0, it informs the **Goal Reasoning Agent** so that this location is removed from the set of potential goals and the process loops until all potential goals are visited.

Planner: This node takes input from those above and outputs a set of plans, *PlanSet*, between the start location, s_0, and the goal, g. As expected, it assumes $\mathcal{G}_G(\overline{o_G})$ holds. We treat each plan as a set of coordinates (in order to avoid duplicates) and we guarantee that the plans are obstacle-free. We also ensure that each plan contains both s_0 and g, as well as ensuring that for each grid coordinate in the plan (that is not s_0 or g which are treated separately) there are two points adjacent to it. This is so that the plan can be sequentialised for execution.

Battery Monitor: Takes input from sensors and assumes that the rover has been turned on, this simple node returns the remaining percentage of battery power.

Plan Reasoning Agent: Takes input from the previous two nodes and outputs its chosen *plan* and the *recharge* boolean flag. The chosen *plan* is guaranteed to be the shortest one in *PlanSet* and is achievable using the remaining battery power otherwise it returns an empty plan. We assume that each step in the plan costs 1% battery power and when the battery is below a certain threshold the *recharge* flag is toggled.

Interface: Receives a *plan* from the **Plan Reasoning Agent**, and issues commands to the hardware to execute it. It tracks if the rover has moved (*haveMoved*) and if it has reached the goal (*atGoal*). When the plan is complete, it alerts the **Goal Reasoning Agent** and updates the start position (s_0), so that the subsequent plans start from this new location.



$\overline{i_V}$**:** from camera, $n \in \mathbb{N}$ (length of square area to be navigated)
$\overline{o_V}$**:** s_0 = (x, y), $Obs = \{(x,y)\}$
set of obstacle locations
$\mathcal{A}_V(\overline{i_V}) : n > 0$
$\mathcal{G}_V(\overline{o_V}) : s_0 \notin Obs \wedge s_0 = position(rover)$
$\quad \wedge \forall x, y \cdot (x,y) \in Obs \Rightarrow x < n \wedge y < n$
$\quad \wedge\ obstacle((x,y))$

$\overline{i_H}$**:** from camera/sensor, $n \in \mathbb{N}$ (length of square area to be navigated)
$\overline{o_H}$**:** $H = \{((x,y),h)\}$ (set of heat locations and heat value)
$\mathcal{A}_H(\overline{i_H}) : n > 0$
$\mathcal{G}_H(\overline{o_H}) : h \in \mathbb{N} \wedge \forall x, y, h \cdot ((x,y),h) \in H \Rightarrow x < n$
$\quad \wedge\ y < n \wedge heat((x,y))$

$\overline{i_G}$: $\overline{o_V} \cup \overline{o_H} \cup \overline{o_A} \cup \overline{o_I}$
$\overline{o_G}$: $\overline{I_G} \cup \{g = (x,y)\}$
$\mathcal{A}_G(\overline{i_G}) : \mathcal{G}_V(\overline{o_V}) \wedge \mathcal{G}_H(\overline{o_H}) \wedge \mathcal{G}_A(\overline{o_A}) \wedge \mathcal{G}_I(\overline{o_I})$
$\mathcal{G}_G(\overline{o_G}) : (GoalSet = H \cup \{(chargePos, 0)\}) \wedge (g \notin Obs)$
$\quad \wedge (\exists h \cdot h \in \mathbb{N} \wedge (g,h) \in GoalSet \wedge$
$\qquad (\forall x, y, h_1 \cdot ((x,y),h_1) \in GoalSet \wedge g \neq chargepos \Rightarrow h \geq h_1))$
$\quad \wedge (recharge \Leftrightarrow g = chargePos)$
$\quad \wedge ((atGoal \wedge s_0 \neq chargePos)$
$\qquad \Rightarrow GoalSet = GoalSet \setminus \{s_0\})$

$\overline{i_P}$**:** $\overline{o_G}$
$\overline{o_P}$**:** $\overline{o_G} \cup PlanSet = \{\{(x,y)\}\}$ set of sets of (x,y) coordinates
$\mathcal{A}_P(\overline{i_P}) : \mathcal{G}_G(\overline{o_G})$
$\mathcal{G}_P(\overline{o_P}) : g = s_0 \Rightarrow PlanSet = \{\varnothing\}$
$\quad \wedge \forall p, x, y \cdot p \in PlanSet \wedge (x,y) \in p \Rightarrow (x,y) \notin Obs$
$\quad \wedge \forall p \cdot p \in PlanSet \wedge |p| \geq 2 \Rightarrow s_0 \in p \wedge g \in p$
$\qquad \wedge (\exists q, r \cdot q, r \in p \wedge adjacent(s_0, q) \wedge adjacent(g, r))$
$\quad \wedge \forall p, p_0 \cdot p \in PlanSet \wedge p_0 \in p \wedge p_0 \neq s_0 \wedge p_0 \neq goal \wedge |p| \geq 3$
$\qquad \Rightarrow (\exists q, r \cdot q, r \in p \wedge adjacent(q, p_0) \wedge adjacent(r, p_0))$

$\overline{i_B}$**:** from sensor
$\overline{o_B}$**:** $b \in \mathbb{N}$
$\mathcal{A}_B(\overline{i_B}) : on(rover)$
$\mathcal{G}_B(\overline{o_B}) : 0 \leq b \leq 100$

$\overline{i_A}$**:** $\overline{o_P} \cup \overline{o_B}$
$\overline{o_A}$**:** $\overline{o_P} \cup \{plan = \{(x,y)\}, recharge \in \{true, false\}\}$
$\mathcal{A}_A(\overline{i_A}) : \mathcal{G}_P(\overline{o_P})$
$\mathcal{G}_A(\overline{o_A}) : plan \in PlanSet$
$\quad \wedge \forall p \cdot p \in PlanSet \Rightarrow p \neq plan \Rightarrow length(plan) \leq length(p)$
$\quad \wedge\ plan = \varnothing \Rightarrow (PlanSet = \{\varnothing\} \vee length(plan) \leq batteryLow(b))$
$\quad \wedge\ recharge \Leftrightarrow length(plan) \leq batteryLow(b)$

$\overline{i_I}$**:** $\overline{o_A}$
$\overline{o_I}$**:** $\overline{i_I} \wedge haveMoved \in \{true, false\} \wedge atGoal \in \{true, false\}$
$\mathcal{A}_I(\overline{i_I}) : \mathcal{G}_A(\overline{o_A})$
$\mathcal{G}_I(\overline{o_I}) : (haveMoved \Rightarrow plan \neq \varnothing)$
$\quad \wedge (atGoal \Rightarrow haveMoved \wedge position(rover) = g \wedge s_0 = g)$

Fig. 2. We have summarised the inputs, $\overline{i}$, outputs, $\overline{o}$, and the FOL Assume-Guarantee descriptions for each node in the system.

