Heterogeneous Verification of an Autonomous Curiosity Rover*

Rafael C. Cardoso(✉), Marie Farrell, Matt Luckcuck, Angelo Ferrando, and Michael Fisher

Department of Computer Science, University of Liverpool, Liverpool L69 3BX, UK
{rafael.cardoso,marie.farrell,m.luckcuck,
angelo.ferrando,mfisher}@liverpool.ac.uk

Abstract. The Curiosity rover is one of the most complex systems successfully deployed in a planetary exploration mission to date. It was sent by NASA to explore the surface of Mars and to identify potential signs of life. Even though it has limited autonomy on-board, most of its decisions are made by the ground control team. This hinders the speed at which the Curiosity reacts to its environment, due to the communication delays between Earth and Mars. Depending on the orbital position of both planets, it can take 4–24 minutes for a message to be transmitted between Earth and Mars. If the Curiosity were controlled autonomously, it would be able to perform its activities much faster and more flexibly. However, one of the major barriers to increased use of autonomy in such scenarios is the lack of assurances that the autonomous behaviour will work as expected. In this paper, we use a Robot Operating System (ROS) model of the Curiosity that is simulated in Gazebo and add an autonomous agent that is responsible for high-level decision-making. Then, we use a mixture of formal and non-formal techniques to verify the distinct system components (ROS nodes). This use of heterogeneous verification techniques is essential to provide guarantees about the nodes at different abstraction levels, and allows us to bring together relevant verification evidence to provide overall assurance.

1 Introduction

We present a case study with a simulation of the Curiosity rover undertaking an exploration mission. Crucially, we have equipped the rover with decision-making capabilities so that it does not rely on human teleoperation. As a result of the added autonomous behaviour, it is important to provide safety assurances about critical components in the system. Usually, components in such systems are modular and each individual component often requires a different verification technique(s) [13,9,6]. We have applied distinct verification techniques to various critical components and at different abstraction levels to ensure the correctness

* Work supported by UK Research and Innovation, and EPSRC Hubs for "Robotics and AI in Hazardous Environments": EP/R026092 (FAIR-SPACE) and EP/R026084 (RAIN).

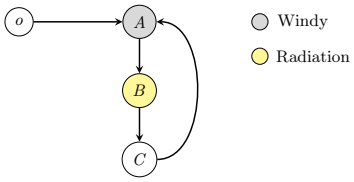

Fig. 1: The Curiosity begins at the origin, o, and then visits the waypoints A, B and C in whichever order is safe. We indicate waypoints with high levels of wind (grey) and radiation (yellow).

of the overall system. All of the artefacts (source code, videos, etc.) discussed in this paper are available in our online repository[1].

2 Mission Description, Simulation and Autonomy

Mission Description: We simulate an inspection mission, where the Curiosity patrols a topological map of the surface of Mars. We assume that the map is known prior to this mission, and in this paper we only consider a small subset of the map (i.e. the agent has map coordinates for each waypoint in the map). Specifically, we consider four different waypoints (o, A, B, and C) that are spread across the Martian terrain. Low-level movement is achieved through a dead reckoning or feedback control.

We begin with the deployment of the Curiosity and a startup period where it initialises all three of its control modules (wheels, arms, and mast). After the agent receives confirmation that the modules are ready, it autonomously controls the Curiosity to move between the waypoints in the following order: $(o \to A \to B \to C \to A \to \ldots)$, as shown in Fig. 1. This is the ideal scenario, however, if one of the waypoints is experiencing high levels of radiation then the rover should skip it until the radiation has reduced to a safe level. For data collection, the mast and arm should be open, however, it is unsafe to do so in windy conditions. We do not model battery power. Instead, we assume that the rover has sufficient battery power to traverse the waypoints and operate the equipment.

