Integration and Evaluation of the AdvoCATE, FRET, CoCoSim, and Event-B Tools on the Inspection Rover Case Study

Hamza Bourbouh  
KBR, NASA Ames Research Center, Moffett Field, CA 94035, USA

Marie Farrell  
Department of Computer Science, Maynooth University, Maynooth, Co. Kildare, Ireland

Anastasia Mavridou  
KBR, NASA Ames Research Center, Moffett Field, CA 94035, USA

Irfan Sljivo  
KBR, NASA Ames Research Center, Moffett Field, CA 94035, USA

March 2020
Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA’s STI. The NASA STI Program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- **TECHNICAL PUBLICATION.** Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- **TECHNICAL MEMORANDUM.** Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- **CONTRACTOR REPORT.** Scientific and technical findings by NASA-sponsored contractors and grantees.

- **CONFERENCE PUBLICATION.** Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- **SPECIAL PUBLICATION.** Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- **TECHNICAL TRANSLATION.** English-language translations of foreign scientific and technical material pertinent to NASA’s mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI Program, see the following:

- Access the NASA STI program home page at [http://www.sti.nasa.gov](http://www.sti.nasa.gov)
- E-mail your question to help@sti.nasa.gov
- Phone the NASA STI Help Desk at 757-864-9658
- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681–2199
Integration and Evaluation of the AdvoCATE, FRET, CoCoSim, and Event-B Tools on the Inspection Rover Case Study

Hamza Bourbouh  
KBR, NASA Ames Research Center, Moffett Field, CA 94035, USA

Marie Farrell  
Department of Computer Science, Maynooth University, Maynooth, Co. Kildare, Ireland

Anastasia Mavridou  
KBR, NASA Ames Research Center, Moffett Field, CA 94035, USA

Irfan Sljivo  
KBR, NASA Ames Research Center, Moffett Field, CA 94035, USA
Acknowledgments

We thank Guillaume Brat, Louise A. Dennis, Michael Fisher, and Dimitra Giannakopoulou for valuable feedback on this work. We gratefully acknowledge the NASA Aeronautics Mission Directorate System-Wide Safety program’s SAAFE project, the UK Research and Innovation, and EPSRC Hubs for “Robotics and AI in Hazardous Environments”: EP/R026092 (FAIR-SPACE) for funding this work. The majority of Marie Farrell’s work on this project took place while she was at the University of Liverpool and, subsequently, the University of Manchester, UK.

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

This report is available in electronic form at http://URL
Executive Summary

The complexity and flexibility of autonomous robotic systems necessitates a range of distinct verification tools. This presents new challenges not only for design verification but also for assurance approaches. Combining the distinct formal verification tools, while maintaining sufficient formal coherence to provide compelling assurance evidence is difficult, often being abandoned for less formal approaches. In this technical memorandum, we demonstrate, through a case study, how a variety of distinct formal techniques can be brought together in order to develop a justifiable assurance case. We use the AdvoCATE assurance case tool to guide our analyses and to integrate the artifacts from the formal methods that we use, namely: FRET, COCOSIM and Event-B. While we present our methodology as applied to a specific Inspection Rover case study, we believe that this combination provides benefits in maintaining coherent formal links across development and assurance processes for a wide range of autonomous robotic systems.

This technical report provides a more detailed overview of the work presented in [1]. In addition to thorough and detailed descriptions, it contains initial design models and describes some of the artifacts that were omitted from discussion in [1].

For full source code and models please contact the authors\textsuperscript{1234}.

\textsuperscript{1}Hamza Bourbouh: bourbouhh@gmail.com
\textsuperscript{2}Marie Farrell: marie.farrell@mu.ie
\textsuperscript{3}Anastasia Mavridou: anastasia.mavridou@nasa.gov
\textsuperscript{4}Irfan Sljivo: irfan.sljivo@nasa.gov
# Contents

1 Introduction 1

2 Background 3
   2.1 Assurance Case Automation Toolset (AdvoCATE) . . . . . . . . . . . . . . 3
   2.2 Formal Requirements Elicitation Tool (FRET) . . . . . . . . . . . . . . . 3
   2.3 Contract-based Compositional Verification of Simulink Models (COCOSIM) 4
   2.4 Event-B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

3 The Case Study Methodology 5

4 The Case Study Step-by-Step 7
   4.1 Step 0: Characterize Initial System . . . . . . . . . . . . . . . . . . . . . . 7
   4.2 Step 1: Create Initial System Model . . . . . . . . . . . . . . . . . . . . . 7
   4.3 Step 2: Perform Preliminary Hazard Analysis . . . . . . . . . . . . . . . . 9
   4.4 Step 3: Define Mitigations and Safety Requirements . . . . . . . . . . . . 13
   4.5 Step 4: Refine System Model According to Mitigations . . . . . . . . . . . 16
   4.6 Step 5: Formalize Requirements and Create Formal Specifications . . . . 24
   4.7 Step 6: Perform Verification and Simulation at System- and Component-Levels 26
      4.7.1 Compositional Verification in CoCoSim . . . . . . . . . . . . . . . . 27
      4.7.2 Component-Level Verification Using Kind2 and Event-B: . . . . . . 27
   4.8 Step 7: Document Verification Results and Build Safety Case . . . . . . . 33

5 Discussion 35

6 Related Work 37

7 Conclusions and Future Work 38
   7.1 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
   7.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
List of Figures

3.1 Our methodology for integrating verification results via an assurance case instantiated with the selected tools for the Inspection Rover case study. The incoming arrows without a source represent all relevant artifacts from previous phases. For system-level analysis these comprise the Lustre requirements and the Simulink system model, while for component-level analysis these comprise the Lustre and Event-B system models and requirements. In the documentation phase we input all artifacts. ........................................ 6

4.1 An overview of the robotic system to be verified. Each rectangle represents a node in the robotic system and each arrow represents data flow between nodes. ................................................................. 8

4.2 AADL initial architecture model. .................................................. 9

4.3 High-level rover architecture. ..................................................... 10

4.4 Preliminary Inspection Rover Functional Decomposition in AdvoCATE. .. 10

4.5 Hazard Identification View in AdvoCATE. The first column indicates the particular rover’s activity that we are analysing for hazards, which are shown in the second column. The third column indicates the part of the system in which the hazard occurs. The fourth column indicates that all of the hazards that we focused on were safety-related. The concrete causes and consequences of the identified hazard are shown in the last two columns. ...................... 11

4.6 Risk Analysis View in AdvoCATE. For each cause that is associated with a hazard we start risk analysis by specifying the mitigations that are used to reduce the effects of the causal effect (column 4). In the last three columns, we estimate the likelihood and severity for each hazard, which are used to calculate the initial risk level per hazard. ................................. 12

4.7 Bow Tie Diagram presenting the running out of battery hazard (orange circle), its causes (blue rectangles to the left) and consequence (red rectangle to the right). ................................................................. 13

4.8 Controlled Event Structure Diagram for the loss of rover hazard in AdvoCATE. The green rectangles represent the various events such as hazards and their causes, while the white rectangles with the purple heading each represent a control mechanism that is applied to mitigate the event specified before in the event chain. ................................. 14

4.9 Controlled Event Structure Diagram for the all heatpoints not visited hazard in AdvoCATE. ................................................................. 15

4.10 Extended Inspection Rover Functional Decomposition in AdvoCATE. .... 16
4.11 Mitigations Requirement View of the Hazard log in AdvoCATE presents the mitigation requirements (column 6) associated with the specified mitigations (column 5) and the type of verification used for the corresponding requirements (column 7).

4.12 Full set of system requirements (part 1/3). Each requirement has an ID, Description and Type, the Source column points to the events from the hazard log that the requirement is associated with, while the Allocations column specifies the system component to which the requirement is allocated. The Verification Method column indicates the verification type used for the verification of the requirement, while the Verification Allocation column points to the exact evidence artifact of the verification of the corresponding requirement. We have specified the fretish version of the requirement in the Notes column.

4.13 Full set of system requirements (part 2/3).

4.14 Full set of system requirements (part 3/3).

4.15 Upgraded inspection rover architecture with additional components and data.

4.16 Top level Subsystem.

4.17 Inside the Rover Subsystem of Figure 4.16.

4.18 The content of the Navigation Subsystem.

4.19 The content of the Reasoning Agent.

4.20 The content of Battery/Interface component.

4.21 The content of Compute Plan To Destination component.

4.22 Screenshot from the fret Update Requirement widget.

4.23 The Navigation Subsystem with its contract attached.

4.24 Lustre implementation of the GRA (Goal Reasoning Agent).

4.25 The argument-fragment for the running out of battery hazard (rectangles represent goals, parallelograms represent strategies, ovals with a 'J' represent justifications, rounded rectangles represent context statements, green rectangles indicate arguments continues elsewhere, green diamonds represent currently undeveloped elements).

7.1 The top level argument for the loss of rover hazard.

7.2 The upper part of the argument assuring the running out of battery hazard.

7.3 The argument fragment for a hazard that leads to the running out of battery hazard.

7.4 The argument fragment that supports the requirement [R1].

7.5 An argument fragment for assuring the consistency of the different design modifications.

7.6 An upper part of the argument for trustworthiness in the formalization of requirement [R1] via fret.

7.7 The part of the argument for assuring the results produced by COCOSIM used for verifying requirement [R1].

7.8 The part of the argument presenting the actual results of the analysis performed through COCOSIM and Kind2 for verification of requirement [R1].

7.9 Event-B context ctx0.

7.10 Event-B context ctx1.

7.11 Event-B machine mac0.

7.12 Event-B context ctx2.

7.13 Event-B machine mac1 part 1/2.
List of Tables

4.1 Time taken by COCOSIM to compositionally verify [R1] and [R3] . . . . . 27
4.2 The verification time increased with the number of cells in the grid . . . . 31
Chapter 1

Introduction

The adoption of formal methods in industry has been slower than their development and adoption in research. One of the main pitfalls is the difficulty in integrating the results from formal methods with non-formal parts of the system development process. A central stumbling block is the formalisation of the (informal) natural language descriptions that are needed to perform the formal analysis, as well as the analysis and interpretation of the formal verification results.

The integrated formal methods approach relies on various tools cooperating to ease the burden of formal methods at various phases of system development. This often involves facilitating the use of one tool/formalism from within another (e.g. Event-B∥CSP [2]), the development of a tool/formalism that incorporates multiple others (e.g. Why3 [3], Circus [4]), or the construction of translation rules to systematically translate between tools/formalisms (e.g. EventB2JML [5]). Recent work has argued that, for autonomous robotic systems, the use of multiple formal and non-formal verification techniques is not only beneficial but actually necessary to ensure that such systems behave correctly [6, 7]. Central to this argument is the fact that the usually modular nature of robotic systems makes them more amenable to an integrated verification approach than monolithic systems [8]. The inherent modularity in robotic systems usually stems from the use of a node-based middleware such as the Robot Operating System (ROS) [9]. However, similar middlewares such as NASA’s core Flight System (cFS) [10] also support the development of similarly complex, modular systems.

In this report, we study the support for integrating formal verification results at both system- and component-level in the design, implementation and assurance of a critical system, namely, an autonomous rover undertaking an inspection mission. In contrast to usual approaches to integrating formal methods, such as those described above, we use an assurance case as the point of integration rather than building bespoke tools or defining mathematical translations between specific formal methods. In this way, we harness the benefits of an integrated approach to verification without the usual overheads. Specifically, we use AdvoCATE [11] to perform safety engineering and assurance, FRET [12] to elicit and formalize requirements, and COCOSIM [13] with Kind2 to perform compositional verification of the system-level requirements. Further, we use Event-B [14] and Kind2 for component-level formal verification. AdvoCATE facilitates the integration of the artifacts/evidence produced from these tools for use in an associated assurance case.

In summary, we contribute an inspection rover case study that demonstrates:

- how these tools can be linked via an argument in an assurance case.
- the benefit of using distinct tools due to limitations that each might have (e.g. Kind2
would time out on certain properties that were verified in Event-B).

- how developing with formal methods in mind from the outset can influence the design of the system, making it more amenable to formal verification.

The remainder of this technical report is structured as follows. Chapter 2 provides a brief overview of the tools that were used. In Chapter 3, we outline our methodology for integrating formal verification results via assurance case development. Chapter 4 describes each of the steps in our methodology and how they were applied to our inspection rover case study. We provide a detailed discussion in Chapter 5. In Chapter 6 we discuss related work. Finally, Chapter 7 concludes the paper and discusses future work directions. We also provide an Annex which contains snippets of our source code and additional figures/screenshots.
Chapter 2

Background

In this chapter, we briefly discuss the tools that we used in the case study.