Thus, we have describe the Assume-Guarantee specifications that we have constructed for our case study example as illustrated in Fig. 2. Next, we show how the calculus presented in §III can be used to derive system-level properties that can be used for certification purposes.

B. Deriving System-Level Properties

After constructing the above FOL Assume-Guarantee specifications for the individual nodes in our system, we employ the inference rules that were introduced in §III to derive the following overall system properties which we envision can be used as evidence for certification. Starting at the top of Fig. 2, the **Vision**, **Infrared**, **Interface** and **Plan Reasoning Agent** nodes each provide input to the **Goal Reasoning Agent** node. Thus we use R3 to account for this joining behaviour:

$$\frac{\begin{array}{l}\forall \overline{i_V}, \overline{o_V} \cdot \mathcal{A}_V(\overline{i_V}) \;\Rightarrow\; \Diamond \mathcal{G}_V(\overline{o_V}) \\ \forall \overline{i_H}, \overline{o_H} \cdot \mathcal{A}_H(\overline{i_H}) \;\Rightarrow\; \Diamond \mathcal{G}_H(\overline{o_H}) \\ \forall \overline{i_A}, \overline{o_A} \cdot \mathcal{A}_A(\overline{i_A}) \;\Rightarrow\; \Diamond \mathcal{G}_A(\overline{o_A}) \\ \forall \overline{i_I}, \overline{o_I} \cdot \mathcal{A}_I(\overline{i_I}) \;\Rightarrow\; \Diamond \mathcal{G}_I(\overline{o_I}) \\ \forall \overline{i_G}, \overline{o_G} \cdot \mathcal{A}_G(\overline{i_G}) \;\Rightarrow\; \Diamond \mathcal{G}_G(\overline{o_G}) \\ \overline{i_G} = \overline{o_V} \cup \overline{o_H} \cup \overline{o_A} \cup \overline{o_I} \\ \vdash\; \forall \overline{i_G}, \overline{o_V}, \overline{o_H}, \overline{o_A}, \overline{o_I} \cdot \mathcal{G}_V(\overline{o_V}) \wedge \mathcal{G}_H(\overline{o_H}) \wedge \mathcal{G}_A(\overline{o_A}) \wedge \mathcal{G}_I(\overline{o_I}) \\ \qquad \Rightarrow \mathcal{A}_G(\overline{i_G})\end{array}}{\begin{array}{l}\forall \overline{i_V}, \overline{i_H}, \overline{i_A}, \overline{i_I}, \overline{o_G} \cdot \mathcal{A}_V(\overline{i_V}) \wedge \mathcal{A}_H(\overline{i_H}) \wedge \mathcal{A}_A(\overline{i_A}) \wedge \mathcal{A}_I(\overline{i_I}) \\ \qquad \Rightarrow \Diamond \mathcal{G}_G(\overline{o_G})\end{array}}$$

We can thus conclude that if the assumptions of the sensor nodes hold, then eventually some goal location as specified by $\mathcal{G}_G(\overline{o_G})$ will be chosen by the **Goal Selector Agent**.

Next, we can use R1 on the simple combination of the **Goal Reasoning Agent** and **Planner** nodes to give the following:

$$\frac{\begin{array}{l}\forall \overline{i_G}, \overline{o_G} \cdot \mathcal{A}_G(\overline{i_G}) \;\Rightarrow\; \Diamond \mathcal{G}_G(\overline{o_G}) \\ \forall \overline{i_P}, \overline{o_P} \cdot \mathcal{A}_P(\overline{i_P}) \;\Rightarrow\; \Diamond \mathcal{G}_P(\overline{o_P}) \\ \overline{o_G} = \overline{i_P} \\ \vdash\; \forall \overline{o_G}, \overline{i_P} \cdot \mathcal{G}_G(\overline{o_G}) \;\Rightarrow\; \mathcal{A}_P(\overline{i_P})\end{array}}{\forall \overline{i_G}, \overline{o_P} \cdot \mathcal{A}_G(\overline{i_G}) \;\Rightarrow\; \Diamond \mathcal{G}_P(\overline{o_P})}$$

In this way, we conclude that if an appropriate goal has been chosen then the **Planner** node behaves as expected.

Then, the **Battery Monitor** and the **Planner** join (using R3) as follows and give us input to the **Plan Reasoning Agent**:

$$\frac{\begin{array}{l}\forall \overline{i_B}, \overline{o_B} \cdot \mathcal{A}_B(\overline{i_B}) \;\Rightarrow\; \Diamond \mathcal{G}_B(\overline{o_B}) \\ \forall \overline{i_P}, \overline{o_P} \cdot \mathcal{A}_P(\overline{i_P}) \;\Rightarrow\; \Diamond \mathcal{G}_P(\overline{o_P}) \\ \forall \overline{i_A}, \overline{o_A} \cdot \mathcal{A}_A(\overline{i_A}) \;\Rightarrow\; \Diamond \mathcal{G}_A(\overline{o_A}) \\ \overline{i_A} = \overline{o_B} \cup \overline{o_P} \\ \vdash\; \forall \overline{i_A}, \overline{o_B}, \overline{o_P}, \cdot \mathcal{G}_B(\overline{o_B}) \wedge \mathcal{G}_P(\overline{o_P}) \Rightarrow \mathcal{A}_A(\overline{i_A})\end{array}}{\forall \overline{i_B}, \overline{i_P} \cdot \mathcal{A}_B(\overline{i_B}) \wedge \mathcal{A}_P(\overline{i_P}) \Rightarrow \Diamond \mathcal{G}_A(\overline{o_A})}$$

Here, when the assumptions of the **Battery Monitor** and **Planner** nodes are met then eventually the **Plan Reasoning Agent** chooses a suitable plan as described by its guarantee.