Simulation: We obtained a Robot Operating System (ROS) [10] version of the Curiosity from a ROS teaching website[2] which uses official data and 3D models of the Curiosity and Martian terrain which have been made public by NASA. This ROS simulation runs in Gazebo[3], a 3D simulator. Most of the Curiosity's effectors are included in the simulation. It has the complete chassis of the rover with all six wheels and the suspension system, a retractable arm with four joints, and a retractable mast with two joints and a camera (Mastcam) on top. Some of the sensors are missing, e.g. MAHLI (Mars Hand Lens Imager), as these would require simulated sensor data.

[1] `https://github.com/autonomy-and-verification-uol/curiosity-NFM2020`
[2] `https://bitbucket.org/theconstructcore/curiosity_mars_rover/src/master/`
[3] `http://gazebosim.org/`

In the original configuration, the standard control method of the Curiosity was implemented using ROS services and it was controlled via teleoperation. ROS services are defined as a pair of request and reply messages that are provided by ROS nodes. In our simulation, we re-implemented the control method through action libraries, which follow a client-server model that is similar to ROS services. Both can receive a request to perform some task and then generate a reply. The difference in using action libraries is that the client can cancel the action, as well as receive feedback about the task execution. Thus, action libraries are more suited for use with decision-making agents since they allow more fine-grained control.

We developed three action libraries: one each for the wheels, arm, and mast. The wheels client receives high-level action commands to move forward, backward, left, and right; or a waypoint from the topological map (using the move base library for path planning). Based on the command received, the server controls each of the six wheels and publishes speed commands to the appropriate wheels depending on the direction or topological waypoint requested in the action. If a direction command is given, then the server expects three parameters: direction of movement, speed, and distance. After a movement action, the server always calls a stop action that sets the speed of all wheels to zero. The arm and mast action libraries control the joints of their respective effectors so that they can be positioned correctly for use.

Enabling Autonomous Decision-Making: We use the GWENDOLEN [4] agent programming language to implement the high-level control and autonomous decision-making behaviour of the Curiosity. Agent programming languages abstract the environment and other external sources, focusing on high-level autonomous control, resulting in smaller and more modular code than other languages. Due to the agent's reasoning cycle an execution trace can clearly show how the agent came to a decision, thus providing us with explainability. Using GWENDOLEN allows us to verify properties of the agent's reasoning, allowing the safeguard of critical behaviours.

GWENDOLEN agents follow the Belief-Desire-Intention (BDI) model [11]. Beliefs, desires, and intentions represent respectively the information, motivational, and deliberative states of the agent. We developed a GWENDOLEN environment that communicates with ROS through the *rosbridge* library. When the agent executes an action in the environment, the action is processed and published to the action's associated ROS topic. The environment creates subscribers that listen to specific topics so that necessary perceptions are created and sent to the agent.

In the Curiosity simulation, the GWENDOLEN agent has four high-level actions. The action *control_wheels* has three parameters: direction of movement (forward, backward, left, or right), speed (an integer with sign to indicate direction), and distance (in seconds). The *move_to_waypoint* action contains one parameter with a waypoint from the topological map. The actions, *control_arm* and *control_mast*, both have one parameter whose possible values are either *open* or *close*.

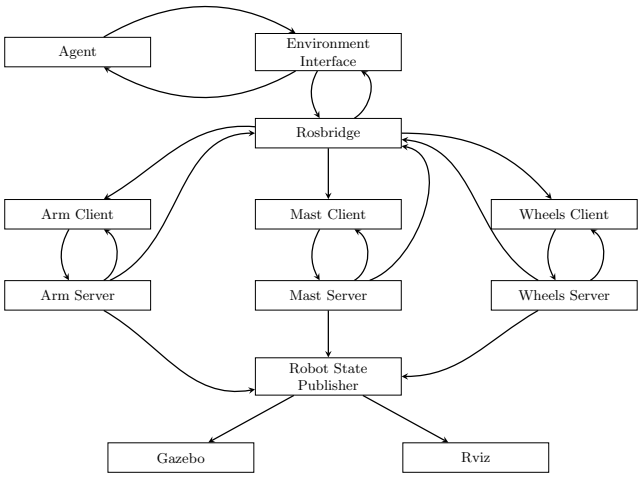

Fig. 2: Overview of the system. Arrows indicate data flow between the nodes.

