Chapter 11

Verification for space robotics

Rafael C. Cardoso[1], Marie Farrell[2], Georgios Kourtis[1], Matt Webster[3], Louise A. Dennis[1], Clare Dixon[1], Michael Fisher[1], and Alexei Lisitsa[4]

Verification and validation are required to help assure the safety, reliability, functional correctness and trustworthiness of systems. Verification demonstrates that the system conforms to its requirements and validation that it meets the needs of the stakeholders. Formal verification involves a mathematical analysis of all behaviours of a system often using logics and tools such as theorem provers or model checkers. Model checkers [1–3] are based on an automated, algorithmic method to show whether a property holds on all runs of the system. As input, model checkers require a model of the system and a property relating to the requirements in some (temporal) logical language. Commonly used logics to express change over time are propositional linear time temporal logic (LTL) [4, 5] and computational tree logic (CTL) [6]. Theorem proving, see, e.g., [7–10] for calculi and tools for temporal resolution, involves specifying both the system and the property in some logical language and using mathematical proof to show that the property is a logical conclusion from the formulae specifying the system. A number of tools and techniques have been developed to carry out theorem proving and model checking. While formal verification has the advantages of being exhaustive (considering all states in the system), precise details of the system often have to be represented in an abstract form to minimise the amount of time and memory required during the formal verification process.

Non-formal techniques can also be used for verification, e.g., simulation-based testing [11], or end-user experiments [12]. While not exhaustive, i.e., not every path through the system will be tested and not every scenario can be examined, the system being tested is more realistic than the abstracted model above.

[1]Department of Computer Science, University of Manchester, Manchester, United Kingdom
[2]Department of Computer Science, Maynooth University, Maynooth, Ireland
[3]School of Computer Science and Mathematics, Liverpool John Moores University, Liverpool, United Kingdom
[4]Department of Computer Science, University of Liverpool, Liverpool, United Kingdom

Different verification methods might be used depending on the system under consideration. A combination of different types of verification using both formal and non-formal techniques can help to improve the confidence in the system [13].

In this chapter, we discuss a range of tools and techniques applicable to the verification and validation of autonomous space systems. In particular we describe:

- verification techniques for robotics and autonomous systems;
- theorem proving for space robotics using modal and temporal logics;
- verifiable space robot architectures;
- simulation and verification of the Mars Curiosity rover;
- verification of astronaut–rover teamwork as modelled by the Brahms agent modelling system originally developed at NASA;
- modelling and verification of multi-objects systems (such as swarms of satellites or sensor network protocols).

11.1 Formal specification and verification techniques

In this section, we provide an overview of formal specification and verification techniques that have been applied to robotics and autonomous systems that can be found in the literature. Additionally, we discuss some particular formal verification techniques and how they might be applied to autonomous space systems. We believe that many of the issues for verifying systems for space are similar to those for robotics and autonomous systems in other extreme and dangerous environments.

11.1.1 Formal specification and verification for autonomous robotic systems

The work summarised in this section was originally published in [14, 15]. In particular, [14] contains a comprehensive survey of the literature in relation to formal specification and verification of autonomous robotic systems. This work identified a number of distinct challenges for formal specification and verification of robotic systems and some of these are described in the position paper which advocates the use of integrated formal methods in this domain [15].

11.1.1.1 Methodology

In this work, we began by identifying the following three research questions:

RQ1: What are the challenges when formally specifying and verifying the behaviour of (autonomous) robotic systems?

RQ2: What are the current formalisms, tools and approaches used when addressing the answer to **RQ1**?

RQ3: What are the current limitations of the answers to **RQ2** and are there developing solutions aiming to address them?

We investigated these questions by carrying out a systematic literature survey on *formal modelling of (autonomous) robotic systems, formal specification of (autonomous) robotic systems* and *formal verification of (autonomous) robotic systems*. This search applied to papers published from 2007 to 2018, inclusive. A summary of the results can be found in [16].

11.1.1.2 Answering RQ1: challenges

By analysing the literature, we were able to identify a number of challenges for formal specification and verification. We partitioned the challenges into those that are *external* and those that are *internal* to the robotic system. The challenges that were deemed as *external* to the robotic system were modelling the physical environment and providing evidence for certification and trust. The challenges that were deemed to be *internal* to the robotic system were agent-based systems, multi-robot systems and self-adaptive and reconfigurable systems. These challenges and current efforts for solving them are summarised in detail in [14, §3–4].

Interestingly, tackling the internal challenges may help to minimise the effects of the external challenges. For example, reconfigurability can help an autonomous robotic system to handle an uncertain and dynamic environment. Similarly, *rational* agents, which can provide reasons for their choices, can help with evidence for certification and public trust.

11.1.1.3 Answering RQ2: formalisms, tools and approaches

To answer RQ2, we quantified and described the formalisms, tools and approaches used in the literature [14, §5–6]. We have summarised these findings briefly in Tables 11.1 and 11.2. In particular, Table 11.1 reveals that most used a state-transition formalism to specify or build a model of their system. Whereas, most used a logic, normally temporal logic, to specify the properties of the system to be verified.

This could be a result of model checkers being the most favourable approach as indicated by Table 11.2, which generally take a state-transition system as input to verify against logical properties, often temporal logic. This choice of model checking may be due to the fact that it is generally easy to explain to stakeholders as 'exhaustive testing', a concept that most are familiar with. The lack of popularity of

Table 11.1 Summary of the types of formalisms for specifying the system and the properties to be checked [14, table 2]

Formalism	System	Property
Set-based	5	0
State-transition	33	0
Logic	6	32
Process algebra	3	1
Ontology	4	0
Other	5	8

Table 11.2 Summary of the verification approaches used throughout the literature [14, table 4]

Approach	Total
Model checking	32
Theorem proving	3
Runtime monitoring	3
Integrated formal methods	8
Formal software frameworks/architectures	10

theorem provers is likely due to their usability issues since they are generally difficult to operate for non-expert users.

11.1.1.4 Answering RQ3: limitations

We address this question in detail in [14, §7]. The most obvious limitation is the lack of adoption of formal methods by robotic software developers. It is often the case that these developers view formal methods as difficult to use and as a complicated additional step in the development process. Further, a lack of appropriate tools often impedes the application of formal methods [16].

As indicated by [14, Table 3], there is a huge variety of tools that have been developed for the same formalism. This indicates a lack of interoperability between different formalisms and tools. The development of a common framework for translating between, relating or integrating different formalisms would be useful in this domain and is an open problem in this domain [15].

Furthermore, formalising the *last link* between specification and implementable code is another limitation not only in this area but also in software engineering, in general. In particular, ensuring that the software implementation matches the associated formal specification requires a formalised translation.

11.1.1.5 Application to space robotics

Of course, this survey [14], the associated position paper [15] and summary [16] are quite broad and do not specifically target space systems, although some of the surveyed materials do. Since space is an appropriate domain where reliable autonomous robotic systems are required, it is important to understand the current state-of-the-art approaches to formal specification and verification of autonomous robotic systems in general and this survey provides the relevant details [14–16].

11.2 Theorem proving for space robotics using modal and temporal logics

We are developing tools and techniques for verification that can be applied to robotics and autonomous systems. In particular, we are interested in the development of

tools and techniques for different dimensions such as temporal logics that consider how systems change over time [3], modal logics [17] that consider possible worlds where the relationships between worlds may represent necessity/possibility, belief or knowledge [18], temporal logics that incorporate probabilities such as Probabilistic Computation Tree Logic (PCTL) [19] or specific time bounds like Metric Temporal Logics (MTLs) [20].

We discuss some deductive methods that have been developed, including calculi and associated theorem-proving tools for different temporal and modal logics. These are resolution-based methods that follow the overall approach developed for LTL in [7] and extended to the branching-time temporal logic CTL in [9]. Using this approach, we can encode the system as a logical formula (S) and a property we want to prove of this system (P) in some logic. If we want to show $S \rightarrow P$ is valid, we negate $S \rightarrow P$ obtaining $S \wedge \neg P$ and show that the negated formula is unsatisfiable. The general approach is to translate the original formula φ into another equi-satisfiable formula φ of a particular form (termed a normal form). Following this, a number of proof rules are applied that generate new normal form formulae (often called clauses). The process stops when no new clauses can be derived or a contradiction can be obtained. With the temporal and modal logics we discuss here, the key thing is to make sure that formulae relate to the same world so that the proof rules can be applied.

11.2.1 The multi-modal logic K

There has been interest in modal logics (see, e.g., [21]) in relation to theoretical results, the development of practical tools such as theorem provers and their application to systems. A calculus [22] and theorem prover [23] have been developed for the multi-modal logic K. This modal logic has two modal operators: $\Box$ denoted *necessity* and $\Diamond$*possibility*. These operators can be indexed for different agents, e.g., below $\Box_a$ ($\Diamond_a$) denotes that it is necessary (possible) for the astronaut, whereas $\Box_r$ ($\Diamond_r$) denotes that it is necessary (possible) for the rover. The example below considers two agents, an astronaut and a rover, where the astronaut can go outside a lunar habitat to survey the moon surface and the rover must accompany the astronaut during dangerous situations.

$\Box_a(out \rightarrow danger)$	For the astronaut it is necessarily the case that if they are out of the habitat then they are in danger
$\Box_a(survey \rightarrow out)$	For the astronaut it is necessarily the case that if they are surveying then they are out of the habitat
$\Box_a(danger \rightarrow \Box_r accompany)$	For the astronaut it is necessarily the case that if they are in danger then the rover necessarily accompanies them
$\Diamond_a survey$	It is possible that the astronaut does a survey

From this, we can prove that it is possible that the astronaut is necessarily accompanied by the rover ($\Diamond_a \Box_r accompany$).

A resolution calculus has been developed for this logic that only allows the deduction rules to be applied to the same modal depth of formula [22]. This makes the prover for this calculus [23] perform well on formulae with a high level of nesting of modal formulae compared to other modal theorem provers for this logic.

11.2.2 Metric temporal logic

MTLs have models that are timed sequences of states. In MTL, the temporal operators such as $\Box$ ('now and at every future moment'), $\Diamond$ ('at sometime in the future') and $\bigcirc$ ('in the next moment in time') include an interval that provides temporal constraints about when formulae should hold. For example, $\Box_{[0,4]}\varphi$ denotes that φ holds at all states that occur between zero and four time points from now, and $\Box_{[3,\infty]}\varphi$ denotes that φ holds at all states that occur at least three time points onwards. In the statements below, we provide some MTL formulae describing some formulae for the astronaut–rover scenario.

