



Deposited via The University of York.

White Rose Research Online URL for this paper:

<https://eprints.whiterose.ac.uk/id/eprint/239116/>

Version: Accepted Version

Proceedings Paper:

Ribeiro, Pedro, Almeida, Dalay, E. R. Bezerra, Paulo et al. (Accepted: 2026) Engineering safe robotics software from simulation models via RoboSim. In: The 7th International School on Engineering Trustworthy Software Systems (SETSS 2025). 7th International School on Engineering Trustworthy Software Systems, 17-23 May 2025 Lecture notes in computer science. Springer, CHN. (In Press)

Reuse

This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here:

<https://creativecommons.org/licenses/>

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

Engineering safe robotics software from simulation models via RoboSim

Pedro Ribeiro¹, Dalay Almeida², Paulo E. R. Bezerra³,
Ana Cavalcanti¹, Thierry Lecomte², and Marcel V. M. Oliveira³

¹ Department of Computer Science, University of York, UK
pedro.ribeiro@york.ac.uk, ana.cavalcanti@york.ac.uk

² CLEARSY, Aix-en-Provence, France

thierry.lecomte@clearsy.com, dalay.almeida@clearsy.com

³ DIMAp - Universidade Federal do Rio Grande do Norte, Natal - RN, Brazil
paulo.rolim.074@ufrn.edu.br, marcel@dimap.ufrn.br

Abstract. Robots have the potential to play important roles in supporting human activities. Simulations can be used to evaluate the design of a robotic system, but are often coded by hand and without a clear architecture in mind. This raises concerns in terms of development cost and of safety, especially if there is reuse of simulation code for deployment. For safety-critical systems, it is common to use safety functions for hazard mitigation. They are key for safety assurance, and need to be designed and verified using rigorous engineering practices. In this tutorial, we show how we can leverage a model-based approach in the development of robotics software. For modelling, we use RoboSim, a domain-specific notation for describing the cyclic design of a robot’s control software; for deployment, we use the RoboSim model to generate code for a commercial embedded safety platform certified to a high Safety Integrity Level (SIL). To illustrate our approach, we apply it to the design of a firefighting UAV and its safety function. With RoboSim and our code-generation technique, costs are reduced by automation. The use of mathematical semantics to reason about the RoboSim models and the code generation provides the required development rigour.

Keywords: Software engineering · Formal methods · Robotics · Safety.

1 Introduction

Trustworthiness needs to be established before modern mobile and autonomous robots can realise their potential to address key societal challenges by collaborating with humans in sectors such as healthcare, agriculture, energy, manufacturing, and many others [32]. Current practice of Software Engineering for Robotics is code centric, with the early stages of development already involving code development for specific simulation or robotic platforms. Even the state of the art on testing is based on ad hoc expensive practices [9, 33]. This approach does not lend itself to provision of evidence that can support arguments of trustworthiness. The RoboStar technology is being co-developed by academics

and industrialists to support roboticists in adopting a cost-effective, usable, and rigorous model-driven approach to software design and verification [14].

RoboStar distinguishes itself in several respects. (1) The RoboStar framework uses diagrammatic and controlled natural languages accessible to roboticists. (2) These languages provide comprehensive support for modelling, design, and verification of many aspects of a robotic system that have an impact on its software: (a) its concurrent design, including artificial-intelligence components; (b) assumptions about the environment (floor, weather conditions, configuration of obstacles, and so on); (c) the physical structure and behaviour of the robotic platform; (d) assumptions about human behaviour; and (e) properties of interest. (3) Models and documents written using RoboStar notations connect well, both syntactically and semantically, to define an overall vision of the system. (4) The various models and documents have a mathematical semantics that can be used for rigorous automated validation and verification of properties. Roboticists are not required to deal with the mathematical models themselves. (5) Development techniques for code and test generation also benefit from the rigour of the mathematical underpinnings. Code can be used for simulations, co-simulations in digital twins, or deployment. The approach provides traceability between the models used in verification, the code used in simulation, and the deployed code.

Coding and hardware implementation must follow safety-oriented practices. Software bugs, arithmetic overflow, or hardware-resource conflicts can introduce safety hazards if not caught. Using the approach presented here can reduce errors. A mistake in an implementation (such as an incorrect sensor calibration constant or a timing bug in the motor control loop) can cause loss of control. Many UAV accidents, for example, trace back to software glitches, so rigorous verification is needed to ensure the implementation meets the safety requirements.

In this tutorial, we focus on one of the RoboStar notations, namely, the RoboSim notation to specify cyclic software designs [17]. Models written using this notation embed the paradigm used in simulations and implementations that adopt a cyclic-executive architecture. We cover use of RoboSim models for verification, and for sound generation of safety-critical code [25] that preserves the properties of the RoboSim models. Figure 1 indicates how RoboSim and its associated design and verification techniques presented here can be used to support development and deployment of safety-critical robotic systems. To illustrate this workflow, we use a firefighting UAV (Unmanned Aerial Vehicle) inspired by a challenge from an international event⁴, where robots compete to extinguish fires.

RoboSim software models define behaviour using a collection of parallel state machines. These are similar to those available in UML or SysML for example, but have some important additional features. We highlight the availability of constructs for definition of time budgets and deadlines. In addition, RoboSim has a process algebraic mathematical semantics that can be used for verification. Tool support for modelling, validation, and verification of RoboSim models is

⁴ youtube.com/watch?v=S0Ok2cnselU

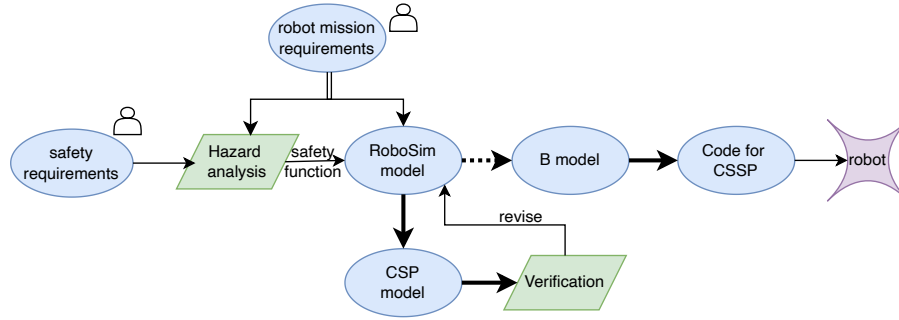


Fig. 1. Workflow for deployment of safety functions in robotic systems. The thin lines indicate processes that require input from a user: development or revision of models, and integration of code in a robot’s API. The thick lines indicate automatic processes. The thick broken line indicates automatic processes under development.

provided by RoboTool⁵. Mathematical models are automatically generated by RoboTool, which also provides support for generation of verification reports.

A UAV must be engineered with safety as a paramount concern: it should not pose unacceptable risks of harm to people, property, or the environment during its operation. So our example is ideal to illustrate our approach to safety, as a property of the entire robotic system (software, robotic platform, and human interactions), which must be built into every stage of development.

In our work, the design is informed by a hazard analysis that leads to the identification of safety functions that must be incorporated in the robotic system. In this tutorial, we give an overview of safety standards that are relevant in the development of an UAV. We also describe a hazard analysis for UAV safety that is needed to demonstrate compliance with standards. A main outcome are design decisions related to the safety requirements. Our example illustrates how we can deal with some of these requirements using a safety function.

In our approach, each safety function is modelled by a RoboSim state machine subjected to rigorous verification. In our example, the safety function ensures that the drone mission is aborted if its communication with the base is lost.

Verification is not enough, however, if we cannot ensure that the behaviour of the model is preserved in the code execution. So, we rely on automatic code generation from the RoboSim model, for execution on the CLEARSY Safety Platform (CSSP) [25]. This platform and its associated toolchain are certified according to EN 50128 and EN 50129, enabling the development of systems that comply with the highest Safety Integrity Levels (SIL 3 and SIL 4), thus facilitating certification. For our example, the CSSP needs to be embedded in the drone to execute the automatically generated code for the safety function.

Ongoing work supports automatic generation of Rust or C code from RoboSim models. This effort, however, does not generate code suitable for the

⁵ Available for download from robostar.cs.york.ac.uk/robotool/

CSSP. Here, we present our ongoing work to generate concrete B models [8] from RoboSim, which is basically the programming language adopted by the CSSP. With the B models we generate, code for the CSSP can be automatically obtained via dedicated tools. So, in the envisaged approach, the link from RoboSim to the CSSP is fully automated, with B used as an intermediary language.

With our approach, we ensure that not only the interactions (inputs and outputs) of control software modelled in RoboSim are preserved in the CSSP code, but also their timing. This means that the scheduling of interactions in RoboSim models into cycles is respected by the CSSP code. This is done, however, under the assumption that all interactions and data operations scheduled by a cycle can in fact finish within that cycle. Discharging this assumption requires worst-case execution time analysis, which is a complementary task not covered here.

In the next section, we briefly present the firefighting UAV. Our approach to hazard analysis for UAVs is described in Section 3. In Section 4, we focus on the description of the UAV’s software, its model in RoboSim and its formal verification. The RoboSim notation for description of software designs, including its tool, are covered in that section. The CSSP and our approach to programming it are presented in Section 5. We conclude in Section 6.

2 A Firefighting UAV

In this section, we present our autonomous firefighting system that employs a UAV, or drone. As part of its mission, the UAV is given a set of predefined GPS coordinates around a building that it needs to patrol. Image processing from thermal and depth cameras is used to identify fires and align the drone with the fire source to deploy fire suppressant. The UAV is shown in Figure 2 during operation. The hardware incorporates a DJI Matrice 600 Pro UAV, with

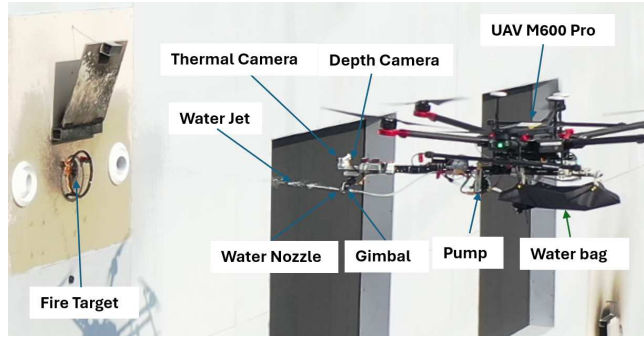


Fig. 2. Firefighting UAV in operation

a custom payload attached below that includes a water bag and pump, and the cameras and a water nozzle located at the end of a carbon fibre pole. The nozzle

is mounted on a gimbal with two Degrees of Freedom (DoF) to allow for minor corrections to orientation when spraying the fire suppressant.

The UAV has three computing units onboard: two computers and a microcontroller. The main computer is connected to the sensors and the UAV’s autopilot, and to a microcontroller that controls the gimbal’s servos. The safety computer is connected to the main computer and executes a safety function.

In what follows, we discuss the design of the safety function based on hazard analysis. Modelling and verification of the UAV using RoboSim [17], including the safety function, is then discussed in Section 4.

3 Safety function and hazard analysis

This section provides an engineering-focused account of the management of safety in autonomous drones. We first define safety in the context of a UAV and identify potential safety issues across the development lifecycle (Section 3.1). We then introduce a hazard analysis method (including the hazard triangle of hazardous element, initiating mechanism, and target) and show how breaking down hazards leads to concrete safety requirements (Section 3.2). Finally, we describe key safety functions that an autonomous UAV (illustrated by our firefighting drone) must implement, such as handling communication loss, monitoring energy, and executing fallback logic, to mitigate hazards (Sections 3.1 and 3.4).

3.1 Safety of Autonomous UAVs

In the context of UAVs, safety refers to the state of being free from unacceptable risk of accidents. Several standards are available that prescribe or recommend how safety needs to be handled. In the aerospace domain, DO-178C [3] defines objectives and processes for airborne software, while DO-254 [2] addresses hardware development. Safety analysis techniques are structured by ARP4761 [1], providing widely adopted methods such as Functional Hazard Assessment (FHA) and Fault Tree Analysis (FTA) to derive safety requirements.

When dealing with unmanned aircraft systems (UAS), additional standards come into play. ASTM F3266-20 [7] and ASTM F3178-16 [5] propose guidelines for system design and loss of control mitigation, respectively. For operational risk evaluation, the Specific Operations Risk Assessment (SORA) [6] methodology, adopted by the European Union Aviation Safety Agency (EASA), defines procedures to assess and mitigate risks in drone operations. Furthermore, DO-326A [4] introduces cybersecurity considerations into the safety framework, reflecting the growing concern with digital threats in avionics.

Aviation regulations explicitly aim to ensure that the risks associated with aircraft operations (including drones) are managed to tolerable levels. This means an autonomous drone should operate without causing injury, collision, or uncontrolled damage even if faults occur. Importantly, safety is not just a feature of one component but a system-wide objective: it emerges from robust design,

correct functioning, and effective risk controls across the UAV’s entire architecture. Modern drones face diverse failure modes – battery failures, mechanical malfunctions, communication dropouts, navigation errors, and environmental hazards are among the leading causes of incidents. The fundamental challenge is to develop comprehensive safety systems that predict, prevent, and respond to such failures while still achieving the mission. Achieving this requires a “safety by design” approach throughout the system development lifecycle.

Safety must be addressed from the earliest requirements to final integration. Each phase of the development lifecycle presents unique challenges that could introduce or lead to the overlooking of hazards.

Requirements Phase At this initial stage, all safety-critical scenarios should be identified and translated into clear requirements. Missing or ambiguous safety requirements pose a risk of leaving some hazards unmitigated. For example, if the requirement to handle a lost communication link is omitted, the UAV may have no defined behaviour on link loss – a serious oversight. Engineering standards (e.g. ARP4754A in aviation) emphasise the need to derive safety goals early so that hazards are considered in system objectives. Incomplete hazard identification or undefined safety constraints can lead to accidents later when the system encounters an unhandled condition.

Design and Modelling Phase During system architecture definition and modelling, safety considerations guide decisions like adding redundancy, fail-safes, and protection mechanisms. A potential issue is designing the control algorithms or system model without considering failure modes or extreme conditions. For instance, if the flight-control design assumes that all sensors always provide valid data, it may become unstable when a sensor malfunctions. Using formal modelling or simulation at this stage can help validate safety properties. An inadequate design (e.g. lacking a backup for a critical sensor or not considering a high-wind scenario) might pass nominal tests but later manifest hazardous behaviour when those unconsidered situations occur.