Fig. 2 illustrates a high-level system diagram with the communication paths between the nodes in the simulation. We have verified distinct components of this simulation using different methods. Specifically, we verify the autonomous agent using the AJPF program model checker; the interface that this agent has with the environment using Dafny; a CSP specification of the action library nodes using FDR4; and we use each of these formal models to guide the generation of runtime monitors. This combination of simulation-based testing, and the use of multiple formal methods at different levels of abstraction, gives us a basis for providing assurances about the use of autonomous decision-making in this extreme environment mission scenario, and could be transferred and applied to other similar case studies as shown in [13,6,9].

3 Verification

This section describes our verification of four critical areas of our simulation of an autonomous Curiosity rover. We verify properties of this system at different levels of abstraction. We begin by describing how we verify that the agent, which is fundamentally controlling the system, makes the correct decisions about which waypoint to visit next.

Next, we discuss our use of an automated theorem prover to verify that the information that the agent receives from the environment sensors is interpreted and acted upon correctly. Then, we outline how we verified that the communication between the *client* and *server* action library nodes (as shown in Fig. 2) functions correctly. Finally, we outline our Runtime Verification (RV) of design-time assumptions about the environment. Interestingly, we used the preceding formal models as a way to focus these runtime checks on appropriate properties.

Verifying the Agent using AJPF: Model-checking [2] exhaustively examines the state space to check if some desired property holds. This can be applied to either a formal model of the system, encoded in some specification language, or directly to the implementation. The property to be verified is usually specified in a logic-based language. For example, we may want to verify that the Curiosity will not move its arm while collecting soil and rock data, in order to protect the sample.

Agent Java PathFinder (AJPF) [5], an extension of Java PathFinder (JPF) [12], is a model-checker that works directly on Java program code. This extension facilitates formal verification of BDI-based agent programs by providing a property specification language based on Linear-time Temporal Logic (LTL) that supports the description of terms usually found in BDI agents.

For example, some of the properties that we verified of the implementation of our agent were as follows:

$$\Box(A_{\texttt{rover}}\mathit{move\_to\_waypoint}(A) \rightarrow \Diamond\mathcal{B}_{\texttt{rover}}\,(\mathit{at}(A)))$$
$$\Box(A_{\texttt{rover}}\mathit{move\_to\_waypoint}(B) \rightarrow \Diamond\mathcal{B}_{\texttt{rover}}\,(\mathit{at}(B)))$$
$$\Box(A_{\texttt{rover}}\mathit{move\_to\_waypoint}(C) \rightarrow \Diamond\mathcal{B}_{\texttt{rover}}\,(\mathit{at}(C)))$$

These properties state that it is always the case ($\Box$) that if the *rover* agent executes the action *move_to_waypoint* (to either A, B, or C), then eventually ($\Diamond$) the *rover* agent will believe that it is currently located in that waypoint.

The syntax of the AJPF specification language is limited to expressing agent related properties, such as beliefs, goals, actions, and intentions of a specific agent that was written in GWENDOLEN. Moreover, properties specified in AJPF must be ground (i.e. cannot be parameterised). For verifying the interface between the agent and the environment, we employ the Dafny program verifier.

Verifying the Agent-Environment Interface: Dafny facilitates the use of specification constructs e.g. pre-/post-conditions, loop invariants and variants [8]. Dafny is used in the static verification of functional program correctness. Programs are translated into the Boogie intermediate verification language [1] and then the Z3 automated theorem prover discharges the associated proof obligations [3].