$\Box_{[0,\infty]}(start\_survey \rightarrow \Diamond_{[0,6]} \neg out)$	It is always the case that once the astronaut starts surveying they must return to the habitat within six time units
$\Box_{[0,\infty]}(end\_survey \rightarrow \Box_{[0,3]} rest)$	It is always the case that once the astronaut finishes the period of surveying then they must rest for three time units

If we consider a natural numbers model where states are mapped to the natural numbers, we can translate such formulae into formulae of LTL. Then we can apply provers that have been developed for LTL to obtain a route to theorem proving for MTL formulae. Two different translations have been developed and applied to two different versions of the semantics. An experimental analysis has been applied translating these alternatives to input to a range of LTL provers to investigate their behaviour [24, 25]. This approach is useful as it allows the re-use of a range of provers for temporal logics that can be applied to problems for MTL over the natural numbers. A related approach is taken in [26], where translations from a similar logic (Mission-Time LTL) into model checkers for LTL are provided.

11.3 Verifiable space robot architectures

Robotic systems combine many hardware and software components, usually represented as node-based architectures. Each node in a robotic system may require different verification techniques, ranging from software testing to formal methods. In fact, *integrating* (formal and non-formal) verification techniques is crucial for the robotics domain [15]. Verification should be carried out using the most suitable

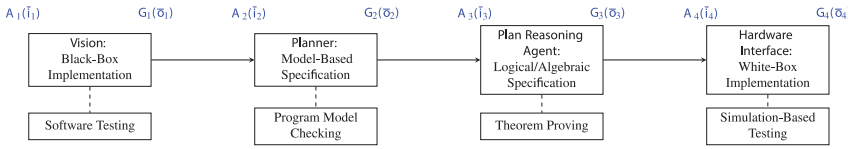


Figure 11.1 *We specify the Assume-Guarantee contracts for each node (denoted by $\mathcal{A}(\bar{i})$ and $\mathcal{G}(\bar{o})$, respectively). These are then used to guide the verification approach applied to each node, denoted by dashed lines, such as software testing for a black-box implementation of the Vision node. The solid arrows represent data flow between nodes and that the assumptions of the next node should follow from the guarantee of the previous node.*

technique or formalism for each node. However, linking heterogeneous verification results of individual nodes is difficult and the current state-of-the-art for robotic software development does not provide an easy way of achieving this. In this section, we summarise ongoing work that was originally presented in [27].

In Figure 11.1, we consider a simple space robotic system: a planetary rover undertaking a remote inspection task. Here, we have nodes representing the Vision system, a Planner that returns a set of potential plans between the current location and the next point to inspect, an autonomous Plan Reasoning Agent that selects a plan and a Hardware Interface that sends commands to the rover's actuators.

As illustrated in Figure 11.1, we could use logical specifications (e.g., temporal logic), model-based specifications (e.g., Event-B [28] or Z [29]) or algebraic specifications (e.g., Communicating Sequential Processes (CSP) [30] or Common Algebraic Specification Language (CASL) [31]) among others to specify the nodes in a robotic system. Each of these formalisms offers its own range of benefits, and each tends to suit the verification of particular types of behaviour. However, in some cases we may only have access to the black-box or white-box implementation of a node and, so, we must use (simulation-based) testing techniques for verification.

Our approach facilitates the use of heterogeneous verification techniques for the nodes in a robotic system. We achieve this by specifying contracts as properties in First-Order Logic (FOL), as high-level node specifications, and we employ temporal logic for reasoning about the combination of these FOL specifications. Thus, we attach the assumptions ($\mathcal{A}(\bar{i})$) and guarantees ($\mathcal{G}(\bar{o})$) to individual nodes (shown in Figure 11.1). This abstract specification can be seen as a logical prototype for individual nodes and thus the entire robotic system.

11.3.1 FOL contract specifications

For each node, N, we specify $\mathcal{A}_N(\bar{i}_N)$ and $\mathcal{G}_N(\bar{o}_N)$, where $\bar{i}_N$ is a variable representing the input to the node, $\bar{o}_N$ is a variable representing the output from the node and $\mathcal{A}_N(\bar{i}_N)$ and $\mathcal{G}_N(\bar{o}_N)$ are FOL formulae describing the assumptions and guarantees, respectively, of this node.

Each individual node, N, obeys the following implication

$$\forall \bar{i}_N, \bar{o}_N \cdot \mathcal{A}_N(\bar{i}_N) \Rightarrow \Diamond \mathcal{G}_N(\bar{o}_N)$$

where '$\Diamond$' is LTL's [4] 'eventually' operator. So, this implication means that if the assumptions, $\mathcal{A}_N(\bar{i}_N)$, hold then *eventually* the guarantee, $\mathcal{G}_N(\bar{o}_N)$, will hold. Note that our use of temporal operators here is motivated by the temporal nature of robotic systems and will be of use in later extensions of this work.

Consider the autonomous Plan Reasoning Agent in Figure 11.1; we can specify the following simple assumption $\mathcal{A}_3(\bar{i}_3)$:

$$\mathcal{A}_3(\bar{i}_3) = \forall p \cdot p \in PlanSet \Rightarrow goal \in p$$

which ensures that every plan that is returned by the Planner contains the *goal* location. Then, we might specify the guarantee that the agent chooses the shortest *plan* as follows:

$$\mathcal{G}_3(\bar{o}_3) = (plan \in PlanSet) \wedge (\forall p \cdot p \in PlanSet) \wedge (p \neq plan) \Rightarrow (length(plan) \leq length(p))$$

Once the FOL assumption and guarantee are specified, then we use these high-level specifications as properties to be verified of the individual nodes. For the autonomous Plan Reasoning Agent, we can use a number of techniques for verifying that it meets its associated FOL specification. For example, we can specify the node using the Gwendolen agent programming language and then use the Agent Java PathFinder (AJPF) model checker to verify that it behaves as specified [32].

Nodes in a modular robotic architecture are linked together and transmit data between them so long as their types/requirements match. Similarly, we can compose the contract specifications of individual nodes in a number of ways and we are working towards a calculus of inference rules that capture this behaviour. To this end, we are developing rules for sequentially composing, joining, branching and looping between nodes.

11.3.2 Measuring confidence in verification

A key question is how using these different verification techniques affects our confidence in the verification of the whole system. One might think that a formal proof of correctness corresponds to a higher level of confidence than simple testing methods (especially over unbounded environments). However, formal verification is usually only feasible on an abstraction of the system whereas testing can be carried out on the implemented code. Therefore, it is our view that we achieve higher levels of confidence in verification when multiple verification methods have been employed for each node in the system [13].

We have broadly partitioned current verification techniques into three categories: testing, simulation-based testing and formal methods. We have determined which of these techniques might be employed for each node in our simple example as shown in Table 11.3. We then provide a score for our level of confidence in the verification of the whole system as 9/12, resulting in a confidence measure of 75 percent. Examining how this metric can be calculated for more complex systems with loops is a future direction for this work.

Table 11.3 Verification techniques applied to each node

	Testing	Simulation-based testing	Formal methods
Vision	✓	✗	✗
Planner	✓	✓	✓
Plan reasoning agent	✓	✓	✓
Hardware interface	✓	✓	✗

When verifying complex robotic systems, it is clear that no single verification technique is suitable for every node in the system [15] and so a logical framework that allows us to integrate the results from distinct verification techniques is needed. We have outlined an initial approach to specifying assumptions and guarantees using FOL for individual nodes in robotic systems and we have used a simple, illustrative example of a planetary rover to convey our approach. Once the FOL specifications have been constructed, they are then used to guide the more detailed verification of each node. Furthermore, we introduce the notion of confidence in verification techniques and provide a broad categorisation.

Our current work involves developing a calculus for reasoning about and combining the contract specifications of individual nodes. In the future, we plan to provide tool support for this and to evaluate it using a set of more complex robotic space missions. We also intend to further investigate the suitability of the confidence levels that we have proposed.

11.3.3 Related work

Our approach draws inspiration from Broy's approach to systems engineering [33], which uses 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 – a domain which has not received much attention in this branch of the literature before.

In terms of compositional verification, related work includes CoCoSpec [34], which allows users to specify contracts for reactive systems in terms of assumptions and guarantees. This work is specialised for synchronous communications and thus it differs from the event-based communications that we target here. Further, their contract semantics is more restrictive than ours. It is also not clear how their support for compositional verification can be extended to support heterogeneous components such as those in our example. Other related approaches include OCRA [35] and AGREE [36], although neither explicitly incorporates heterogeneous verification techniques.

11.4 Case study 1: Simulation and verification of the Mars Curiosity rover

Autonomous robots are especially relevant in scenarios with communication bottlenecks. For example, in planetary space exploration it can take a long time for human operators to send commands from Earth to the robot, and then the same amount of time to receive any feedback data from the command that was sent. The Curiosity rover[1] is one of the most complex rovers successfully deployed in a planetary exploration mission to date. It was sent by NASA to explore the surface of Mars. Its main objectives include determining signs of life, characterising climate and geology and preparing for human exploration.

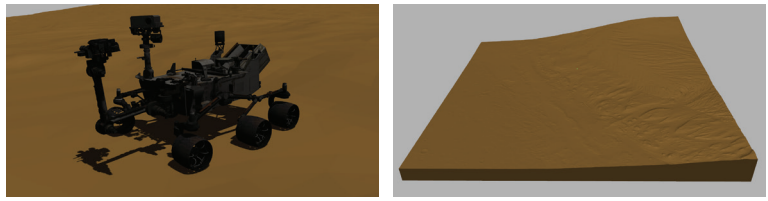
However, one of the biggest challenges faced by the Curiosity is the long communication delay between Earth and Mars. Depending on the orbital position of both planets, it can take anywhere from 4 to 24 minutes for a message to be transmitted between Earth and Mars. Thus, if the Curiosity could be controlled autonomously it would be able to perform its activities much faster. One of the major challenges preventing the use of autonomy in such scenarios is the lack of assurance that the autonomous behaviour will work as expected. To this end, it is important to take a corroborative approach [13] when trying to provide assurances about autonomy.

In [37], system scenario tests are described for the validation of the Mars Curiosity rover surface operations. These tests were performed before the launch, over the period of one year, to test high-priority objectives in typical missions that would take place for the duration of a Martian day. Testing-based approaches are essential for validating a system; however, they are not exhaustive and can often miss edge cases, particularly when testing autonomous systems.

The presence of autonomy (without any input from Earth) in the original mission was restricted to the AEGIS (Autonomous Exploration for Gathering Increased Science) component [38]. This component provides the autonomous targeting of surfaces to be processed by the remote geochemical spectrometer. It has been validated through comprehensive testing both in simulation before the launch and on Mars after the launch, but no concrete formal verification was made public. While we do not have the AEGIS in our (much simpler) simulation, we use a rational agent to perform autonomous operations that would usually be delegated to Earth operators and use formal techniques to verify the behaviour of the agent.