2.1 Assurance Case Automation Toolset (AdvoCATE)

AdvoCATE [15] is a tool that supports the development and management of safety assurance cases. A safety assurance case is composed of all the artifacts that are created during system development to assure that the system is acceptably safe. Such a safety case is often documented in form of a graphical argument that presents how the system safety goals have been achieved and supported by the various evidence. AdvoCATE uses Goal Structuring Notation [16] to document the safety cases. To enable automation of the development and management of assurance cases, AdvoCATE is built with a formal basis where all of the artifacts relevant from the safety assurance perspective can be defined and formally related. Some of the safety assurance artifacts can be created directly in AdvoCATE (e.g., hazard log, bow tie diagrams, safety arguments), while some other artifacts such as testing results or formal verification, can be imported into the tool so that the evidence can be collectively viewed.

2.2 Formal Requirements Elicitation Tool (FRET)

FRET [12] is an open source tool [17] developed at NASA Ames for writing, understanding, formalizing, and analyzing requirements. Users enter system requirements in a restricted natural language called fretish [18]. FRET helps understanding and review of semantics by utilizing a variety of forms for each requirement: natural language description, formal logics, and informal diagrams. Requirements can be defined in a hierarchical fashion and can be exported in a variety of forms to be used by formal analysis tools such as COCOSIM [19] (see next section for a detailed description of COCOSIM).

FRET and COCOSIM are connected [20]: COCOSIM exposes model details to FRET to support the mapping between requirement- and model- variables by FRET users; FRET generates verification code that COCOSIM can process to analyze requirements against models.

The fretish language and the connection with the COCOSIM tool has previously been evaluated in the Ten Lockheed Martin Challenge Problems (LMCP) case study [19, 21], which comprises a set of industrial Simulink model benchmarks and natural language functional requirements developed by domain experts. At present, fretish is being evaluated in the context of several NASA projects but also from external software development experts and academic researchers, with and without formal methods background.
2.3 Contract-based Compositional Verification of Simulink Models (COCOSIM)

COCOSIM [13] is an open source framework [22] for Simulink/Stateflow that can be used for both code generation (e.g. C/Rust or Lustre) and formal verification. With respect to formal verification, COCOSIM translates a Simulink model into Lustre code [23], which is the intermediate language of COCOSIM. Using the Simulink API, COCOSIM iterates over Simulink blocks and produces equivalent Lustre nodes. The translation technique preserves the structural and modal behavior of the system. Then, COCOSIM annotates the generated Lustre code with assume-guarantee contracts. Verification can be performed in a compositional way by checking that component-level contracts imply those at system-level and/or by checking the component-level contracts against individual component behavior.

Different Lustre-based analysis tools integrated in the COCOSIM tool check the validity of the generated Lustre nodes, by using SMT-based model checking. In this work, we performed analysis with the Kind2 [24] model-checker that uses k-induction, IC3/PDR [25], and invariant generation [26].

COCOSIM supports most of the frequently used Simulink block libraries and is easily extensible should others wish to be added. COCOSIM has been successfully used in a number of aerospace related examples and case studies [13,19].

2.4 Event-B

Event-B [14] is a formal specification language that is used predominantly in the verification of cyber-physical systems. Event-B evolved from the B-Method [27]. It uses a set-theoretic modelling notation and supports formal refinement. In Event-B, models are composed of machines, which model the dynamic components of a systems’ specification (variables, invariants and events), and contexts, which model the static components (constants, carrier sets and axioms). Machines can refine one another which enables the user to gradually add complexity to their model. Similarly, contexts can be extended to add more detail.

Event-B has tool support via the Rodin Platform, an Eclipse-based IDE, which generates proof obligations corresponding to a given specification and provides support for automatic and interactive proof [28] with the supported provers (e.g. Atelier B). Some notable applications of Event-B include medical devices [29] and rail systems [30].
Chapter 3

The Case Study Methodology

The objective of this work is to study the integration of formal verification results via the development of an assurance case, as applied to a robotic system, using a tool palette that includes the three NASA Ames tools fret, cocosim, and AdvoCATE, as well as Event-B. To this end, we provide a step-by-step methodology that builds on top of existing NASA guidelines [31,32] that can be used in the design and development of mission-critical systems. In particular, existing guidelines [32] suggest the following phases: 1) characterization; 2) modeling; 3) specification; 4) analysis; and 5) documentation. Each phase consists of constituent processes and the overall process is iterative rather than sequential.

Our methodology focuses on the application of formal methods and connects it to parts of a greater system safety assurance methodology [33] needed to perform and assure the application of formal methods. Our methodology is guided by the need to devise a detailed assurance case that integrates verification results from a number of distinct tools. The steps that we followed are the following:

**Step 0:** Characterize initial system.
**Step 1:** Create initial system model.
**Step 2:** Perform preliminary hazard analysis.
**Step 3:** Define mitigations and safety requirements.
**Step 4:** Refine system model according to mitigations.
**Step 5:** Formalize requirements and create formal specification(s).
**Step 6:** Perform verification and simulation at system- and component-levels.
**Step 7:** Document verification results and build safety case.

Fig. 3.1 presents a detailed view of our methodology instantiated with the selected tools for the Inspection Rover case study. The upper part of Fig. 3.1 shows the system-level concept, design, and assurance steps that are mainly performed by the AdvoCATE tool (steps 2–3 and step 7), while the lower part shows the formal methods application steps performed by the FRET (step 5), COCOSIM (steps 1, 4, and 6), and Event-B tools (steps 1 and steps 4–6).

These steps belong to the five development phases [32]. For instance, as indicated at the bottom of Fig. 3.1, step 0 belongs to the characterization phase, while step 5 belongs to the specification phase. In the analysis phase (step 6) we perform two types of analysis. We use COCOSIM to perform compositional system-level analysis with Kind2. We also perform verification at component-level against the system model using the Atelier-B and Kind-2 tools. Finally, in the documentation phase (step 7), we use AdvoCATE to integrate the evidence produced by the tools within the assurance case.

Over the years, we have worked with a variety of formal approaches for the assurance
Figure 3.1: Our methodology for integrating verification results via an assurance case instantiated with the selected tools for the Inspection Rover case study. The incoming arrows without a source represent all relevant artifacts from previous phases. For system-level analysis these comprise the Lustre requirements and the Simulink system model, while for component-level analysis these comprise the Lustre and Event-B system models and requirements. In the documentation phase we input all artifacts.

of safety-critical systems. The goal of this study is to explore how such approaches can work together and be integrated within the development process of an autonomous system. With this aim, we developed a case study of a rover system. Our case study is not extracted from an actual mission. Rather, it is developed by iteratively using our expertise on various assurance approaches. The resulting Inspection Rover case study has a reasonable complexity, and demonstrates a variety of generic challenges in formal methods techniques and their integration. Most importantly, we make the details of our case study publicly available, since we believe that it can serve as a good basis for discussion and comparison of approaches and tools across the research community.

We target rovers for a variety of reasons. First, rovers are used in many autonomous systems, and present challenges that are typical of autonomous applications. Second, some of the authors have prior experience with autonomous robotic systems that are deployed in hazardous environments, such as the nuclear, offshore, and space domains through their involvement in projects\(^1\). Third, our research group at NASA Ames is in the process of building rover applications to experiment with AI technologies and their assurance techniques.

Four formal methods experts were involved: 1) a safety expert; 2) a requirements expert; 3) a Simulink and Lustre verification expert; and 4) a verification expert of robotic systems that also served as the domain expert. Step 0 was performed by the domain expert, step 1 was performed together by the Simulink and domain experts. Steps 2 and 3 were performed by the safety expert. Steps 4 and 6 were performed together by the safety, domain and Simulink experts. Step 5 was performed together by the requirement and domain experts, and finally step 7 was performed mainly by the safety expert with contributions from all others.

\(^1\)UKRI and EPSRC Hubs for “Robotics and AI in Hazardous Environments”.
Chapter 4

The Case Study Step-by-Step

In this chapter, we describe how we followed the step-by-step methodology that we defined in Chapter 3 (Fig. 3.1) with respect to our inspection rover case study.

4.1 Step 0: Characterize Initial System

We performed our case study in the context of the navigation system for an autonomous rover undertaking an inspection mission. The objective of this rover is to explore a square grid of known size and to autonomously navigate to points of interest whilst avoiding obstacles and recharging as necessary. We assumed that this system would be operated indoors to minimize environmental uncertainty. Our preliminary outline sketch of this system architecture is illustrated in Fig. 4.1.

The rover’s goal is to navigate to all heat positions on a 2D grid map of a given size. The Vision system is used by the rover to detect obstacles that it should avoid. The Infrared component identifies grid locations that are hotter than expected. Based on these heat locations, the autonomous Goal Reasoning Agent selects the hottest location as the goal, unless the Battery Monitor (via the Interface) indicates that it must recharge. The Planner returns a set of obstacle-free plans for navigating from the current position to the goal. The autonomous Plan Reasoning Agent selects the shortest plan. Finally, the Interface translates the navigation actions of the plan into the instructions required by the hardware components and alerts the Goal Reasoning Agent when it reaches the goal or that it does not have enough battery to execute the chosen plan so it must recharge.

4.2 Step 1: Create Initial System Model

We began with the initial system overview as illustrated in Fig. 4.1. We then produced the AADL model as shown in Fig. 4.2. We used AADL at this stage because it is a widely-used tool for designing system architectures.

Fig. 4.2 contains an AADL diagram describing the components in the rover system and the communication links between them. This diagram was automatically generated from the AADL specification contained in the Annex. Hardware components are indicated as rectangles, software components are indicated as parallelograms and arrows indicate data transmission between components.

As part of the process of using AADL we also included hardware components such as the Camera that provides input to the Vision component and the HeatSensor which provides input to the Infrared component. These hardware components are referred to using the
\( i_{I} \): from camera, \( n \in \mathbb{N} \) (length of square area to be navigated)

\( i_{V} \): \( s_0 = (x, y) \), \( \text{Obs} = \{(x, y)\} \)

set of obstacle locations

\( i_{V} \): \( s_0 = (x, y) \), \( \text{Obs} = \{(x, y)\} \)

\( i_{H} \): \( H = \{(x, y)\} \)

\( i_{G} \): \( G = \{(x, y)\} \)

\( i_{P} \): \( P = \{(x, y)\} \)

\( i_{B} \): \( b \in \mathbb{N} \)

\( i_{I} \): \( \text{haveMoved} \in \{\text{true, false}\} \), \( \text{atGoal} \in \{\text{true, false}\} \)

\( i_{V} \): \( i_{V} \cup i_{H} \cup i_{G} \cup i_{P} \cup i_{B} \cup i_{A} \cup i_{I} \)

\( i_{V} \): \( i_{V} \cup i_{H} \cup i_{G} \cup i_{P} \cup i_{B} \cup i_{A} \cup i_{I} \)

\( i_{V} \): \( \text{from camera/sensor, } n \in \mathbb{N} \)

\( i_{H} \): \( H = \{(x, y), h\} \)

\( i_{G} \): \( G = \{(x, y)\} \)

\( i_{P} \): \( P = \{(x, y)\} \)

\( i_{B} \): \( b \in \mathbb{N} \)

\( i_{I} \): \( \text{haveMoved} \in \{\text{true, false}\} \), \( \text{atGoal} \in \{\text{true, false}\} \)

Figure 4.1: An overview of the robotic system to be verified. Each rectangle represents a node in the robotic system and each arrow represents data flow between nodes.

device keyword and software components are identified by the process keyword. These were not present in our original system diagram in Fig. 4.1 but are present in the final system so we include them here.

We assume that \( n \) (size of the grid) is a global variable so it is not marked as input anywhere. Note that we also model the MainProcessor and use bus_access to represent communication links between the hardware and software components.

We outline below the variables used in the AADL model:

- **camera**: contains the images that are recorded by the camera.
- **Obs**: is a set of obstacle locations as detected by the Vision node.
- **s0**: is the current position of the rover.
- **sensorinp**: is the input to the Infrared node from the heat sensor.
- **H**: is the set of heat locations (pair containing coordinate and temperature measured at that point).
- **recharge**: is a boolean flag that is toggled when the rover needs to recharge.
- **atGoal**: is a boolean flag that is toggled when the rover has reached its goal location.
- **g**: the current goal location.
- **PlanSet**: is a set of plans between the current \( s_0 \) and \( g \).
- **b**: is the battery remaining.
- **plan**: is the current plan chosen for execution.
- **sensorInp**: is the input from the battery to the Battery node which measure how much battery is remaining.
- **haveMoved**: is a boolean flag that is toggled when the rover has successfully executed a movement.

This AADL model provided a useful point of reference as we began to characterise the initial system and its components.

After constructing the AADL model described previously, we realised that compositional
verification approaches for AADL, such as AGREE [34] would not be expressive enough to adequately capture the type of properties that we were interested in. Thus, we constructed a Simulink model of the system as illustrated in Fig. 4.3. We also created an Event-B model of the system and automatically generated the Lustre model via COCOSIM.

Simulink [35] is usually used by engineers for developing an executable specification of the whole system, which the user may use to simulate and validate before producing the source code. However, in this case study, we used Simulink in two different ways: 1) as an architecture description language, which allowed us to specify the architecture of the rover without providing implementation of low-level components (for compositionally verifying properties using assume-guarantee reasoning); 2) as a behavioral specification language for the implementation of some of the low-level components (for checking properties against component behavior).