We then capture the branching behaviour (using R2) from the **Plan Reasoning Agent** as follows:

$$\begin{array}{l}
\forall \overline{i_A}, \overline{o_A} \cdot \mathcal{A}_A(\overline{i_A}) \;\Rightarrow\; \Diamond \mathcal{G}_A(\overline{o_A}) \\
\forall \overline{i_G}, \overline{o_G} \cdot \mathcal{A}_G(\overline{i_G}) \;\Rightarrow\; \Diamond \mathcal{G}_G(\overline{o_G}) \\
\forall \overline{i_I}, \overline{o_I} \cdot \mathcal{A}_I(\overline{i_I}) \;\Rightarrow\; \Diamond \mathcal{G}_I(\overline{o_I}) \\
\overline{o}_A = \overline{i_G} \cup \overline{i_I} \\
\vdash \forall \overline{o_A}, \overline{i_G}, \overline{i_I} \cdot \mathcal{G}_A(\overline{o_A}) \;\Rightarrow\; \mathcal{A}_G(\overline{i_G}) \wedge \mathcal{A}_I(\overline{i_I}) \\
\hline
\forall \overline{i_A}, \overline{o_G}, \overline{o_I} \cdot \mathcal{A}_A(\overline{i_A}) \;\Rightarrow\; \Diamond \mathcal{G}_G(\overline{o_G}) \wedge \Diamond \mathcal{G}_I(\overline{o_I})
\end{array}$$

Now, we consider the loops in Fig 2. Note that we treat the **Vision** and **Infrared** nodes here as one to simplify the following application of R4 (which we extend, as mentioned earlier to account for multiple nodes in the middle) for the loop between the **Goal Reasoning Agent** and the **Plan Reasoning Agent**:

$$\begin{array}{l}
\forall \overline{i_V}, \overline{o_V}, \overline{i_H}, \overline{o_H} \cdot \mathcal{A}_V(\overline{i_V}) \wedge \mathcal{A}_H(\overline{i_H}) \Rightarrow \Diamond \mathcal{G}_V(\overline{o_V}) \wedge \Diamond \mathcal{G}_H(\overline{o_H}) \\
\forall \overline{i_G}, \overline{o_G} \cdot \mathcal{A}_G(\overline{i_G}) \;\Rightarrow\; \Diamond \mathcal{G}_G(\overline{o_G}) \\
\forall \overline{i_P}, \overline{o_P} \cdot \mathcal{A}_P(\overline{i_P}) \;\Rightarrow\; \Diamond \mathcal{G}_P(\overline{o_P}) \\
\forall \overline{i_A}, \overline{o_A} \cdot \mathcal{A}_A(\overline{i_A}) \;\Rightarrow\; \Diamond \mathcal{G}_A(\overline{o_A}) \\
\forall \overline{i_I}, \overline{o_I} \cdot \mathcal{A}_I(\overline{i_I}) \;\Rightarrow\; \Diamond \mathcal{G}_I(\overline{o_I}) \\
\overline{i_G} = \overline{o_V} \cup \overline{o_H} \cup \overline{o_A} \wedge \overline{i_A} = \overline{o_P} \wedge \overline{o_A} \subseteq \overline{i_I} \\
\vdash \forall \overline{i_G}, \overline{o_A} \cdot \mathcal{A}_G(\overline{i_G}) \Rightarrow \Diamond \mathcal{G}_A(\overline{o_A}) \\
\vdash \forall \overline{i_A}, \overline{o_G}, \overline{o_I} \cdot \mathcal{A}_A(\overline{i_A}) \;\Rightarrow\; \Diamond \mathcal{G}_G(\overline{o_G}) \wedge \Diamond \mathcal{G}_I(\overline{o_I}) \\
\vdash \forall \overline{i_V}, \overline{i_H}, \overline{i_A}, \overline{o_G} \cdot \mathcal{A}_V(\overline{i_V}) \wedge \mathcal{A}_H(\overline{i_H}) \wedge \mathcal{A}_A(\overline{i_A}) \;\Rightarrow\; \Diamond \mathcal{G}_G(\overline{o_G}) \\
\hline
\forall \overline{i_V}, \overline{i_H}, \overline{o_I} \cdot \mathcal{A}_V(\overline{i_V}) \wedge \mathcal{A}_H(\overline{i_H}) \;\Rightarrow\; \Diamond \mathcal{G}_I(\overline{o_I})
\end{array}$$

and, assuming that the loop exiting the **Interface** terminates (which we can show since the size of *GoalSet* $\setminus$ *chargePos* decreases with each iteration), we use the same approach to derive:

$$\forall \overline{i_v}, \overline{i_H} \cdot \mathcal{A}_V(\overline{i_V}) \wedge \mathcal{A}_H(\overline{i_H}) \Rightarrow \Diamond \mathit{true}$$

These properties can be used as certification evidence for regulators of robotic systems. Moreover, they allow us to relate different nodes so long as their FOL Assume-Guarantee specifications have been verified. This verification can take place in a number of distinct formalisms and we provide examples of this in the next section.

C. Heterogeneous Specification and Verification

Each of the nodes in a system can present their own verification challenges to ensure compliance with their corresponding FOL specifications. Thus, for each distinct node, we have chosen a suitable specification and verification approach[1].

We have modelled the **Planner** node using the Event-B formal specification language which is a model-based approach to specification. The **Goal Reasoning Agent** and **Plan Reasoning Agent** nodes are implemented using the GWENDOLEN agent programming language and verified using its associated model-checker, Agent Java PathFinder (AJPF). The **Interface** is required to communicate with hardware components and, as such, we have used the CSP process algebra which is specialised for verifying communication protocols. Note that the **Vision**, **Infrared** and **Battery Monitor** sensor nodes were not formally verified as they contain an environmentally dependent component (e.g. machine learning) so their verification is achieved through *testing*.

[1]Link to full specifications omitted due to double blind review.

1) Testing: Sensor Nodes: The sensor nodes (**Vision**, **Infrared**, and **Battery Monitor**) are "black box" implementations in some programming language. This kind of node is generally very difficult to apply formal verification techniques to as it may contain adaptive and/or learning components that are notoriously difficult to formally verify. However, we assume that a range of software testing techniques, including field tests and simulation, were used to validate and thus verify the sensor nodes. These testing techniques would include testing of the Assume-Guarantee specifications for each node. There has been much work in the area of using formal specifications to guide testing and a detailed overview of the various approaches is given in [27].