In this section, we present a simulation of the Mars Curiosity rover controlled via an autonomous agent. Then, we discuss the formal verification of this agent through the use of model checking. Finally, we verify it at runtime by deploying runtime monitors. This combination of simulation-based testing, and the use of two

[1]https://mars.nasa.gov/msl/



(a) (b)

Figure 11.2 The Mars Curiosity rover simulation in Gazebo. (a) The Mars Curiosity model in Gazebo. (b) The Mars world in Gazebo.

distinct formal methods, gives us a basis for providing assurances about the use of autonomy in extreme environments that could be transferred and applied to other similar case studies. We refer the reader to [39] for a more in-depth discussion about the use of different verification techniques applied to a similar case study to the one presented in the section.

11.4.1 Simulation

Modular architectures are typically employed to speed up and make the development of robotic systems easier. The Robot Operating System (ROS) [40] is an example of a popular middleware that can be used to develop a modular robotic system. In ROS, nodes are used to effectively capture robotic software in terms of a graph that describes the communication between distinct nodes. Some of the advantages of decoupling the system in this way include more precise failure handling and recovery mechanisms, since failures can be traced to individual nodes and the complexity of the code is reduced when compared to monolithic systems, making it easier to add, replace or remove functionality (i.e., nodes).

Even though the software deployed with the real Curiosity was not ROS-based, a ROS version has been developed by the ROS teaching website The Construct[2] using official data and 3D models of Curiosity and Mars terrain that NASA made public. The simulation uses ROS and runs in Gazebo, a 3D simulator with several high-performance physics engines. The 3D model of the Curiosity running in Gazebo is shown in Figure 11.2a, and the Mars world used in the simulation is shown in Figure 11.2b.

RVIZ is a 3D visualiser tool that displays state information about the virtual model of the robot and live sensor data such as camera feeds, infrared measurements and more. Most of the Curiosity's effectors are included in the simulation, as shown in Figure 11.3. It has all six wheels, along with the suspension system, the complete chassis of the rover, a 7-foot retractable arm with four joints and a retractable mast

[2]https://bitbucket.org/theconstructcore/curiosity_mars_rover/src/master/

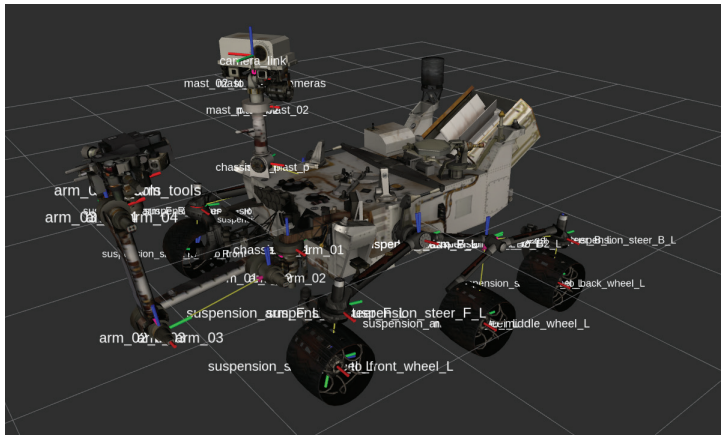


Figure 11.3 RVIZ view with all of the effectors in the simulated Mars Curiosity rover

with two joints and a camera (Mastcam) on top. Some of the sensors are missing (such as chemical and weather sensors), as these would require the sensor data to be simulated in some way.

The standard control of the Curiosity rover in the original simulation was implemented using ROS services and needs to be teleoperated by a human. Services can be provided by ROS nodes and are defined by a pair of request and reply messages. This interaction is similar to a remote procedure call. Action libraries 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 user can cancel the action, as well as receive feedback about the task execution. Thus, action libraries are more suited when autonomy is used, allowing more fine-grained autonomous control of the robot.

We implemented three action libraries: wheels, arm and mast. The client of the wheels can receive high-level action commands to move forward, backward, turn left and turn right, which are then passed over to the server. The server has control over each of the six wheels and publishes speed commands to the appropriate wheels depending on the direction requested in the action. After any movement action, the server always calls a stop action that sets all wheels speed to zero. The arm and mast action libraries are responsible for controlling the joints of the arm and the mast, respectively.

Agent-based control allows a system to dynamically adapt to changes in the environment through the use of modularity, decentralisation, autonomy, scalability and reusability [41]. We use the Gwendolen agent language to program the high-level control and autonomous behaviour of the Mars Curiosity rover. Agent programming languages abstract the environment and other external sources, focusing on programming autonomous control at high-level, resulting in smaller and more modular code than other languages. Furthermore, due to the agent's reasoning cycle

and mental attitudes, an execution trace can clearly show how the agent came to a decision, thus providing us with explainability. Finally, using Gwendolen, properties of the agent's reasoning can be formally verified, allowing us to safeguard critical behaviours.

Gwendolen [42] agents follow the Belief-Desire-Intention (BDI) model [43]. These mental attitudes represent, respectively, the information, motivational and deliberative states of the agent. The *belief revision function* is used to process incoming perceptions from the environment (e.g., obtained through sensors) and triggers the update of the *belief* (what the agent believes to be true about its environment and other agents) base. The *option generation function* uses the belief base and the intention base to generate more options and update the *desire* (the desired states that the agent hopes to achieve) base. The *filter* is responsible for updating the *intention* (a sequence of actions that an agent wants to carry out to achieve a desired state) base, taking into account its previous intentions and the belief and desire bases. Finally, the *action selection function* outputs an action from the intention that was chosen to be executed.

It is not possible to integrate Gwendolen, which is implemented in Java, directly with ROS, which is implemented in Python/C++. There are two possible solutions to this problem: use the *rosjava* library or use the *rosbridge* library. The former is a third-party library that is not available in all recent versions of ROS; since it re-implements parts of ROS in Java it can be an arduous process to update it for newer versions. The latter is a compact library that is less dependent on ROS version, as it does not change anything in the core, but rather uses WebSocket communication to interface ROS with external languages through JavaScript Object Notation (JSON) messages.

We developed a Gwendolen environment[3] that can communicate with ROS through the *rosbridge* library. This environment is not domain-specific, i.e., it could be used in other robots as long as they are running ROS. It allows Gwendolen agents to publish and subscribe to ROS topics. Generally, when the agent executes an action in the environment, the action is processed and published to the appropriate ROS topic associated with that action. The environment can also create subscribers that keep listening to predetermined topics, and then when a message is received it is processed and, if it is the case, perceptions are created and sent to the agent.

The Gwendolen agent has access to three high-level actions. The action *control_wheels* has three parameters: direction of movement (forward, backward, left, or right), speed (an integer, if moving backwards should be negative, otherwise it should be positive) and distance (in seconds). This action is defined in the Gwendolen environment, as shown in Listing 11.1, which basically receives the parameters set by the agent, converts it into a *Move3* message (a message type defined in ROS) and then publishes the message to the appropriate ROS topic. The remaining actions

[3]Source code is available at: https://github.com/autonomy-and-verification-uol/gwendolen-rosbridge

are *control_arm* and *control_mast*, both require one parameter with possible values either *open* or *close*.

```
public void control_wheels(String modereq, float speedreq, int distancereq) {
        Publisher wheels = new Publisher("/gwen_wheels", "curiosity/Move3", bridge);
        wheels_control.publish(new Move3(modereq, speedreq, distancereq));
}
```

Listing 11.1 Environment code for the control wheels action

Our simulation[4] contains an inspection mission, where the Curiosity patrols between four different waypoints (A, B, C and D) that are spread across Mars terrain. The agent has previous knowledge about the terrain, with map coordinates to each of the waypoints.

The simulation starts with the deployment of the Curiosity and a start-up period where it initialises all three control modules (wheels, arms and mast). After the agent receives confirmation that all modules are ready, it autonomously controls the Curiosity to move to the waypoints. Movement through the waypoints is done in order and loops back when arriving at the last one (A → B → C → D → A). When moving to waypoint A and D, the agent ensures that the arm is retracted and that the mast is extended upwards, since in both of these waypoints, the mission of the Curiosity is to take images of the terrain. Otherwise, when moving to waypoint B and C, the arm is fully extended to manipulate soil and rock samples, and the mast is retracted due to extreme weather conditions, to preserve power and the condition of the mast.

The plan of the Gwendolen agent that shows the start of the movement from waypoint A to B is shown in Listing 11.2. The head of the plan (movement_complete) is a trigger event that is activated when the belief with the same name is added to the agent's belief base. The guard of the plan, enclosed by curly brackets, determines the precondition of the plan (what has to be true for the plan to be selected for execution), which in this case is the belief that the agent is patrolling waypoint A and that it is time to turn to go to the next waypoint. The body of the plan begins after the left arrow sign, with each operation (e.g., the removal of the belief patrol turn) or action (e.g., closing the mast) separated by a comma and a semicolon after the last element in the body of the plan.

```
+movement_completed :
    { B patrol("A"), B patrol("turn")  }
    <-
    -movement_completed, -patrol("turn"), print("Turning to go to Waypoint B"),
    control_wheels("right",5,10.0), control_mast("close"), control_arm("open");
```

Listing 11.2 Plan for turning to move to waypoint B

[4]Source code is available at: https://github.com/autonomy-and-verification-uol/gwendolen-ros-curiosity

11.4.2 Model checking

Model checking [1] is the process of exhaustively performing a state space search to check if some desired property holds. This can be done either with a formal model of the system, encoded in some specification language, or directly within the program, called program model checking. The property that we want to verify also has to be specified using some language, usually logic-based. For example, we may want to verify a property that states that the Curiosity will not move its arm while it is collecting soil and rock data, to prevent damaging the collection.

AJPF [32] is an extension of Java PathFinder [44], 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 on LTL that supports the description of terms usually found in BDI agents.

Some of the properties that we verified of the implementation of our agent were:

$$\Box(A_{\texttt{rover}}\mathit{move\_waypoint}(A) \rightarrow \Diamond \mathsf{B}_{\texttt{rover}}(\mathit{patrol}(A)))$$
$$\Box(A_{\texttt{rover}}\mathit{move\_waypoint}(B) \rightarrow \Diamond \mathsf{B}_{\texttt{rover}}(\mathit{patrol}(B)))$$
$$\Box(A_{\texttt{rover}}\mathit{move\_waypoint}(C) \rightarrow \Diamond \mathsf{B}_{\texttt{rover}}(\mathit{patrol}(C))$$
$$\Box(A_{\texttt{rover}}\mathit{move\_waypoint}(D) \rightarrow \Diamond \mathsf{B}_{\texttt{rover}}(\mathit{patrol}(D))$$

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

11.4.3 Runtime verification

Runtime verification (RV) [45] is a more lightweight approach that is usually more suitable for examining 'black box' software components. RV focuses on analysing only what the system produces while it is being executed and, because of this, it can only conclude the satisfaction/violation of properties regarding the current observed execution.