4.3 Step 2: Perform Preliminary Hazard Analysis

To perform the preliminary hazard analysis in AdvoCATE as part of the safety assurance methodology [33], we defined a functional decomposition of the Inspection Rover based on Fig. 4.3. This functional decomposition is shown in Fig. 4.4 and it was used as the basis for
the traditional hazard analysis (FMEA [36]) that we documented in the AdvoCATE hazard log. A snapshot of the Hazard Log table from the Hazard Identification View is shown in Fig. 4.5. We further analysed the risk, based on the likelihood of the hazard occurring and the associated severity/impact, as shown via the risk analysis view in AdvoCATE (Fig. 4.6).

We specifically identified two top-level hazards: 1) loss of rover, and 2) inspection finished before visiting all of the heatpoints. In total, we identified 25 hazards including these two. For example, we identified the running out of battery and collision with an obstacle hazards as causes of loss of rover. AdvoCATE uses the information from the hazard log to automatically create a safety architecture documented via interconnected Bow Tie Diagrams (BTD) for each hazard [37]. A single BTD shown in Fig. 4.7 details the causes and consequences of the running out of battery hazard.

The parts of the Hazard Log that are shown in Figures 4.5 and 4.6 focus on the hazardous activity called Manoeuvring towards a goal, where the top hazard is loss of rover. When it comes to the hazardous activity called Finishing inspection, the top hazard is inspection finished before visiting all the heatpoints. Since not all heatpoints may be reachable in practice, we focus here on ensuring that each reachable heatpoint is visited.
Figure 4.5: Hazard Identification View in AdvoCATE. The first column indicates the particular rover’s activity that we are analysing for hazards, which are shown in the second column. The third column indicates the part of the system in which the hazard occurs. The fourth column indicates that all of the hazards that we focused on were safety-related. The concrete causes and consequences of the identified hazard are shown in the last two columns.
Figure 4.6: Risk Analysis View in AdvoCATE. For each cause that is associated with a hazard we start risk analysis by specifying the mitigations that are used to reduce the effects of the causal effect (column 4). In the last three columns, we estimate the likelihood and severity for each hazard, which are used to calculate the initial risk level per hazard.
4.4 Step 3: Define Mitigations and Safety Requirements

After preliminary hazard analysis, we conducted a risk analysis (Fig. 4.6) that qualitatively analysed the severity and likelihood of the identified hazards to estimate the risk level. From this, we defined mitigations to minimize the risk of those hazards and their consequences. E.g., the loss of rover hazard is characterized with catastrophic severity, but its likelihood is calculated based on the events causing it. The combination of the two defines the risk associated with the hazard.

Next, we describe mitigations against the identified hazards. While the Hazard Log includes all of the hazard and risk assessment information, its tabular form does not offer a good holistic overview of the safety architecture of the system. To achieve that, complementary graphical representations in the form of Controlled Event Structure (CES) and Bow-Tie Diagrams are automatically generated from the Hazard Log. In fact, those diagrams are more suited to perform mitigations planning than the Hazard Log itself.

In particular, Fig. 4.7 contains a BTD corresponding to the running out of battery hazard. For example, in order to minimize the risk of running out of battery: (1) we formally analysed the navigation system and battery controller, (2) we ensured that the charging station position is predefined so that we can estimate at every point whether we have enough battery to go to recharge, and (3) if the basic assumptions about battery consumption are violated, then we abort and return to the charging station. Besides mitigating the causes to prevent the hazard from happening, we add the recovery barrier between the hazard and the consequence to reduce the severity of the consequence in the case that the hazard still occurs.

While BTDs describe each hazard individually, a holistic overview of all of the hazards,
Figure 4.8: Controlled Event Structure Diagram for the loss of rover hazard in AdvoCATE. The green rectangles represent the various events such as hazards and their causes, while the white rectangles with the purple heading each represent a control mechanism that is applied to mitigate the event specified before in the event chain.
their relations and the applied mitigations is shown in the corresponding CES diagram. Fig. 4.8 presents the CES diagram with all of the events leading to loss of rover, including the employed mitigations. A similar diagram is created for the inspection finished before visiting all the heatpoints hazard in Fig. 4.9. For the rover to visit all of the heatpoints, the loss of rover hazard should also not occur before visiting all of the heatpoints. Hence, the loss of rover hazard is used as a cause for the all heatpoints not visited hazard. However, since the safety architecture for the loss of rover hazard is presented in the first CES diagram, the second CES diagram is shortened so that it does not fully decompose the safety architecture behind the loss of rover hazard.

Since some of the mitigations required design modifications. As a result, we updated the corresponding functional decomposition accordingly. The extended functional decomposition is shown in Fig. 4.10.

The safety mitigation requirements that resulted from the mitigation planning are added to the Hazard Log as shown in the Mitigation Requirements View in Fig. 4.11. The requirements added through the Hazard Log are at the same time visible in a separate requirements table that shows all the requirements specified for the system. The parts of the requirements table are shown in Figs. 4.12 - 4.14.

The information about the requirements is gathered as the system development process progresses. When we start creating mitigation requirements during mitigation planning in the Hazard Log, the requirements will have minimal details associated with them. For example, description, type and source are usually the first fields filled in for a requirement created though Hazard Log. To formalize these requirements and perform verification, we export the initial set of natural language mitigation requirements in excel and import them into FRET to perform the formalization. Once we have performed the formalization and verification, we import that information back into AdvoCATE. The final column in the requirements table represents the FRETISH representation of the corresponding natural language requirement which will be described in Section 4.6.

For each of the two top-level hazards, loss of rover and inspection finished before visiting all the heatpoints, the corresponding CES diagrams are shown in Fig. 4.8 and Fig. 4.9. The diagrams illustrate the causal relationships between the events and the mitigations applied to manage the hazards. The diagrams are created using AdvoCATE, a tool for safety management and validation.

Figure 4.9: Controlled Event Structure Diagram for the all heatpoints not visited hazard in AdvoCATE.
all of the heatpoints, we define system-level requirements:

- [R1]: The rover shall not run out of battery.
- [R2]: The rover shall not collide with an obstacle.
- [R3]: The rover shall visit all reachable heat points.

The requirements [R1] and [R2] correspond to the causes of loss of rover, while [R3] relates to the inspection finished before visiting all of the heatpoints hazard. We have decomposed these system-level requirements further into child (component-level) requirements detailing the specific mitigation mechanisms captured in the BTDs. For example, the mitigations from Fig. 4.7 are related to the child requirements of [R1], while [R3] scopes which heat points should be visited to those that are reachable and not visited before.

Based on the defined mitigations, we specified the corresponding mitigation requirements for each of the hazards and their causes. The full set of system requirements addressing the identified hazards is shown in Figures 4.12, 4.13 and 4.14. In this AdvoCATE view, the verification allocation column describes which verification artifacts/evidence is produced during the verification of particular requirements.

### 4.5 Step 4: Refine System Model According to Mitigations

Some of the identified mitigations required design modifications resulting in a refined system architecture (Fig. 4.15). The initial rover position and a static charging station position are given as user input. Note that the charging station position is static and the rover always starts its missions from a pre-defined initial position.

We modified the original architecture, adding the `MapValidator` component to check that the initial position, charging position, obstacles and heat points are valid, i.e., that the initial position and the charging station position are not obstacles or heat points, as well as that the obstacle and heat point sets are mutually exclusive. Furthermore, the `MapValidator` checks that the initial position, as recognized by the `Vision` component, is in
Figure 4.11: Mitigations Requirement View of the Hazard log in AdvoCATE presents the mitigation requirements (column 6) associated with the specified mitigations (column 5) and the type of verification used for the corresponding requirements (column 7).
<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Type</th>
<th>Source</th>
<th>Allocation</th>
<th>Verification Method</th>
<th>Verification Allocation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>The rover shall not run out of battery</td>
<td>Safety</td>
<td>Loss of rover</td>
<td>navigation: Navigation subsystem</td>
<td>VM1: Formal Analysis</td>
<td>Kind2: Kind2 verification of the Navigation System</td>
<td>Rover shall always satisfy battery &gt; 0</td>
</tr>
<tr>
<td>R1.1</td>
<td>Charging station location shall be static</td>
<td>Safety</td>
<td>Running out of battery</td>
<td>rover: InspectionRover</td>
<td>VM2: Testing</td>
<td>GRA shall always satisfy chargePosition = x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(goal=chargePosition)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1.2</td>
<td>Charging station shall be selected as the next destination whenever the recharge flag is set to true</td>
<td>Safety</td>
<td>Running out of battery inspection</td>
<td>GRA: Goal Reasoning Agent</td>
<td>VM1: Formal Analysis</td>
<td>Kind2: Kind2 verification of the Navigation System</td>
<td>GRA shall always satisfy recharge =&gt; (goal=chargePosition)</td>
</tr>
<tr>
<td></td>
<td>mitigation B13 of Hazard E123</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mitigation B12 of Hazard E11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1.3</td>
<td>Battery monitor shall show battery charge 5% lower than currently estimated</td>
<td>Safety</td>
<td>Running out of battery</td>
<td>batteryMonitor: Battery monitoring subsystem</td>
<td>VM2: Testing</td>
<td>BatteryMonitor shall always satisfy battery = pre_battery - 1/n - 0.05</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mitigation B6 of Hazard E15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1.4</td>
<td>The Interface recharge output shall be set to true when the current battery charge is lower than battery needed to reach the charging station from the current position plus 5% battery charge</td>
<td>Safety</td>
<td>Running out of battery inspection</td>
<td>Interface: Interface controller</td>
<td>VM1: Formal Analysis</td>
<td>Kind2: Kind2 verification of the Navigation System</td>
<td>Interface shall always satisfy ((0.95 * battery) &lt;= chargeNeeded(plan)) =&gt; recharge</td>
</tr>
<tr>
<td></td>
<td>mitigation B12 of Hazard E11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mitigation B6 of Hazard E15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1.5</td>
<td>Once at charging station, rover shall remain there until battery reaches full charge</td>
<td>Safety</td>
<td>Loss of rover</td>
<td>Interface: Interface controller</td>
<td>VM2: Testing</td>
<td>Interface shall always satisfy (goal=chargePosition &amp; atGoal) =&gt; batteryFull</td>
<td></td>
</tr>
<tr>
<td>R1.6</td>
<td>Each step in the plan shall not use more than 1/n amount of battery</td>
<td>Safety</td>
<td>Running out of battery</td>
<td>Interface: Interface controller</td>
<td>VM2: Testing</td>
<td>Interface shall always satisfy (battery = pre_battery - 1/n * length(plan))</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.12: Full set of system requirements (part 1/3). Each requirement has an ID, Description and Type, the Source column points to the events from the hazard log that the requirement is associated with, while the Allocations column specifies the system component to which the requirement is allocated. The Verification Method column indicates the verification type used for the verification of the requirement, while the Verification Allocation column points to the exact evidence artifact of the verification of the corresponding requirement. We have specified the FRETISH version of the requirement in the Notes column.
<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Type</th>
<th>Source</th>
<th>Allocation</th>
<th>Verification Method</th>
<th>Verification Allocation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2.1</td>
<td>Initial rover position shall be static</td>
<td>Safety</td>
<td>Collision with an obstacle</td>
<td>rover; InspectionRover</td>
<td></td>
<td></td>
<td>Navigation shall always satisfy InitialPosition = y</td>
</tr>
<tr>
<td>R2.2</td>
<td>Current rover position shall be accurate</td>
<td>Safety</td>
<td>Collision with an obstacle</td>
<td>rover; InspectionRover</td>
<td>VM2: Testing</td>
<td></td>
<td>Navigation shall always satisfy currentPosition = currentPhysicalPosition</td>
</tr>
<tr>
<td>R2.2.2</td>
<td>The Vision subsystem shall ensure that the initial current rover position as recognised by the Vision is equal to the static initial position</td>
<td>Safety</td>
<td>Collision with an obstacle</td>
<td>vision: Vision subsystem</td>
<td></td>
<td></td>
<td>Map_Validator shall immediately satisfy start = s0</td>
</tr>
<tr>
<td>R2.2.3</td>
<td>The Vision subsystem shall ensure that the current position is not a position with an obstacle</td>
<td>Safety</td>
<td>Collision with an obstacle</td>
<td>vision: Vision subsystem</td>
<td></td>
<td></td>
<td>Vision shall always satisfy</td>
</tr>
<tr>
<td>R2.3</td>
<td>The path to the next destination shall be calculated from the current rover position</td>
<td>Safety</td>
<td>Collision with an obstacle</td>
<td>GRA: Goal Reasoning Agent computePlan: Compute plan</td>
<td>VM1: Formal Analysis</td>
<td>Kind2: Kind2 verification of the Navigation System</td>
<td>&quot;forall plan in PlanSet =&gt; exists start in plan&quot;</td>
</tr>
<tr>
<td>R2.4</td>
<td>All obstacles shall be correctly identified</td>
<td>Safety</td>
<td>Path to destination includes an obstacle</td>
<td>rover; InspectionRover</td>
<td></td>
<td></td>
<td>&quot;forall o in Obl =&gt; obstaclePhysicalyDetected(o)&quot;</td>
</tr>
<tr>
<td>R2.4.1</td>
<td>An obstacle shall not be in the location of the initial rover position</td>
<td>Safety</td>
<td>Collision with an obstacle</td>
<td>Path to destination includes an obstacle</td>
<td>mapValidator: Map validation subsystem</td>
<td></td>
<td>&quot;forall o in Obs =&gt; o != s0&quot;</td>
</tr>
<tr>
<td>R2.4.2</td>
<td>An obstacle shall not be in the location of the charging station</td>
<td>Safety</td>
<td>Collision with an obstacle</td>
<td>Path to destination includes an obstacle</td>
<td>mapValidator: Map validation subsystem</td>
<td></td>
<td>&quot;forall o in Obs =&gt; o != chargingPosition&quot;</td>
</tr>
<tr>
<td>R2.4.3</td>
<td>An obstacle shall not be in the same location as a heatpoint</td>
<td>Safety</td>
<td>Collision with an obstacle</td>
<td>Path to destination includes an obstacle</td>
<td>mapValidator: Map validation subsystem</td>
<td></td>
<td>&quot;forall o in Obs =&gt; !heat(o)&quot;</td>
</tr>
<tr>
<td>R2.5</td>
<td>The calculated path to destination shall not include a location with an obstacle</td>
<td>Safety</td>
<td>Collision with an obstacle</td>
<td>planner: Planner subsystem</td>
<td>VM1: Formal Analysis</td>
<td>Event-B: Event-B verification of the Planner and PRA components</td>
<td>&quot;forall plan in PlanSet (forall o in plan =&gt; obstacles(o))&quot;</td>
</tr>
<tr>
<td>R2.6</td>
<td>Rover shall not go faster than 10kmph</td>
<td>Safety</td>
<td>Loss of rover</td>
<td>rover; InspectionRover</td>
<td></td>
<td></td>
<td>Rover shall always satisfy speed &lt;= 10 km</td>
</tr>
</tbody>
</table>