By addressing safety at each stage, the UAV development follows a safety lifecycle similar to those in the aerospace and automotive domains (such as those mentioned in DO-178C for software, and in ISO 26262 for the automotive sector). The end result should be a drone system whose design and verification have considered and mitigated foreseeable hazards from the ground up, greatly reducing the chances of accidents in operation and producing evidence of safety.

3.2 Hazard Analysis Method for UAV Safety

Once safety goals are established, engineers perform a hazard analysis to discover what could go wrong and how to prevent it. A hazard is typically defined as a potential condition or event that could lead to an accident (mishap) if not controlled. In system safety engineering (e.g. MIL-STD-882), a hazard is seen as the precursor to an accident, and the analysis focuses on identifying hazards,

their causes, and their effects. A key concept is that a hazard can be broken down into three components – often visualized as a hazard triangle:

- **Hazardous element:** an inherent source of potential harm – an energy source, moving object, or dangerous substance. This is the agent that can do damage if uncontrolled. For example, in a drone, a spinning rotor or a high-density battery are examples of hazardous elements (capable of cutting through obstacles, including humans, or igniting fires).
- **Initiating mechanism:** The trigger or failure that unleashes the hazard. An initiating mechanism is some event or condition that causes a hazardous element to actually pose a threat. In a UAV, an initiating mechanism could be a software error that causes loss of control, a component failure (motor seizing up), or an external factor, like extreme wind.
- **Target or threat:** The person, property, or environment that could be harmed. This defines what is at risk and often determines the severity of the hazard’s outcome. For drones, targets might include people on the ground, other aircraft in the air, or sensitive infrastructure. For instance, if a UAV carrying a heavy payload flies over a crowd, the people below are the targets that could be harmed if the drone falls due to an initiating mechanism.

With hazards identified and defined in this structured way, we can systematically analyse risk. Engineers assess each hazard’s severity (how bad the consequences would be if it occurs) and likelihood (how probable it is). This helps to identify and prioritise the hazards that need the most stringent controls.

Common techniques used at this stage include Preliminary Hazard Analysis (PHA) [19], Failure Mode and Effects Analysis (FMEA) [13, 24], fault tree analysis [28, 35], and newer methods like STPA (Systems-Theoretic Process Analysis) [30]. In particular, STPA (a technique based on system control theory) has been applied to UAVs in research; one study [34] found that applying STPA to a small drone system yielded 70 distinct safety requirements addressing hazards across the drone’s operation and stakeholders. This underscores how thorough hazard analysis translates into a large set of necessary safety-related constraints and mechanisms.

A crucial outcome of a hazard analysis is a set of concrete safety requirements or design constraints that can eliminate, mitigate, or control each hazard. Essentially, for every identified hazard, we ask: “what do we need the system to do (or not to do) to prevent this hazard from leading to an accident?”. By decomposing hazards into their causes, we can target requirements at breaking the chain between cause and harm. We provide some examples.

Example 1. A loss of communication link is a hazard. The *hazardous element* is the UAV with running propellers (kinetic energy); *initiating mechanisms* are loss of command and control link (e.g., due to interference or out-of-range scenarios); *targets* are people or property on the ground (if the UAV falls or flies away). This leads to the following safety requirements: the UAV must detect communication loss and immediately transition to a safe state. Common requirements are an automatic Return-To-Base (RTB) or hover-then-land failsafe after a defined

timeout of link loss. This ensures the drone does not just drop or fly randomly if it loses contact. In firefighting operations, for instance, if a drone loses its radio link amid a wildfire, it might autonomously climb to a safe altitude and return to a predefined safe landing point. Such a behaviour prevents the worst-case outcome of falling onto firefighters or bystanders – a risk highlighted by agencies warning that an out-of-control UAV “that loses its communication link could fall from the sky, causing serious injuries or deaths” [20].

Example 2. Battery depletion or failure mid-flight, causing a crash, is also a hazard. The *hazardous element* is the high-energy battery (and the UAV’s potential energy at altitude); *initiating mechanisms* are battery running empty or a sudden power failure; *targets* are once again people and property on the ground, but also the environment, if the battery catches fire. A potential safety requirement is that the UAV must continuously monitor the battery-energy levels and health. This includes real-time battery diagnostics and an algorithm to anticipate power failure. A low-battery warning threshold should trigger an automatic RTB or landing while there is still sufficient power for a controlled descent. Some advanced drones even carry redundant batteries or cells so that, if one fails, another can temporarily take over the provision of power. In a firefighting drone, which may spend battery faster when powering thermal cameras or spotlight payloads, energy management is critical – the UAV should plan its mission to return before battery is critically low, or find a safe landing zone if power is about to run out. This requirement addresses the initiating mechanism (battery exhaustion) by ensuring a graceful degradation (safe landing rather than fall).

Example 3. Another hazard is the possibility that the UAV enters a restricted or unsafe zone, so that it could collide with other aircraft or endanger people. The *hazardous element* is the UAV in flight; *initiating mechanisms* are GPS navigation error or loss of control; *targets* are other aircraft or people in the restricted zone. This leads to the following safety requirements: implement geo-fencing and no-fly zone enforcement so the drone cannot stray outside its allowed area. The drone should autonomously correct course or initiate safe return if it approaches a geo-fence boundary. This requirement directly addresses an initiating mechanism (navigation error) by adding a containment function.

Through these examples, we see how hazard analysis “decomposes” each potential mishap into scenarios and triggers, and then informs the elicitation of specific safety requirements that become part of the UAV’s design. With this approach, every safety requirement can be traced back to one or more hazards it mitigates – this traceability is often required in certification processes. In summary, hazard analysis provides a blueprint of what could go wrong and what the system must do to prevent or control it. The next step is to implement these requirements as functional safety mechanisms in the UAV.

3.3 Key Safety Functions in Autonomous Firefighting Drone

Building on the hazard analysis described and exemplified above, autonomous drones include several safety functions – special operational features and proto-

cols that maintain safety even when something goes awry. We describe key safety functions that must be implemented, using an example of a firefighting UAV. A firefighting drone often operates in high-risk, dynamic environments: near intense heat and smoke, possibly beyond the operator’s line of sight, alongside manned aircraft like water-drop helicopters, and over firefighters on the ground. This scenario puts a premium on robust safety functions. The primary safety functions include Communication Loss Handling, Energy Monitoring and Management, and Fallback or Emergency Logic described in detail next.

Communication Loss Handling For any UAV, especially one coordinating with a firefighting team, losing the communications link with ground control is a critical hazard. The drone must not spiral into chaos if it goes out of radio range or experiences interference. Communication loss handling refers to the built-in behaviour the UAV will follow upon detecting a loss of command or control signal. The safety requirement derived from this hazard is typically implemented as a failsafe mode like RTB or Emergency Landing.

In practice, drones continuously monitor the health of their control links. For example, a professional UAV might be programmed such that if it has not received any command or heartbeat from the ground for, say, 60 seconds, it will automatically initiate an RTB. During RTB, the drone might ascend to a safe altitude (to avoid obstacles), then navigate back to a predefined home location and land. This behaviour is designed to protect both the drone and people on the ground when the human operator is no longer in control.

In a wildfire scenario, communication signals can be lost due to terrain, smoke, or radio interference from firefighting equipment. A firefighting drone should have a pre-agreed failsafe: for instance, if communication is lost, it could hover in place for a short time (in case the link is momentarily dropped), then proceed to either return to a base at the edge of the fire zone or perform a vertical descent in a safe area if return is not possible. This prevents the drone from drifting unpredictably over active firefighting zones.

The importance of such a function is underscored by the U.S. Forest Service’s warning that a drone with a lost link could literally fall out of the sky and endanger lives. Thus, lost-link logic is mandatory. Typically, regulatory guidelines also require operators to have predefined “lost link procedures” for UAVs. The UAV’s autopilot should implement this autonomously, since a disconnected operator cannot command it. Modern UAV autopilots (e.g., ArduPilot, DJI systems) indeed have configurable failsafe responses for loss of RC signal or telemetry, ranging from RTB to hovering or landing after a timeout.

Moreover, communication loss handling is not limited to total link loss. It also covers high latency or degraded communications. An autonomous drone might start by downgrading its dependency on the link if signal quality drops – for instance, reducing speed and holding position more frequently as a precaution. Ultimately, if the link quality falls below a critical threshold for a sustained period, the failsafe triggers fully. All these measures ensure the UAV does not become a unguided missile if it loses its “eyes and ears” to the ground control.

Energy Monitoring and Safe Energy Management Power failure is a leading cause of drone crashes, making energy monitoring and management a cornerstone safety function. An autonomous UAV must actively supervise its battery state and make intelligent decisions to avoid sudden power loss. For a firefighting drone, this is vital – such drones may carry heavy payloads (water or fire retardant, sensors) and operate in high-throttle situations (battling updrafts from fires), which can drastically shorten flight time. Additionally, high ambient temperatures near fires can affect battery performance.

Battery monitoring systems are implemented to continuously track key metrics: remaining charge percentage, battery voltage, current draw, temperature of battery packs, and even the health of individual cells. The drone’s onboard computer uses these to estimate remaining flight time. A critical safety feature is setting thresholds that will trigger automated actions. A Low-Battery RTB can be configured: for example, if the battery level falls below X% (for some value of X enough to return home with some margin), the drone will alert the operator and initiate an RTB if no action is taken. If the battery reaches an even more critical level (where continued flight is impossible), the drone might perform an immediate autoland – essentially descending vertically to land as soon as possible to avoid a high-altitude power loss.

Another aspect is energy management in mission planning. The UAV’s flight control software might actively limit its mission if it detects higher than expected power consumption. For example, if battling wind gusts is draining battery faster, the drone could shorten its waypoint route or jettison non-critical payload to conserve energy. For example, a firefighting drone might drop remaining water early if needed. While this is an extreme measure, it could be considered if keeping a heavy payload would cause a crash – though it would have its own safety requirements: ensuring the drop occurs only over safe areas.

In summary, energy monitoring ensures the UAV is never blindsided by an empty battery. Safe-energy management functions guarantee that the drone has enough reserve to execute a safe RTB or landing before power is exhausted. These functions directly mitigate the hazard created by a power failure by actively managing the initiating mechanism (battery depletion).

Collision Avoidance and Airspace Safety While not explicitly considered in our case study, in which, for the sake of simplicity, we assume that the route to and around the building on fire is free of obstacles, collision avoidance is unquestionably a key safety function for UAVs, particularly relevant to firefighting operations in more general scenarios. Firefighting drones share airspace with water-bomber planes, helicopters, and other drones, and they operate in environments with trees, poles, and uneven terrain. A mid-air collision or a collision with obstacles can be catastrophic. Hence, UAVs must have Detect-And-Avoid (DAA) capabilities, or at minimum geospatial awareness to avoid known obstacles.

Regulators require that UAVs not pose collision risks to other aircraft – for example, the FAA mandates that drone operators have a means to avoid crewed aircraft, either via human observers or an onboard DAA system. For a

UAV, especially if operating beyond visual line of sight, this implies onboard sensors (like radar, LiDAR, optical cameras) and algorithms to detect other aircraft or obstacles and manoeuvre away in time. In our firefighting scenario, the drone might be equipped with a thermal camera that can also detect the heat signature of other aircraft engines or a radar to detect helicopters through smoke. The collision avoidance function would autonomously alter the drone's path if another aircraft comes within a dangerous range, or if the drone is on a collision course with terrain (e.g., ascending terrain or a building).

Geofencing, as mentioned earlier, is another aspect: the drone can have predefined “no-fly zones” (such as areas above where firefighters are actively working, or around an incident commander's helicopter), and the drone will actively prevent itself from entering those zones. If the mission requires it to cross such an area, only explicit override with proper authorization would allow it (and typically, in real deployments, drones are kept away from where aerial firefighting with manned aircraft is happening at the same moment, to avoid conflicts).

Finally, post-mission and continuous improvement are worth noting. A safety-oriented design does not stop at deployment: data from each flight (especially data related to any incident or near-miss) should feed back into refining the hazard analysis and safety requirements. For example, if a firefighting drone had an unexpected near-collision with a power line that was not in its map, the team would investigate and perhaps update the obstacle database or add a new sensor. This is part of the safety management process – a loop of analysis, requirements, implementation, and operation feedback. It ensures the UAV's safety functions remain effective as the environment or use cases evolve.

3.4 Safety function for our firefighting drone

From this analysis, we designed the safety function for our example, which directly addresses several of the identified hazards. We have also added an operational exported constraint: the drone cannot be operated from a moving base because in that case the Return-To-Base would fail.

Our safety function performs two key safety checks:

1. Verifying that a communication link is maintained during the whole mission. This communication link, from ground base to drone only, is used to interrupt the mission if decided by the operator or if some on-board conditions are not met. Recovering the communication link re-enables the mission.
2. Checking the battery charge. Insufficient charge requires recharging the battery. This is the only way to cancel the “low battery” alarm.

If the safety check fails, the flight software goes to a mode where an RTB is mandatory. The safety function takes three inputs (see Figure 3):

1. **Communication link** is the result of transforming an analogue radio signal received by the drone into a digital signal (bit stream). The frequency of the signal and bit alternation is constant.

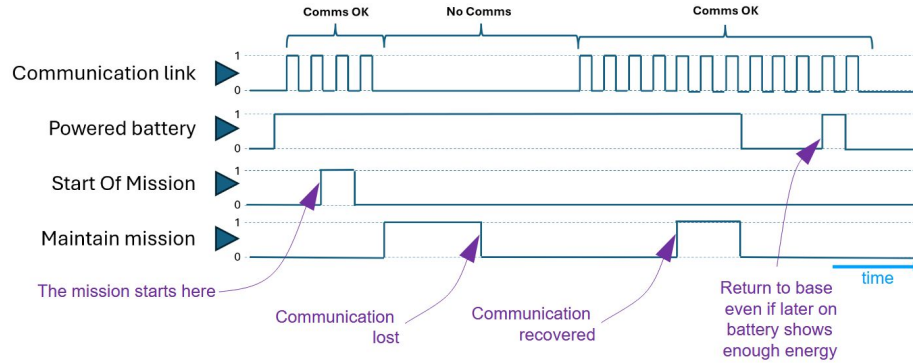


Fig. 3. An example of scenario

2. **Powered battery** represents the capability of the drone to return to its base, as it is supposed to start its mission with full charge. The data required for the low battery alarm is usually complex (real value fluctuating over time). For our case study, the Boolean input signal represents the fact that the output voltage is greater than a threshold.
3. **Start of Mission** represents the first moment when the safety check must be done. This event is characterised by the first rising edge of this input.