ROSMonitoring[5] is a framework for runtime monitoring of ROS topics. It creates monitors that are placed between ROS nodes to intercept messages on relevant topics and check the events generated by these messages against formally specified properties in an oracle. We applied this framework to the Curiosity case study using the filter action to intercept external messages sources from the agent that violate our property.

As an example of the filter action, consider an action library in ROS that controls the wheels of the rover. The content of the message includes the parameters discussed previously in the control wheels action: the *direction* for the rover to

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

move, the *speed* of the wheels and the *distance* that it should move. The configuration file in ROSMonitoring for this example is shown in Listing 11.3.

```
monitors:
  - monitor:
    id: monitor_0
    log: ./log_0.txt
    oracle:
      port: 8080
      url: 127.0.0.1
    topics:
      - name: wheels_control
        type:
        curiosity_mars_rover_description.msg.Move3
        action: filter
        warning: True
        side: subscriber
          - node: wheels_client
            path: /curiosity/launch/wheels.launch
```

Listing 11.3 Configuration file for the first Curiosity example

Due to the gravity and rocky/difficult terrain in Mars, the Curiosity has to be careful with its speed. Thus, when we intercept a message in the *wheels_control* topic, the message is sent to the oracle to verify the following property:

```
left_speed matches {topic:'wheels_control', direction:'left',
    speed:val} with val <= 10;
right_speed matches {topic:'wheels_control', direction:'right',
    speed:val} with val <= 10;
forward_speed matches {topic:'wheels_control', direction:'forward',
    speed:val} with val <= 15;
backward_speed matches {topic:'wheels_control', direction:'backward',
    speed:val} with val <= 15;
Main = (left_speed \/ right_speed \/ forward_speed \/ backward_speed)*;
```

That is, if the direction is left or right (i.e., a turn action) then the speed cannot be greater than 10, and if the direction is forward or backward then the speed cannot be greater than 15. These are arbitrary numbers that were defined based on testing to prevent the Curiosity from suffering any accidents. After the error is intercepted by the oracle, the agent could use this information to adapt its plan. For instance, using the agent reconfigurability approach introduced in [46], the agent could detect that a failure happened and better understand why it happened to reconfigure itself accordingly.

11.5 Case study 2: Verification of astronaut–rover teams

Sections 11.1, 11.3 and 11.4 used the example of a planetary surface rover performing autonomous inspection and surveying tasks on nearby planetary bodies such as the Moon or Mars, e.g., the Curiosity rover used on the Mars Science Laboratory mission since 2012. Such planetary rovers have been used in several missions

starting with the Soviet Union's Lunokhod 1 in 1970, and a number of further rover-based missions are being planned by various space agencies. Early rovers were limited in autonomous operations and were primarily remotely operated. However, autonomous systems are increasingly used to increase the reliability and efficiency of mission activities [47]. The increased use of autonomy also enables the proposed use of astronaut–rover teams, in which astronauts are assisted during missions by autonomous rovers [48–50]. For example, astronaut–rover teams have been evaluated for use in planetary outpost assembly [51], may employ multiagent planning systems to distribute tasks between team members [52] and can be assessed at the mission conception stage using simulators [53] and terrestrial field tests with robot prototypes [54].

As described in the previous sections, it is possible to use formal methods in the form of model checking to formally verify the rover's behaviour in the astronaut–rover team. In Section 11.4, the Gwendolen agent programming language was used to specify the behaviour of a decision-making agent in control of the rover. In this case, however, a different approach was taken due to the need to model the behaviour of one or more astronauts in the astronaut–rover team. This new approach used a multiagent workflow specification language called Brahms [55, 56] to model the behaviour of the astronaut–rover team. Brahms has been used before for modelling human–robot teamwork as part of the Mobile Agents Architecture at NASA Ames Research Center [57]. It has also been used to implement systems for mission control for the International Space Station [58] and health monitoring of astronauts [59].

Brahms consists of a modelling language and an integrated development environment (IDE). The modelling language allows multiagent systems to be specified in terms of interacting agents. Each agent has a set of beliefs that resemble common programming language variables types such as Booleans and integers. Agents can be given a location within a topological 'geography' within the Brahms model. Agent can perform activities or movements that can take a period of time. Each agent's behaviour is specified through workframes and thoughtframes. The former allows the agent to perform activities, move and update beliefs, whereas the latter only allows instantaneous belief updates, known as inferences. Workframes can also be interrupted by the receipt of new information from another agent. Communication between agents is modelled using a primitive 'communicate' construct. The design of the Brahms language allows for detailed models of multiagent systems to be developed in an intuitive way. Once a model has been developed, it is possible to run simulations of the model using the Brahms IDE, known as the 'Composer'. The results of these simulations can be displayed in the form of a timeline showing agent locations, workframes and activities as horizontal bars, and with communications shown as vertical lines (see Figure 11.4).

A Brahms model of an astronaut–rover team was developed. The model was based on a scenario similar to those used during NASA field tests and demonstrations of astronaut–rover teamwork [48, 50, 57], in which astronauts and a rover work together to achieve mission goals at a (simulated) outpost on the Moon or Mars. During the scenario, the rover assists the astronaut and performs autonomous behaviours when its assistance is not needed. For example, when the astronaut is

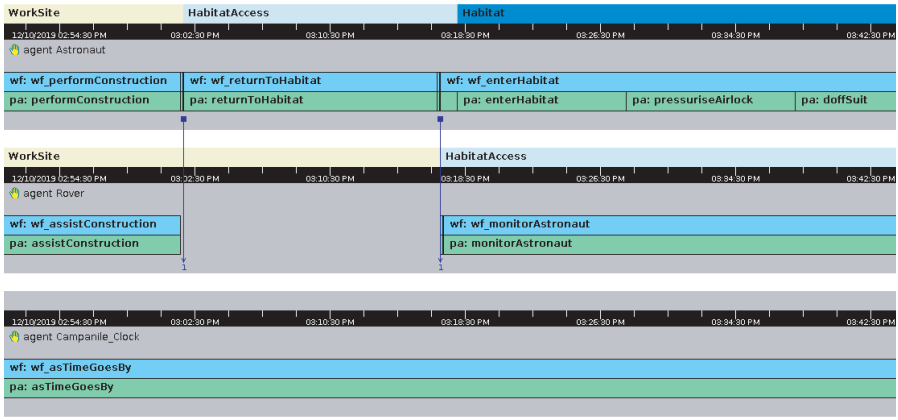


Figure 11.4 Simulation of the astronaut–rover scenario using the Brahms Composer IDE

performing construction or geological surveying tasks, the rover will assist the astronaut, e.g., by following the astronaut and providing a mobile platform for tools and materials. During the video extra-vehicular activity (EVA) the rover assists by turning on the camera stream and filming the astronaut so that they can be monitored by the ground team. A similar behaviour is performed when the astronaut enters or leaves the habitat as these operations are especially hazardous. If the astronaut chooses to perform a miscellaneous activity then the rover will recognise that its assistance is not needed and will perform a solo geological survey. Also, whenever the astronaut is in the habitat, the rover performs a habitat-monitoring activity to ensure that the integrity of the habitat's life-support systems is maintained.

The Brahms model of the scenario used three agents: one to model an astronaut's behaviour, the second to model the autonomous rover's behaviour and the third, called the campanile clock, to assist in measuring the passage of time. The model examines the events over the course of a typical work day. The astronaut begins the day in the habitat and at some point decides to leave the habitat and start work. Leaving the habitat involves donning a spacesuit, depressurising the airlock and moving outside through the external door. Once the astronaut is outside, they can choose from a number of different behaviours: construction, geological surveying, video-recorded EVA or miscellaneous activities. At the end of the work day the astronaut returns to and enters the habitat. After entering the astronaut repressurises the airlock and doffs the spacesuit. The astronaut then remains in the habitat for the rest of the work day. The astronaut's choice between different behaviours is modelled as a non-deterministic choice between workframes in the astronaut agent.

As mentioned earlier, the Brahms Composer IDE can be used to display the results of simulations using a Brahms model. An excerpt of a simulation of the astronaut–rover model is shown in Figure 11.4. The behaviour of the three agents (astronaut, rover and campanile clock) are shown as horizontal bands. Time progresses from the left to the right. The locations of the agents are shown at the top of

the bands. For example, at the start of this excerpt the astronaut agent is at the work site, is in the 'perform construction' workframe and is performing the 'perform construction' activity. While this is happening the rover is in the 'assist construction' workframe and is performing the 'assist construction' activity. After completing the construction task the astronaut decides to return to the habitat. The rover stops working while the astronaut moves to the habitat access point. When the astronaut enters the habitat the rover begins monitoring the astronaut by entering the 'monitor astronaut' workframe. During the entire simulation the campanile clock agent is monitoring the time and announcing it to the other agents. The campanile clock is used as a means of synchronising agent behaviours. For example, at the end of the work day the campanile clock informs the other agents that the work day has ended. When this happens the astronaut agent will stop work and return to the habitat.

It is possible to simulate this scenario many times using Brahms to determine whether the behaviour of the autonomous rover satisfies mission requirements. However, it is difficult to tell using simulation whether these requirements are satisfied in all cases. Therefore it may be useful to perform an exhaustive analysis using model checking to determine whether requirements hold for the agent-based decision-making system. This can be done for the Brahms model of the astronaut–rover scenario by translating the Brahms model code into the input language for a model checker. This was done automatically using the BrahmsToPromela software, which translates Brahms model code into Promela, the input language for the Spin model checker [2]. Once translation is complete, Spin can be used to exhaustively analyse the Promela model to determine whether it satisfies requirements encoded formally as *properties* in LTL. The results of the model checking will also apply to the Brahms model as long as we have validated the automatic translation performed by BrahmsToPromela. Validation was achieved by developing BrahmsToPromela with respect to a formal semantics of Brahms [60] and through extensive applications in other human–robot team scenarios [61, 62].

To demonstrate the approach, an initial set of four mission- or safety-critical requirements were examined:

1. The rover should perform a solo geological survey whenever the astronaut is performing a miscellaneous activity. [Mission-critical.]
2. The rover should assist the astronaut during construction tasks. [Mission-critical.]
3. The rover should monitor the astronaut when they are leaving the habitat. [Safety-critical.]
4. The rover should monitor the habitat whenever the astronaut is located inside the habitat. [Safety-critical.]