Figure 4.13: Full set of system requirements (part 2/3).
<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Type</th>
<th>Source</th>
<th>Allocation</th>
<th>Verification Method</th>
<th>Verification Allocation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>The rover shall visit all reachable heatpoints</td>
<td>Safety</td>
<td>Inspection finished before visiting all the heatpoints</td>
<td>navigation: Navigation subsystem</td>
<td>VM1: Formal Analysis</td>
<td>Kind2: Kind2 verification of the Navigation System</td>
<td><em>forall h \in H. reachable(h) =&gt; visited(h)</em></td>
</tr>
<tr>
<td>R3.1</td>
<td>All heatpoints shall be correctly identified</td>
<td>Safety</td>
<td>Inspection finished before visiting all the heatpoints</td>
<td>rover: InspectionRover</td>
<td></td>
<td></td>
<td><em>forall h \in H =&gt; heatPhysicallyMeasured(h)</em></td>
</tr>
<tr>
<td>R3.1.1</td>
<td>A valid heatpoint position shall not be the same as the initial rover position</td>
<td>Safety</td>
<td>Inspection finished before visiting all the heatpoints</td>
<td>mapValidator: Map validation subsystem</td>
<td></td>
<td></td>
<td><em>forall h \in H =&gt; h \neq s0</em></td>
</tr>
<tr>
<td>R3.1.2</td>
<td>A valid heatpoint position shall not be the same as the charging station position</td>
<td>Safety</td>
<td>Inspection finished before visiting all the heatpoints</td>
<td>mapValidator: Map validation subsystem</td>
<td></td>
<td></td>
<td><em>forall h \in H =&gt; h \neq chargingPosition</em></td>
</tr>
<tr>
<td>R3.1.3</td>
<td>A valid heatpoint shall not be on the same location as a valid obstacle</td>
<td>Safety</td>
<td>Inspection finished before visiting all the heatpoints</td>
<td>mapValidator: Map validation subsystem</td>
<td></td>
<td></td>
<td><em>forall h \in H =&gt; obstacle(h)</em></td>
</tr>
<tr>
<td>R3.2</td>
<td>Visited heatpoint shall be removed from the list of destinations to visit</td>
<td>Safety</td>
<td>Inspection finished before visiting all the heatpoints</td>
<td>GRA: Goal Reasoning Agent</td>
<td></td>
<td>if atGoal GRA shall after 1 tick satisfy removeGoalFromSet</td>
<td></td>
</tr>
<tr>
<td>R3.3</td>
<td>The hottest heatpoint that was not visited before shall be the current goal when recharge flag is false</td>
<td>Safety</td>
<td>Inspection finished before visiting all the heatpoints</td>
<td>GRA: Goal Reasoning Agent</td>
<td></td>
<td><em>atGoal = false =&gt; (forall h \in H \Rightarrow heat(goal) &gt; heat(h))</em></td>
<td></td>
</tr>
<tr>
<td>R3.4</td>
<td>The shortest path to the current goal shall be selected</td>
<td>Safety</td>
<td>PRA: Plan Reasoning Agent</td>
<td>VM1: Formal Analysis</td>
<td>Event-B: Event-B verification of the Planner and PRA components</td>
<td><em>forall p \in PlanSet =&gt; length(plan) = length(p)</em></td>
<td></td>
</tr>
<tr>
<td>R3.5</td>
<td>The Interface subsystem shall set atGoal flag to true only when the current rover position is equal to the current goal position.</td>
<td>Safety</td>
<td>Mitigation B13 of Hazard EP16</td>
<td>interface: interface controller</td>
<td></td>
<td>interface shall always satisfy atGoal =&gt; currentPosition = goal</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.14: Full set of system requirements (part 3/3).
Next, we defined the NavigationSystem which contains the ReasoningAgent and the Battery_Interface components. We emphasise these two components as we focus on formally verifying them. We further decompose these components.

The ReasoningAgent takes as input the identified and validated obstacle locations, the current rover position, the heat points and the position of the charger. It outputs three things: (1) a plan from the current position to the goal ($plan_{2D}$), (2) a plan from the goal to the charger location ($plan_{2C}$), and (3) the list of visited locations which it is responsible for keeping track of. Within the ReasoningAgent, the goal reasoning agent (GRA) chooses the goal as either the next hottest heat point or as the charging location if the recharge flag has been set to true by the Battery_Interface. The GRA updates the visited locations.

Further, the ReasoningAgent contains ComputePlan2Charging and ComputePlan2Destination which both contain a Planner and plan reasoning agent (PRA). These are essentially two copies of the same thing but we split them for simplicity. Specifically, ComputePlan2Charging returns the shortest plan from the goal to the charger, whilst ComputePlan2Destination returns the shortest plan from the current position to the goal.

The Battery_Interface is composed of a BatteryMonitor and a hardware Interface. The Interface takes the plans from NavigationSystem and the battery status from the Battery-Monitor as inputs, and returns two flags indicating whether the rover has reached the goal ($atGoal$) and the status of the battery charge ($recharge$). The recharge output is set to true when the current battery charge is not sufficient to follow the plan to the goal ($plan_{2D}$) and then return to the charging station ($plan_{2C}$).

If the recharge flag is false, then the Interface executes the plan and returns $atGoal$ as true once it reaches the goal. However, if the recharge flag is true and the rover first needs to recharge, then the Interface sets $atGoal$ to false. We note that we actually do not need both of these outputs since we have always $recharge \Rightarrow \neg atGoal$. However, we include both for simplicity. These outputs are fed back to the ReasoningAgent that generates the next plan, and this loop executes until all of the heat points have been visited. Note that we assume that the frequency of execution for the NavigationSystem is similar to the Interface.

For illustrative purposes, we include the top-level Simulink subsystem in Fig. 4.16, the subsystem takes as input the infrared and camera values and outputs the battery status and the visited array that maps each index (cell position) to a boolean value denoting whether the cell has been visited.

Fig. 4.17 shows the content of the Rover Subsystem. Here, the Vision component...
starts to goal. We compute two plans, one from start to goal and the second from goal to start points and the tery all the heatpoints flag is sent back to the Reasoning Agent produces the plan to destination and updates the visited signal, whereas the obstacles Infrared Subsystem generates the obstacles and the current position, the Navigation Subsystem is executed to visit all the heatpoints. The Navigation System is then executed to visit all the heatpoints without running out of battery and hitting an obstacle.

The content of the Navigation Subsystem is illustrated in Fig. 4.18, the Reasoning Agent produces the plan to destination and updates the visited signal, whereas the battery_interface is executing the plan if the amount of the battery allows it, if not the recharge flag is sent back to the Reasoning Agent to compute a plan to charging position instead.

The content of the Reasoning Agent is shown in Fig. 4.19, the GRA computes the goal and start points and the Compute Plan To Destination computes the shortest path from start to goal. We compute two plans, one from start to goal and the second from goal to
charging position, the latter is used later by the Battery/Interface component to check if the rover have enough battery to go to goal and then back to charging position if needed. The content of the Battery/Interface component is in Figure 4.20.

The content of Compute Plan To Destination is presented in Fig. 4.21. First, the Planner computes all possible plans from start to goal, then the PRA chooses the shortest path from start to goal that was computed by the Planner.

Figure 4.19: The content of the Reasoning Agent.

Figure 4.20: The content of Battery/Interface component.

Figure 4.21: The content of Compute Plan To Destination component.
4.6 Step 5: Formalize Requirements and Create Formal Specifications

We manually encoded the requirements in the restricted natural language of FRET, i.e., FRETISH, which has a precise, unambiguous meaning. A FRETISH requirement contains up to six fields: scope, condition, component*, shall*, timing, and response*, where mandatory fields are indicated by an asterisk. ‘component’ specifies the component that the requirement refers to. ‘shall’ is used to express that the component’s behavior must conform to the requirement. ‘response’ is a Boolean condition that the component’s behavior must satisfy. ‘scope’ specifies intervals where the requirement is enforced. For instance, ‘scope’ can specify system behavior before a mode occurs, or after a mode ends, or when the system is in a mode. The optional ‘condition’ field is a Boolean expression that triggers the need for a ‘response’ within the scope. When triggered, the response must occur as specified by field timing, e.g., immediately, always, after/for/within N time units.

A FRET requirement template is defined by a template key with values for fields [scope, condition, timing]. For example, [in, null, always] identifies requirements of the form In M mode, the software shall always satisfy R. Condition null (as opposed to regular), means that the response is triggered at the beginning of each scope interval. The most common key is [null, null, always], i.e., The software shall always satisfy R. Scope null indicates global scope, which means that the requirement is enforced on the entire execution interval. At the time of this case study, FRETISH supported 8 values for field mode, 2 values for field condition, and 7 values for field timing, for a total of $8 \times 2 \times 7 = 112$ semantic templates. More details on FRETISH and its semantics are available in [18].
The majority of the requirements that we formalized did not have scope or condition but they did have *always* timing, e.g.:

[R1]: Navigation shall *always* satisfy battery > 0.

Other requirements use the *condition* field and *immediately* timing, e.g.:

[R1.2]: if recharge GRA shall *immediately* satisfy goal = chargePosition.

In Fig. 4.22, a screenshot of the FRET update requirement widget is shown. A user may define a requirement ID (e.g., R1.2), declare parent-child relations between requirements (e.g., R1 is the parent of requirement R1.2), and specify a rationale or comments for each requirement. The rationale is usually the natural language description of a requirement. Once a FRETISH requirement is entered and correctly parsed by the requirement editor, the semantics of the requirement are generated in various forms, as shown in the right hand side of Fig. 4.22. In particular, FRET produces natural language and diagrammatic explanations of its exact meaning, and formalizes the requirement in temporal logic. For example, notice that in the generated semantics for requirement [R1.2], if recharge is a “trigger”: the requirement is only enforced when the condition becomes true from false. The use of ‘immediately’ states that the response must hold simultaneously with each trigger point. The natural language version of [R1] was previously presented in §4.4, while the natural language version of [R1.2] is “Charging station shall be selected as the next destination whenever the recharge flag is set to true” (also shown in the rationale field of Fig. 4.22).

Some requirements needed first-order temporal logic, which is not currently supported in FRET. For these, we used auxiliary variables that we instantiated with quantifiers at the Lustre level. For instance, the natural language version of requirement [R3.3] is “The hottest heatpoint that was not visited before shall be the current goal when recharge flag is false” was written in FRETISH as follows:

[R3.3]: GRA shall *always* satisfy if ! recharge then (if forAll_i & i_inGrid then (if ! visited[i] then heatpoints[goal] > heatpoints[i]))

where forAll_i represents the universal quantification over heatpoints. In total, our case study contains 28 requirements, 7 of these required first-order temporal logic formulae. We were able to write all 28 requirements in FRETISH and formalize them (we used auxiliary variables for first order logic quantifiers).