Using these inputs, the safety function calculates one output: **maintain mission** determines the ability of the UAV to continue the mission.

The transmission pattern of the communication link must be determined. When the communication link is down, the mission is not maintained until either the communication link is reestablished, or the drone reaches base and is reset and restarted. If the battery power is lower than the threshold, then the low battery alarm is raised. Once a low battery alarm is raised, the RTB is forced until the drone returns to base and is reset, reenergized, and restarted.

The Maintain Mission output has a restrictive position (low value, 0 or FALSE) and a permissive position (high value, 1 or TRUE). The restrictive position ("return-to-base") should correspond to "absence of power" of the computer calculating it, to be able to handle such failure. The permissive position corresponds to an energized, live computer where the communication link is established and the battery has enough charge to continue the mission.

By formalising these safety functions, the UAV's flight software can contain and isolate hazardous situations, ensuring that the UAV operates safely even under unexpected conditions. Our work demonstrates a systematic approach for translating hazard analysis into actionable control strategies, reinforcing mission reliability and protecting both the UAV and the environment. Next, we present how properties of the RoboSim model for our safety function can be established.

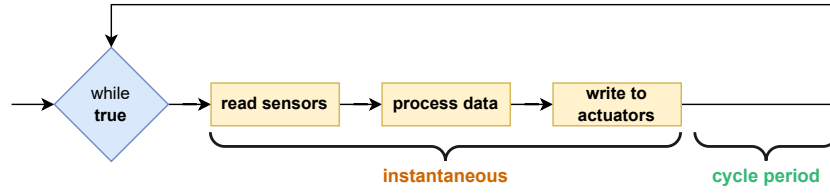


Fig. 4. Idealised simulation cycle control flow

4 Modelling and verification using RoboSim and RoboTool

RoboSim is concerned with the last level of the design of a control software for a robotic system. It adopts a component model based on notions of modules, controllers, state machines, and operations that provide guidance on the development of models. RoboSim adopts an idealised cyclic simulation paradigm, whose abstract control flow is depicted in Figure 4. It can be described by an infinite loop whose iterations define the behaviour of the simulation cycles. In each simulation cycle, inputs are read from registers of sensors, data computations are performed, and outputs are written to actuators, all instantaneously with respect to simulation time. Afterwards, time is incremented by a fixed period, defined by a positive natural number, and a new cycle begins.

The inputs and outputs are abstractions for services of the robotic platform. In this sense, the sensors and actuators can correspond to low-level devices (infrared sensors and servo-motors, for example) or to functions of a sophisticated API. These can provide vision algorithms to identify obstacles, and control systems to move a robot based on a GPS, for example. It is up to the modeller to define the appropriate level of abstraction of the services, based on the needs for verification (that is, on the reliability of available code) and for code generation.

Although RoboSim embeds the idealised simulation paradigm depicted in Figure 4, a RoboSim model is also an appropriate account of a cyclic implementation. Obviously, inputs, processing, and outputs cannot take place infinitely fast in a real implementation. This is not an issue, however, if all that can happen within the cycle. This time analysis, based on the computational resource of a deployment, is a usual requirement for a real-time control system.

In what follows, we present the main components of a RoboSim model. To follow the explanations, it is recommended to install RoboTool and create a RoboSim project as described in a complementary step-by-step tutorial⁶.

4.1 Modules and platform

The main structuring element of a RoboSim model is the module. It defines the control software for a single robot, and contains a single robotic platform: a block

⁶ robostar.cs.york.ac.uk/publications/techreports/reports/robosim-tutorial.pdf

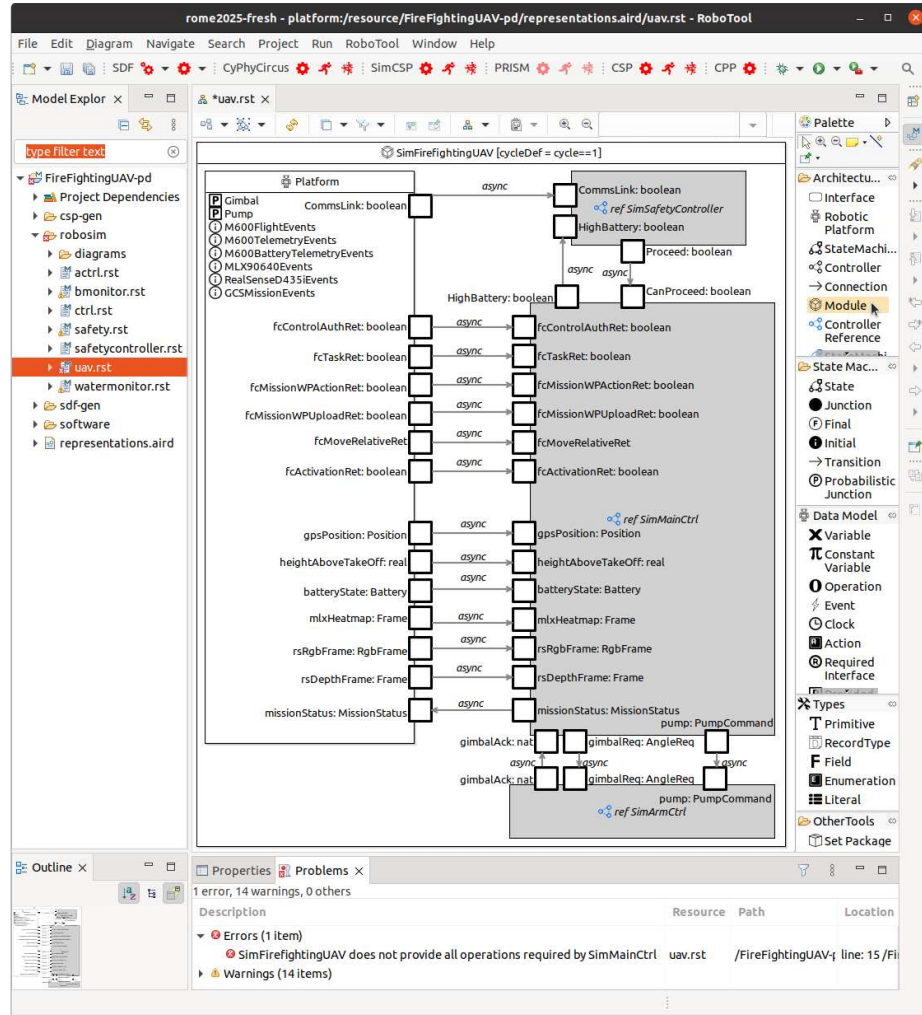


Fig. 5. RoboTool showing three panes: on the left the Model Explorer, showing the project files tree, then the RoboSim graphical editor on the right, and the Problems view at the bottom. An error is shown in the Problems list, indicating that the module `SimFirefightingUAV` does not provide all required operations required by the controller `SimMainCtrl`. To the right of the diagram, there is a palette with the tools available for editing and the mouse cursor hovering the `Module` component.

describing the services that can be used by the software. In a module, behaviour is specified by one or more controllers that are connected to the robotic platform.

In Figure 5 we reproduce a screenshot of RoboTool showing the RoboSim module (indicated by the symbol ) named `SimFirefightingUAV` for the firefighting UAV. This and other screenshots give an idea of the modelling environment

offered by RoboTool. The diagrammatic editor has a palette, shown on the right of the figure, with clickable tools for adding new components to a diagram. For example, to create a module, we select the **Module** tool in the **Architectural Constructs** section of the palette, and click in the graphical area at the centre of the editor: the user is then asked to define the simulation cycle period of the module, as reproduced in Figure 6, before it appears as a block in the editing area. Here, the expression `cycle==1` to the right of the equals sign (`=`) defines that 1 is the only value allowed for the cycle period. For a RoboSim model to be well-formed, the conjunction of the expressions defining the cycle period of every component, including the module, its controllers, and state machines, cannot be false.

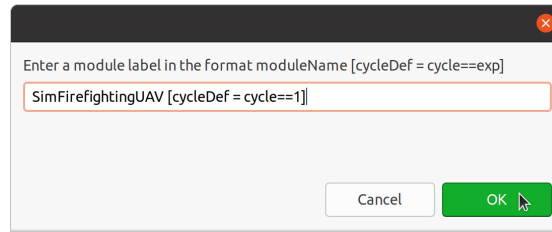


Fig. 6. Setting the cycle period for the module

As we construct the model, guidance is provided interactively. The problems view, shown at the bottom in Figure 5, identifies any warnings or errors related to the model. In this case, the error message shown indicates that the module `SimFirefightingUAV` does not provide all required operations by one of the controllers. It can be fixed by recording that the robotic platform provides the interface named `M600Flight`, reproduced in Figure 7, that defines the signature of operations (**O**) corresponding to services of the M600’s API. To add a provided interface to fix this problem, using the diagram editor, we can click on the option **P** Provided Interface tool available under the Data Model section of the Palette and then click on the robotic Platform. A dialog will appear, where the name `M600Flight` should be typed, and then we should click Ok. Afterwards, the model should be saved to ensure it is parsed and validated.

More generally, the services available to the software are captured in a robotic platform and described by events (**E**), operations, and variables (**X**) that can be shared with the software. These correspond to abstractions for sensors and actuators. Any physical platform implementing these services can use the software defined by the module. Thus, a robotic platform records assumptions about the robot and its embedded software. Constants (**π**) may also be declared in components, which become parameters of the model if their value is not defined.

For example, for the Platform in `SimFirefightingUAV`, the provision (indicated by the symbol **P**) of interface `M600Flight` includes the operation `fcDroneTaskControl` with a parameter `task` of type `M600Task`, an enumerated type that can be used to represent instructions for the autopilot to `GoHome`, `Land` or `TakeOff`. We

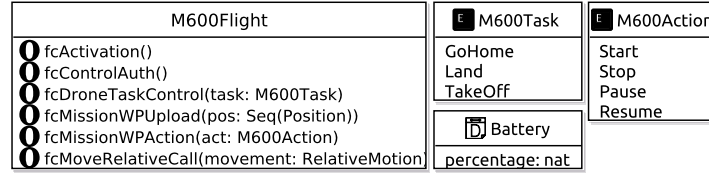


Fig. 7. M600Flight interface with the signature of operations for the M600 API and related data types captured in RoboSim

observe that the robotic platform can be augmented with further events to explicitly account for other capabilities, such as DAA. For our use case, we assume that the route to and around the building is free of obstacles.

Variables, events, and operations can be declared directly in a robotic platform block (⚙️), or indirectly by provided (📦) or defined interfaces (Ⓢ). The notion of interface here is different from that in UML. Interfaces group variables, operations, and clocks (⌚), or events. Interfaces can be provided, required, or defined by robotic platforms, controllers, state machines, or operations defined in the model. In the case of a robotic platform, interfaces can be just provided or defined, because a platform is a service provider: it cannot require any services.

Defined interfaces cannot contain operations. This is because no component can or needs to declare an operation locally. First, modules do not have local declarations. The data model of a module is given by its robotic platform. Second, in a robotic platform, operations are either provided, as opposed to defined, by the platform, and not further defined in the module. Thirdly, in a controller, operations are either required or defined, not just declared, locally.

4.2 Controllers and state machines

The module `SimFirefightingUAV`, reproduced in Figure 5, contains references to three controllers, accounting for the computing units onboard the UAV. Controller `SimMainCtrl` is connected directly with the Platform and two other controllers, `SimArmCtrl`, that controls the gimbal and pump, and `SimSafetyController`, that contains the safety function. The definition of `SimSafetyController`, reproduced in Figure 8, uses two interfaces `SMonitorInput` and `SMonitorOutput`, that define the input events `HighBattery`, `MissionStart`, and `CommsLink`, and the only output event `Proceed`. Apart from `MissionStart`, that is assumed to be received just once, all other events are of type `boolean`. For example, the occurrence of the input event `CommsLink` corresponds to receiving one bit value from the communication link bit stream, while its absence represents a loss of signal.

In what follows, we describe the safety machine defined as `SimBCMonitor` and referenced in the controller `SimSafetyController`.

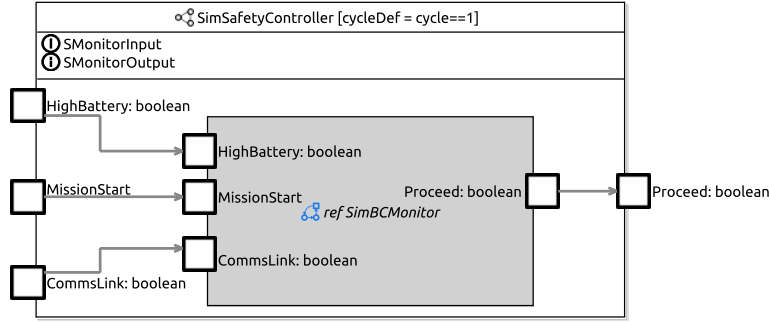


Fig. 8. RoboSim controller `SimSafetyController`, showing three events `HighBattery`, `MissionStart` and `CommsLink` connected to the statemachine `SimBCMonitor` as inputs, and a single event `Proceed` connected as an output.