Our Dafny model centres on the decisions made by the agent in response to the input that it receives from the environment. In this simple model, we verify an important safety property that the rover will not select any actions if the arm, mast or wheels have not been initialised yet. This is specified as follows:

```
ensures (wheelsready && armready && mastready) == false ==> actions ==[];
```

Here, `wheelsready`, `armready` and `mastready` are boolean flags that are toggled by the associated modules, and `actions` is the sequence of returned actions.

Our Dafny model has functions for accessing the environmental conditions at a given waypoint e.g. `getEnvironment()` and `getWind()`. This allows us to verify properties about the how the environmental conditions affect where the rover goes. The `getEnvironment()` method then checks the wind and radiation at a particular waypoint and we verify that the following condition is met where `e` is a variable that represents the current status of the environment:

```
ensures windspeed < 5 && radiation < 5 ==> e == Fine;
```

In this way, our Dafny model allows us to verify conditions about the safety of the agent and also that the information coming from the environment is interpreted correctly by the agent. We provide other verified methods including `getRad()` which is a high-level implementation of how the radiation at waypoint B decays over time. Our loop invariant in the `CuriosityAgent()` also ensures that the rover can't be at waypoint B when the radiation is too high:

```
invariant !(current == B && env == Radiation);
```

We included radiation at B in the Dafny implementation to examine how the rover reacts to radiation at a particular waypoint, as per the mission description (Sect. 2). Next, we verify the action library client and server nodes using CSP.

Verifying Action Library Communication: We verify the communication between the pairs of action library client–server nodes that interface between the software and hardware (arm, mast, and wheels). Each client accepts instructions from the agent (via *rosbridge*) which it then sends to the relevant server node as a goal (task to complete). Since the AJPF model checker can only check agent-programs, we decided to use Communicating Sequential Processes (CSP) to verify this critical link. CSP processes describe sequences of events; $a \rightarrow b \rightarrow \mathit{Skip}$ is the process where events a and b occur sequentially, then terminates (*Skip*).

The CSP model is constructed from the Curiosity ROS code, capturing both the program-specific and the generic action library behaviour. Each of the client–server pairs is modelled by one CSP file, with one further file modelling the generic behaviour of an action library server. We use the FDR4 model-checker [7] to check three properties: (1) when a client sends a goal, it will begin execution on the correct server, (2) when a client sends a goal, eventually it receives a result from the server, and (3) when the agent instructs a client node to perform an action, the server informs the agent that it is ready and then eventually the agent receives a result. Here we give an example of (2), where we check that if the arm client sends a goal then eventually it will receive a result:

$$\mathit{send\_goal\_arm}?\_ \rightarrow \mathit{executeGoal.arm} \rightarrow \mathit{SKIP}$$

Runtime Verification: It is achieved by examining the current execution of the system at runtime against a formal specification. Since runtime monitors only observe the current system execution, the resulting approach is not exhaustive in the sense that model-checking is (which examines the entire state space). However, monitor implementations are usually extremely efficient since they do not consider all possible system executions and they can remain as safeguards after deployment. In this way, a monitor helps to ensure correct system behaviour.

ROSMonitoring[4] (ROSMon) is a flexible and formalism-agnostic RV framework for ROS. ROSMon creates gaps in the communication between nodes in the system. These gaps are then filled by monitors which are automatically synthesised by ROSMon. In this way, the messages of interest are forced to pass through the monitors and are checked against a corresponding formal specification. We applied ROSMon to our simulation to check properties at runtime.

[4] `https://github.com/autonomy-and-verification-uol/ROSMonitoring`

For example, using Dafny, we verify the agent-environment interface; ROSMon bridges the gap between the Dafny model and the real environment by checking at runtime if the assumptions used in the Dafny model are satisfied by the real system.