**FRETISH to Verification Code:** FRET automatically formalizes requirements in pure future-time (fmLTL) and pure past-time (pmLTL) Linear Temporal Logic. pmLTL formulae exclusively use past-time temporal operators, i.e., Y, 0, H, S (meaning Yesterday, Once, Historically, Since, respectively). We used the pmLTL variant since Lustre-based analysis tools only accept pmLTL specifications. The automatically generated pmLTL formulae for [R1] and [R1.2] are:

[R1]: H(battery > 0);

[R1.2]: H((recharge & (Y(!recharge) | FTP)) ⇒ (goal = chargePosition));

where FTP means First Time Point of execution (equivalent to ¬ Y TRUE). From the pmLTL formulae we automatically generated Lustre-based verification contracts that can be directly fed into COCOSIM for verification with the Kind2 model checker. The full process is described in detail in [20]. For example, below is the generated Lustre code for [R1] and [R1.2]:

```luster
1 guarantee "R1" (battery > 0);
1 guarantee "R1.2" ((recharge and ((pre (not recharge)) or FTP)) ⇒ (goal = chargePosition));
```
Figure 4.23: The Navigation Subsystem with its contract attached.

where FTP = true → false. If requirements were based only on model inputs, then FRET generates assumptions (instead of guarantees) As mentioned earlier, some requirements used first-order logic quantification such as \([R3.3]\) which was generated as follows:

\[
\text{guarantee "R3.3" not recharge} \Rightarrow (\forall (i:\text{int}) (0 \leq i \text{ and } i < \text{width}) \Rightarrow (\text{not visited}[i] \Rightarrow \text{heatpoints}[\text{goal}] \geq \text{heatpoints}[i]));
\]

Notice that the \(\forall i\) placeholder was replaced by \(\forall (i:\text{int})\), and \(i\text{ inGrid}\) was replaced by \((0 \leq i \text{ and } i < \text{width})\) during generation of the Lustre code.

The Lustre contracts that were generated from FRET can be directly fed (together with traceability information) into the COCOSIM tool. COCOSIM then translates the Lustre contracts into Simulink components and uses the traceability information, which is also generated from FRET, to automatically attach these Simulink contracts at the correct hierarchical level of the model. Fig. 4.23 shows the Navigation Subsystem with its contract attached.

Additionally, we specified the requirements in Event-B. Event-B does not support temporal logic but we used the FRETISH requirements to guide our Event-B modelling since they were simple enough and more useful as a starting point for formalization than the natural language requirements. E.g., the natural language requirement \([R3.4]\) is “The shortest path to the current goal shall be selected”. The FRETISH version is: \([R3.4]\): Planner shall always satisfy if \((\text{planningCompleted} \& \text{returnPlan})\) then \((\text{if} (\forall x \& x\text{inPlanSet})\) then \((\text{card(chosenPlan)} <\text{card(x)})\)), where the \(\text{card()}\) function computes the length of a path. The corresponding Event-B invariant was based on the FRETISH version:

\[
(\text{planningCompleted} = \text{TRUE}) \land (\text{returnplan} = \text{TRUE}) \Rightarrow (\forall x \cdot x \in \text{PlanSet} \Rightarrow \text{card(chosenplan)} \leq \text{card(x)})
\]

Similarly, \([R2.5]\): The calculated path to destination shall not include a location with an obstacle. This was defined in Event-B as follows:

\[
\forall p, x \cdot p \in \text{PlanSet} \land x \in p \Rightarrow x \notin \text{Obs}
\]

where every element of \(\text{PlanSet}\) is a set of grid locations.

4.7 Step 6: Perform Verification and Simulation at System- and Component-Levels

In this section, we describe how we performed compositional verification of the system- level requirements and used more classical, non-compositional, verification approaches for
component-level verification.

4.7.1 Compositional Verification in CoCoSim

Our objective was to attach the component-level child requirements to the relevant component(s) and then to compositionally verify, using COCOSIM, that the system-level parent requirements hold. There were some requirements that we could not formally verify. For example, a requirement stating that the current position as recognized by the rover is its current physical position, cannot be formally modelled or verified but rather needs to be physically tested.

COCOSIM has a front-end that translates a Simulink model into Lustre code, the latter can be verified by Kind2. COCOSIM comes with specification blocks to annotate the model with properties (Assume and Guarantees). It also supports Simulink Subsystems with no implementation and the Lustre node associated to that Subsystem will be translated as imported node. Therefore, COCOSIM allows us to use Simulink as a high level architecture language, where you can add your components, describe the connections between components (Datatype, dimensions of signals) and attach contracts to each components describing the component behaviour. COCOSIM can then verify top level properties against the sub-components contracts.

Compositional verification in COCOSIM involves defining a top system node with associated system-level assumptions and guarantees. During verification, the model checker attempts to show that these system-level properties can be successfully derived from the component-level contracts. Using compositional assume guarantee reasoning, we were able to verify system-level requirements [R1] and [R3], defined in Section 4.4, using COCOSIM, but not requirement [R2]. Table 4.1 captures the time taken by COCOSIM to verify [R1] and [R3].

<table>
<thead>
<tr>
<th>number of cells</th>
<th>[R1]</th>
<th>[R3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>0.487 seconds</td>
<td>0.487 seconds</td>
</tr>
<tr>
<td>16</td>
<td>2.395 seconds</td>
<td>9.08 hours</td>
</tr>
<tr>
<td>25</td>
<td>4.298 seconds</td>
<td>Timeout (&gt; 2 days)</td>
</tr>
</tbody>
</table>

Table 4.1: Time taken by COCOSIM to compositionally verify [R1] and [R3].

Specifically, compositional verification of [R1] was achieved quite quickly (< 20 seconds), this was because the model checker actually only had to analyse two components: the Interface and the BatteryMonitor to verify [R1]. [R3] was more complex since it involved a loop between the Interface and the ReasoningAgent. This required Kind2 to carry out a lot of unrolling to adequately assess this property as well as dealing with more complex contracts which included quantifiers and arrays. As a result, we were only able to prove [R3] for specific grid widths (3 × 3 took a few minutes, 4 × 4 took a few hours, and larger grids timed out).

4.7.2 Component-Level Verification Using Kind2 and Event-B:

Previously, we used compositional verification to verify that the system-level parent requirements hold based on the component-level requirements. Here, our objective was to verify that the more detailed specification/implementation of individual components obey the associated component-level requirements. Recognising that, for autonomous robotic systems, it is often necessary to use a range of verification techniques for individual components, we
used two distinct formal methods in this section [6,7]. Specifically, we used Kind2 to verify a simple implementation of the GRA and, Event-B to model and verify the ComputePlan component.

**Specification and Verification of the GRA:**
We represent a grid by a n-by-n matrix or \( n^2 \) array where \( n \) is the grid width. The following is 4-by-4 grid, the coordinates are expressed as inlined indices following a column-wise convention. For example index 5 represents the coordinate (2,2) in the matrix.

\[
\begin{bmatrix}
0 & 4 & 8 & 12 \\
1 & 5 & 9 & 13 \\
2 & 6 & 10 & 14 \\
3 & 7 & 11 & 15
\end{bmatrix}
\]

\[0 \ 1 \ 2 \ 3 \ \ldots \ 15\]

The set of obstacles is also represented by a matrix denoting whether a cell is an obstacle or not. In Lustre, we keep track of where the obstacles are using a boolean array: \( \text{bool}^\text{grid_width*grid_width} \). For example, the following matrix (transformed into a simple array following a column-wise convention) describes the obstacles in a 4-by-4 grid where cells 5 and 9 contain obstacles.

\[
\begin{bmatrix}
F & F & F & F \\
T & T & F & F \\
F & F & F & F \\
F & F & F & F
\end{bmatrix}
\]

\[F \ F \ F \ F \ F \ T \ F \ \ldots \ F\]

The set of heatpoints is also described as a matrix of integers denoting the heat value of each cell. In Lustre, heatpoints are represented in an integer array: \( \text{int}^\text{grid_width*grid_width} \). The following is an example of a 4-by-4 heat map represented as an integer array of length 16:

\[
\begin{bmatrix}
10 & 42 & 83 & 121 \\
14 & 0 & 0 & 13 \\
62 & 16 & 110 & 114 \\
43 & 74 & 131 & 135
\end{bmatrix}
\]

\[10 \ 14 \ 62 \ 43 \ 42 \ \ldots \ 135\]

The set of visited locations is represented by a matrix where we map each cell to a boolean indicating whether it has been visited or not. In Lustre, we declare the visited structure as a boolean array: \( \text{bool}^\text{grid_width*grid_width} \). In the following example, cells 1 and 2 are visited.

\[
\begin{bmatrix}
T & F & F & F \\
T & F & F & F \\
F & F & F & F \\
F & F & F & F
\end{bmatrix}
\]

\[T \ T \ F \ F \ F \ F \ \ldots \ F\]

The full Lustre architecture is presented in the Annex including the specifications for the components that are defined in the Lustre architecture.

Once these basic data representations were completed, we encoded our implementation of the GRA component, as shown in Fig.4.24. Here, the GRA computes the start (lines 33–35), goal (lines 48–50) and the visited (lines 44–46) cells. The start output is initialised
by the `currentPosition`, if we reached the goal in the last execution (`atGoal` is true) then the start is the previous goal (`pre_goal`), if the `recharge` flag is active then the `start` point is the previous start position since the rover did not move.

The `goal` output is set to `chargingPosition` if the `recharge` flag is active. Otherwise we choose the hottest heatpoint, the latter is computed with the help of the `hottestPoint` local array that keeps track of the hottest heatpoint traversing all `heatpoints`\(^1\).

\(^1\)The specification of the node `GRA` is given in the Annex. The Kind2 command is `kind2 --lus_main GRA main.lus`. Kind2 does not terminate verifying all properties specified in the specification.
The assumptions used are:

```plaintext
-- Assumptions on locations
assume "chargePosition_is_within_range"
0 <= chargePosition and chargePosition <= grid_width_square -1;
assume "currentPosition_is_within_range"
0 <= currentPosition and currentPosition <= grid_width_square - 1;

-- assume all heatpoints >= 0
assume "R3.1.3" forall (i: int)
0 <= i and i < grid_width_square
=> heatpoints[i] >= 0;
assume "R3.1.2" heatpoints[chargePosition] = 0;

assume "do not recharge twice" pre_recharge => not recharge;
assume "recharge_Eq_not_atGoal" recharge = not atGoal;
assume "R2.2.1.2" ( atGoal => currentPosition = pre_goal );
```

The following properties were verified in less than 1 second:

```plaintext
-- When goal is reached, in the next step we start from the previous goal
guarantee "R2.2.1.1" ( atGoal => start = pre_goal );
-- If recharge then go to charging position
guarantee "R1.2.1" ((recharge and ((pre (not recharge)) or FTP)) => (goal = chargePosition));
-- If recharge then we did not move from our last position.
guarantee "R1.2.2" (recharge => start = pre_start );
-- Mark goal as visited when atGoal is true.
guarantee "R3.2.1" true => ( atGoal => visited[pre_goal]);
```

The following requirements were verified for a specific grid width.

```plaintext
(*Don't visit an already visited cell*)
var end_Of_Execution : bool = goal = pre_goal;
guarantee "R3.2.2" not(goal = chargePosition) and not(end_Of_Execution) => not visited[goal];

(*Req visited is initialized by 0 for all heatpoints*)
guarantee "initialize visited at t=0" (forall (i: int) 0 <= i and i < grid_width_square
=> not visited[i]);

(*Once visited, it stays visited*)
guarantee forall (i: int) 0 <= i and i < grid_width_square
  => (false -> pre visited[i]) => visited[i] );

(*Pick up the hottest point*)
guarantee "R3.3" not recharge =>
  forall (i: int) 0 <= i and i < grid_width_square
  => ( not visited[i] => heatpoints[goal] >= heatpoints[i] )
```

The verification time increased with the number of cells, as shown in Table 4.2.

The following requirements took even longer than the previous ones.

```plaintext
(*End of execution properties*)
guarantee "all visited at end_Of_Execution" end_Of_Execution =>
  (forall (i: int) 0 <= i and i < grid_width_square
   => (heatpoints[i] > 0 => visited[i]));
```
Table 4.2: The verification time increased with the number of cells in the grid.

### Specification and Verification of ComputePlan using Event-B

Our Event-B model contains three contexts (for modelling the static parts of the system including constants, carrier sets and axioms) and two machines (for modelling the dynamic parts of the system including variables, invariants and events). Event-B supports formal refinement and thus our contexts extend one another and our machines indicate refinement steps. The detailed Event-B model is contained in the Annex (Figs. 7.9, 7.10, 7.11, 7.12, 7.13 and 7.14).