4.3 Safety machine

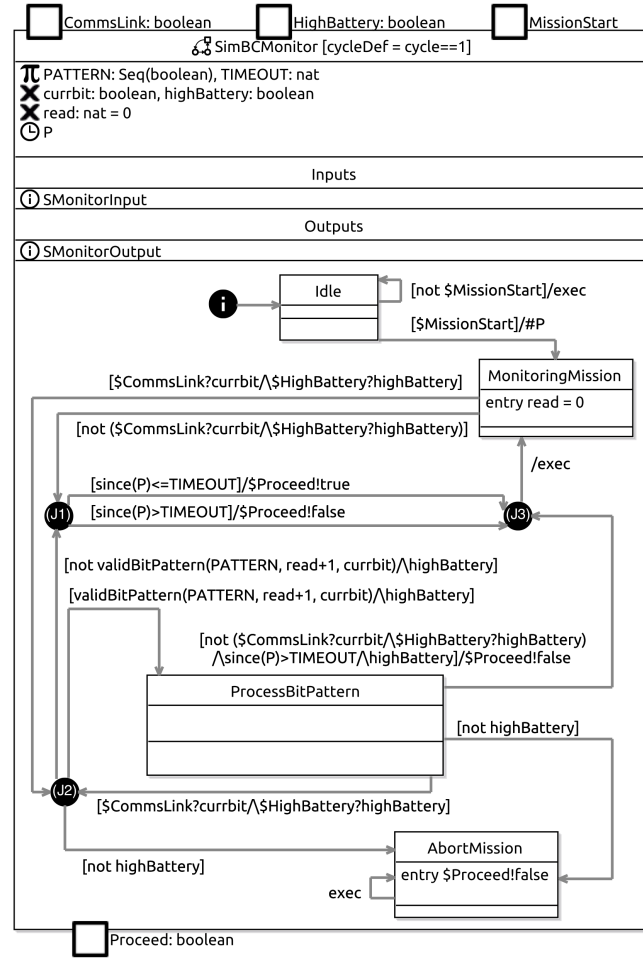
The definition of the safety machine, following on from the hazard analysis in the previous section, is reproduced in Figure 9, with the states and transitions of the composite state `ProcessBitPattern` elided for brevity. Similarly to a controller, the specification of a RoboSim state machine gives its name, `SimBCMonitor` in the example, followed by an expression defining its cycle period. Distinctively, for a machine we also have an explicit record of inputs and outputs.

`SimBCMonitor` declares two local constants: `PATTERN`, a `Sequence` of `booleans` that defines the expected transmission pattern, and `TIMEOUT`, which defines the maximum amount of time allowed for the communication link to recover, beyond which the mission should not proceed. The variables `currbit` and `highBattery` are used to store values communicated via the events, `CommsLink` and `HighBattery` respectively, and `read` is used to track the position in `PATTERN` matched so far. `SimBCMonitor` also declares a clock `P` that is used to enforce the `TIMEOUT`.

An incomplete version of `SimBCMonitor` is provided as a RoboSim project, alongside a set of complementary exercises⁷ to be completed using RoboTool. Next, we describe the control flow of `SimBCMonitor` in more detail.

The control flow of a state machine starts in the initial junction (indicated by ❶), whose outgoing transition identifies the starting state of the control flow. In `SimBCMonitor`, that state is `Idle`, which has two outgoing transitions with mutually exclusive guards: if the `MissionStart` event is received (that is, if the guard `$MissionStart` holds, where `$MissionStart` is a boolean expression indicating whether the event has happened in the current cycle), then state `Idle` is interrupted, the clock `P` is reset (`#P`) on the transition's action and the state `MonitoringMission` is entered afterwards. Otherwise, if no `MissionStart` event has taken place in the current cycle (`not $MissionStart`), there is a self transition on

⁷ robo-star.cs.york.ac.uk/events/setss2025/RoboSim-practical-1.pdf

Fig. 9. RoboSim model of the safety machine named **SimBCMonitor**

Idle with the action **exec** to indicate completion of the current cycle. Once the cycle period has elapsed, then execution resumes by entering state **Idle** again.

Once the mission has started, that is, the transition with guard $\$MissionStart$ is enabled, **SimBCMonitor** transitions to a state **MonitoringMission**, where an entry action assigns the value 0 to the variable **read** before actually entering **MonitoringMission**. From that state there are two outgoing transitions with mutually exclusive guards that go to junctions, **J1** and **J2**, (each indicated by ●). The RoboSim notation does not define labels for junctions, but we include them in Figure 9 to help us explain the control flow of **SimBCMonitor**.

The first outgoing transition from **MonitoringMission** is available exactly when both **CommsLink** and **HighBattery** have been provided as inputs in a cycle. When that transition is taken, the values communicated via these events are assigned

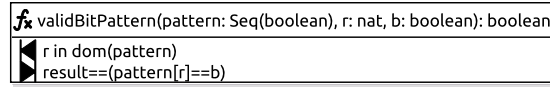


Fig. 10. Specification of function validBitPattern using pre- and post-conditions.

to local variables `currbit` and `highBattery`, respectively. For example, the notation `$CommsLink?currbit` indicates that the (boolean) value received via `CommsLink` is assigned to `currbit`. From the transition's target junction J2, there are three outgoing transitions whose guards use `highBattery`: if the battery level is not high (not `highBattery`), then `SimBCMonitor` transitions to the state `AbortMission`, otherwise it proceeds to `ProcessBitPattern` or to junction J3.

Junction J2, which is also the target of a transition from `MonitoringMission`, has two possible outgoing transitions whose conditions depend on the time since the clock `P` was reset. If it is less than or equal to `TIMEOUT`, then an output (`Proceed!true`) via the event `Proceed` of the value `true` is produced, and otherwise the value provided is `false`. In both cases, the transitions ultimately lead to an `exec` action being executed before re-entering the state `MonitoringMission`.

`AbortMission` is a sink state, that is, a state that the machine cannot leave, with an entry action that produces an output (`$Proceed!false`) via the event `Proceed` of value `false`, indicating that the mission should not proceed. The self-transition with `exec` as a trigger ensures that the simulation can proceed.

Finally, the state `ProcessBitPattern` encapsulates the specific control flow for processing one bit at a time against the `PATTERN`. Firstly, the transitions into `ProcessBitPattern` are guarded by the application of the function `validBitPattern` to the constant `PATTERN`, the position `read + 1` of the current bit `curbit` just input. The result of this function application is true exactly when `currbit` is equal to the value at position `read+1` in the sequence `PATTERN`.

The specification of `validBitPattern` itself is reproduced in Figure 10. It is given using a pre ($\blacktriangleleft$) and a post-condition ($\blacktriangleright$), using the operators of the Z mathematical toolkit [36, 37], which are available in RoboSim. The precondition requires that the input index `r` (`read + 1` in our example) must be in the domain of the `pattern` sequence given as argument. The postcondition defines that the result of the function is the boolean determined by the equality between `pattern` at position `r`, and the argument bit `b` (`currbit` in our example).

A state machine for the composite state `ProcessBitPattern` that is able to handle multiple bits is reproduced in Figure 11. Here, the transition from the initial junction to another junction has an action to increment the value of `read` up to the number of bits in the sequence as obtained using `size(PATTERN)`. In the target junction, if the index `read` has reached the end of the sequence, the pattern has been fully processed, so clock `P` is reset, `read` is set to zero, and `true` is produced as an output via `Proceed`, otherwise no action is taken until the state `s0` is entered. In `s0`, a self-transition ensures that while no inputs are received, and the `TIMEOUT` is not exceeded, passage of cycles is allowed via

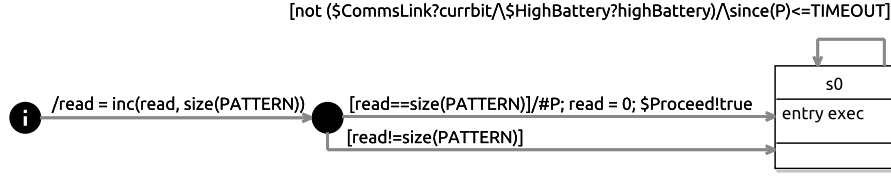


Fig. 11. Machine for composite state ProcessBitPattern

```

1 assertion A0 : SimBCMonitor is deadlock-free
2 assertion A1 : SimBCMonitor is divergence-free
3 assertion A2 : SimBCMonitor is deterministic

```

Listing 1.1. RoboSim assertions.

exec. Transitions out of ProcessBitPattern handle other cases (see Figure 9), for example, when a TIMEOUT occurs or the battery is not high.

4.4 Verification of the safety machine

In this section, we illustrate the use of RoboTool’s integration with formal verification tools to check that SimBCMonitor behaves as intended. Here, we are interested in verifying that the Proceed output is set to **false** whenever the battery goes low or the communication link has been lost for a set amount of time.

First, we specify these properties in a mixture of controlled English and *tock*-CSP [11], a discrete-timed process algebra that is used to give semantics to RoboSim models. Afterwards, we verify that the model is conformant with the properties, via the notion of refinement, using the CSP model-checker FDR [21].

Core properties RoboSim has an assertion language⁸ that allows us to specify core properties, namely, deadlock and divergence freedom, and determinism, without the need to know any CSP. These assertions can be specified as illustrated in Listing 1.1. Each assertion has a unique identifier, such as A0, and the name of the RoboSim component it applies to, followed by the type of assertion.

Low battery property The first application-specific property is specified via the *tock*-CSP process BatterySpec in Listing 1.2. Lines 1 to 12 define a **csp** block, a construct of the assertion language that encapsulates definitions written in CSPM⁹, the machine-readable form of CSP accepted by FDR. The CSP process BatterySpec itself is defined on lines 2 to 7, which we explain in detail next.

⁸ Further described in Section 5.4 of the RoboTool manual.

⁹ <https://cocotec.io/fdr/manual/>

```

1 csp BatterySpec csp-begin
2 BatterySpec =
3   SimBCMonitor::registerRead.SimBCMonitor::i_HighBattery.True?b ->
4   if b == False
5     then NoP
6   else (SimBCMonitor::registerWrite.SimBCMonitor::o_Proceed$_ -> SKIP
7         |~| SKIP) ; SimBCMonitor::endexec -> tock -> BatterySpec
8
9   NoP = SimBCMonitor::registerWrite.SimBCMonitor::o_Proceed!False ->
10         SimBCMonitor::endexec -> tock ->
11         SimBCMonitor::registerRead.SimBCMonitor::i_HighBattery.True?_ -> NoP
12 csp-end

```

Listing 1.2. RoboSim assertion block with CSP process definitions.

Initially, the behaviour of `BatterySpec` is defined by a prefixing (`->`) on the channel `SimBCMonitor::registerRead`, defining that it can communicate a value `SimBCMonitor::i_HighBattery.True?b`, where `i_HighBattery` is a type constructor corresponding to the event `HighBattery` in the RoboSim model, the value `True` indicates that the event has occurred in the current cycle, and `b` is any boolean. This captures the fact that the machine `SimBCMonitor` accepts the input indicated by the CSP event `registerRead`, namely, an occurrence of `HighBattery`.

In *tock*-CSP, `c.t?b -> P` is syntactic sugar for an external choice over all possible values of `b`, as defined by the data-type constructor `t`, with the free occurrences of `b` in `P` bound to the actual value chosen by the environment. So, the behaviour that follows on lines 4-7 in Listing 1.2 depends on the value of `b` input. When `b` is `False`, the behaviour is that of `NoP`, and otherwise it is defined by a sequential composition (`;`). We explain each process in what follows.

The behaviour of `NoP` is defined by a sequence of prefixings, starting with a communication over the channel `SimBCMonitor::registerWrite` of the value `SimBCMonitor::o_Proceed!False`, which corresponds to outputting `False` via the event `Proceed` of the RoboSim model. This is followed by the prefixing on `SimBCMonitor::endexec`, which records the end of a simulation cycle, and `tock`, which records the passage of one discrete time unit. Afterwards, the next cycle starts by reading inputs (prefixing on `SimBCMonitor::registerRead`) and then there is a recursion. So, in further cycles, the output for `Proceed` remains `False`.

In `BatterySpec`, when the value for `b` is `True` (lines 6-7 of Listing 1.2), there is a sequential composition. First, there is a nondeterministic choice (`|~|`) between a prefixing or the terminating process (**SKIP**). The prefixing allows for an output via `Proceed` of any boolean value chosen nondeterministically, as indicated by the notation `$_`, followed by **SKIP**. This allows for the output on `Proceed` to be optional with any value chosen non-deterministically, given that, when the input `HighBattery` is `True`, the property does not constrain the outputs. After the sequential composition, the process proceeds by synchronising on `SimBCMonitor::endexec` and `tock`, like in `NoP`, and then there is a recursion.

```

1 csp Impl1 associated to SimBCMonitor csp-begin
2 IEnable = SimBCMonitor::registerRead.SimBCMonitor::i_MissionStart.True ->
3           SimBCMonitor::registerRead.SimBCMonitor::i_HighBattery.True?_ ->
4           SimBCMonitor::registerRead.SimBCMonitor::i_CommsLink.True?_ ->
5           SimBCMonitor::endexec -> tock -> IEnable
6
7 Impl1 = (SimBCMonitor::D__(0,1,<True,False>,TIMEOUT)
8         [|{|SimBCMonitor::registerRead, SimBCMonitor::endexec, tock|}|
9         IEnable)
10        |\ {|SimBCMonitor::registerRead.SimBCMonitor::i_HighBattery,
11             SimBCMonitor::registerWrite,SimBCMonitor::endexec, tock|}|
12 csp-end
13 assertion A3 : Impl1 refines BatterySpec

```

Listing 1.3. Assertion block specifying the verification context for SimBCMonitor.

The verification of the safety machine `SimBCMonitor` is performed in a context where we assume all inputs to be present in every cycle. The cyclic paradigm embedded in `RoboSim` requires that all inputs can be read. Here, in addition, we require that those inputs provide a value for all inputs (so the value `True` is communicated for all events). This matches the assumptions made about the safety platform, where the safety function is to be deployed. This assumption is specified by the process `IEnable` in Listing 1.3 (lines 2 to 5).

The process `Impl1` used for verification is defined by the parallel composition ($[|\dots|]$) of `IEnable` with the process `SimBCMonitor::D__(...)`, which defines the *tock*-CSP semantics of `SimBCMonitor` automatically calculated by `RoboTool`, synchronising on `SimBCMonitor::registerRead`, `SimBCMonitor::endexec` and `tock`, and with every CSP channel other than those representing `HighBattery`, outputs (`registerWrite`), `SimBCMonitor::endexec` and `tock` hidden using the projection operator ($|\backslash$) of CSP. For model-checking with FDR, the values of every constant need to be defined, so on line 7 of Listing 1.3 `SimBCMonitor::D__(...)` has four values passed as parameters: the first value is an internal identifier used in the semantics, the second defines the period of the cycle as 1, and the last two define values for constants `PATTERN` and `TIMEOUT`, respectively.