2) Event-B: Planner Node: We have used the industrial-strength Event-B formal specification language [28] to model and verify the **Planner** node. Event-B uses a set-theoretic notation and first-order logic to model and verify a state-based system. Event-B specifications consist of *contexts*, that model the static components of a system's specification, and *machines* for the dynamic components. Contexts typically contain carrier sets, constants and axioms. Machines typically define variables, invariants, variants and events. Core to the Event-B approach is the notion of formal refinement where a developer can specify an abstract system specification and gradually refine to a concrete implementation in a provably correct way [29]. Tool support for Event-B is achieved via its Eclipse-based IDE, the Rodin Platform that facilitates the generation of proof obligations with support for both automatic and interactive proof [30].

Since Event-B supports the use of first-order logic, it is relatively easy to encode the FOL Assume-Guarantee specifications for the **Planner** node that was presented in Fig. 2. In particular, $\mathcal{G}_P(\overline{o_P})$ is captured by the Event-B invariants:

$$\begin{array}{ll}
\texttt{inv1}: & \forall p \cdot p \in \mathit{PlanSet} \Rightarrow p \subseteq \mathit{grid} \wedge (\mathit{card}(p) \geq 2 \Rightarrow (g \in p \wedge s0 \in p)) \\
\texttt{inv2}: & g = s0 \;\Rightarrow\; \mathit{PlanSet} = \{\varnothing\} \\
\texttt{inv3}: & \forall p, x, y \cdot p \in \mathit{PlanSet} \wedge (x \mapsto y) \in p \Rightarrow (x \mapsto y) \notin \mathit{Obs} \\
\texttt{inv6}: & \forall p \cdot p \in \mathit{PlanSet} \Rightarrow p \subseteq \mathit{grid} \wedge (\mathit{card}(p) \geq 2 \\
 & \quad \Rightarrow (\exists q, r \cdot q \in p \wedge r \in p \wedge (\mathit{adjacent}(s0 \mapsto q) = \mathit{TRUE}) \\
 & \qquad \wedge (\mathit{adjacent}(r \mapsto g) = \mathit{TRUE}))) \\
\texttt{inv7}: & \forall p, p0 \cdot p \in \mathit{PlanSet} \wedge p0 \in p \Rightarrow p \subseteq \mathit{grid} \wedge p0 \in \mathit{grid} \\
 & \quad \wedge (\mathit{card}(p) \geq 3 \wedge p0 \neq g \wedge p0 \neq s0 \\
 & \quad \Rightarrow (\exists q, r \cdot q \in p \wedge r \in p \wedge (\mathit{adjacent}(p0 \mapsto q) = \mathit{TRUE}) \\
 & \qquad \wedge (\mathit{adjacent}(r \mapsto p0) = \mathit{TRUE})))
\end{array}$$

Here, $\mapsto$ is the tuple notation used in Event-B. The above invariants have been shown to hold before and after each event in the corresponding Event-B machine which contains the following events: *addstart*, *addcurrentplan*, *moveLeft*, *moveRight*, *moveUp* and *moveDown*. In Event-B, events are triggered so long as their guards hold and so we traverse adjacent, obstacle-free squares until we reach g from s_0. In total 43 proof obligations were generated by Rodin for this machine and the majority were proven automatically with 10 requiring interactive proof. Note that the *adjacent* function has been specified using axioms in a context as shown below and the cardinality function, *card*, is native to Event-B.

axm2: $grid \subseteq \mathbb{N} \times \mathbb{N}$
axm6: $adjacent \in (grid \times grid) \rightarrow BOOL$
axm8: $distance \in (grid \times grid) \rightarrow \mathbb{N}$
axm9: $\forall a,b,c,d \cdot a \mapsto b \in grid \land c \mapsto d \in grid$
$\land (distance((a \mapsto b) \mapsto (c \mapsto d)) \neq 1)$
$\Rightarrow (adjacent((a \mapsto b) \mapsto (c \mapsto d)) = FALSE)$

This Event-B specification describes a simple path finding algorithm and subsequent refinement steps in this specification would potentially refine this to a more efficient and detailed algorithm to be implemented on the final system.

Our Event-B specification was guided by the FOL Assume-Guarantee specification of the **Planner** (Fig. 2) and it was relatively straightforward to show that the constructed Event-B specification preserved these properties since it supports first-order logic, although there are some minor syntactic differences. The main difficulties that we encountered were in proving that the *adjacent* function was well defined and our use of *card* also generated some extra proof obligations that required interactive proof. Overall, linking the FOL Assume-Guarantee specification and the one in Event-B required minimal effort since the latter supports first-order logic.

3) GWENDOLEN: Agent Nodes: We specify the agent nodes in GWENDOLEN [31] and then use the AJPF [32] model-checker to verify that both agents comply with their FOL guarantees. GWENDOLEN agents follow the Belief-Desire-Intention (BDI) model [33]. 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.

AJPF is an extension of Java PathFinder (JPF) [34], a model-checker that works directly on Java program code instead of on a mathematical model of the program's execution. This extension allows for formal verification of BDI-based agent programming languages by providing a property specification language based in LTL that supports the description of terms usually found in BDI agents.

The **Goal Reasoning Agent** receives a *GoalSet* from the **Infrared** node where each element contains a goal triplet (X, Y, H) where X and Y are the coordinates of the goal and H is the heat value. The set of intentions available to an agent is generated by reasoning over plans in the agent's plan library (not to be confused with the plans that come from the **Planner** node). The specification of a plan starts with an *event*. For example, the plan shown below in line 1 is activated when the goal (!) `select_goal` is added (+). A plan is selected if the formulae in the *guard* (followed by a colon) are true. After a plan is selected, a sequence of actions in the plan body (denoted by $\leftarrow$) is executed.

```
1  +!select_goal [perform]
2       : { ℬ recharge }
3       ← +goal(recharge);
4  +goal(recharge)
5       : { ℬ chargePos(X,Y) }
6       ← +goal(X,Y);
7  +atGoal
8       : { ℬ initPos(X,Y), ~ ℬ chargePos(X,Y) }
9       ← −goal(X,Y),
10        +!remove_goal_from_set(goal(X,Y)) [perform];
```

The first plan is selected if the agent has the `recharge` belief. Then, the `goal(recharge)` belief is added, triggering the second plan (line 4) which queries the agent's belief base and unifies X and Y with the coordinates of the charger, adding it as the next goal. The last plan (line 7) is activated when the belief `atGoal` is added (input from the **Interface** node). The guard of the plan tests if the agent's current position (initial position updated by the **Interface**) is different to the charger position. If that is the case, then the goal that was previously selected is removed.