Our most primitive context, \(\text{ctx0}\) (Fig. 7.9), contains basic information about the size of the grid, valid grid locations, obstacles and heat points. We do not explicitly list the elements of these sets since this specification is for a generic planner. However, it includes a series of axioms that impose constraints on the values that these constants can hold. In the spirit of refinement, \(\text{ctx0}\) is extended via \(\text{ctx1}\) (Fig. 7.10) which specifies functions that capture the behavior of the planning component. For example, the following axioms (lines 11-12 of Fig. 7.10) capture what it means for the rover to move left in the grid.

\[
\begin{align*}
\text{axm1: } & \text{left } \in \mathbb{N} \rightarrow \mathbb{N} \\
\text{axm2: } & \forall x \cdot x \in \mathbb{N} \Rightarrow \text{left}(x) = (x-1)
\end{align*}
\]

The abstract machine, \(\text{mac0}\) (Fig. 7.11), models a simple search-based planning algorithm which produces a set of plans containing the start and goal. Event-B uses sets as primitive so we ensure that these plans, represented as sets, can be linearized using the \text{adjacent} function specified in \(\text{ctx1}\) (line 19 of Fig. 7.10). This is captured by invariants in \(\text{mac0}\) (lines 8–9 of Fig. 7.11) as shown below.

\[
\begin{align*}
\text{inv6: } & \forall p \cdot p \in \text{PlanSet} \Rightarrow p \subseteq \text{grid} \land (\text{card}(p) \geq 2) \Rightarrow (\exists q, r \cdot q \in p \land r \in p \land (\text{adjacent}(s_0 \mapsto q) = \text{TRUE}) \land (\text{adjacent}(r \mapsto g) = \text{TRUE})) \not\text{ theorem} \\
\text{inv7: } & \forall p, p_0 \cdot p \in \text{PlanSet} \land p_0 \in p \Rightarrow p \subseteq \text{grid} \land p_0 \in \text{grid} \land (\text{card}(p) \geq 3 \land p_0 \neq g \land p_0 \neq s_0) \Rightarrow (\exists q, r \cdot q \in p \land r \in p \land (\text{adjacent}(p_0 \mapsto q) = \text{TRUE}) \land (\text{adjacent}(r \mapsto p_0) = \text{TRUE})) \not\text{ theorem}
\end{align*}
\]

The refinement, \(\text{mac1}\) (Figs. 7.13 and 7.14), has the added functionality of a plan reasoning agent and chooses the shortest plan from \text{PlanSet}. This is captured by the \text{PRA} event shown below (lines 68–75 of Fig. 7.14):

\[
\begin{align*}
\text{Event} \text{ PRA} & : \text{ordinary} \\
\text{any} & : p \\
\text{when} \ p & : p \in \text{PlanSet} \not\text{ theorem} \\
\text{grid1: } & \text{card}(p) \leq \text{card}(q) \not\text{ theorem} \\
\text{grid2: } & \text{planningCompleted} = \text{TRUE} \not\text{ theorem} \\
\text{then} \ a & : \text{chosenplan} := p \\
\text{act2: } & \text{returnplan} := \text{TRUE}
\end{align*}
\]

We also constructed another context, \(\text{ctx2}\), (Fig. 7.12) to be used by \(\text{mac1}\) which
contains a constant which limits the number of plans that are generated.

Linking (FRET) Requirements and Event-B: We encoded [R2.1], [R2.4.1], [R2.4.3], [R2.4.4], [R2.5] and [R3.4] in our Event-B model. Although we could not verify [R2] compositionally, its child requirements feature in our Event-B model (e.g. [R.2.5] above). This ensures that the planning components do not inadvertently cause the rover to collide with an obstacle. The vast majority of the Event-B proof obligations were discharged automatically by the Atelier B theorem prover in the Rodin Platform. Some required interactive proof but they were relatively straightforward.

We describe how each of the above requirements were encoded in our Event-B model as follows:

[R2.1:] Initial rover position shall be static.
This corresponds to axm6 in ctx0 (Fig. 7.9) which states that
\texttt{axm6: s0 ∈ grid}
Since this is specified in a context, it is assumed by the prover that this information is static.

[R2.4.1:] An obstacle shall not be in the location of the initial rover position.
This corresponds to axm10 in ctx0 (Fig. 7.9) which states that
\texttt{axm10: s0 ∉ Obs}
where \texttt{Obs} denotes the set of obstacle locations.

[R2.4.3:] An obstacle shall not be in the same location as a heatpoint.
This corresponds to axm12 in ctx0 (Fig. 7.9) which states that
\texttt{axm12: ∀ a, b · (a ↦→ b) ∈ H ⇒ a ∉ Obs}
where \( ↦→ \) is the tuple notation used in Event-B.

[R2.4.4:] All obstacles should be within the grid.
This corresponds to axm5 in ctx0 (Fig. 7.9) which states that
\texttt{axm5: Obs ⊆ grid}

[R2.5:] The calculated path to destination shall not include a location with an obstacle.
This corresponds to inv3 in mac0 (Fig. 7.11) which states that
\texttt{inv3: ∀ p, x · p ∈ PlanSet ∧ x ∈ p ⇒ x ∉ Obs}
Note here that every element of \texttt{PlanSet} is a set of grid locations.

[R3.4:] The shortest path to the current goal shall be selected.
This corresponds to inv3 in mac1 (Fig. 7.13) which states that
\texttt{inv3: (planningCompleted = TRUE) ∧ (returnplan = TRUE) ⇒ (∀ x · x ∈ PlanSet ⇒ card(chosenplan) ≤ card(x))}
where \texttt{card()} is a function that is native to Event-B for getting the cardinality (size) of a set.

Further to these requirements, we examine the relevant COCOSIM contracts and ensure that these are adequately and appropriately represented in our Event-B model. For example, consider the following guarantees in COCOSIM:

1. \texttt{guarantee "plan\_ends\_in\_goal" plan[plan\_sizeMax-1] = goal;}
2. \texttt{guarantee "plan\_starts\_in\_start" plan[0] = start}
These are captured by \textit{inv1} and \textit{inv6} in \textit{mac0} (Fig. 7.13) which state that

\textit{inv1}: \(\forall p.p \in \text{PlanSet} \Rightarrow p \subseteq \text{grid} \land (\text{card}(p) \geq 2 \Rightarrow (g \in p \land s0 \in p))\)

and

\textit{inv6}: \(\forall p.p \in \text{PlanSet} \Rightarrow p \subseteq \text{grid} \land (\text{card}(p) \geq 2 \Rightarrow (\exists q, rq \in p \land r \in p \land (\text{adjacent}(s0 \mapsto q) = \text{TRUE}) \land (\text{adjacent}(r \mapsto g) = \text{TRUE})))\)

We used Event-B because it is not limited by the state space explosion problem that was causing Kind2 to time out. We specified more complex component-level properties that would have been difficult to verify using a model-checker. The full Event-B specification is available in the Annex.

4.8 Step 7: Document Verification Results and Build Safety Case

All of the verification results produced by the tools are a part of the safety case that was constructed in AdvoCATE. Some artifacts were imported automatically into AdvoCATE, while others were added manually. In this way, we ensured that the necessary data was available in AdvoCATE. Since this case study did not include a full system implementation, the safety case that we report here is an interim version and contains the current safety assurance status.

The skeleton of the overall argument is generated automatically from the information defined and imported into AdvoCATE such as hazards, mitigation requirements, formalized requirements, and evidence artifacts. We have further extended the skeleton argument based on the specific application and the tools that we used. Fig. 4.25 presents an argument fragment about mitigation of the \textit{running out of battery} hazard that causes \textit{loss of rover}. Similar arguments exist for other causes of \textit{loss of rover} and the other hazards. For brevity, Fig. 4.25 only contains a fragment of the existing argument. This argument focuses on two aspects: the requirements directly related to this hazard (right branch), and the causes that lead to the hazard (left branch). Parts of the argument that come before and after the fragment in Fig. 4.25 can be found in the Annex.

The goal \textbf{G14} focuses on \textbf{[R1]} that was verified using Cocosim. We built a similar argument for each system-level requirement previously verified compositionally with Cocosim. For each argument, we extended the automatically generated part with a combination of existing argumentation patterns [38, 39] to support application-specific goals (base of Fig. 4.25):

- The formalisation of the natural language requirement is correct (\textbf{G3-A1}).
- The results from Cocosim are trustworthy (\textbf{G4-A1}).
- The different design representations are consistent (\textbf{G5-A1}).
- The Cocosim verification result for \textbf{[R1]} is valid (\textbf{G6-A1}).

To ensure that the different design representations were consistent across the tools, we performed manual reviews where automated consistency validation was not available. For example, we carried out manual reviews to verify that the design as specified in AdvoCATE was consistent with the design modelled in Simulink as well as those used by Kind2 and Event-B.

The goal \textbf{G3-A1} focuses on the correct specification of \textbf{[R1]} in Fretish and the correct functioning of Fret. While we have to verify through a manual review that the natural language requirement is correctly represented in Fretish, the correct Fret functioning and generation of the corresponding Cocosim contracts is supported by the automated verification framework of Fret.
The goal **G6-A1** focuses on the validity of [R1] through analysis with the COCOSIM tool. This part of the argument points out the dependencies to the properties of the other components, but also implicit assumptions on which these results rely. Finally, to have confidence in the results from COCOSIM, we argued the trustworthiness of COCOSIM with the goal **G4-A1**. Since COCOSIM relies on model transformations and external tools for verification, the correctness of these has to be established. For example, we argued about the correctness of the translation from Simulink to Lustre code that is used by Kind2.
Chapter 5

Discussion

Using the given formal verification tools we were able to fully verify that the Navigation System will not cause the rover to run out of battery. We could not verify that a collision will never occur at system-level with COCOSIM due to the specification complexity. However, we were able to verify with Event-B that the Navigation System will not generate plans that contain obstacles at component-level. Finally, we were able to verify that the rover will visit all of the heat points with COCOSIM, but only for a small grid size of 4x4. Verifying the property for greater grid sizes did not finish even after several days of analysis.

This case study showed us that by following our methodology we were able to leverage multiple formal tools and use them in a complementary fashion. In this way, we applied formal methods to small, manageable chunks of the system to ease the verification burden and to avoid becoming trapped by the limitations of any single tool. Using FRET to bridge the gap between the informal and formal steps by formalizing our requirements was particularly useful because it helped us to clarify any details that were implicit in the natural language requirements.

The explanations produced by FRET were instrumental in ensuring that the FRETIISH requirements captured our intended semantics. Even though FRETIISH is intended to be intuitive, translating the natural language requirements into FRETIISH was not always straightforward. However, most of the Inspection Rover requirements fall within a small number of patterns, an issue that we have observed in other studies within our organization. We were not able to encode everything in FRETIISH. For example, some requirements had to be defined using first order temporal logic. We were able to get around this problem by using auxiliary variables as placeholders for the quantifiers at the requirements level, but a FRETIISH-level solution would be desirable.

The choice of COCOSIM and in particular Kind2 greatly influenced our design decisions. For example, in our original design, we represented cells in the grid as $(x, y)$-coordinates. However, we subsequently simplified this by using indices so that they were easier to represent and reason about in formal tools. Our choice of a compositional verification approach caused us to output specific variables such as the remaining battery power to verify [R1] compositionally. Furthermore, we had to adapt the hierarchical structure of the system to accommodate compositional verification. If the choice of formal verification tools is made early on in the system development process, the design of the system can be created so that it is more suitable to formal verification using the chosen method(s).

Not all of the formalized requirements were formally verifiable, some described hardware constraints and some required physical testing. This supports the claim that the robotics domain requires both formal and informal verification processes [6]. For example, everything
depends on the accuracy of the current position of the rover which is a property that we could not formally verify in this case. However, by formalizing the requirements that will be verified via testing, we can potentially incorporate run-time analysis. Specifically, the formalized properties can be used to generate formal run-time monitors, which will help with fault management during operation. These could potentially help to create recovery barriers in the bow-tie diagrams. In this way, we could include the development of fault management at design time.

Integrating the verification results from the different formal methods in an assurance case required intensive cooperation between the assurance and formal methods experts. The effort required identifying dependencies between different tools, understanding the techniques and the tool implementations, implicit assumptions on which analyses were ran and results interpreted. The activity was greatly performed ad hoc. A more systematic approach to gathering the assurance information from formal methods applications would be beneficial.

The case study helped us to identify limitations in the used tools (AdvoCATE, FRET, COCOSIM and Event-B) for robotics applications. In fact, this case study prompted an update to COCOSIM to incorporate abstract unimplemented components. In particular, COCOSIM now generates Lustre code for these components using the imported keyword when no implementation is available. Other limitations that were identified include the lack of support in FRET for abstract data types which caused us to manually edit the FRET-generated COCOSIM contracts. We encountered some difficulties when attempting to automatically import verification artifacts directly from the tools into AdvoCATE which caused us to insert some information manually.

Our methodology follows the development phases of existing development guidelines [32, 33] and builds on top of them through a set of steps (Chapter 3), which are guided by the need to devise an assurance case that integrates artifacts from different tools. Although in the presented work we used specific tools, we believe that our methodology can be followed irrespective of the choice of tools.
Chapter 6

Related Work

Heterogeneous verification techniques were used to verify an autonomous Mars Curiosity rover simulation [8]. This work uses distinct verification methods for specific components but does explicitly link the verification artifacts produced. Recent work proposes first-order logic to unify heterogeneous formal methods via a compositional approach but this work currently lacks tool support [40].