Conformance of `Impl1`, which captures the semantics of `SimBCMonitor`, to `BatterySpec` is stated as a refinement **assertion** on line 13, requiring that every behaviour of `Impl1` is permitted by `BatterySpec`. The default semantic model of CSP used here for checking refinement accounts for both safety, concerned with what the implementation may do, and liveness, concerned with what the implementation must do. From this assertion, `RoboTool` automatically produces a `.csp` file suitable for checking with FDR, which can also be launched from within `RoboTool` to produce a verification report.

Communications link property The second application-specific property is specified by the process `CommsSpec` in Listing 1.4. This process is defined via

```

1 csp CommsSpec associated to SimBCMonitor csp-begin
2 CommsSpec =
3 let
4   p = <True,False>
5   max = {0..TIMEOUT+1}
6   Send(<b>^xs,t) =
7     SimBCMonitor::registerRead.SimBCMonitor::i_CommsLink.True!b ->
8     (if xs == <>
9       then SimBCMonitor::registerWrite.SimBCMonitor::o_Proceed!True ->
10        SimBCMonitor::endexec -> tock -> Send(p,1)
11     else SimBCMonitor::endexec -> tock -> Send(xs,Plus(t,1,max)))
12   []
13   SimBCMonitor::registerRead.SimBCMonitor::i_CommsLink.True!(not b) ->
14   (if t > TIMEOUT
15     then SimBCMonitor::registerWrite.SimBCMonitor::o_Proceed!False ->
16      SimBCMonitor::endexec -> tock -> Send(p,Plus(t,1,max))
17     else SimBCMonitor::registerWrite.SimBCMonitor::o_Proceed!True ->
18      SimBCMonitor::endexec -> tock -> Send(p,Plus(t,1,max)))
19 within
20   Send(p,0)
csp-end

```

Listing 1.4. *tock*-CSP specification for the communications link property.

a CSPM **let** ... **within** block that defines other processes and a pattern *p*, and behaves as the process *Send*(*p*,0) in the **within** clause. This process takes as parameters the pattern to be checked, and the time elapsed so far in the checking. So, initially the pattern is set to *p* and the time is 0. The process *Send* is defined for the case where the pattern is non-empty as follows.

A non-empty pattern has a head *b* and a tail *xs*. The behaviour of *Send* for such a pattern is defined on lines 6 to 17. There is an external choice ([1]) between two prefixings depending on the value of the input *CommsLink*. If it matches the head *b*, then there are two cases: if we are at the end of the sequence, that is, the tail *xs* is empty (lines 9 to 10), the output on *Proceed* is *True*, followed by passage of the cycle and the recursion with *Send*(*p*,1), otherwise there is no output but the recursion takes the tail *xs* and increments the time *t*. In the recursion (*Send*(*p*,1)) the second parameter with value 1 accounts for the fact that clock *P* is reset in the current cycle when the pattern has been fully processed, as seen in the transition guarded by *read==size(PATTERN)* from the junction to state *s0* in Figure 11, and so in the following cycle its value is 1.

If the value received is not *b*, then the behaviour depends on whether the *TIMEOUT* has been breached: if it has, then the output on *Proceed* is *False*; otherwise it is *True*. In both cases the recursion takes the initial sequence *p* and *t* is incremented. Here, the incrementation is specified using an auxiliary CSPM function *Plus* that ensures that the value is bounded by the set *max* that contains discrete values from 0 to *TIMEOUT*+1. This ensures the process is finite.

Results of analysis of assertions in rcheck.rsa using FDR

Assertion	States	Transitions	Result
SimBCMonitor is deadlock free (A0) [failures divergences model]	9672	24066	true
SimBCMonitor is divergence free (A1) [failures divergences model]	9672	24066	true
SimBCMonitor is deterministic (A2) [failures divergences model]	9672	24066	true
Impl1 refines CommsSpec (A3) [failures divergences model]	1763	3425	true
Impl2 refines BatterySpec (A4) [failures divergences model]	2691	5277	true

Fig. 12. Report produced by RoboTool after verifying assertions with FDR.

For verification the machine `SimBCMonitor` is considered in a context where inputs are available in every cycle, however, unlike for `Impl1`, we assume that the input `HighBattery` is always `True`. This is captured by the process `Impl2`, defined similarly to `Impl1` in Listing 1.5, where instead the semantics of `SimBCMonitor` is composed in parallel with the process `IE0kB`, itself the parallel composition of `IEnable` and `HB`, that constrains the value of input `HighBattery` to `True`. The **assertion** A4 is stated as a refinement on line 13.

```

1 csp Impl2 associated to SimBCMonitor csp-begin
2 Impl2 = (SimBCMonitor::D__ (0,1,<True,False>,TIMEOUT)
3     [|{|SimBCMonitor::registerRead, SimBCMonitor::endexec, tock|}|]
4     IE0kB)
5     |\ {|SimBCMonitor::registerRead.SimBCMonitor::i_CommsLink,
6         SimBCMonitor::registerWrite,SimBCMonitor::endexec, tock|}
7 IE0kB = (IEnable
8     [|{|SimBCMonitor::registerRead.SimBCMonitor::i_HighBattery|}|]
9     HB)
10
11 HB = SimBCMonitor::registerRead.SimBCMonitor::i_HighBattery.True.True-> HB
12 csp-end
13 assertion A4 : Impl2 refines CommsSpec

```

Listing 1.5. Assertion block specifying the verification context for `SimBCMonitor`.

Using RoboTool it is possible to invoke FDR to verify the assertions and produce a table of results as reproduced in Figure 12. In this case, all assertions pass, indicating that the model satisfies both properties as intended, in addition to being free of deadlocks and divergences, and deterministic.

This concludes the discussion of the safety machine as modelled in RoboSim. In the next section, we focus on its implementation for the safety platform.

5 Sound implementation of the safety function

In this section, we present a systematic approach to implement a RoboSim model (for a safety function) using the CSSP. Section 5.1 describes how we use the formal mathematical notation B in the process. Details of the CSSP and its programming environment are given in Section 5.2. Our model-transformation approach to translating RoboSim models is described in Section 5.3.

5.1 Use of B-method for the system development

For many years, the adoption of formal methods in industrial contexts faced significant resistance, largely due to concerns about scalability, tool maturity, and the steep learning curve associated with formal specification languages [23]. This scenario began to change with the successful use of the B-method in the validation of the SACEM system [22] and the implementation of the METEOR metro control system [12]. These pioneering projects demonstrated not only the technical maturity of the B-method but also the practical viability of its supporting tools in large-scale, safety-critical industrial applications.

Following these milestones, the B-method gained increasing recognition as a reliable approach for the development of dependable systems. It has since been widely adopted in the railway domain, where it has been used to specify, verify, and implement systems ranging from train control and interlocking systems to on-board software and trackside logic [27]. Beyond railways, the B-method has also been applied to other safety-critical domains, such as power generation and distribution systems, where functional correctness and fault tolerance are crucial; industrial automation, particularly in the control of machinery subject to strict safety constraints; and safety mechanisms, including the opening and closing of secure doors in public transport and industrial environments.

These diverse applications highlight the versatility of the method and its ability to provide formal guarantees of correctness throughout the system development lifecycle, from specification to executable code, backed by mathematical proof. The work we present here allows practitioners to ally the ease of developing and reading diagrammatic models of control software for robotics with the formal facilities of the B-method and its tools to write code for the CSSP.

The B-method ensures through the generation and discharge of proof obligations that the implementation faithfully satisfies the formally specified safety requirements. Its development approach has been successfully adopted for decades in the design and certification of Communication-Based Train Control (CBTC) systems, and is now considered a mature and efficient industrial practice for delivering safety assurance in critical railway applications.

Over the years, additional tools and methods have emerged to support the development and certification of safety-critical systems. Next, we describe a notable example: the creation of the CSSP, a hardware and software co-engineered platform specifically designed to guarantee safety properties by construction.

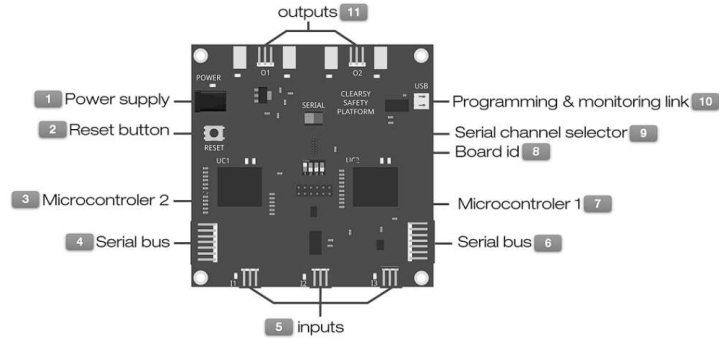


Fig. 13. SK0 Board – Clearsy Safety Platform.
github.com/CLEARSY/CSSP-Programming-Handbook

5.2 The CLEARSY Safety Platform

The CSSP, shown in Figure 13, combines the characteristics of a generic Programmable Logic Controller (PLC), used for controlling inputs and outputs in safety-critical applications, with a comprehensive Integrated Development Environment (IDE) that supports formal specification, analysis, proof, and implementation. This section presents the hardware and software components of the platform that support the reliable implementation of a RoboSim state machine.

The safety computer embedded in the CSSP is designed to meet a set of standardised safety requirements. When a failure is detected, the hardware immediately transitions to a restrictive mode, referred to as Panic Mode, in which all outputs are deactivated, that is, outputs are de-energised, leading to a restrictive state, so that no dangerous action can be triggered. This mechanism ensures that no hazardous outputs are maintained, fulfilling the platform’s central objective of preserving human safety in the presence of faults [25].

To achieve high safety integrity, the CSSP uses a dual-processor architecture (see Figure 13). Each processor performs the computation independently, and a voting system is used to validate the results. Three different turnkey systems including the CSSP have been certified at SIL3/SIL4 levels on different continents by different certification bodies. This remarkable achievement is evidence of its robustness and compliance with safety standards [25].

Two hardware models are currently available: SK0, with 3 digital inputs and 2 outputs, and SK1, with 20 inputs and 8 outputs. Both models share the same execution platform, based on two PIC32 microcontrollers¹⁰, capable of updating 50,000 interlocking Boolean equations per second [25, 26]. Future versions will include analogue inputs and outputs and networking capabilities.

The firmware operates cyclically; in each cycle, it reads inputs, performs computations, and sets outputs, matching the paradigm adopted by RoboSim. In case the CSSP enters the Panic Mode, restart is only possible by resetting the

¹⁰ PIC32MX795F512L providing 105 DMIPS at 80 MHz.

board, which reloads the flash memory contents into RAM. If the fault persists, the board remains in the restrictive state of the Panic Mode.

As SK0 and SK1 are intended for research and development, they do not implement all the features required by production-grade safety electronics. However, this does not prevent their use for prototyping and experimentation.

From a B implementable model, two binaries can be generated for deployment in the CSSP. The first one can be obtained via a dedicated compiler developed by CLEARSY to transform a B model into a HEX file. The second is produced with the Atelier B C code generator [18], then compiled with the GCC compiler into another HEX file. Each binary encodes the same B machine, but is supposed to be made of different sequences of instructions because of the diversity of the tool chains. The two binaries are linked with a sequencer, accountable for reading inputs, executing each binary and setting the outputs, using a safety library to perform safety verification. If verification fails, the CSSP enters the Panic Mode.

The final program is uploaded on the two micro-controllers. The bootloader, on the electronic board, checks the integrity of the program, and then both micro-controllers start to execute it. During execution, an internal verification of the identity between memory output states and physical output states is carried out. In addition, it is checked if the board is unable to command the outputs. Finally, an external verification is executed every 50ms to compare memory spaces (variables) in the micro-controllers. If any of these verifications fails the LED indicator blinks to indicate that the CSSP is in Panic Mode.

To summarise, the development cycle includes the following steps.

1. Specification of an abstract B model, capturing the system constraints and requirements.
 2. Development of the implementation model also in the B language.
 3. Automatic verification of the model using Atelier B's theorem prover.
 4. Automatic generation of two binaries.
 5. Linking of both binaries with a high-level sequencer and a safety library.
 6. Deployment of the software to the flash memory of both microcontrollers.
- During execution, the contents of the flash memory are copied to RAM.

In our approach, since B models are produced by transformation of a RoboSim model, we obtain the implementation model mentioned in (2) above without effort. Moreover, we do not need to subject that model to verification as suggested in (3), if that has been done at the RoboSim level. For validation of our work, however, the possibility of verification is very useful.

Programming the CSSP requires a dedicated version of Atelier B, shown in Figure 14¹¹. It adopts a default project structure, which already includes low-level access to digital inputs and outputs. The developer needs to edit only four files: `logic`, `logic_i`, `user_ctx`, and `user_ctx_i`. Figure 14 shows part of this structure. The interface with the safety library defines all types and operations available in a CSSP project. These include access to digital input values, elapsed time since the last reset, and the ability to control digital outputs. The behaviour

¹¹ Available at: atelierb.eu/en/atelier-b-support-maintenance/download-atelier-b/

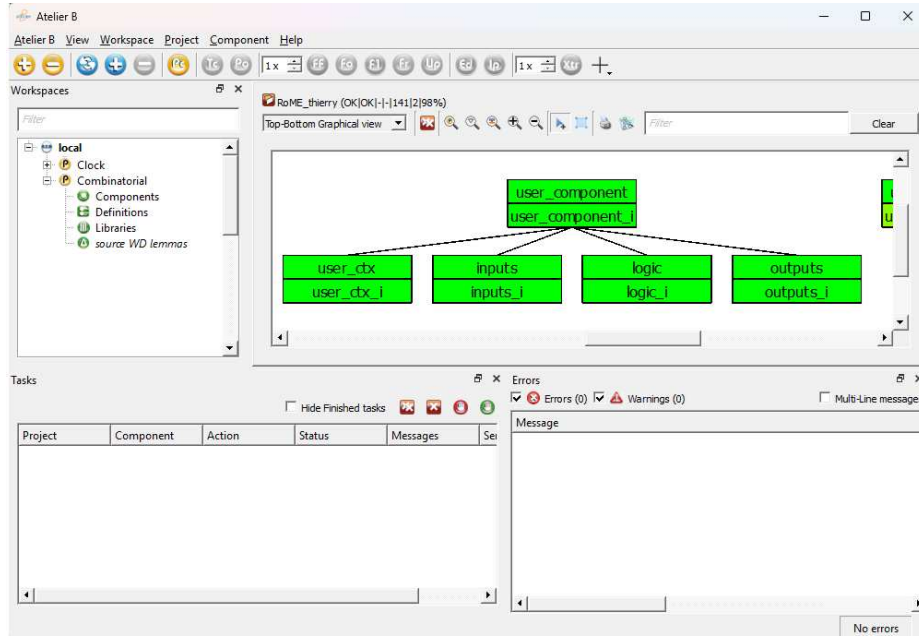


Fig. 14. The Atelier B interface for CSSP development.

of the system is defined in `logic` and `logic.i`, while constants and sets are declared in `user_ctx`, with their values defined in `user_ctx.i`.