Using AJPF, we prove the following property that represents the $G_G(\overline{o_G})$ of the **Goal Reasoning Agent** node:

$$\square\,(\mathcal{B}_{\texttt{rover}}\, recharge \Rightarrow \mathcal{B}_{\texttt{rover}}\, goal(recharge))$$

This property states that it is always the case ($\square$) that if the rover agent believes it needs to recharge, then it will select recharge as the next goal.

The **Plan Reasoning Agent** receives a *PlanSet* from the planner where each *Plan* is a set of coordinates. Each *Plan* in *PlanSet* is translated into a belief `plan(P,L)`, where `P` is the sequence of actions and `L` is the plan's length (number of actions in a plan).

```
1  +!select_plan(ExpectedBattery)
2       : { ℬ threshold(T), T < ExpectedBattery }
3       ← +execute;
4  +!select_plan(ExpectedBattery)
5       : { ℬ threshold(T), ExpectedBattery < T }
6       ← +recharge;
```

Both plans above match the event for the addition of the goal `select_plan(P, ExpectedBattery)`, where P is the plan with lowest length and *ExpectedBattery* is the current battery of the rover (as provided by the **Battery Monitor** node) minus the length of P. The guard of the first plan (line 2) tests that given a belief `threshold(T)`, T is lower than *ExpectedBattery*, where T is the minimum amount of battery required to arrive at a charge position from anywhere in the map (given by the *batteryLow* function). If this is true, then the agent adds the belief that it should execute P. Otherwise, the agent adds the belief that it should recharge its batteries.

The following AJPF property returns no error when proving the $G_A(\overline{o_A})$ of the **Plan Reasoning Agent** node:

$$\lozenge\,(\mathcal{B}_{\texttt{rover}}\, execute \lor \mathcal{B}_{\texttt{rover}}\, recharge)$$

That is, that the agent will eventually ($\lozenge$): select a *Plan* from the *PlanSet* and send it to the **Interface** for execution; or send the recharge flag to the **Goal Reasoning Agent**.

For brevity, other parts of the Gwendolen code and its related AJPF properties were omitted, such as how to choose the shortest plan in the **Plan Reasoning Agent** and the highest heat value in the **Goal Reasoning Agent**. Plans that treat the belief additions of `recharge`, `goal(X,Y)`, and `execute`, are in place to send messages to the **Goal Reasoning Agent**, **Planner**, and **Interface** nodes, respectively.

4) CSP: Interface Node: This section shows how we verify the **Interface** node using the process algebra Communicating Sequential Processes (CSP) [35], which is designed for concurrent communicating systems. We specify both the behaviour that characterises a desired system property and the system itself in CSP, then use the FDR 4 model checker [36] to show that the system behaves according to the specified property. This can be thought of as checking that the system implements the specified behaviour.

Here, we briefly describe the relevant CSP notation. CSP specifications are built out of processes. A process may take parameters, and describes a sequence of events; $a \rightarrow b \rightarrow Skip$ is the process where the events a and b happen sequentially, followed by *Skip* which is the terminating process. Processes are connected by bi-directional channels. An event is a communication on a channel, which may accept input ($a?in$), produce output ($b!out$), or require a particular event on that channel ($c.param$). Parameters must match the specified type for that channel. Further, a restricted input can be specified ($d?p : set$) which allows any communications on d that matches the channel's type and are in the *set*. $P \Box Q$ provides the option of either P or Q to the process's environment, once one process is picked the other becomes unavailable. Additionally, processes can occur in sequence or run in parallel.

The **Interface** node is specified by the `INTERFACE` process, which is composed of three parallel processes: `EXECUTOR` takes each waypoint in the plan and issues a move command, `MOTORS` accepts move commands, and `LOCATION` tracks the robot's location. `MOTORS` and `LOCATION` run until the robot turns off, `EXECUTOR` executes one plan and recurses.

The `INTERFACE` process has two channels that are visible to its environment: `input` and `output`. These two channels correspond to the node's input and output, as specified in §IV-A. The `input` channel takes two parameters: the plan, represented as a set of coordinate pairs; and the goal location, represented as a coordinate pair. The `output` channel also takes two parameters, both are booleans: `hasMoved` is true when the robot has moved from its start location, and `atGoal` is true when the robot is at the goal location. We use these two channels in the verification step, to link the node's input to its output.

We capture the **Interface** node's guarantees as three CSP specifications. The first guarantee is described by the `emptyPlan` and `nonEmptyPlan` processes, and the second is described by the `atTheGoal` process.

The **Interface** node's first guarantee is that if `haveMoved` is `True` then the `plan` $\neq \varnothing$. This is captured by the `emptyPlan` and `nonEmptyPlan` processes, shown below. The `emptyPlan` process accepts only the empty set for the first parameter of the `input` channel (the plan) and then always communicates `False` as the first parameter of the `output` channel (`hasMoved`). Note that the second parameter of the `input` channel can be any value of the right type (represented by `?_`) because we are not interested in its value for this property.

```
emptyPlan = (
  input.∅?_  → output.False.False→ emptyPlan
  □
  input.∅?_  → output.False.True→ emptyPlan )
```

We use the `emptyPlan` process to check that if the `INTERFACE` process is given an empty set for its plan, then it will only communicate `haveMoved` as `False` on the `output` channel. We use a check on the process's failures, which means that the `INTERFACE` process can perform the same behaviour as `emptyPlan` and cannot refuse to do anything specified by `emptyPlan`.