These requirements were formalised as four LTL properties. These are shown in Table 11.4 LTL allows the formalisation of concepts relating to time, e.g., 'now and at all points in the future' (via the $\Box$ operator), 'now or at some point in the future' ($\Diamond$) and 'in the next state' ($\bigcirc$) [63]. This enables formalisation of safety requirements (something bad never happens, $\Box\neg$bad), liveness properties (e.g., something good eventually happens, $\Diamond$good) and fairness properties (e.g., if one

Table 11.4 Properties verified for the Brahms model of the astronaut–rover scenario

Req.	Property	Description
1	$\Box\left[\begin{array}{l} B_{Astro}(goalPerformMisc) \Longrightarrow \\ \Diamond\, B_{Rover}(goalSoloGeoSurvey) \end{array}\right]$	It is always the case that if the astronaut agent believes that it is performing a miscellaneous activity (i.e., an activity that does not require the assistance of the rover), then the rover will perform a solo geological survey after a period of time.
2	$\Box\left[\begin{array}{l} B_{Astro}(goalPerformConstr) \Longrightarrow \\ \Diamond\, B_{Rover}(goalAssistConstr) \end{array}\right]$	It is always the case that if the astronaut agent believes that it has a goal to start construction, then the rover agent will form a corresponding goal to assist in the construction task after a period of time.
3	$\Box\left[\begin{array}{l} B_{Astro}(goalLeaveHabitat) \Longrightarrow \\ \Diamond\, B_{Rover}(cameraStream) \end{array}\right]$	It is always the case that if the astronaut decides to leave the habitat, the rover will start monitoring by setting the camera stream variable to true, indicating that a video stream is being sent back to the habitat (and then back to the ground station for monitoring, if needed).
4	$\Box\left[\begin{array}{l} Astro.location = Habitat \Longrightarrow \\ \Diamond\, B_{Rover}(goalSoloMonitorHab) \end{array}\right]$	It is always the case that if the astronaut is in the habitat, then the rover will form a goal to autonomously monitor the habitat.

thing occurs infinitely often so does another, e.g., $\Box\Diamond \text{send} \Longrightarrow \Box\Diamond \text{receive}$). Using BrahmsToPromela extends Spin's property specification language with a *belief* operator, 'B'. This allows us to specify that an agent has a belief, e.g., $B_{Rover}x$ means that the Rover agent believes x is true.

No errors were found by the model checker and therefore all properties held for the Promela model, meaning that the autonomous behaviour of the robot, was correct with respect to the requirements. Using this approach we were able to determine that it is possible to use formal methods, in particular, model checking, to formally verify the behaviour of an autonomous rover within a realistic astronaut–rover scenario.

11.6 Modelling and verification of multi-objects systems

1.6.1 Motivation

Sections 11.1, 11.3 and 11.4 presented various ways to model and verify the behaviour of a single autonomous planetary rover operating on nearby planetary bodies such as the Moon or Mars. Section 11.5 provided a methodology to model and verify the rover's behaviour in an astronaut–rover team. In this section, we consider generalisations of the above scenarios with two or more *identical* rovers working in cooperation. Such scenarios present many challenges with regard to coordination

and resource management, hence it is important to address these challenges at the appropriate level of abstraction.

To motivate our presentation, let us discuss a simple generalisation of the scenario in Section 1.5. In our version of the scenario, we have a team of an astronaut working with k ($k > 1$) autonomous planetary rovers $r_1, \ldots, r_k$ to perform an action. Our mission- and safety-critical requirements will be as follows:

1. Each rover should perform a solo geological survey whenever the astronaut is performing a miscellaneous activity. [Mission-critical.]
2. One rover should assist the astronaut during construction tasks. [Mission-critical.]
3. One rover should monitor the astronaut when they are leaving the habitat. [Safety-critical.]
4. One rover should monitor the habitat whenever the astronaut is located inside the habitat. [Safety-critical.]

Given the above requirements, we would like to state analogues of the properties in Table 11.4. To simplify matters, let us forget about the belief operators in Section 5.

If the number k is fixed, the most obvious way to encode the above requirements is to use LTL. For the astronaut, we can introduce four propositional variables 'astrGoalPerformMisc', 'astrGoalPerformConstruction', 'astrGoalLeaveHabitat' and 'astrInHabitat', to be viewed as stating, respectively, that 'the astronaut has a goal to perform miscellaneous activity', 'the astronaut has a goal to start construction', 'the astronaut decides to leave the habitat' and 'the astronaut is in the habitat'. For the rover requirements, we can introduce for each rover r_i ($1 \leq i \leq k$) four propositional variables 'rvGoalSoloSurvey$_i$', 'rvGoalAssistConstruction', 'rvCameraStream$_i$' and 'rvGoalSoloMonitorHab$_i$', to be viewed as stating, respectively, that 'rover r_i will perform a solo geological survey', 'rover r_i will assist the astronaut's construction task', 'rover r_i will send a camera stream back to the ground station' and 'rover r_i will autonomously monitor the habitat'. Now, requirements 1–4 can be stated as follows:

1. $\Box[\text{astrGoalPerformMisc} \Longrightarrow \bigwedge_{1 \leq i \leq k} \Diamond\, \text{rvGoalSoloSurvey}_i]$
2. $\Box[\text{astrGoalPerformConstruction} \Longrightarrow \bigvee_{1 \leq i \leq k} \Diamond\, \text{rvGoalAssistConstruction}_i]$
3. $\Box[\text{astrGoalLeaveHabitat} \Longrightarrow \bigvee_{1 \leq i \leq k} \Diamond\, \text{rvCameraStream}_i]$
4. $\Box[\text{astrInHabitat} \Longrightarrow \bigvee_{1 \leq i \leq k} \Diamond\, \text{rvGoalSoloMonitorHab}_i]$

The above specification of our requirements has two main disadvantages. First, it depends on the number k being fixed. Thus, it can only answer the question 'given a value of k (e.g., 5), does the above system of the astronaut and the rovers $r_1, \ldots, r_k$ have a given property $\mathcal{P}$?', whereas it would be ideal to answer the more general question 'does the system have the property $\mathcal{P}$ *for all* values of k?'. Second, the above specification is not succinct, since it requires each rover to be mentioned individually in every formula that refers to the totality of rovers. This also makes it difficult to keep track of messages in the system in more complex scenarios that involve communication. It is

natural then to seek more expressive languages than LTL that allow each individual in a system like the above to be referred to in a more abstract manner.

11.6.2 Logics for parameterised systems

Recognising the need for better abstractions and formal languages in the verification of parameterised systems, i.e., systems comprising arbitrary numbers of identical components (such as the above system of rovers), various approaches have been proposed in recent decades. Two of the most popular are model checking for parameterised and infinite state-systems [64, 65] and constraint-based verification using counting abstractions [66–68]. The model-checking approach has been applied to several scenarios verifying safety properties and some liveness properties, but is in general incomplete. Constraint-based approaches [67] do provide complete procedures for checking safety properties, but these procedures have non-primitive recursive upper bounds, and thus do not scale well for large instances. In addition, they usually lead to undecidability when applied to liveness properties.

Another approach is *first-order temporal logic* (FOTL), which can be viewed as a first-order generalisation of LTL. Although this logic is incomplete (not finitely axiomatisable) [69] and generally undecidable [70], it is valuable from a practical standpoint because a certain syntactic restriction to it, referred to as *monodic* FOTL, is *finitely axiomatisable* [71], in many cases *decidable* [70] and can naturally model systems of identical, communicating finite-state machines arising frequently in the verification of distributed systems and protocols [72, 73]. In the ensuing part, we briefly present the syntax and semantics of FOTL, as well as the syntactic restriction of monodicity and show how it can be used in the specification and verification of practical systems.

The symbols used in FOTL are predicate symbols $P_0, P_1, \ldots$, each of fixed arity (0-ary or nullary predicate symbols are allowed and correspond to propositions); variables $x_0, x_1, \ldots$, constants $c_0, c_1, \ldots$; the propositional constants $\top$ (true) and $\bot$ (false); the usual Boolean connectives ($\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow$); the quantifiers $\forall$ (for all) and $\exists$ (exists); and the temporal operators $\Box$ (always in the future), $\Diamond$ (sometime in the future), $\bigcirc$ (at the next moment), $\mathcal{U}$ (until) and **start** (at the first moment). Neither equality nor function symbols are allowed. The syntax of FOTL is as follows [70, 74]:

- $\top$and $\bot$ are atomic FOTL-formulae;
- if P is an n-ary predicate symbol and ti, $1 \leq i \leq n$, are variables or constants, then $P(t_1, \ldots, t_n)$ is an atomic FOTL-formula;
- if ϕ and ψ are FOTL-formulae, so are $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \Rightarrow \psi$, and $\phi \Leftrightarrow \psi$;
- if ϕ is an FOTL-formula and x is a variable, then $\forall x\, \phi$ and $\exists x\, \phi$ are FOTL-formulae;
- if ϕ and ψ are FOTL-formulae, then so are $\Box\phi$, $\Diamond\phi$, $\bigcirc\phi$, $\phi\, \mathcal{U}\, \psi$, and **start**.

FOTL-formulae are interpreted in *first-order temporal structures*, i.e., sequences $\mathfrak{M} = \mathfrak{A}_0, \mathfrak{A}_1, \ldots$ of first-order structures over a common universe A. In more detail, if A is a non-empty set, each $\mathfrak{A}_n$ ($n \in \mathbb{N}$) is a pair $\langle A, I_n \rangle$, where I_n is an interpretation