A B model is structured into modules (not to be confused with RoboSim modules) and refinements. A module is used to break down the model for a large software into smaller parts. A B module has a specification, called a machine, where a static and a dynamic description of the requirements are formalised. It defines a mathematical model of the subsystem concerned with an abstract description of its state space and possible initial states, and an abstract description of operations to query or modify the state. It is good modelling practice to separate constants (usually seen by many modules) and variables (implemented in a single module) to avoid architecture constraints (dependencies).

A B module establishes an external interface: every implementation needs to conform to this specification. Conformance is assured by proof during the formal development process, in which a module specification is refined. This means that it is re-expressed with more information: adding some requirements, refining abstract notions with more concrete notions, and finally getting to an implementable code level. Data refinement introduces new variables to represent the state variables for the refined component, with their linking invariant. Algorithmic refinement consists in transforming the operations for the refined component. A refinement may also be refined. The final refinement of a refinement column is called the implementation, it contains only B0-compliant models (deterministic, using concrete types and operators).

Context B machines (constants only) and implementation B machines (variables only) are distinguished using the pragmas `CONSTANTS` and `SAFETY_VARS`, respectively, as reproduced in Figure 15 [26].

```

IMPLEMENTATION
  logic_i
REFINES
  logic
SEES
  g_types,
  g_operators,
  io_constants,
  lchip_interface,
  user_ctx,
  inputs

  // pragma SAFETY_VARS

```

```

IMPLEMENTATION
  user_ctx_i
REFINES
  user_ctx

  // pragma CONSTANTS
SEES
  g_types
VALUES
  DELTA_T = 1000 // 1000 ms == 1s
END

```

Fig. 15. Examples of pragmas.

Available types include `uint8_t`, `uint16_t`, `uint32_t`, and `B00L`. The first three represent unsigned integers encoded on 8, 16, and 32 bits. Inputs and outputs are stored as `uint8_t` with possible values `IO_ON` or `IO_OFF`, rather than booleans, to increase robustness against memory corruption¹².

To prevent arithmetic overflows, the platform provides built-in arithmetic operations that do not overflow: `add_uint32`, `sub_uint32`, and `mul_uint32`. These are implemented using lambda functions and are integrated into the safety library. Division is not replaced, as it cannot cause overflow. Time is represented as a `uint32_t` in milliseconds, accessible via the `get_ms_tick` operation.

The inputs and outputs are accessed using the functions `get_<input>` and `get_<output>`, where the placeholder corresponds to the configured name of each I/O port.

The CSSP uses a restricted subset of the B language, called B0, which simplifies compilation and improves confidence in the correctness of generated code during certification. The following constraints apply:

- The **IF-THEN-ELSE** construct supports only a single condition; compound tests must be nested.
- Logical operators are limited to `<`, `<=`, and `=`.
- Assignments support at most two operands on the right-hand side.
- Valuations involving more than two operands must be decomposed into successive operations.
- Variables declared in **VAR IN END** blocks must be typed explicitly using a “becomes such that” substitution.

Examples of B0 specifications satisfying the constraints are shown in Figure 16.

¹² Using two very different representations of True and False, encoded on 8 bits, allows to protect against a single (or few) “bit flip(s)”, as only 2 values out of 256 are valid.

<pre> user_logic = BEGIN VAR i1_, i2_, i3_ IN i1_ : (i1_ : uint8_t); i2_ : (i2_ : uint8_t); i3_ : (i3_ : uint8_t); i1_ <-- get_I1; i2_ <-- get_I2; i3_ <-- get_I3; O1 <-- triAND(i1_, i2_, i3_); O2 <-- negIO(O1) END END; </pre>	<pre> res <-- triAND(v1, v2, v3) = BEGIN res := IO_OFF; IF v1 = IO_ON THEN IF v2 = IO_ON THEN IF v3 = IO_ON THEN res := IO_ON END END END END END; </pre>
---	--

Fig. 16. Local variable typing and nested conditional structures.

The (bare metal) CSSP platform is executing its program as fast as possible (there is no underlying Operating System sharing resources, no notion of predefined cycle nor cycle time), corresponding to the sequence: (1) read inputs, (2) perform calculations, (3) write outputs. This sequence can be executed several tens of thousands times per second. With (1), Boolean input values are read once, saved in memory, and used for the current sequence. If the inputs are accessed several times during (2), it is always the value saved in memory that is used, even if the current physical value of the input is different. (2) represents the function `user_logic` which has access to the input values saved in memory. It performs computations based on these input values and on program variables. In particular, it calculates the Boolean value of the outputs to be established when the current sequence terminates. It also has access to the current number of milliseconds elapsed since the last reboot - enabling the programming of temporal behaviour. With (3), the Boolean output values computed in (2) are used to modify the state of the outputs with relays. The inertia of electronic relays prevents any change from one state to another at the computation speed of the CSSP platform. Hence, it is highly recommended to avoid short output state transitions to prevent fatal and final hardware failure.

In summary, we have presented an overview of the CLEARSY Safety Platform, describing its hardware and software components, the use of formal methods for safe system development, and the constraints used to ensure reliable execution. The CSSP provides a robust foundation for the prototyping and development of safety-critical systems.

5.3 Formal specification and implementation strategy

In this section, we present the approach used to translate a RoboSim model into the B language used on the CSSP platform. Usually, this kind of translation from state machines into executable code incurs some overhead mainly because of the control flow's encoding alongside the natural pre-existing control-flow of the host language. This duplication leads to efficiency degradation. In our particular case,

however, efficiency is not an issue as we have no difficulties meeting real-time constraints.

The RoboSim and the CLEARSY Safety Platform compliant B models share the same structure and execution model (reading inputs, performing computations, writing outputs). This simplifies the translation from RoboSim to B, which is currently done manually following the guidelines presented in this tutorial, but will be automated by a dedicated tool currently under development.

The approach being presented, the translation of RoboSim into B, may be conceived as the process of converting a diagrammatic and intuitive architectural design (RoboSim) into a set of rigorously defined B machines. We validate the RoboSim-to-B translation by checking that forbidden traces of a RoboSim model, calculated by RoboTool's forbidden trace generator¹³, cannot be executed by a corresponding B model. This additional validation is achieved using the ProB model checker [29] via specification of Linear Temporal Logic (LTL) formulae encoding the forbidden traces. It is part of a dedicated tool under development to be released in the near future. All code snippets presented in the sequel refer to the translation of the model represented in Figure 9.

Generating the CSSP context Initially, our translation approach extracts the constants from the RoboSim model and declares them as **CONCRETE_CONSTANTS** in a B context **MACHINE** `user_ctx`. In our example, the `SimBCMMonitor` constants are `PATTERN`, `TIMEOUT`, and the cycle related constant `cycleDef` (declared as constant `SimBCMMonitor_cycle_def`). The context machine also contains a constant `cycle_unit` that can be used to define the time unit considered in the RoboSim model. Furthermore, this machine contains constants that are used in our translation strategy and later described in this section. They are constants used to refer to the cycle state (`st_READ_INPUTS`, `st_STATE_MACHINE`, `st_WRITE_OUTPUTS`, and `st_TIME`) and to the states in the RoboSim state machine that are relevant to our translation (`INIT` and `EXEC_n`). All these constants are instantiated in the context **IMPLEMENTATION** `user_ctx_i`. These B **MACHINE** and **IMPLEMENTATION** components are rather trivial and are omitted here for conciseness. They can be found online¹⁴ along with the full generated CSSP code of our example.

The remainder of the translation generates a B **IMPLEMENTATION** component `logic_i`, sketched in Listing 1.6, which implements the main behaviour of the CSSP model. This implementation declares types and initialises the variables needed to implement the behaviour of the RoboSim model with default values. These are variables related to the RoboSim model's variables (`read`, `currbit`, `highBattery`) and clock (`P`), the inputs (`MissionStart`, `CommsLink`, and `HighBattery`), and the only output (`Proceed`). The name of B variables corresponding to RoboSim model inputs and outputs are prefixed, respectively, with `i_` and `o_`. Furthermore, there are variables for writing to the board's outputs (`board_0_01`

¹³ Documented in Section 5.6 of the RoboTool manual: <https://robostar.cs.york.ac.uk/publications/techreports/reports/robotool-manual.pdf#section.5.6>

¹⁴ https://robostar.cs.york.ac.uk/events/setss2025/UAV_firefighter-main.zip

```

1 IMPLEMENTATION logic_i
2 REFINES logic
3 SEES (...)
4 CONCRETE_VARIABLES
5     var_read_1, var_currbit_2, var_highBattery_3, var_P_1,
6     i_MissionStart, i_CommsLink, i_HighBattery, o_Proceed,
7     board_0_01, board_0_02,
8     first_time, SM_SimBCMonitor_state, cycle_timer, cycle_state,
9     (...)
10 INVARIANT
11     (... types the variables ...)
12 INITIALISATION
13     (... initialises the variables with default values ...)
14 LOCAL_OPERATIONS
15     (... specifies the operations ...)
16 OPERATIONS
17     (... implements the operations ...)
18 END

```

Listing 1.6. Main behaviour of CSSP model.

and board_0_02) and variables used by our translation approach (first_time, SM_SimBCMonitor_state, cycle_timer, and cycle_state).

In our translation, the entry actions in a RoboSim state machine are converted into operation calls, input events are mapped to the inputs of the CSSP platform, and operations are defined to link the RoboSim machine’s outputs to the platform’s outputs. Finally, transition guards and conditions are translated into **IF-THEN-ELSE** clauses within the B operation that represents the RoboSim machine. All operations are specified as **LOCAL_OPERATIONS** and implemented in the **OPERATIONS** part of the B component. Their specification, omitted here, simply includes information for typing the variables that can be written by an operation. Their implementation, however, complies with the RoboSim model as we describe in the sequel.

Synchronising the RoboSim cycle with the CSSP cycle As discussed in Section 4, the cycle of each RoboSim component (module, controller, or state machine) is characterised by the following control flow: reading the inputs, executing the component’s logic, writing the outputs, and then waiting for the remaining cycle period.

The RoboSim cycle is implemented using the `user_logic` operation defined in Listing 1.7, which updates the state machine using the `execute_model_cycle` operation every `cycle_duration` milliseconds, starting at the very moment when the board is switched on. In its first execution, the `user_logic` operation updates the cycle timer to the current time (line 5), executes the model cycle (line 6) – reading inputs, executing the state machine, and writing outputs – and sets the current state to the reading inputs state (line 7). This guarantees that the

state machine is instantly executed when we start the board. The next cycles will always wait a RoboSim cycle duration before executing the evolution of the state machine, which is guaranteed by the **ELSE** part of the operation body. In this case, the operation calculates the time elapsed (line 14) and checks if the cycle duration has been reached (line 16), in which case it executes the model cycle once again (line 18). Otherwise, this operation simply skips. The operation since is omitted here for conciseness. It calculates and returns the time elapsed in the timer given as an argument.

```

1  OPERATIONS
2  user_logic =
3  BEGIN
4      IF first_time = TRUE THEN
5          cycle_timer <-- get_ms_tick;
6          execute_model_cycle;
7          cycle_state:= st_READ_INPUTS;
8          first_time := FALSE
9      ELSE
10         VAR time_elapsed, cycle_duration IN
11             time_elapsed:(time_elapsed:uint32_t);
12             cycle_duration:(cycle_duration:uint32_t);
13
14             time_elapsed <-- since(cycle_timer);
15             cycle_duration := mul_uint32(SimSMovement_cycleDef,cycle_unit);
16             IF (cycle_duration <= time_elapsed) THEN
17                 cycle_timer <-- get_ms_tick;
18                 execute_model_cycle;
19                 cycle_state:= st_READ_INPUTS
20             END
21         END
22     END
23 END;

```

Listing 1.7. Implementation of the RoboSim cycle.

Executing the RoboSim model cycle The operation `execute_model_cycle`, specified in Listing 1.8, invokes the operations for reading the inputs, executing the RoboSim model’s state machines, and writing the outputs. The operation (`SM_SimBCMonitor`) that represents the state machine is specific to the RoboSim model being translated. The operations for reading inputs and writing outputs are described next (Listing 1.9).

```

1  execute_model_cycle =
2  BEGIN
3      read_model_inputs;
4      SM_SimBCMonitor;
5      write_model_outputs
6  END;

```

Listing 1.8. Execution of the RoboSim machine model cycle.

The operation that reads the model inputs (`read_model_inputs`) invokes the CSSP operations that retrieve the input values (`get_board_0_In`) assigning the results to the corresponding variables. On the other hand, the operation that writes the model outputs (`write_model_outputs`) assigns the values of the variables that represent the outputs to the corresponding CSSP outputs. We recall that the name of B variables corresponding to RoboSim model inputs and outputs are prefixed, respectively, with `i_` and `o_`. In our example, we have a single output (`Proceed`), thus to avoid any mistakes, we keep the unused board output (`board_0_02`) in a restrictive state (`I0_OFF`).

```

1 read_model_inputs =
2 BEGIN
3   i_MissionStart <-- get_board_0_I1;
4   i_CommsLink <-- get_board_0_I2;
5   i_HighBattery <-- get_board_0_I3;
6   cycle_state := st_STATE_MACHINE
7 END;
8 write_model_outputs =
9 BEGIN
10  board_0_01 := o_Proceed;
11  board_0_02 := I0_OFF;
12  cycle_state:= st_TIME
13 END;
```

Listing 1.9. Operations for reading model inputs and writing model outputs.