Other approaches to compositional verification include AGREE [34] and OCRA [41]. We explored these as potential alternatives to COCOSIM in this work but neither offered the level of expressivity that we sought. They also did not accommodate for the use of distinct verification techniques at component-level.

Developers should choose the most appropriate formal method on a per-component basis based on the suitability of the formal method and the user’s level of expertise. As such, there are many alternatives to Event-B and Kind2, including Gwendolen [42], TLA+ [43] and Dafny [44].

Isabelle/SACM [45, 46] extends the Isabelle proof assistant to support assurance case development. In Isabelle/SACM, a UTP semantics must be defined for each formal verification artifact that is to be included in the assurance case.

In this report, we have illustrated the benefits of using various formal verification techniques. Related to this, [47–49] demonstrate that a collaborative approach to verification, encompassing static verification and testing, is advantageous as it finds more errors and proves more properties than a single technique.
Chapter 7

Conclusions and Future Work

In this Chapter, we conclude and identify future directions.

7.1 Conclusions

This paper presented our methodology for integrating results from distinct formal methods via the development of an assurance case. We applied this methodology in the design of an Inspection Rover system and used the AdvoCATE, FRET, COCOSIM, and Event-B tools. This is the first effort to integrate the four aforementioned tools. We illustrated how the choice of verification method can impact the system design and discussed how a heterogeneous set of verification results can be linked during assurance with AdvoCATE. Further, we have made our case study artifacts publicly available to fuel discussion in the research community.

7.2 Future Work

This work has opened up a number of avenues for future research. In particular, it is likely that requirements for complex robotic systems will increasingly contain probabilistic properties. Therefore, supporting the definition of these requirements in FRET is a future direction. Additionally, we intend to develop a DSL to facilitate the integration of AdvoCATE with different verification tools. In this case study, we mainly used tools that output artifacts in JSON or XML format, which are straightforward to parse in AdvoCATE, but this may not be the case for other analysis tools. These formats are straightforward to parse in AdvoCATE but a DSL would be more beneficial, particularly for tools that do not support these output formats. Finally, we intend to explore the definition of a ‘Taxonomy of Requirements’ and classify those in this case study. This will help developers to design their system with verification in mind by demonstrating how to classify requirements based on the ways that they will be verified and argued in an assurance case early at design phase.

We have identified a number of phases to the development of the inspection rover system which will allow us to gradually increase the complexity of the system. These phases will support us in investigating how our methodology and tools scale with increasing system complexity. For the work presented in this technical report, we focused on a simplified model based on Phase 1 below. In future work, we intend to gradually increase the complexity of this system guided by the Phases 2 and 3 as outlined below:

Phase 1: Basic functionality and assumptions:
– Assume a constant velocity once the rover starts to move.
– Assume that the rover does not deviate from the chosen plan.
Assume that the Infrared component only identifies reachable heatpoints. Assume that the charger is reachable and that all heatpoints can be reached from the position of the charger. Traverse a grid of known size to visit all identified heatpoints, avoid obstacles and charging as required.

Phase 2: Additional functionality:
- Rover has variable velocity (method of accelerating and decelerating) once in motion.
- The Infrared component may identify heatpoints that are not reachable (e.g., completely surrounded by obstacles or too far away from the charger). Here, the MapValidator must examine the map and produce the set of reachable heatpoints. It may potentially also seek to inform the user of the list of heatpoints which were not reachable so that they may decide to use another means for inspecting them (e.g. UAV).
- Use the Vision component continuously to provide extra redundancy with respect to detecting the current position of the rover accurately.
- Added functionality to support an ‘emergency stop’ in the case where an inaccurate map might cause the rover to collide with an obstacle.

Phase 3: Increased Autonomy and Verification
- The rover has the ability to deviate from the chosen plan if an unforeseen situation arises, e.g. suddenly an object appears in a grid location that was not identified as an obstacle location. This could be caused by inaccuracies in the original map or adverse weather conditions causing debris to blow into another location.
- Addition of runtime monitors to support the verification of critical requirements and/or those that were difficult to formally verify.
- Addition of a learning component.


Annex

AADL Model

```
package rover
public
system roverSystem
end roverSystem;

system implementation roverSystem.with_devices
subcomponents
    this_camera: device Camera;
    this_heatsensor: device HeatSensor;
    this_vision: process Vision;
    this_infrared: process Infrared;
    this_goal_agent: process GoalReasoningAgent;
    this_planner: process Planner;
    this_plan_agent: process PlanReasoningAgent;
    this_battery: process BatteryMonitor;
    this_interface: process HardwareInterface;
    this_bus: bus HWConnection.impl;
    this_processor: processor MainProcessor.impl;
connections
    camera_vision: port this_camera.images -> this_vision.camera;
    sensor_infrared: port this_heatsensor.sensorReading -> this_infrared.sensorinp;
    vision_goal: port this_vision.Obs -> this_goal_agent.Obs;
    vision_goal2: port this_vision.s0 -> this_goal_agent.s0;
    infrared_goal: port this_infrared.H -> this_goal_agent.H;
    planagent_goal: port this_plan_agent.recharge -> this_goal_agent.recharge;
    interface_goal: port this_interface.atGoal -> this_goal_agent.atGoal;
    goal_planner: port this_goal_agent.g -> this_planner.g;
    planner_planagent: port this_planner.PlanSet -> this_plan_agent.PlanSet;
    battery_planagent: port this_battery.b -> this_plan_agent.b;
    planagent_interface: port this_plan_agent.plan -> this_interface.plan;
    bus_camera: bus access this_bus -> this_camera.bus_access;
    bus_heat: bus access this_bus -> this_heatsensor.bus_access;
    bus_processor: bus access this_bus -> this_processor.bus_access;
end roverSystem.with_devices;

processor MainProcessor
features
    bus_access: requires bus access HWConnection;
end MainProcessor;

processor implementation MainProcessor.impl
subcomponents
    this_ram: memory Ram;
end MainProcessor.impl;

memory Ram
end Ram;

bus HWConnection
end HWConnection;
```
bus implementation HWConnection.impl
end HWConnection.impl;

device Camera
  features
    images: out data port;
    bus_access: requires bus access HWConnection;
end Camera;

device implementation Camera.impl
end Camera.impl;

device HeatSensor
  features
    sensorReading: out data port;
    bus_access: requires bus access HWConnection;
end HeatSensor;

device implementation HeatSensor.impl
end HeatSensor.impl;

process Vision
  features
    camera: in data port;
    Obs: out data port;
    s0: out data port;
end Vision;

process implementation Vision.impl
end Vision.impl;

process Infrared
  features
    sensorinp: in data port;
    H: out data port;
end Infrared;

process implementation Infrared.impl
end Infrared.impl;

process GoalReasoningAgent
  features
    H: in data port;
    Obs: in data port;
    recharge: in data port;
    atGoal: in data port;
    s0: in data port;
    g: out data port;
end GoalReasoningAgent;

process implementation GoalReasoningAgent.impl
end GoalReasoningAgent.impl;

process Planner
  features
    g: in data port;
    PlanSet: out data port;
end Planner;

process implementation Planner.impl
end Planner.impl;

process PlanReasoningAgent
  features
    PlanSet: in data port;
    b: in data port;
    plan: out data port;
    recharge: out data port;
end PlanReasoningAgent;

process implementation PlanReasoningAgent.impl
end PlanReasoningAgent.impl;

process BatteryMonitor
  features
sensorInp: in data port;

b: out data port;
end BatteryMonitor;

process implementation BatteryMonitor.impl
end BatteryMonitor.impl;

process HardwareInterface
features
plan: in data port;
atGoal: out data port;
haveMoved: out data port;
end HardwareInterface;

process implementation HardwareInterface.impl
end HardwareInterface.impl;
end rover;
System architecture in Lustre

-- Implementation Choices:
-- The grid is seen as a Matrice.
-- The position in the grid is indicated by the inlined index of the Matrix
-- We use Column wise indexing
-- For example this is a 4-by-4 grid:
-- ________________
-- | 0 | 4 | 8 | 12 |
-- ________________
-- | 1 | 5 | 9 | 13 |
-- ________________
-- | 2 | 6 | 10 | 14 |
-- ________________
-- | 3 | 7 | 11 | 15 |
-- ________________

-- Obstacles Representation: Obstacles : bool ^ (grid_width_square1)
-- a boolean array indicating if a cell is an obstacle.
-- Obstacles[i] indicates if cell i is an obstacle.

-- Heat Points Representation: Heatpoints : int ^ (grid_width_square1)
-- Heatpoints[i] indicates the heat value of cell i

-- We Store Visited HeatPoints: Visisted : bool ^ (grid_width_square1)
-- Visisted[i] indicates if the cell "i" was visited.

include "helpers.lus"
include "Map_ValidatorSpec.lus"
include "GRASpec.lus"
include "Battery_MonitorSpec.lus"
include "InterfaceSpec.lus"
include "NavigationSpec.lus"
include "ComputePlanSpec.lus"

-- Components

-- Map Validator
node imported Map_Validator ( 
    const grid_width1 : int;
    const grid_width_square1 : int;
    const chargingPosition : int;
    currentPosition_in : int;
    const heatpoints_in : int^ (grid_width_square1);
    const obstacles_in : bool^ (grid_width_square1);
    const initialPosition:int;
) 
returns ( 
    currentPosition_out : int;
    heatpoints_out : int^ (grid_width_square1);
    obstacles_out : bool^ (grid_width_square1);
);
(*@contract
import Map_ValidatorSpec ( 
    grid_width1,
    grid_width_square1,
    chargingPosition,
    currentPosition_in,
    heatpoints_in,
    obstacles_in,
    initialPosition )
)
returns (currentPosition_out,
  heatpoints_out,
  obstacles_out);
*)

-- GRA
node imported GRA (
  const grid_width1 : int;
  const grid_width_square1 : int;
  currentPosition : int;
  const heatPoints : int ^ grid_width_square1;
  const chargePosition : int;
  atGoal : bool;
  recharge : bool;
) returns (goal : int;
  start : int;
  visited : bool ^ grid_width_square1);
(* @contract
import GRASpec (
  grid_width1,
  grid_width_square1,
  currentPosition,
  heatPoints,
  chargePosition,
  atGoal,
  recharge)
returns (goal ,
  start ,
  visited);
*)

-- Remove imported keyword and uncomment the implementation if you want to check the
GRA Spec
(*
var
pre_goal , pre_start : int;
hottestPoint : int ^ grid_width_square1;
let
  pre_goal = currentPosition -> pre goal;
  pre_start = currentPosition -> pre start;
  start = currentPosition ->
    if atGoal then pre_goal
    else pre_start;
  hottestPoint[i] =
    if i = 0 then 0
    else
      if visited[hottestPoint[i-1]] or
        ( not(visited[i]) and heatPoints[i] >= heatPoints[hottestPoint[i-1]])
      then i
      else hottestPoint[i-1];
  visited[i] = false ->
    if atGoal and pre_goal = i then true
    else pre visited[i];
  goal = if recharge then chargePosition
    else
      if visited[hottestPoint[grid_width_square1 - 1]]
        then pre_goal else hottestPoint[grid_width_square1 - 1];
  tel
*)

-- Planner: compute a plan set for going from start to goal
node imported Planner(
  const grid_width1 : int;
  const grid_width_square1 : int;
  const plan_sizeMax : int;
const planSet_sizeMax : int;
chargePosition : int;
const obstacles : bool^ (grid_width_square1);
start : int;
goal : int;
returns {
planSet : int^planSet_sizeMax^plan_sizeMax;
};

-- PRA
node imported PRA (const plan_sizeMax : int; const planSet_sizeMax : int; planSet : int^planSet_sizeMax^plan_sizeMax; goal : int; returns (plan : int^plan_sizeMax));

-- ComputePlan
node ComputePlan (const grid_width1 : int; const grid_width_square1 : int; const plan_sizeMax : int; const planSet_sizeMax : int; chargePosition : int; const obstacles : bool^ (grid_width_square1); start : int; goal : int; returns (plan : int^plan_sizeMax));