$\mathfrak{A}_n \models^{\mathfrak{a}} \top$	iff	$\mathfrak{A}_n \not\models^{\mathfrak{a}} \bot$
$\mathfrak{A}_n \models^{\mathfrak{a}} \mathbf{start}$	iff	$n = 0$
$\mathfrak{A}_n \models^{\mathfrak{a}} P(t_1, \ldots, t_m)$	iff	$\langle I_n^{\mathfrak{a}}(t_1), \ldots I_n^{\mathfrak{a}}(t_m)\rangle \in I_n(P)$, where $I_n^{\mathfrak{a}}(t_i) = I_n(t_i)$ if t_i is a constant, and $I_n^{\mathfrak{a}}(t_i) = \mathfrak{a}(t_i)$ if t_i is a variable
$\mathfrak{A}_n \models^{\mathfrak{a}} \neg\phi$	iff	$\mathfrak{A}_n \not\models^{\mathfrak{a}} \phi$
$\mathfrak{A}_n \models^{\mathfrak{a}} \phi \wedge \psi$	iff	$\mathfrak{A}_n \models^{\mathfrak{a}} \phi$ and $\mathfrak{A}_n \models^{\mathfrak{a}} \psi$
$\mathfrak{A}_n \models^{\mathfrak{a}} \phi \vee \psi$	iff	$\mathfrak{A}_n \models^{\mathfrak{a}} \phi$ or $\mathfrak{A}_n \models^{\mathfrak{a}} \psi$
$\mathfrak{A}_n \models^{\mathfrak{a}} \phi \Rightarrow \psi$	iff	$\mathfrak{A}_n \models^{\mathfrak{a}} \neg\phi \vee \psi$
$\mathfrak{A}_n \models^{\mathfrak{a}} \phi \Leftrightarrow \psi$	iff	$\mathfrak{A}_n \models^{\mathfrak{a}} (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$
$\mathfrak{A}_n \models^{\mathfrak{a}} \forall x\, \phi$	iff	$\mathfrak{A}_n \models^{\mathfrak{b}} \phi$ for every assignment $\mathfrak{b}$ that may differ from $\mathfrak{a}$ only in x and such that $\mathfrak{b}(x) \in A$
$\mathfrak{A}_n \models^{\mathfrak{a}} \exists x\, \phi$	iff	$\mathfrak{A}_n \models^{\mathfrak{b}} \phi$ for some assignment $\mathfrak{b}$ that may differ from $\mathfrak{a}$ only in x and such that $\mathfrak{b}(x) \in A$
$\mathfrak{A}_n \models^{\mathfrak{a}} \bigcirc\phi$	iff	$\mathfrak{A}_{n+1} \models^{\mathfrak{a}} \phi$;
$\mathfrak{A}_n \models^{\mathfrak{a}} \Diamond\phi$	iff	there exists $m \geq n$ such that $\mathfrak{A}_m \models^{\mathfrak{a}} \phi$;
$\mathfrak{A}_n \models^{\mathfrak{a}} \Box\phi$	iff	for all $m \geq n$, $\mathfrak{A}_m \models^{\mathfrak{a}} \phi$
$\mathfrak{A}_n \models^{\mathfrak{a}} \phi\, \mathscr{U}\, \psi$	iff	there exists $m \geq n$, such that $\mathfrak{A}_m \models^{\mathfrak{a}} \psi$ and, for all $i \in \mathbb{N}, n \leq i < m$ implies $\mathfrak{A}_i \models^{\mathfrak{a}} \phi$.

Figure 11.5 Semantics of FOTL

of predicate and constant symbols over A, assigning to each predicate symbol P a predicate $P^{\mathfrak{A}_n}$ on A of the same arity as P (if P is a nullary predicate, $P^{\mathfrak{A}_n}$ is simply one of the propositional constants $\top$ or $\bot$), and to each constant symbol c an element $c^{\mathfrak{A}_n}$ of A. We require that the interpretation of constants be *rigid*, i.e., $I_n(c) = I_m(c)$, for all $n, m \in \mathbb{N}$. Intuitively, each $\mathfrak{A}_n$ ($n \in \mathbb{N}$) represents the state of the world at time n and truth values in different worlds are associated via temporal operators. An *assignment* $\mathfrak{a}$ in A is a function from the set of variables $\{x_0, x_1, \ldots\}$ to A. The *truth* relation $(\mathfrak{M}, \mathfrak{A}_n) \models^{\mathfrak{a}} \phi$ (or simply $\mathfrak{A}_n \models^{\mathfrak{a}} \phi$) in the model $\mathfrak{M}$ is defined in a manner analogous to LTL. See Figure 11.5 for details. We say that $\mathfrak{M}$ is a *model* for a formula ϕ or that ϕ is *true* in $\mathfrak{M}$ if there exists an assignment $\mathfrak{a}$ such that $\mathfrak{A}_0 \models^{\mathfrak{a}} \phi$. A formula is *satisfiable* if it has a model and *valid* if it is true in any temporal structure under any assignment.

As discussed earlier, FOTL is incomplete [69] and generally undecidable [70]. We now define a subset of all possible FOTL-formulae, the *monodic FOTL-formulae*, that allow us to regain finite axiomatisability and in many cases decidability. An FOTL-formula is called *monodic* if any subformula of ϕ of the form $\bigcirc\psi$, $\Diamond\psi$, $\Box\psi$, or $\psi_1\, \mathcal{U}\, \psi_2$ has *at most one* free variable. For example, the formulae $\Box\, \forall x(p(x) \Rightarrow \bigcirc q(x))$ and $\forall x\, \Diamond \exists y\, p(x,y)$ are monodic, whereas the formula $\forall x \forall y(p(x,y) \Rightarrow \bigcirc q(x,y))$ is *not* monodic. The set of all monodic FOTL-formulae form the *monodic fragment*, abbreviated MFOTL, of FOTL. MFOTL is *finitely axiomatisable* [71] and *decidable* (in 2-NExpTime) if, roughly speaking, one restricts

the pure classical (first-order) part of monodic formulae to any decidable fragment of first-order logic [70].

For a larger example, let us revisit the requirements for the autonomous planetary rovers considered earlier. We shall keep the propositional variables (nullary predicates) 'astrGoalPerformMisc', 'astrGoalPerformConstruction', 'astrGoalLeaveHabitat' and 'astrInHabitat' and their intended meaning. But instead of the propositional variables corresponding to each individual rover, we now use four unary predicates 'rvGoalSoloSurvey(x)', 'rvGoalAssistConstruction(x)', 'rvCameraStream(x)' and 'rvGoalSoloMonitorHab(x)' referring to an arbitrary rover x, to be viewed as stating, respectively, that 'rover x will perform a solo geological survey', 'rover x will assist the astronaut's construction task', 'rover x will send a camera stream back to the ground station' and 'rover x will autonomously monitor the habitat'. Using MFOTL, requirements 1–4 stated previously, can now be stated as follows:

1. $\Box[\text{astrGoalPerformMisc} \Longrightarrow \forall x \Diamond\ \text{rvGoalSoloSurvey}(x)]$
2. $\Box[\text{astrGoalPerformConstruction} \Longrightarrow \exists x \Diamond \text{rvGoalAssistConstruction}(x)]$
3. $\Box[\text{astrGoalLeaveHabitat} \Longrightarrow \exists x \Diamond\ \text{rvCameraStream}(x)]$
4. $\Box[\text{astrInHabitat} \Longrightarrow \exists x \Diamond\ \text{rvGoalSoloMonitorHab}(x)]$

Notice that at no point is a reference made to an individual rover or the number of rovers in the system. If our specification is proved correct, its correctness applies to systems of rovers of *any* size.

From a practical standpoint, two theorem provers are available for FOTL: TeMP [75] and TSPASS [76, 77]. TeMP has been successfully applied to problems from several domains [78], in particular, to examples specified in the temporal logics of knowledge (the fusion of propositional LTL with multi-modal S5) [79–81]. TSPASS has been used (among other things) to reason about contract violations [82] and accountability [83, 84] in distributed protocols as well as the behaviour of robots and robot swarms [85]. We remark that the above provers implement FOTL with so-called *expanding domain* semantics. This allows them to use a simplified clausal resolution calculus [86]. In contrast, in our presentation of FOTL we used so-called *constant domain* semantics. This detail does not affect the reader: satisfiability with constant domain semantics and satisfiability with expanding domain semantics can be reduced to each other with only a polynomial increase in the size of the formulae [87].

11.6.3 Translating broadcast protocols to MFOTL

Writing specifications in MFOTL can seem difficult at first to people unfamiliar with logic. Since a lot of verification tasks involve distributed protocols, it is natural to ask whether some commonly used language for the specification of distributed protocols can automatically be translated to MFOTL. Indeed, distributed protocols are often represented as collections of finite-state machines exchanging messages. We now give a brief overview of a distributed system model comprising an arbitrary

number of identical finite-state machines communicating by broadcasting messages. This model is quite expressive, capturing many interesting and useful systems, and distributed protocols described in its terms can be *automatically* be translated in MFOTL [72, 73]. In particular, it is rich enough to describe (possibly with small extensions) such diverse systems as cache coherence protocols [67] or distributed atomic commitment protocols including the two- and three-phase commit protocols [88, 89] and their modifications [90, 91].

In the aforementioned distributed system model, we have a collection of k ($k > 1$) *identical* finite-state machines in a network environment. The transitions of these finite-state machines correspond to three types of actions: (a) broadcast a message μ (denoted μ); (b) receive a message μ (denoted $\overline{\mu}$) and (c) local (i.e., an action not related to the network). The delivery of messages in the network is guaranteed. At each moment of time, each machine in the network performs an action depending on its local state at that time or is *idle*, i.e., performs no action at all. (The latter is useful for modelling asynchrony.) See [72, 73] for more technical details.

Now, given such a distributed system, say D, we can construct a MFOTL-formula $\mathcal{T}_D$ such that to each valid execution (run) of D corresponds a temporal model of $\mathcal{T}_D$ and vice versa. In other words, $\mathcal{T}_D$ completely captures the operation of D. (Again, for more details, see [72, 73].) Thus, to check whether the operation of D has a property $\mathcal{P}$, we can (assuming that $\mathcal{P}$ is expressible in MFOTL) check whether the MFOTL-formula $\mathcal{T}_D \Rightarrow \mathcal{P}$ is valid. This obviates the need to specify the operation of D in MFOTL: the specification in MFOTL can be obtained *automatically* from the state machine description of the system. $\mathcal{T}_D$ is written over the signature Σ_D containing a unary predicate symbol $P_q(x)$ for each state q of the machines, a unary predicate $A_\tau(x)$ for each transition τ of the machines (corresponding to actions (a), (b) or (c) above) and a nullary predicate (proposition) μ for each message μ that can be broadcast. Intuitively, $P_q(x)$ is to be viewed as stating that 'machine x is in state q', $A_\tau(x)$ as stating that 'machine x performs action τ' and μ as stating that 'the message μ is in transition'.

To clarify the above, let us consider an example relevant to the system of planetary rovers described earlier. Suppose that all rovers in the system are away from the astronaut, each performing a solo geological survey as described in requirement 1. Suppose, then, the astronaut requests the assistance of a rover for a construction task. Adhering to requirement 2, one of the rovers must move towards the astronaut for assistance. Suppose, further, that, for energy conservation, we would like exactly one of the rovers to go to the astronaut. (This is something that we can include in our requirements.) In distributed systems terminology, this is a scenario in which the rovers must achieve *consensus* [92]. That is, the rovers must agree on which of them will go to the astronaut. Suppose, now, there is a proposal that rover r_i ($1 \leq i \leq k$), e.g., go to the astronaut, and the rovers vote on whether to reject (0) or accept (1) the proposal.

Let the rovers use the following simplified (asynchronous) variant of the *FloodSet algorithm* [92, p. 105] modelled in FOTL in [73]. Each rover has a preset default bit d (0 for reject and 1 for accept). Each rover has a result bit r, which will eventually contain the result of each rover's decision (0 for reject and 1 for accept).