Executing the RoboSim state machine We now turn our attention to the operation `SM_SimBCMonitor` that implements the RoboSim model’s state machine.

First, we need to characterise the states from which the RoboSim model cycles start their execution. One of these states is the initial junction (`INIT`). For ease of illustrating our approach to mapping the cycles into B, we assume that the first cycle corresponds to `INIT`, so that `MissionStart` can only be triggered from subsequent cycles, and that the battery is initially in a charged state, so that the input `highBattery` has value true in the first cycle. The other states can be identified by looking for states in the RoboSim machine whose outgoing transitions have `exec`, to be encoded by `EXEC_n` states in the B model. For each identified state, we specify a **CONCRETE_CONSTANT** in the context machine `user_ctx` of the CSSP platform and trivially value them (from 0 to n) in the implementation `user_ctx.i` of the context machine. For example, in Figure 17, our states are `INIT`, `EXEC_1`, `EXEC_2`, and `EXEC_3`.

The skeleton of the implementation `SM_SimBCMonitor` of the state machine is presented in Listing 1.10. It uses the variable `SM_SimBCMonitor_state` to keep track of the current state, changing according to the transitions extracted from the RoboSim model. For example, if we are in the initial state `INIT`, we simply move to the next state `EXEC_1` (line 4), in which case we either move to the state `EXEC_1` if `Condition 1` is satisfied (line 6), or to `EXEC_2` if `Condition 2` is satisfied (line 7). Observe that like in the RoboSim machine, state `EXEC_3` is a sink state; that is, a

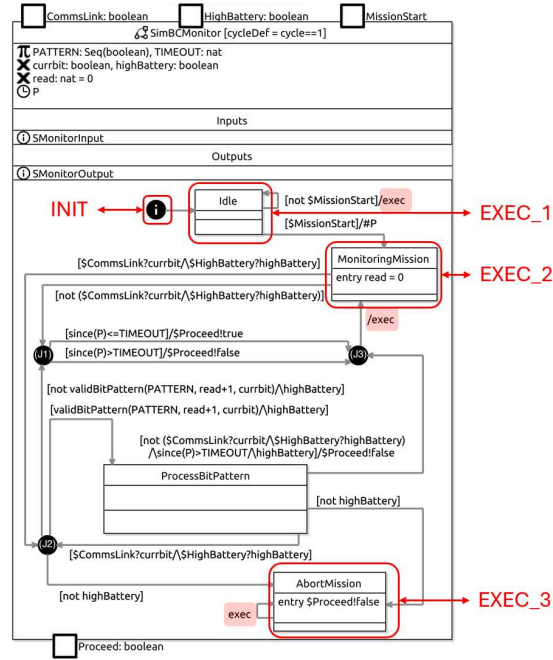


Fig. 17. CSSP constants representing states

state that, once reached, the state machine remains there regardless of the input. Then, after executing a cycle of the state machine, the operation `SM_SimBCMonitor` updates the variable `cycle_state` (line 16) to `st_WRITE_OUTPUTS`.

```

1 SM_SimBCMonitor =
2 BEGIN
3   IF (SM_SimBCMonitor_state = INIT) THEN
4     SM_SimBCMonitor_state := EXEC_1
5   ELSIF (SM_SimBCMonitor_state = EXEC_1) THEN
6     (Condition 1) SM_SimBCMonitor_state := EXEC_2
7     (Condition 2) SM_SimBCMonitor_state := EXEC_1
8     (...)
9   ELSIF (SM_SimBCMonitor_state = EXEC_2) THEN
10    (Multiple conditions) SM_SimBCMonitor_state := EXEC_2
11    (Other Multiple conditions) SM_SimBCMonitor_state := EXEC_3
12    (...)
13  ELSIF (SM_SimBCMonitor_state = EXEC_3) THEN
14    (... this is a sink state ...)
15  END;
16  cycle_state := st_WRITE_OUTPUTS
17 END;

```

Listing 1.10. Operation encoding the RoboSim state machine.

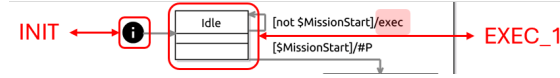


Fig. 18. State INIT

An initial junction does not have incoming transitions and it must have exactly one outgoing transition. In the case of our model (see Figure 18), this is a transition without a guard. Its translation to B is captured by the assignment of EXEC_1 to SM_SimBCMonitor_state (line 4 in Listing 1.10).

When in state EXEC_1 (Figure 19), the state machine can either evolve to state EXEC_2 or remain in state EXEC_1, depending on the input MissionStart. This conditional is modelled as a guarded transition as reproduced by the sketch in Listing 1.11. If the value for the input MissionStart is false (IO_OFF), the state machine remains in state EXEC_1 (line 4); otherwise (IO_ON), the machine stores in the variable var_P_1 the time when the mission start was raised (line 6) and moves to the EXEC_2 state (line 7).

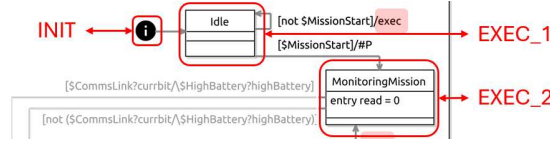


Fig. 19. States EXEC_1 and EXEC_2

```

1  (...)
2  ELSIF (SM_SimBCMonitor_state = EXEC_1) THEN
3    IF (i_MissionStart = IO_OFF) THEN
4      SM_SimBCMonitor_state:= EXEC_1
5    ELSIF (i_MissionStart = IO_ON) THEN
6      var_P_1 <- get_ms_tick;
7      SM_SimBCMonitor_state:= EXEC_2
8    END
9  ELSIF
10 (...)

```

Listing 1.11. Encoding of transitions out of state EXEC_1.

The transitions leaving from state EXEC_2 are slightly more complex, as they involve (1) composite predicates, such as conjunctions, and (2) junctions. With the CSSP, to evaluate a composite predicate, we need to store the logical value

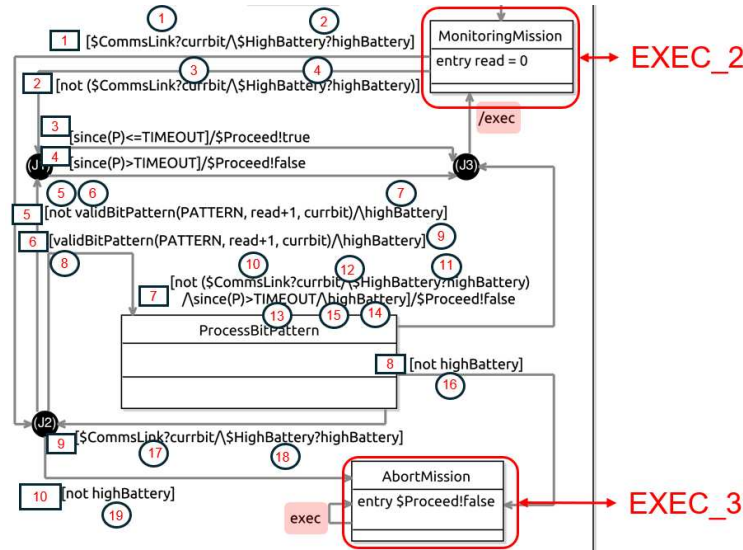


Fig. 20. States EXEC_2 and EXEC_3 with guards' predicates numbered from 1 to 19 and transitions numbered from 1 to 10, as captured in the translation by the variables `predicaten` (with n from 1 to 19) and `guardn` (with n from 1 to 10), respectively.

of each atomic predicate in a separate local variable and build the value of the entire predicate using the CSSP logical binary operators while storing the intermediate values in local variables. For junctions J1, J2 and J3, we need to treat the composite predicates used in the guards of both ingoing and outgoing transitions. We use local variables `predicaten` to store the logical values of the atomic predicates and their composition, except when the composition is used as a guard, in which case we store its value in a variable `guardn`. There are also variables to store the evaluation of the RoboSim clock P (`local_since`) and the variable `read` (`local_read`). In general, we assume that the CSSP is input-enabled, so that every input is read on every cycle, but to stay close to the RoboSim semantics we introduce a local variable `x_received` for every input x , that models availability of the input and that is assigned the value `TRUE`.

The skeleton of the alternation branch for state EXEC_2 of our example is presented in Listing 1.12. It initially declares and types the 19 predicates needed to construct the 10 existing guards, which are numbered according to the numbering presented in Figure 20: circles and rectangles indicate how we numbered the predicates and the guards, respectively.

By way of illustration, we present in Listing 1.13 the code that calculates the guard `$CommsLink?currbit /\ $HighBattery?highBattery`, which is identified as guard 1 in Figure 20. This guard is the result of the conjunction of two atomic predicates. The first one (predicate 1), encoded on line 2, states that the input `CommsLink` has been received, as recorded by the variable `commsLink_received`. We use the operation `bool` to convert the boolean value into one of type `B00L`. On

```

1  (...)
2  ELSIF (SM_SimBCMonitor_state = EXEC_2) THEN
3  (...)
4      VAR predicate_1, (... similar declarations ...), predicate_19,
5          guard_1, (... similar declarations ...), guard_10,
6          local_since, local_read_1,
7          commsLink_received, highBattery_received
8  IN predicate_1:(predicate_1:B00L);
9      (... similar declarations ...); predicate_19:(predicate_19:B00L);
10     guard_1:(guard_1:B00L);
11     (... similar declarations ...); guard_10:(guard_10:B00L);
12     var_clock_local:(var_clock_local:uint32_t);
13
14     (... compute local variable values ...)
15     (Multiple conditions) SM_SimBCMonitor_state:= EXEC_2
16     (Other Multiple conditions) SM_SimBCMonitor_state:= EXEC_3
17 ELSIF
18 (...)

```

Listing 1.12. Sketch illustrating the encoding of transitions out of state EXEC_2.

line 3, we assign the result of converting the input `i_CommsLink` to `var_currbit_2`. The second predicate (predicate 2) is translated similarly. Once we have the values of these two predicates assigned to local variables, we can calculate their conjunction using the safe operation `land`, which implements a logical conjunction, the definition of which is trivial and is omitted here for conciseness. Similarly, we also have operations for logical disjunction (`lor`) and negation (`lnot`).

```

1  (... compute local variable values ...)
2  predicate_1:= bool(commsLink_received = TRUE);
3  var_currbit_2:= bool(i_CommsLink = IO_ON);
4  predicate_2:= bool(highBattery_received = TRUE);
5  var_highBattery_3:= bool(i_HighBattery = IO_ON);
6  guard_1 <-- land(predicate_1,predicate_2);
7  (...)

```

Listing 1.13. Encoding of guard 1 as the conjunction of two atomic predicates.

Similarly, we build the values of the following local variables related to the guards of the state machine reproduced below:

```

– guard_1: CommsLink?currbit /\ $HighBattery?highBattery
– guard_2: not($CommsLink?currbit /\ $HighBattery?highBattery)
– guard_3: since(P) <= TIMEOUT
– guard_4: since(P) > TIMEOUT
– guard_5: not validBitPattern(...) /\ highBattery
– guard_6: validBitPattern(...) /\ highBattery
– guard_7: not($CommsLink?currbit /\ $HighBattery?highBattery)
           /\ since(P) > TIMEOUT /\ highBattery

```

```

1      IF (guard_1 = TRUE) THEN
2          IF (guard_5 = TRUE) THEN
3              IF (guard_3 = TRUE) THEN
4                  o_Proceed:= IO_ON;
5                  SM_SimBCMonitor_state := EXEC_2
6              ELSIF (guard_4 = TRUE) THEN
7                  o_Proceed:= IO_OFF;
8                  SM_SimBCMonitor_state := EXEC_2
9              END
10             ELSIF (guard_6 = TRUE) THEN
11                 (...)
12             ELSIF (guard_10 = TRUE) THEN
13                 (...)
14             END
15             ELSIF (guard_2 = TRUE) THEN
16                 (...)
17         END

```

Listing 1.14. Encoding of alternation starting from state EXEC_2.

- guard_8: not highBattery
- guard_9: \$CommsLink?currbit /\ \$HighBattery?highBattery
- guard_10: not highBattery

Next, we translate the alternation that considers all possible scenarios for the calculated guard values and their conjunctions based on the model's junctions. First, from the state EXEC_2, we have only two possibilities depending on whether guard_1 or guard_2 is true. In the following Listing 1.14, we first present the translation of the branch in which the guard_1 is true, with other cases covered by Listings 1.15 to 1.17. In this case, the model leads us to junction J2 from which we can leave based on the truth values of guards guard_5, guard_6 or guard_10.

In the first case (via guard_5), we reach junction (J1) from which we can only leave if guard guard_3 or guard guard_4 is true. In both cases, the state machine leads to junction J3, which waits for the next model cycle, taking us back to the state EXEC_2. If guard_3 is true, the state machine sets the output variable o_Proceed to IO_ON; otherwise, if guard_4 is true, the state machine sets the output variable o_Proceed to IO_OFF. As previously explained, the CSSP output values IO_ON and IO_OFF correspond, respectively, to the boolean values true and false.

In the second case, when guard_5 is false and guard_6 is true, the state machine is taken to the ProcessBitPattern state, from which we have three possible behaviours based on the guards guard_7, guard_8 and guard_9. This is sketched in Listing 1.15, where if guard_7 is true, the state machine sets the output variable o_Proceed to IO_OFF and stays in state EXEC_2; otherwise, if guard_8 is true, the state transitions to state EXEC_3; otherwise, if guard_9 is true, we go back to junction J2 and depend again on the evaluation of guards guard_5 and guard_10 as explained in relation to the transitions in the previous Listing 1.14.

```

1      IF (guard_1 = TRUE) THEN
2          IF (guard_5 = TRUE) THEN
3              (...)
4          ELSIF (guard_6 = TRUE) THEN
5              IF (guard_7 = TRUE) THEN
6                  o_Proceed:= IO_OFF;
7                  SM_SimBCMonitor_state := EXEC_2
8              ELSIF (guard_8 = TRUE) THEN
9                  SM_SimBCMonitor_state := EXEC_3
10             ELSIF (guard_9 = TRUE) THEN
11                 IF (guard_5 = TRUE) THEN
12                     IF (guard_3 = TRUE) THEN
13                         o_Proceed:= IO_ON;
14                         SM_SimBCMonitor_state := EXEC_2
15                     ELSIF (guard_4 = TRUE) THEN
16                         o_Proceed:= IO_OFF;
17                         SM_SimBCMonitor_state := EXEC_2
18                     END
19                 ELSIF (guard_10 = TRUE) THEN
20                     SM_SimBCMonitor_state := EXEC_3
21                 END
22             END
23             ELSIF (guard_10 = TRUE) THEN
24                 (...)
25             END
26         ELSIF (guard_2 = TRUE) THEN
27             (...)
28         END

```

Listing 1.15. Encoding of alternation starting from state EXEC_2 when guard 5 is false and guard 6 is true.