We used a property, written in Runtime Monitoring Language (RML), to synthesise a monitor to check the constraint used in the Dafny `getEnvironment` method. Here, we check that the wind speed and radiation are always positive, and if the wind speed and radiation are less than 5 each, then the environment is "Fine". This is (partially) written as follows:

```
Main = (GetEnvironmentConstraints /\ (wind_speed(_) >> wind_speed_at_least(0)*) /\
        (radiation_units(_) >> radiation_units_at_least(0)*));
GetEnvironmentConstraints =
  wind_speed_up_to(4) GetEnvironmentConstraints1
  \/ radiation_units_up_to(4) GetEnvironmentConstraints2
  \/ any GetEnvironmentConstraints;
...
```

In this way, we used abstract formal system models to guide the development of corresponding runtime monitors to examine these properties at runtime.

4 Discussion

This paper has reported on our case study of using multiple verification techniques to provide assurance for an autonomous Curiosity rover undertaking an exploration/sampling mission. We used the GWENDOLEN agent programming language to implement an autonomous agent in a ROS-based simulation of the Curiosity. We verified this agent using AJPF, how it responds to discrete input from its environment using Dafny, the message passing between the action library nodes using CSP, and we synthesised runtime monitors using ROSMon.

We employed a myriad of verification techniques to verify the behaviour of distinct aspect(s) of the system. Our aim was to streamline the process of verifying the system by verifying each system component using a suitable technique, rather than attempting to verify everything using only one technique. For example, we use an agent programming language for the agent and CSP for message passing. The tool used to verify the agent program is not appropriate (and would generally not work) to verify message passing.

Our use of RV is of particular interest here since the system is implemented in C++ or Python for which formal verification at code level is not currently feasible/possible. However, the tools and techniques that were chosen are not necessarily the only ones that were suitable for any specific component and certainly other choices could have been made. Future work includes investigating these alternatives.

Our use of heterogeneous techniques for various critical components of the system was motivated by the work done in [13,6,9]. Our case study exhibits how heterogeneous verification techniques can be applied to various components of an autonomous robotic system at different levels of abstraction. Future work seeks to link the results of these heterogeneous techniques in a holistic framework so that they might inform one another.

References

1. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Formal Methods for Components and Objects. LNCS, vol. 4111, pp. 364–387. Springer (2005)
2. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
3. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 4963, pp. 337–340. Springer (2008)
4. Dennis, L.A., Farwer, B.: Gwendolen: A BDI language for verifiable agents. In: Logic and the Simulation of Interaction and Reasoning. AISB, Aberdeen (2008)
5. Dennis, L.A., Fisher, M., Webster, M.P., Bordini, R.H.: Model checking agent programming languages. Automated Software Engineering **19**(1), 5–63 (2012)
6. Farrell, M., Luckcuck, M., Fisher, M.: Robotics and integrated formal methods: Necessity meets opportunity. In: Furia, C., Winter, K. (eds.) Integr. Form. Methods. LNCS, vol. 11023, pp. 161–171. Springer (2018)
7. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: FDR3 – A Modern Model Checker for CSP. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 8413, pp. 187–201. Springer (2014)
8. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Logic for Programming Artificial Intelligence and Reasoning. LNCS, vol. 6355, pp. 348–370. Springer (2010)
9. Luckcuck, M., Farrell, M., Dennis, L.A., Dixon, C., Fisher, M.: Formal specification and verification of autonomous robotic systems: A survey. ACM Computing Surveys (CSUR) **52**(5), 100 (2019)
10. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.: ROS: an open-source robot operating system. In: Workshop on Open Source Software at the International Conference on Robotics and Automation. IEEE, Japan (2009)
11. Rao, A.S., Georgeff, M.: BDI agents: From theory to practice. In: International Conference on Multi-Agent Systems. pp. 312–319. AAAI (1995)
12. Visser, W., Havelund, K., Brat, G., Park, S.J., Lerda, F.: Model checking programs. Automated Software Engineering **10**(2), 3–11 (2002)
13. Webster, M., Western, D., Araiza-Illan, D., Dixon, C., Eder, K., Fisher, M., Pipe, A.G.: A corroborative approach to verification and validation of human–robot teams. The International Journal of Robotics Research **39**(1), 73–99 (2020)