The `nonEmptyPlan` process, below, accepts a possibly empty plan on the `input` channel (the first parameter) and then communicates on `output`. If the plan received on the `input` channel is not empty (it is in `setOfNonEmptyPlans`), then the first parameter (`haveMoved`) will always be `True`. As in the `emptyPlan` process, if the plan is the empty set, then and the first parameter of `output` will always be `False`.

```
nonEmptyPlan =
  input?p:setOfNonEmptyPlans?_ →
    ((output.True.False → nonEmptyPlan)
      □
     (output.True.True → nonEmptyPlan))
  □
  input.∅?_ →
    output.False.False → nonEmptyPlan
```

We use this specification to check that if the `INTERFACE` process is given a non-empty plan, then it will only communicate `haveMoved` as `True` on the `output` channel; or, if it is given an empty plan, then it will only communicate `haveMoved` as `False`. We use a check on the process's traces, which means that the `INTERFACE` process can perform all the behaviours specified by the `nonEmptyPlan` process. This is a weaker model than for the refinement of the `emptyPlan` process, because `nonEmptyPlan` does not capture enough information about the robot's location to specify the value of the `output` channel's second parameter (`atGoal`). Thus, it may refuse to perform, for example, `output.True.True` if the plan it receives does not get the rover to the goal location. Checking `atGoal` is the focus of the second guarantee.

The **Interface** node's second guarantee is that if `atGoal` is `True`, then `haveMoved` is `True` and the rover's position is the goal location. This property is specified by the `atTheGoal` process, which we omit for brevity. The `atTheGoal` process accepts a plan and a goal on the `input` channel, then outputs `haveMoved` as described above and `atGoal` as `True` if the robot's final position matches the goal location, and `False` if it is not. This process assumes that if the plan is not empty, then the robot will eventually move to the last location in the plan, which is supported by the `nonEmptyPlan` process above.

We then check that if the `INTERFACE` process is given a plan where the final location is the goal location, then it will output `atGoal` as `True`. We use a check on the process's failures and divergences, which means that `INTERFACE` can perform the same events as, and will not refuse any of the events in, the `atTheGoal` process and will not perform an endless sequence of events that are not in `atTheGoal`.

D. Modularity

One of the advantages of our approach is that we can swap a node with another (similar) node and apply different specification languages and verification tools, as long as they still conform with the FOL. For example, we might not need a BDI agent as the plan reasoner if we can instead use a simpler technique that can satisfy the same properties in $G_A(\overline{o_A})$. To demonstrate this modularity, we specify the **Plan Reasoning Agent** in Dafny.

Dafny [37], initially developed at Microsoft Research, is an imperative programming language that supports formal verification through the use of its embedded program verifier. It mixes pre- and post-conditions (similar to Assume-Guarantees) with more traditional object-oriented programming features such as classes and datatypes. The keyword `ensures` is used to encode the post-conditions, which in this case are the $G_A(\overline{o_A})$ of the **Plan Reasoning Agent** node:

```
1 ensures plan in PlanSet;
2 ensures b < threshold ⇒ recharge == true && plan == [];
3 ensures threshold ≤ b && |plan| ≤ b ⇒ recharge == false;
4 ensures threshold ≥ b && recharge == false && plan != [] &&
|plan| ≤ b ⇒ forall x :: x in PlanSet ⇒ |x| ≥ |plan|;
```

The Dafny code above shows the four $G_A(\overline{o_A})$ properties as established in our FOL model in Fig. 2. The first property (Line 1) is straightforward and exactly follows the FOL. The property on line 2 combines the last two properties of our FOL model, that is, if the rover does not have enough battery then it will not choose a plan to execute. Instead, it will decide to recharge its batteries. Lines 3 and 4 verify the property that states that the plan with the smallest length will be selected for execution, as long as the rover has enough battery to execute this plan and charge its batteries after (indicated by the threshold value). The remaining Dafny code is omitted, but it follows the specification in GWENDOLEN, except that instead of using BDI concepts it uses traditional imperative programming languages constructs.

Compared to the GWENDOLEN specification and AJPF verification, Dafny required fewer lines of code and the properties more closely resemble the FOL model. However, it loses the reasoning capabilities that BDI agents provide, and is not directly executable. But because Dafny is an imperative programming language, it can be easier than with other verification techniques to refine it into executable code.

Specifying and verifying the same node using multiple techniques and tools, assuming that all of these techniques have proved to comply with the FOL model, can increase the confidence in the verification of the node. Such evidence could be useful in the reconfigurability [38], [39] of modular robotic systems, for example, to aid in the selection of a potentially safer node during on-line node replacement.

V. DISCUSSION

In this section, we discuss how our approach fits into the process of developing heterogeneous robotic systems. To validate our work, we leverage existing tool support to examine the case study that we have illustrated in the previous section. Furthermore, we discuss how to define a measure of confidence in the verification approach used for the individual nodes and how these might be combined to give an overall measure of confidence in the system.

A. Development Process

As previously mentioned, our approach enables the specification of high-level requirements to guide the specification and verification of a system using the most suitable techniques. This relies on the existing tool(s) for whichever verification approaches are chosen. This can include testing frameworks, model checkers, or theorem provers. Our approach is, therefore, useful throughout the development process.

Our approach produces development artefacts that unambiguously describe the system's requirements, and provide traceability of those requirements to the specification and verification of the modules in the system. First, this can help improve reuse or refactoring of the system. Second, with robotic systems being increasingly used in safety-critical situations, it can also help to provide robust evidence of a system's safety or security properties [12].

Our approach assumes that the system being developed is modular. We envisage a node-based system (such as Robot Operating System (ROS) [40]) but the approach could be applied to a system where the modules are program classes or methods. Our approach can be applied to both top-down and bottom-up development.

For a top-down approach, we begin by defining the nodes in the system and writing the FOL Assume-Guarantee conditions for them. This specification of the node's requirements is used to guide the specification and verification of the nodes. This is the approach we describe in §IV. For a bottom-up approach, we begin with the nodes (either as software modules or as a formal or non-formal specification of the node) and any existing verification of the node (either formal checks or non-formal tests). From these we construct the FOL Assume-Guarantee conditions, which describe system-level properties.

Section IV shows how we utilise existing tool support to verify the individual nodes in our case study. The variety of distinct tools available for formal verification makes it difficult to list all possible translations from FOL. However, we have chosen FOL as a unifying language because it forms the basis of several verification tools and techniques (e.g. [11], [28]).