Finally, in the third case (*guard_10*), the state machine simply evolves to state EXEC_3 as sketched in Listing 1.16. Now, we turn to the case in which the *guard_1* is false and *guard_2* is true. In this case, the state machine goes directly to junction J1 from which the behaviour has already been previously explained.

The last state, EXEC_3 as identified in Figure 21, represents a restrictive position that should either correspond to “absence of power” or “safety conditions not met”. Upon entry into this state, the mission monitoring function is halted. The system enforces a safe state by assigning the value *IO_OFF* to the output *o_Proceed*, thereby mandating the immediate termination of the mission. Its

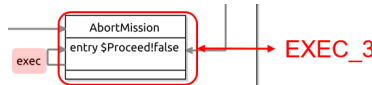


Fig. 21. State EXEC_3

```

1      IF (guard_1 = TRUE) THEN
2          IF (guard_5 = TRUE) THEN
3              (...)
4          ELSIF (guard_6 = TRUE) THEN
5              (...)
6          ELSIF (guard_10 = TRUE) THEN
7              SM_SimBCMonitor_state := EXEC_3
8          END
9          ELSIF (guard_2 = TRUE) THEN
10             (...)
11         END

```

Listing 1.16. Encoding of alternation from EXEC_2 when guard 10 is true.

```

1      IF (guard_1 = TRUE) THEN
2          (...)
3      ELSIF (guard_2 = TRUE) THEN
4          IF (guard_3 = TRUE) THEN
5              o_Proceed:= IO_ON;
6              SM_SimBCMonitor_state := EXEC_2
7          ELSIF (guard_4 = TRUE) THEN
8              o_Proceed:= IO_OFF;
9              SM_SimBCMonitor_state := EXEC_2
10         END
11     END

```

Listing 1.17. Encoding of alternation from EXEC_2 when guard 2 is true.

translation, presented in Listing 1.18, is rather simple because it is a sink state from which the machine sets the output value to IO_OFF and never leaves.

```

1      (...)
2      ELSIF (SM_SimBCMonitor_state = EXEC_3) THEN
3          o_Proceed:= IO_OFF
4      END;
5      (...)

```

Listing 1.18. Encoding of state EXEC_3.

In summary, the translation strategy presented in this section establishes a systematic bridge between RoboSim models and B implementations compliant with the CLEARSY Safety Platform (CSSP). By exploiting the close correspondence between the execution models of RoboSim and the CSSP, the proposed approach avoids unnecessary control-flow duplication while preserving both functional behaviour and timing assumptions. The ongoing validation of the translation through the comparison of forbidden traces further strengthens confidence that safety properties established at the modelling level are not compromised at the implementation level.

6 Conclusions

We have presented here an end-to-end design and verification of a safety function for a UAV. Our demonstrator is a firefighting UAV, but our work is immediately relevant for the many drone applications whose safety requirements are similar. Moreover, our approach can be applied to any robotic systems that raise safety concerns. This certainly includes all robotic systems aimed at close collaboration, that is, outside cages and isolated areas, with humans.

Ongoing work is dealing with the automation of the translation from RoboSim to B, and its validation (see Figure 1). This will make the cost of code development very low, and appealing outside the scope of safety functions. Another line of ongoing work is providing support to generate ROS code from RoboSim models. Integration of the CCSP in that context is not difficult.

Support for describing physical robotic platforms via block diagrams and differential equations, is available as part of RoboSim [31]. From such models, RoboTool can automatically generate Scene Description Format (SDF¹⁵) files that describe the physical structure of a robot suitable for use with a robotics simulator, such as Gazebo¹⁶ and CoppeliaSim¹⁷. That work, however, is beyond the scope of this tutorial. Here, we focus on the control software.

When hardware, software, and possibly human operators are integrated, new issues can emerge. Interfaces between components might not align with design assumptions. For example, a slight difference in sensor update timing could lead to control instability, or electromagnetic interference from the motors might disrupt the GPS receiver. Thorough testing under realistic conditions (hardware-in-the-loop simulations, field flight tests) is critical to reveal any hidden hazards.

If integration testing is insufficient, hazards may go unnoticed until deployment. For instance, a drone might work well in lab tests but, once deployed, a combination of high temperature and high payload could induce a failure that was never tested. Ensuring all safety functions work together as intended (e.g. the low-battery auto-landing engages properly in a real flight) is key. Any issues discovered require refining the design or implementation before final deployment.

RoboStar provides some support for automatic test generation [10, 15, 16]. Ongoing work is using the needs of the firefighter UAV as a basis to devise generation of test drivers for ROS and integration tests.

Additional resources to support the use RoboSim, RoboTool, Atelier-B, and CCSP are available¹⁸. This includes additional tutorials and course material.

Acknowledgments

This work was partially funded by the UK EPSRC Grants EP/R025479/1 and EP/V026801/2, and by the UK Royal Academy of Engineering Grant number

¹⁵ sdformat.org

¹⁶ gazebosim.org

¹⁷ www.coppeliarobotics.com

¹⁸ roboStar.cs.york.ac.uk and www.clearsy.com/en/tools/clearsy-safety-platform/

CiET1718/45. This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) www.ines.org.br, CNPq grant 408817/2024-0.

References

1. ARP4761: Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. Tech. Rep. ARP4761, SAE International, Warrendale, PA, USA (1996), available from <https://www.sae.org>
2. DO-254: Design assurance guidance for airborne electronic hardware. Tech. Rep. DO-254, RTCA, Inc., Washington, D.C., USA (2000), available from <https://www.rtca.org>
3. DO-178C: Software considerations in airborne systems and equipment certification. Tech. Rep. DO-178C, RTCA, Inc., Washington, D.C., USA (2011), available from <https://www.rtca.org>
4. DO-326A: Airworthiness security process specification. Tech. Rep. DO-326A, RTCA, Inc., Washington, D.C., USA (2014), available from <https://www.rtca.org>
5. Standard practice for operational risk assessment of small unmanned aircraft systems (suas) (2016), available from <https://www.astm.org>
6. Specific operations risk assessment (SORA) (2019), available from <https://www.easa.europa.eu>
7. ASTM F3266-20: Standard guide for training for remote pilot in command of unmanned aircraft systems (UAS) endorsement (2020), available from <https://www.astm.org>
8. Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)
9. Afzal, A., Goues, C.L., Hilton, M., Timperley, C.S.: A study on challenges of testing robotic systems. In: 13th IEEE International Conference on Software Testing, Validation and Verification. pp. 96–107 (2020). <https://doi.org/10.1109/ICST46399.2020.00020>
10. Baxter, J., Cavalcanti, A.L.C., Gazda, M., Hierons, R.M.: Testing Using CSP Models: Time, Inputs, and Outputs. ACM Transactions in Computational Logic **24**(2) (2023). <https://doi.org/10.1145/3572837>
11. Baxter, J., Ribeiro, P., Cavalcanti, A.L.C.: Sound reasoning in tock-CSP. Acta Informatica **59**, 125–162 (2022). <https://doi.org/10.1007/s00236-020-00394-3>
12. Behm, P., Benoit, P., Faivre, A., Meynadier, J.M.: Météor: A successful application of B in a large project. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) International Symposium on Formal Methods. pp. 369–387. Springer (1999). https://doi.org/10.1007/3-540-48119-2_22
13. Carlson, C.S.: Effective FMEAs: Achieving Safe, Reliable, and Economical Products and Processes using Failure Mode and Effects Analysis. John Wiley & Sons, Hoboken, NJ (2012)
14. Cavalcanti, A.L.C., Barnett, W., Baxter, J., Carvalho, G., Filho, M.C., Miyazawa, A., Ribeiro, P., Sampaio, A.C.A.: Robostar technology: A roboticist’s toolbox for combined proof, simulation, and testing. In: Cavalcanti, A.L.C., Dongol, B., Hierons, R., Timmis, J., Woodcock, J.C.P. (eds.) Software Engineering for Robotics, pp. 249–293. Springer International Publishing (2021). https://doi.org/10.1007/978-3-030-66494-7_9, [papers/CBBcfmrs21.pdf](https://papers.cbbcfmrs21.pdf)

15. Cavalcanti, A.L.C., Baxter, J., Hierons, R.M., Lefticaru, R.: Testing Robots using CSP. In: Beyer, D., Keller, C. (eds.) *Tests and Proofs*. pp. 21–38. Springer (2019). https://doi.org/10.1007/978-3-030-31157-5_2
16. Cavalcanti, A.L.C., Miyazawa, A., Schulze, U., Timmis, J.: Bringing RoboStar and RT-Tester together, pp. 16–33. Springer (2023). https://doi.org/10.1007/978-3-031-40132-9_2
17. Cavalcanti, A.L.C., Sampaio, A.C.A., Miyazawa, A., Ribeiro, P., Filho, M.C., Didier, A., Li, W., Timmis, J.: Verified simulation for robotics. *Science of Computer Programming* **174**, 1–37 (2019). <https://doi.org/10.1016/j.scico.2019.01.004>
18. CLEARSY: C4B User Manual, v1.2, 19/09/2016, CLEARSY (September 2016)
19. Ericson, C.A.: *Hazard Analysis Techniques for System Safety*. John Wiley & Sons, Hoboken, NJ, 2nd edn. (2015)
20. FAA: Drones & wildfires - digital toolkit, https://www.faa.gov/sites/faa.gov/files/uas/resources/community_engagement/FAA_drones_wildfires_toolkit.pdf
21. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3 - A Modern Refinement Checker for CSP. In: *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 187–201 (2014). https://doi.org/10.1007/978-3-642-54862-8_13
22. Guiho, G.D., Hennebert, C.: SACEM software validation (experience report). In: Valette, F., Freeman, P.A., Gaudel, M. (eds.) *Proceedings of the 12th International Conference on Software Engineering*, Nice, France, March 26–30, 1990. pp. 186–191. IEEE Computer Society (1990), <https://dl.acm.org/doi/10.5555/100296.100321>
23. Hall, A.: Seven myths of formal methods. *IEEE Software* **7**(5), 11–19 (1990). <https://doi.org/10.1109/52.57887>
24. International Electrotechnical Commission: *Failure modes and effects analysis (FMEA and FMECA)*. Tech. Rep. IEC 60812:2018, IEC, Geneva, Switzerland (2018)
25. Lecomte, T., Déharbe, D., Fournier, P., Oliveira, M.: The CLEARSY safety platform: 5 years of research, development and deployment. *Science of Computer Programming* **199**, 102524 (2020). <https://doi.org/10.1016/j.scico.2020.102524>
26. Lecomte, T.: Programming the CLEARSY safety platform with B. In: Raschke, A., Méry, D., Houdek, F. (eds.) *International Conference on Rigorous State-Based Methods. Lecture Notes in Computer Science*, vol. 12071, pp. 124–138. Springer (2020). https://doi.org/10.1007/978-3-030-48077-6_9
27. Lecomte, T., Déharbe, D., Prun, É., Mottin, E.: Applying a formal method in industry: A 25-year trajectory. In: da Costa Cavalheiro, S.A., Fiadeiro, J.L. (eds.) *Brazilian Symposium on Formal Methods. Lecture Notes in Computer Science*, vol. 10623, pp. 70–87. Springer (2017). https://doi.org/10.1007/978-3-319-70848-5_6
28. Lee, W.S., Grosh, D.L., Tillman, F.A., Lie, C.H.: Fault tree analysis, methods, and applications – a review. *IEEE Transactions on Reliability* **R-34**(3), 194–203 (1985). <https://doi.org/10.1109/TR.1985.5222114>
29. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) *FME2003: Formal Methods. Lecture Notes in Computer Science*, vol. 2805, pp. 855–874 (2003)
30. Leveson, N.G.: *Engineering a Safer World: Systems Thinking Applied to Safety*. The MIT Press (01 2012). <https://doi.org/10.7551/mitpress/8179.001.0001>

31. Miyazawa, A., Ahmadi, S., Cavalcanti, A., Baxter, J., Post, M., Ribeiro, P., Timmis, J., Wright, T.: Diagrammatic physical robot models. *Software and Systems Modeling* pp. 1–45 (2025). <https://doi.org/10.1007/s10270-025-01270-9>
32. Mousavi, M., Cavalcanti, A.L.C., Fisher, M., Dennis, L., Hierons, R., Kaddouh, B., Law, E.L.C., Richardson, R., Ringert, J.O., Tyukin, I., Woodcock, J.C.P.: Trustworthy Autonomous Systems through Verifiability. *IEEE Software Magazine* **56**(2) (2023). <https://doi.org/doi.org/10.1109/MC.2022.3192206>
33. Ortega, A., Hochgeschwender, N., Berger, T.: Testing service robots in the field: An experience report. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. pp. 165–172. IEEE (2022). <https://doi.org/10.1109/IR0547612.2022.9981789>
34. Plioutsias, A., Karanikas, N., Chatzimihailidou, M.M.: Hazard analysis and safety requirements for small drone operations: To what extent do popular drones embed safety? *Risk Analysis* **38**(3), 562–584 (2018). <https://doi.org/https://doi.org/10.1111/risa.12867>
35. Ruijters, E., Stoelinga, M.: Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Comput. Sci. Rev.* **15**, 29–62 (2015). <https://doi.org/10.1016/J.COSREV.2015.03.001>
36. Spivey, J.M.: *The Z Notation: A Reference Manual*. 2nd, Prentice-Hall (1992)
37. Woodcock, J.C.P., Davies, J.: *Using Z-Specification, Refinement and Proof*. Prentice-Hall (1996)