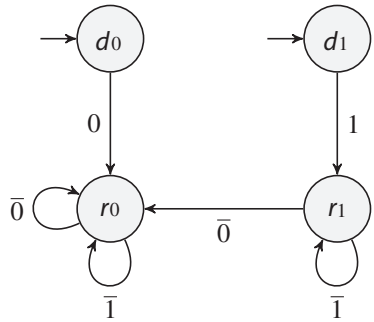


Figure 11.6 Each rover's voting behaviour

The goal of the algorithm is for all rovers to reach a consensus, i.e., to eventually produce the same result bit. It is also required that if all rovers have been initialised with the same default bit, that bit should be produced as a result. The protocol proceeds in the following rounds:

- At the first round, every rover broadcasts (the value of) its preset default bit.
- At every round the result bit is set to the minimum value ever received so far.

Thus, each rover's behaviour is described by the finite-state machine in Figure 11.6, where the states d_0, d_1, r_0 and r_1 denote, respectively, that 'the preset default bit is 0', 'the preset default bit is 1', 'the result bit is 0' and 'the result bit is 1'; and the transitions 0, 1, $\overline{0}$ and $\overline{1}$ correspond, respectively, to 'broadcasting 0', 'broadcasting 1', 'receiving 0' and 'receiving 1'. Now, as discussed earlier, let $\mathcal{T}_{\text{Flood}}$ be the MFOTL-formula that captures the above system, over the signature Σ_{Flood} containing the unary predicates $P_{d_0}(x)$ ('rover x is in state d_0'), $P_{d_1}(x)$ ('rover x is in state d_1'), $P_{r_0}(x)$ ('rover x is in state r_0') and $P_{r_1}(x)$ ('rover x is in state r_1'); $A_0(x)$ ('rover x broadcasts 0'), $A_1(x)$ ('rover x broadcasts 1'), $A_{\overline{0}}(x)$ ('rover x receives 0') and $A_{\overline{1}}(x)$ ('rover x receives 1'); and the nullary predicates (propositions) 0 ('message 0 in transition') and 1 ('message 1 in transition'). Recall that $\mathcal{T}_{\text{Flood}}$ can be obtained automatically from Figure 11.6, so we need not know its content. To prove that the algorithm achieves the informal goal stated above, we check that the following MFOTL-formula is valid:

$$\mathcal{T}_{\text{Flood}} \implies \Diamond(\forall x\, P_{r_0}(x) \vee \forall x\, P_{r_1}(x)).$$

Thus using FOTL provides a route to verifying multi-robot systems with an arbitrary number of identical components. As we have already mentioned, while full FOTL is incomplete and in general undecidable, using monodic FOTL-formulae, we regain finite axiomatisability and in many cases decidability. However, the disadvantage is that this restriction limits what we can express in the logic. Also the theorem-proving tools for FOTL are not as developed in terms of usability as many model checkers.

11.7 Conclusions, recommendations and future trends

Verification techniques such as formal verification, simulation and testing are useful when ensuring systems are safe, trustworthy and meet their stated requirements. They are needed for space robotics as failures in space may be much more critical, costly and harder to resolve. We have discussed several tools and techniques for verification of space robotics with reference to some simple space scenarios.

Recommendations include designing systems for verification using a modular approach separating concerns, embedding verification and validation into engineering process, and using a range of tools and techniques to improve confidence in space systems. Future trends include the greater need for and use of autonomy in space robotics, e.g., to support planetary missions with robots working closely with astronauts, where safety and functional correctness is crucial. Additionally, verification and validation is needed for the New Space sector to conform regulation and standards for applications such as satellite communication, imaging, navigation, space tourism and mining.

References

[1] Clarke E.M., Grumberg O., Peled D. *Model checking*. MIT Press; 1999.
[2] Holzmann G.J. *The spin model checker: primer and reference manual*. Addison-Wesley; 2003.
[3] Fisher M. *An introduction to practical formal methods using temporal logic*. Wiley; 2011.
[4] Pnueli A. 'The temporal logic of programs'. *Symposium on the Foundations of Computer Science*; IEEE; 1977. pp. 46–57.
[5] Gabbay D., Pnueli A., Shelah S., *et al*. 'On the temporal analysis of fairness'. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*; 1980. pp. 163–73.
[6] Clarke E.M., Emerson E.A. 'Design and synthesis of synchronisation skeletons using branching time temporal logic'. *Workshop on the Logic of Programs. Vol. 131 of LNCS*. Springer; 1981. pp. 52–71.
[7] Fisher M., Dixon C., Peim M. 'Clausal temporal resolution'. *ACM Transactions on Computational Logic*. 2001;**2**(1):12–56.
[8] Hustadt U., Konev B. 'TRP++ 2.0: a temporal resolution prover'. *Automated Deduction—CADE-19. Vol. 2741 of LNAI*. Springer; 2003. pp. 274–8.
[9] Zhang L., Hustadt U., Dixon C. 'A resolution calculus for the branching-time temporal logic CTL'. *ACM Transactions on Computational Logic*. 2014;**15**(1):1–38.
[10] Zhang L., Hustadt U., Dixon C. 'CTL-RP: a computation tree logic resolution prover'. *AI Communications*. 2010;**23**(2-3):111–36.
[11] Araiza-Illan D., Western D., Pipe A.G. 'Systematic and realistic testing in simulation of control code for robots in collaborative human-robot interactions'.

Towards Autonomous Robotic Systems. Vol. 9716 of LNCS. Springer; 2016. pp. 20–32.

[12] Salem M., Lakatos G., Amirabdollahian F. 'Would you trust a (faulty) robot?: effects of error, task type and personality on human-robot cooperation and trust'. ACM/IEEE International Conference on Human-Robot Interaction; 2015. pp. 141–8.

[13] Webster M., Western D., Araiza-Illan D., *et al*. 'A corroborative approach to verification and validation of human–robot teams'. *The International Journal of Robotics Research*. 2020;**39**(1):73–99.

[14] Luckcuck M., Farrell M., Dennis L.A., *et al*. 'Formal specification and verification of autonomous robotic systems'. *ACM Computing Surveys*. 2019;**52**(5):1–41.

[15] Farrell M., Luckcuck M., Fisher M. 'Robotics and integrated formal methods: necessity meets opportunity'. *Integrated formal methods. Vol. 11023 of LNCS*. Springer; 2018. pp. 161–71.

[16] Luckcuck M., Farrell M., Dennis L.A. 'A summary of formal specification and verification of autonomous robotic systems'. *Integrated Formal Methods. Vol. 11918 of LNCS*. Springer; 2019. pp. 538–41.

[17] Cresswell M.J., Hughes G.E. *A new introduction to modal logic*. Taylor and Francis; 1997.

[18] Fagin R., Halpern J.Y., Moses Y., *et al*. *Reasoning about knowledge*. MIT Press; 1995.

[19] Hansson H., Jonsson B. 'A logic for reasoning about time and reliability'. *Formal Aspects of Computing*. 1994;**6**(5):512–35.

[20] Koymans R. 'Specifying real-time properties with metric temporal logic'. *Real-Time Systems*. 1990;**2**(4):255–99.

[21] Blackburn P., van Benthem J., Wolter F. (eds.). *Handbook of modal logic: vol. 3 of studies in logic and practical reasoning*. North-Holland; 2007. Available from https://www.sciencedirect.com/bookseries/studies-in-logic-and-practical-reasoning/vol/3/suppl/C.

[22] Nalon C., Dixon C., Hustadt U. 'Modal resolution: proofs, layers, and refinements'. *ACM Trans Comput Log*. 2019.

[23] Nalon C., Hustadt U., Dixon C. 'A resolution-based theorem prover for kn: architecture, refinements, strategies and experiments'. *Journal of Automated Reasoning*. 2020;**64**(3):461–84.

[24] Hustadt U., Ozaki A., Dixon C. 'Theorem proving for metric temporal logic over the naturals'. *Automated Deduction - CADE 26. vol. 10395 of LNCS*. Springer; 2017. pp. 326–43.

[25] Hustadt U., Ozaki A., Dixon C. 'Theorem proving for pointwise metric temporal logic over the naturals via translations'. *Journal of Automated Reasoning*. 2020;**64**(8):1553–610.

[26] Li J., Vardi M.Y., Rozier K.Y. 'Satisfiability checking for mission-time LTL' in Dillig I., Tasiran S. (eds.). *Computer Aided Verification – 31st International Conference, CAV 2019, New York City, NY, USA, July 15–18, 2019, Proceedings, Part II. vol. 11562 of Lecture Notes in Computer Science*. **2019**. Springer; 2019. pp. 3–22.

[27] Farrell M., Cardoso R.C., Dennis L.A., *et al*. 'Modular verification of autonomous space robotics'. *Assuring Autonomy for Space Missions Workshop*; 2019.
[28] Abrial J.R. *Modeling in Event-B: system and software engineering*. Cambridge University Press; 2010.
[29] Spivey J.M. *Understanding Z: a specification language and its formal semantics*. **3**. Cambridge University Press; 1988.
[30] Hoare C.A.R. 'Communicating sequential processes'. *Communications of the ACM*. 1978;**21**(8):666–77.
[31] Astesiano E., Bidoit M., Kirchner H., *et al*. 'CASL: the common algebraic specification language'. *Theoretical Computer Science*. 2002;**286**(2):153–96.
[32] Dennis L.A., Fisher M., Webster M.P., *et al*. 'Model checking agent programming languages'. *Automated Software Engineering*. 2012;**19**(1):5–63.
[33] Broy M. 'A logical approach to systems engineering artifacts: semantic relationships and dependencies beyond traceability—from requirements to functional and architectural views'. *Software & Systems Modeling*. 2018;**17**(2):365–93.
[34] Champion A., Gurfinkel A., Kahsai T. CoCoSpec: A mode-aware contract language for reactive systems. *International Conference on Software Engineering and Formal Methods. vol. 9763 of LNCS*; Springer; 2016. pp. 347–66.
[35] Cimatti A., Dorigatti M., Tonetta S. OCRA: A tool for checking the refinement of temporal contracts. *International Conference on Automated Software Engineering (ASE)*; IEEE; 2013. pp. 702–5.
[36] Cofer D., Gacek A., Miller S., *et al*. 'Compositional verification of architectural models'. *NASA Formal Methods Symposium. vol. 7226 of LNCS*; Springer; 2012. pp. 126–40.
[37] Kornfeld R.P., Prakash R., Devereaux A.S., Greco M.E., Harmon C.C., Kipp D.M. 'Verification and validation of the Mars science laboratory/Curiosity rover entry, descent, and landing system'. *Journal of Spacecraft and Rockets*. 2014;**51**(4):1251–69.
[38] Francis R., Estlin T., Doran G., *et al*. 'AEGIS autonomous targeting for ChemCam on mars science laboratory: deployment and results of initial science team use'. Science Robotics. 2017. Available from https://robotics.sciencemag.org/content/2/7/eaan4582.
[39] Cardoso R.C., Farrell M., Luckcuck M., *et al*. 'Heterogeneous verification of an autonomous curiosity rover'. *NASA Formal Methods*. Cham: Springer International Publishing; 2020. pp. 353–60.
[40] Quigley M., Conley K., Gerkey B., *et al*. 'ROS: an open-source robot operating system'. *Workshop on Open Source Software*. Japan: IEEE; 2009.
[41] Leitão P. 'Agent-based distributed manufacturing control: a state-of-the-art survey'. *Engineering Applications of Artificial Intelligence*. 2009;**22**(7):979–91.
[42] Dennis L.A., Farwer B. 'Gwendolen: a BDI language for verifiable agents'. *Logic and the Simulation of Interaction and Reasoning*. AISB; 2008. pp. 16–23.