Some of the nodes in our case study (§IV) were specified using notations that do not include FOL, for example GWENDOLEN and CSP. These were intentional choices to show how our approach can be effective without an implementation of FOL in the target specification language. This is especially evident in §IV-C1, where the FOL requirements are used to guide the verification of the sensors nodes by generating test cases. The same sort of process is used to go from the FOL to GWENDOLEN or CSP.

Specifying agents in GWENDOLEN is very different from FOL, however, using the FOL specification to guide the agent verification made it easier to address the properties that we wanted to guarantee. These properties are written in LTL (which is inherently similar to FOL), but contain specific BDI concepts with their own semantics [32]. Using CSP to specify

TABLE I
CONFIDENCE LEVELS

Confidence Level	Verification Techniques
$CL1$	**Practical Testing:** field tests, software testing, etc.
$CL2$	**Simulation-Based Testing:** a combination of simulation and testing.
$CL3$	**Formal Verification:** encompassing both model-checking and theorem proving approaches.

the **Interface** node was relatively easy. Guided by the FOL specification in §III, the assertions were simple to encode. The challenge lay in transforming the declarative FOL statements into representative processes amenable to model checking.

B. Measuring Confidence in Verification

Since the nodes in our example have been verified using three different verification approaches (testing, theorem proving and model-checking), an obvious question is how confident are we in the verification of the whole system, when parts of it are verified using different approaches. Here, we present a notion of the confidence level of the verification of each node and of the system as a whole.

Our approach is related to, but distinct from, the Safety Integrity Levels (SILs) [41] used in certifying safety-critical systems. We assess the verification approach used, so the evidence that we provide could help in obtaining appropriate SIL certification for a particular system.

We add a measure of the 'confidence' in the verification of these Assume-Guarantee specifications. Basically, a formal proof gives us a high level of confidence, whereas, simple testing methods (especially over unbounded environments) give a lower degree of confidence. There are a number of ways that confidence levels could be partitioned based on the verification techniques used and we initially propose the following three confidence levels in Table I.

These confidence levels could be further decomposed if necessary but we keep them broad for now. In our case study, the **Vision** node contains a black-box implementation that can only be verified using, at most, simulation-based testing and thus its confidence level can be $CL2$. (More likely it will revert to $CL1$, since the environment might be difficult to reconstruct in simulation.) The **Planner**, agent and **Interface** nodes have all been formally verified and are all therefore $CL3$.

Given that these components correspond to a number of varying confidence levels, we also need a way to chain these confidence levels together in order to gauge the confidence level of the entire system as a whole. A simple but cautious approach would be to evaluate the maximum confidence level of the overall system as the minimum confidence level of each of its constituent nodes. Thus:

$$CL(S) = minimum(CL(1), \ldots, CL(n))$$

This is a conservative measure, where the use of formal methods at all seems an unnecessary overhead if not applied to every node in the system. To amplify the effect of formal verification we could calculate the average of the confidence levels for each nodeThus:

$$CL(S) = average(CL(1), \ldots, CL(n))$$

However, this could be misleading if a large number of sensor nodes could only be tested. A more sophisticated approach would encompass a notion of criticality, so that the confidence level of the system reflects the confidence level of its most critical node(s). In this way we could weight the nodes so that the more critical nodes carry a higher weight than the less critical ones

$$CL(S) = CL(n_w)$$

where n_w is the node in the system with the maximum weight, w. If multiple nodes are paired with the highest criticality weighting then we could apply either of the other approaches to calculating confidence to this set of critical nodes.

All of these approaches to chaining confidence levels offer benefits, but are not without their limitations. For example, none of the approaches dependencies between nodes. In order to get a more representative measure of confidence, it is important to consider how tightly or loosely coupled the nodes are. This is of particular relevance in complex systems, where the propagation of data between the nodes contains loops and there may be a many-to-many relationship between nodes. Thus, our future work in this vein seeks to understand how these confidence levels are distributed in a larger, more complex robotic system and how this relates to the certification requirements and risk analysis required by regulatory bodies.

VI. CONCLUSIONS AND FUTURE WORK

Our primary contribution is a framework that uses FOL to specify a robotic system's requirements and uses this as a guide for the specification and verification of the system's components. The key application area for our approach is where the system is composed of modules that benefit from verification using different techniques. We applied our approach to the specification and verification of a robotic system in a search and rescue scenario, where we show how the specifications of each node preserve the high-level FOL specification for each node.

This research presents several avenues for future work. In particular, the application and expansion of these techniques to a larger and more complex system is our current goal. We intend to provide tool support for writing and reasoning about the FOL Assume-Guarantee properties. We also intend to explore the idea of the level of confidence we can have in a system that has been verified using a mixture of methods, as described in §V-B.

Additionally, the inference mechanisms outlined in §III can be expanded in a number of ways, depending on the requirements of the system being verified. In particular, the power of temporal logics allows us to describe much more complex possibilities. For example, we might specify that the output be produced within 8 seconds:

$$\mathcal{A}_C(\bar{i}_C) \;\Rightarrow\; \Diamond^{\leq 8}\mathcal{G}_C(\bar{o}_C)$$

or that the output will be achieved with a certain probability:

$$\mathcal{A}_C(\bar{i}_C) \;\Rightarrow\; P^{0.6}\Diamond\mathcal{G}_C(\bar{o}_C)$$

Such stronger high-level requirements will likely also require using more sophisticated, and complex, verification techniques.

References

[1] H. Hastie, K. Lohan, M. Chantler, D. A. Robb, S. Ramamoorthy, R. Petrick, S. Vijayakumar, and D. Lane, "The ORCA Hub: Explainable offshore robotics through intelligent interfaces," in *Explainable Robotic Systems Workshop*, 2018, article, pp. 1–2.

[2] R. Bogue, "Robots in the nuclear industry: a review of technologies and applications," *Industrial Robot: An International Journal*, vol. 38, no. 2, pp. 113–118, 2011. [Online]. Available: https://doi.org/10.1108/01439911111106327

[3] J. M. Aitken *et al*, "Autonomous Nuclear Waste Management," *IEEE Intelligent Systems*, vol. 33, no. 6, pp. 47–55, 2018.

[4] B. H. Wilcox, "Robotic vehicles for planetary exploration," *Applied Intelligence*, vol. 2, no. 2, pp. 181–193, 1992. [Online]. Available: https://doi.org/10.1007/BF00058762

[5] A. Flores-Abad, O. Ma, K. Pham, and S. Ulrich, "A review of space robotics technologies for on-orbit servicing," *Progress in Aerospace Sciences*, vol. 68, pp. 1–26, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0376042114000347

[6] M. Luckcuck, M. Farrell, L. Dennis, C. Dixon, and M. Fisher, "Formal Specification and Verification of Autonomous Robotic Systems: A Survey," Jun 2018, http://arxiv.org/abs/1807.00048. [Online]. Available: https://arxiv.org/abs/1807.00048http://arxiv.org/abs/1807.00048

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

[8] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: CoqArt: the calculus of inductive constructions*. Springer, 2013.

[9] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Comms. of the ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[10] C. B. Jones, "Tentative Steps Toward a Development Method for Interfering Programs," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 4, pp. 596–619, 1983.

[11] C. A. R. Hoare, "An axiomatic basis for computer programming," *Comms. of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[12] M. Farrell, M. Luckcuck, and M. Fisher, "Robotics and Integrated Formal Methods: Necessity meets Opportunity," in *Integrated Formal Methods*. Springer, 2018, pp. 161–171. [Online]. Available: http://arxiv.org/abs/1805.11996

[13] 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*, vol. 17, no. 2, pp. 365–393, 2018. [Online]. Available: https://doi.org/10.1007/s10270-017-0619-4

[14] L. A. Dennis, "Reconfigurable autonomy: Architecture and configuration language," University of Liverpool, Tech. Rep. ULCS-18-002, 2018.

[15] M. Broy and K. Stølen, *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer Science & Business Media, 2012.

[16] V. Aravantinos, S. Voss, S. Teufl, F. Hölzl, and B. Schätz, "Autofocus 3: Tooling concepts for seamless, model-based development of embedded systems." in *ACES-MB&WUCOR@ MoDELS*, 2015, pp. 19–26.

[17] C. Luckeneder and H. Kaindl, "Systematic top-down design of cyber-physical models with integrated validation and formal verification," in *International Conference on Software Engineering*, 2018, pp. 274–275.

[18] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger, "Specification patterns for robotic missions," *arXiv preprint arXiv:1901.02077*, 2019.

[19] K. A. Elkader, O. Grumberg, C. S. Păsăreanu, and S. Shoham, "Automated circular assume-guarantee reasoning," in *International Symposium on Formal Methods*. Springer, 2015, pp. 23–39.

[20] S. Graf, R. Passerone, and S. Quinton, "Contract-based reasoning for component systems with rich interactions," in *Embedded Systems Development*. Springer, 2014, pp. 139–154.

[21] I. Ruchkin, J. Sunshine, G. Iraci, B. Schmerl, and D. Garlan, "Ipl: An integration property language for multi-model cyber-physical systems," in *International Symposium on Formal Methods*. Springer, 2018, pp. 165–184.

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

[23] A. Pnueli, "The Temporal Logic of Programs," in *18th Symposium on the Foundations of Computer Science*. IEEE, 1977, pp. 46–57.

[24] C. Morgan, K. Robinson, and P. Gardiner, *On the Refinement Calculus*. Springer, 1988.

[25] R. R. Murphy, "Trial by fire [rescue robots]," *IEEE Robotics Automation Magazine*, vol. 11, no. 3, pp. 50–61, Sept 2004.

[26] R. R. Murphy, "Emergency informatics: Using computing to improve disaster management," *Computer*, vol. 49, no. 5, pp. 19–27, May 2016.

[27] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause *et al.*, "Using formal specifications to support testing," *ACM Computing Surveys (CSUR)*, vol. 41, no. 2, p. 9, 2009.

[28] J.-R. Abrial, *Modeling in Event-B*. Cambridge University Press, 2010. [Online]. Available: http://ebooks.cambridge.org/ref/id/CBO9781139195881

[29] J.-R. Abrial and S. Hallerstede, "Refinement, decomposition, and instantiation of discrete models: Application to Event-B," *Fundamenta Informaticae*, vol. 77, no. 1-2, pp. 1–28, 2007.

[30] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in Event-B," *International Journal on Software Tools for Technology Transfer*, vol. 12, no. 6, pp. 447–466, 2010.

[31] L. A. Dennis and B. Farwer, "Gwendolen: A BDI Language for Verifiable Agents," in *Workshop on Logic and the Simulation of Interaction and Reasoning*. AISB, 2008, pp. 16–23.

[32] L. A. Dennis, M. Fisher, M. P. Webster, and R. H. Bordini, "Model checking agent programming languages," *Automated Software Engineering*, vol. 19, no. 1, pp. 5–63, 2012. [Online]. Available: https://www.aaai.org/ocs/index.php/WS/AAAIW15/paper/viewFile/10119/10131

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

[34] W. Visser, K. Havelund, G. Brat, S.-J. Park, and F. Lerda, "Model Checking Programs," *Automated Software Engineering*, vol. 10, no. 2, pp. 3–11, 2002. [Online]. Available: http://link.springer.com/10.1023/A:1022920129859https://doi.org/10.1023/A:1022920129859

[35] C. A. R. Hoare, "Communicating sequential processes," *Comms. of the ACM*, vol. 21, no. 8, pp. 666–677, 1978. [Online]. Available: http://portal.acm.org/citation.cfm?doid=359576.359585

[36] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. Roscoe, "FDR3 – A Modern Model Checker for CSP," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 187–201. [Online]. Available: http://www.cs.ox.ac.uk/projects/fdr/manual/

[37] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming Artificial Intelligence and Reasoning*, ser. LNCS, vol. 6355, 2010, pp. 348–370.

[38] X. Huang, Q. Chen, J. Meng, and K. Su, "Reconfigurability in reactive multiagent systems," in *Proceedings of the 25th International Joint Conference on Artificial Intelligence*. New York, USA: AAAI Press, 2016, pp. 315–321.

[39] R. C. Cardoso, L. A. Dennis, and M. Fisher, "Plan library reconfigurability in bdi agents," in *Proc. of the 7th International Workshop on Engineering Multi-Agent Systems (EMAS)*, 2019.

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

[41] TC65/SC65A, "IEC 61508 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems," 2010. [Online]. Available: http://www.iec.ch/functionalsafety/standards/page2.htm