[43] Rao A.S., Georgeff M. 'BDI agents: From theory to practice'. *International Conference on Multi-Agent Systems*. AAAI; 1995. pp. 312–9.
[44] Visser W., Havelund K., Brat G., *et al*. 'Model checking programs'. *Automated Software Engineering*. 2002;**10**(2):3–11.
[45] Leucker M., Schallhart C. 'A brief account of runtime verification'. *The Journal of Logic and Algebraic Programming*. 2009;**78**(5):293–303.
[46] Cardoso R.C., Dennis L.A., Fisher M., EMAS. 'Plan library reconfigurability in BDI agents'. *International Workshop on Engineering Multi-Agent Systems*; 2019. pp. 1–16.
[47] Grotzinger J.P., Crisp J., Vasavada A.R., *et al*. 'Mars science laboratory mission and science investigation'. *Space Science Reviews*. 2012;**170**(1):5–56.
[48] Trevino R.C., Kosmo J.J., Ross A., Cabrol N. *'First astronaut-rover interaction field test'*. International Conference On Environmental Systems; 2000.
[49] Landis G.A. 'Robots and humans: synergy in planetary exploration'. *AIP conference proceedings. Vol. 654*. American Institute of Physics; 2003. pp. 853–60.
[50] Pedersen L. *'Field demonstration of surface human-robotic exploration activity'*. **9**. AAAI Spring Symposium; 2006.
[51] Medina A., Pradalier C., Paar G., *et al*. 'A servicing rover for planetary outpost assembly'. *Advanced Space Technologies for Robotics and Automation*; 2011.
[52] Ransan M., Atkins E.M. *'A collaborative model for astronaut-rover exploration teams'*. AAAI Spring Symposium; 2006. pp. 52–8.
[53] Heiskanen P., Heikkilä S., Halme A. *'Development of a Dynamic Mobile Robot Simulator for Astronaut Assistance. Workshop on Advanced Space Technologies for Robotics and Automation'Development of a dynamic mobilerobot simulator for astronaut assistance'. Workshop on Advanced SpaceTechnologies for Robotics and Automation*; 2008.
[54] Fong T., Nourbakhsh I. 'Peer-to-peer human-robot interaction for space exploration from interfaces to intelligence'. *AAAI Fall Symposium: The Intersection of Cognitive Science and Robotics*; 2004.
[55] Sierhuis M. Modeling and Simulating Work Practice. BRAHMS: a multiagent modeling and simulation language for work system analysis and design [PhD. Thesis]. Social Science and Informatics (SWI), University of Amsterdam, SIKS Dissertation Series No. 2001-10, Amsterdam, The Netherlands; 2001.
[56] Sierhuis M., Clancey W.J. 'Modeling and simulating practices, a work method for work systems design'. *IEEE Intelligent Systems*. 2002;**17**(5):32–41.
[57] Sierhuis M., Clancey W.J., Alena R.L., *et al*. 'NASA's mobile agents architecture: a multi-agent workflow and communication system for planetary exploration'. International Symposium on Artifical Intelligence, Robotics and Automation in Space; 2005.
[58] Sierhuis M., Clancey W.J., Vanhoof R., *et al*. 'ASA's OCA mirroring system: an application of multiagent systems in mission control'. International Conference on Autonomous Agents and Multi-Agent Systems; 2009.

[59] Clancey W.J., van Hoof R. *'The metabolic rate advisor: Using agents to integrate sensors and legacy software'*. NASA; 2013.
[60] Stocker R., Sierhuis M., Dennis L.A. 'A formal semantics for Brahms'. *International workshop on computational logic in multi-agent systems. Vol. 6814 of lncs*. Springer; 2011. pp. 259–74.
[61] Stocker R., Dennis L.A., Dixon C., *et al*. 'Verification of brahms human–robot teamwork models'. *European Conference on Logics in Artificial Intelligence. vol. 7519 of LNCS*; Springer; 2012. pp. 385–97.
[62] Webster M., Dixon C., Fisher M., *et al*. 'Toward reliable autonomous robotic assistants through formal verification: a case study'. *IEEE Transactions on Human-Machine Systems*. 2016;**46**(2):186–96.
[63] Fisher M. *An introduction to practical formal methods using temporal logic*. Wiley; 2011.
[64] Abdulla P.A., Jonsson B., Nilsson M., *et al*. 'Regular model checking for LTL(MSO)'. *International Conference on Computer Aided Verification. vol. 3114 of LNCS*; Springer; 2004. pp. 348–60.
[65] Abdulla P.A., Jonsson B., Rezine A., *et al*. 'Proving liveness by backwards reachability'. *International Conference on Concurrency Theory. vol. 4137 of LNCS*; Springer; 2006. pp. 95–109.
[66] Delzanno G. 'Automatic verification of parameterized cache coherence protocols'. *International Conference on Computer Aided Verification. vol. 1855 of LNCS*; Springer; 2000. pp. 53–68.
[67] Delzanno G. 'Constraint-based verification of parameterized cache coherence protocols'. *Formal Methods in System Design*. 2003;**23**(3):257–301.
[68] Esparza J., Finkel A., Mayr R. 'On the verification of broadcast protocols'. Symposium on Logic in Computer Science. IEEE Computer Society Press; 1999. pp. 352–9.
[69] Szalas A., Holenderski L. 'Incompleteness of first-order temporal logic with until'. *Theoretical Computer Science*. 1988;**57**(2–3):317–25.
[70] Hodkinson I., Wolter F., Zakharyaschev M. 'Decidable fragments of first-order temporal logics'. *Annals of Pure and Applied Logic*. 2000;**106**(1–3):85–134.
[71] Wolter F., Zakharyaschev M. 'Axiomatizing the monodic fragment of first-order temporal logic'. *Annals of Pure and Applied Logic*. 2002;**118**(1–2):133–45.
[72] Fisher M., Konev B., Lisitsa A. 'Practical infinite-state verification with temporal reasoning'. *Verification of infinite state systems and security. Vol. 1 of NATO security through science series: information and communication*. IOS Press; 2006. pp. 91–100.
[73] Dixon C., Fisher M., Konev B., *et al*. 'Practical first-order temporal reasoning'. *International Symposium on Temporal Representation and Reasoning. IEEE*; 2008. pp. 156–63.
[74] Degtyarev A., Fisher M., Konev B. 'Monodic temporal resolution'. *ACM Transactions on Computational Logic*. 2006;**7**(1):108–50.
[75] Hustadt U., Konev B., Riazanov A., *et al*. 'TeMP: A temporal monodic prover'. *International Joint Conference on Automated Reasoning. vol. 3097 of LNCS*; Springer; 2004. pp. 326–30.

[76] Ludwig M., Hustadt U. 'Fair derivations in monodic temporal reasoning'. *International Conference on Automated Deduction*; Springer; 2009. pp. 261–76.
[77] Ludwig M., Hustadt U. 'Implementing a fair monodic temporal logic prover'. *AI Communications*. 2010;**23**(2–3):69–96.
[78] Fernández-Gago M.C., Hustadt U., Dixon C., *et al*. 'First-order temporal verification in practice'. *Journal of Automated Reasoning*. 2005;**34**(3):295–321.
[79] Dixon C., Fisher M., Wooldridge M. 'Resolution for temporal logics of knowledge'. *Journal of Logic and Computation*. 1998;**8**(3):345–72.
[80] Dixon C. 'Using temporal logics of knowledge for specification and verification—a case study'. *Journal of Applied Logic*. 2006;**4**(1):50–78.
[81] Dixon C., Fisher M., Konev B. 'Is there a future for deductive temporal verification?' *International Symposium on Temporal Representation and Reasoning. IEEE Computer Society Press*; 2006. pp. 11–18.
[82] Halle S. 'Causality in message-based contract violations: A temporal logic "Whodunit"'. *International Enterprise Distributed Object Computing Conference*. IEEE; 2011. pp. 171–80.
[83] Benghabrit W., Grall H., Royer J.C. 'Checking accountability with a prover'. *Computer Software and Applications Conference*. **2**. IEEE; 2015. pp. 83–8.
[84] Benghabrit W., Grall H., Royer J.C. 'Abstract accountability language: Translation, compliance and application'. *Asia-Pacific Software Engineering Conference. IEEE*; 2015. pp. 214–21.
[85] Behdenna A., Dixon C., Fisher M. 'Deductive verification of simple foraging robotic behaviours'. *International Journal of Intelligent Computing and Cybernetics*. 2009;**2**(4):604–43.
[86] Konev B., Degtyarev A., Dixon C., Fisher M., Hustadt U. 'Mechanising first-order temporal resolution'. *Information and Computation*. 2005;**199**(1–2):55–86.
[87] Wolter F., Zakharyaschev M. 'Decidable fragments of first-order modal logics'. *Journal of Symbolic Logic*. 2001;**66**(3):1415–38.
[88] Gray J. 'Notes on database operating systems'. *Operating systems: an advanced course*. Springer; 1978. pp. 393–481.
[89] Skeen D. 'Nonblocking commit protocols'. *SIGMOD International Conference on Management of Data. ACM Press*; 1981. pp. 133–42.
[90] Chkliaev D., van der Stok P., Hooman J. 'Mechanical verification of a non-blocking atomic commitment protocol'. *Workshop on Distributed System Validation and Verification*. IEEE; 2000. pp. 96–103.
[91] Chkliaev D., Hooman J., van der Stok P. 'Mechanical verification of transaction processing systems'. *International Conference on Formal Engineering Methods. IEEE*; 2000. pp. 89–97.
[92] Lynch N.A. *Distributed algorithms*. Elsevier; 1996.