(* @contract
import ComputePlanSpec (grid_width1, grid_width_square1, plan_sizeMax, planSet_sizeMax, chargePosition, obstacles, start, goal)
returns (plan);*)

var planSet : int^planSet_sizeMax^plan_sizeMax;

let
planSet = Planner(
grid_width1, grid_width_square1, plan_sizeMax, planSet_sizeMax, chargePosition, obstacles, start, goal);
plan = PRA(
plan_sizeMax, planSet_sizeMax, planSet, goal);
node ReasoningAgent()
    const grid_width1 : int;
    const grid_width_square1 : int;
    const plan_sizeMax : int;
    const planSet_sizeMax : int;
    const obstacles : bool^(grid_width_square1);
    currentPosition:int;
    const heatPoints : int^(grid_width_square1);
    const chargingPosition:int;
    pre_atGoal:bool;
    pre_recharge:bool;
returns(
    goal : int;
    plan_To_Destination : int^plan_sizeMax;
    plan_From_Destination_To_ChargPos : int^plan_sizeMax;
    visited : bool^(grid_width_square1));
var
    start : int;
let
    (goal, start, visited) = GRA(
        grid_width1,
        grid_width_square1,
        currentPosition,
        heatPoints,
        chargingPosition,
        pre_atGoal,
        pre_recharge);
    plan_To_Destination = ComputePlan(
        grid_width1,
        grid_width_square1,
        plan_sizeMax,
        planSet_sizeMax,
        chargingPosition,
        obstacles,
        start,
        goal);
    plan_From_Destination_To_ChargPos = ComputePlan(
        grid_width1,
        grid_width_square1,
        plan_sizeMax,
        planSet_sizeMax,
        chargingPosition,
        obstacles,
        goal,
        chargingPosition);
end

node imported Battery_Monitor (battery_in:int)
returns (battery_out : int;);
(*@contract
import Battery_MonitorSpec (battery_in) returns (battery_out);
*)

node imported Interface(
    const plan_sizeMax:int;
    const chargePosition : int;
    battery_in:int;
goal: int;
plan_to_dest: int^plan_sizeMax;
plan_from_dest_to_chargPos: int^plan_sizeMax)
returns (atGoal: bool;
 recharge: bool;
battery_out: int);
(*@contract
import InterfaceSpec ( plan_sizeMax,
 chargePosition, battery_in,
goal,
plan_to_dest,
plan_from_dest_to_chargPos)
returns (atGoal, recharge, battery_out);
*)

-- Battery + Interface
node battery_interface(
 const plan_sizeMax: int;
 const planSet_sizeMax: int;
 plan_To_Destination: int^plan_sizeMax;
 plan_From_Destination_To_ChargPos: int^plan_sizeMax;
 const chargingPosition : int;
goal:int;
) returns ( atGoal, recharge : bool ; battery_out : int ;)
(**@contract
-- assume 1 < plan_sizeMax ;
-- assume 1 < planSet_sizeMax ;
-- guarantee "battery_always_positive" battery_out > 0.0 ;
*)
var battery_in : int;
let
battery_in = Battery_Monitor (100 -> pre battery_out);
(atGoal, recharge, battery_out) = Interface ( plan_sizeMax,
 chargingPosition, battery_in,
goal,
plan_To_Destination,
plan_From_Destination_To_ChargPos );

-- Navigation System: Reasoning agent + Interface + Battery Monitor
node NavigationSystem(
 const grid_width1 : int;
 const grid_width_square1 : int;
 const plan_sizeMax: int;
 const planSet_sizeMax: int;
 const chargingPosition : int;
 const initialPosition:int;
currentPosition: int;
 const heatpoints : int^(grid_width_square1);
 const obstacles : bool ^(grid_width_square1);
) returns(
atGoal: bool;
battery_out : int;
visited: bool^(grid_width_square1);
);
(*@contract
import NavigationSpec ( grid_width1,
grid_width_square1, plan_sizeMax, planSet_sizeMax, chargingPosition, initialPosition, currentPosition, heatpoints, obstacles)
returns (battery_out, visited)
)

var

plan_To_Destination : int ^ plan_sizeMax;
plan_From_Destination_To_ChargPos : int ^ plan_sizeMax;
pre_atGoal : bool;
recharge, pre_recharge : bool;
goal : int;

let

pre_atGoal = true -> pre atGoal;
pre_recharge = false -> pre recharge;
(goal, plan_To_Destination, plan_From_Destination_To_ChargPos, visited) =
ReasoningAgent (
    grid_width1,
    grid_width_square1,
    plan_sizeMax,
    planSet_sizeMax,
    obstacles, 
    currentPosition,
    heatpoints,
    chargingPosition, 
    pre_atGoal, 
    pre_recharge ) ;

(atGoal, recharge, battery_out) = battery_interface(
    plan_sizeMax, 
    planSet_sizeMax, 
    plan_To_Destination, 
    plan_From_Destination_To_ChargPos, 
    chargingPosition, 
    goal 
);
System specification in Lustre

```plaintext
contract GRASpec (  
const grid_width:int;  
const grid_width_square:int;  
currentPosition:int;  
const heatpoints: int ^ grid_width_square;  
const chargePosition: int;  
atGoal: bool;  
recharge: bool; )  
returns ( goal: int;  
start: int;  
visited: bool ^ grid_width_square);
let

var pre_goal : int = currentPosition -> pre goal;  
var pre_start : int = currentPosition -> pre start;  
var pre_recharge : bool = false -> pre recharge;  
var end_Of_Execution : bool = false -> goal = pre_goal;

-- uncomment this only when you are proving GRA individually
--assume grid_width_square = 9;
--assume grid_width = 3;

-- Assumptions on locations
assume "chargePosition_is_within_range"
0 <= chargePosition and chargePosition <= grid_width_square -1;  
assume "currentPosition_is_within_range"
0 <= currentPosition and currentPosition <= grid_width_square - 1;

-- un-comment this if you want to prove GRA implementation
-- assume "do not recharge twice" pre_recharge => not recharge;
-- assume if recharge then Interface did not execute plan and atGoal is negative
-- assume "recharge_Eq_not_atGoal" recharge = not atGoal;

-- This assumption cannot be verified since currentPosition is free input in Navigation component
-- assume "R2.2.1.2" ( atGoal => currentPosition = pre_goal );

guarantee "goal_is_within_range"
0 <= goal and goal <= grid_width_square - 1;
guarantee "start_is_within_range"
0 <= start and start <= grid_width_square - 1;

-- Req text: GRA shall always satisfy atGoal => start=goal

guarantee "R2.2.1.1" ( atGoal => start = pre_goal );

-- Req text: if recharge GRA shall immediately satisfy (goal=chargePosition)
guarantee "R1.2.1" ((recharge and ((pre (not recharge)) or FTP)) => (goal = chargePosition));
guarantee "R1.2.2" (recharge => start = pre_start );

-- Req text: if atGoal GRA shall after 1 tick satisfy removeGoalFromSet

guarantee "R3.2.1" true -> ( atGoal => visited[pre_goal]);
```

53
-- Don't visit an already visited cell

guarantee "R3.2.2" not(goal = chargePosition) and not(end_Of_Execution) => not visited[goal];

-- Req visited is initialized by 0 for all heatpoints

guarantee "initialize visited at t=0" (forall (i: int ) 0 <= i and i < grid_width_square => not visited[i]) -> true;

-- Once visited, it stays visited

guarantee "Once visited, it stays visited" forall (i: int ) 0 <= i and i < grid_width_square => (false -> pre visited[i]) => visited[i]);

-- Pick up the hottest point

guarantee "R3.3" not recharge => (forall (i: int ) 0 <= i and i < grid_width_square => (not visited[i] => heatpoints[goal] >= heatpoints[i]) );

-- End of execution properties

guarantee "all visited at end_Of_Execution" end_Of_Execution => (forall (i: int ) 0 <= i and i < grid_width_square => (heatpoints[i] > 0 => visited[i]) ) ;

tel

contract ComputePlanSpec (const grid_width : int ;
const grid_width_square : int ;
const plan_sizeMax : int ;
const planSet_sizeMax : int ;
chargePosition : int ;
start : int ;
goal : int )
returns (plan : int^ plan_sizeMax);

let

-- Note: we avoid any nonlinear operations when using quantifiers.
-- We are using this array to store multiplications of grid_width: 0, grid_width, ..., grid_width*grid_width
var MultipleGridWith : int^(grid_width+1) = constructMultipleGridWidth(grid_width);

assume grid_width_square = MultipleGridWith[grid_width];

assume "start_different_from_goal" start = goal => goal = chargePosition;

guarantee "plan_ends_in_goal" plan[plan_sizeMax-1] = goal;
guarantee "plan_starts_in_start" plan[0] = start;
guarantee "plan_steps_in_bounds"
forall (i: int)
0 <= i and i < plan_sizeMax
=> 0 <= plan[i] and plan[i] <= grid_width_square - 1;

guarantee "plan_steps_not_repeated"
forall (i, j: int)
0 <= i and i < plan_sizeMax - 1
and i < j and j < plan_sizeMax
=>
( plan[i] <> goal and plan[i] <> plan[j])
or (plan[i] = goal and plan[j] = goal);

guarantee "next_valid_step_simple_version"
forall (i, k: int)
0 <= i and i < plan_sizeMax - 1
and 0 <= k and k <= grid_width
=> ((plan[i+1] - plan[i]) = 1 and (plan[i] <> MultipleGridWith[k] - 1))
or ((plan[i+1] - plan[i]) = -1 and (plan[i] <> MultiplGridWith[k]))
or (plan[i+1] - plan[i]) = grid_width
or (plan[i+1] - plan[i]) = -grid_width
or ((plan[i+1] - plan[i]) = 0 and plan[i] = goal);

contract InterfaceSpec (
  const plan_sizeMax : int;
  const chargePosition : int;
  battery_in : int;
  goal : int;
  plan_to_dept : int^plan_sizeMax;
  plan_from_dept_to_chargPos : int^plan_sizeMax;
)
returns (atGoal : bool;
  recharge : bool;
  battery_out : int);

let

-- We can assume that the longest path does not consume more than
-- some defined constant of charge.
-- We therefore set the charge needed as double of that defined constant.
-- enough to go to destination and then go to the Charging position if needed.
var longest_path_charge : int = 10;
var charge_needed : int = 2 * longest_path_charge;

(* Req text: Interface shall always satisfy (battery = pre_battery - charge_needed) +* )
guarantee "R1.6_and_R1.5" battery_out =
    if goal = chargePosition and atGoal then 100
    else if recharge then battery_in
    else battery_in = longest_path_charge;

(* Req text: Interface shall always satisfy ((0.95 * battery) <= charge_needed(plan)) => recharge *)
guarantee "R1.4.1" recharge = (goal <> chargePosition and battery_in < charge_needed)

guarantee "R1.4.2" atGoal = (goal <> chargePosition and battery_in >= charge_needed )
    or (goal = chargePosition and battery_in >= longest_path_charge);

guarantee "recharge_Eq_not_atGoal" recharge = not atGoal;

tel

contract NavigationSpec ( const grid_width : int;
    const grid_width_square : int;
    const plan_sizeMax : int;
    const planSet_sizeMax: int;
    const chargePosition: int;
    const initialPosition: int;
    const currentPosition: int;
    const heatpoints : int ^ grid_width_square;
    const obstacles : bool ^ grid_width_square) returns ( battery : int;
    visited : bool ^ grid_width_square;)

let

assume grid_width_square = 9;
assume "grid_width" grid_width = 3;
(*Some assumptions on sizes*)

assume plan_sizeMax = 2*grid_width;
assume planSet_sizeMax = 2;

-- Assumptions on locations
assume "chargePosition_is_within_range"
    0 <= chargePosition and chargePosition <= grid_width_square -1;
assume "initialPosition_is_within_range"
    0 <= initialPosition and initialPosition <= grid_width_square - 1;
assume "currentPosition_is_within_range"
    0 <= currentPosition and currentPosition <= grid_width_square - 1;

-- Infrared assumptions

-- assume all heatpoints >= 0
assume "R3.1.3" forall (i:int)
    0 <= i and i < grid_width_square
    => heatpoints[i] >= 0;

(* Req text: Map_Validator shall always satisfy forall_h_in_H => h != initialPosition *)
assume "R3.1.1" heatpoints[initialPosition] = 0;

(* Req text: Map_Validator shall always satisfy forall_h_in_H => h != chargingPosition *)
assume "R3.1.2" heatpoints[chargingPosition] = 0;
(* Req text: Map_Validator shall always satisfy forall_h_in_H => ! obstacle(h) *)
assume "R3.1.3" forall (i:int)
  0 <= i and i < grid_width_square
  => (obstacles[i] => heatpoints[i] = 0);

(* Req text: Map_Validator shall always satisfy for_all_o_in_Obs => (o != initialPosition) *)
assume "R2.4.1" not obstacles[initialPosition];

(* Req text: Map_Validator shall always satisfy forall_o_in_Obs => o != chargingPosition *)
assume "R2.4.2" not obstacles[chargingPosition];

(* Req text: Map_Validator shall immediately satisfy initialPosition = currentPosition *)
assume "R2.2.2" (initialPosition = currentPosition) => true;

(* Req text: Navigation shall always satisfy ! obstacle(currentPosition) *)
assume "R2" not obstacles[currentPosition];

(* guarantee that the rover is never out of battery*)
guarantee "battery_positive" battery > 0;

def var t: int = 0 -> 1 + pre t;
var end_Of_Execution : bool = t >= 2*grid_width_square;
guarantee "all visited at end_Of_Execution" end_Of_Execution =>
  (forall (i:int) 0 <= i and i < grid_width_square
   => (heatpoints[i] > 0 => visited[i]));
tel
Figure 7.1: The top level argument for the loss of rover hazard.
Figure 7.2: The upper part of the argument assuring the running out of battery hazard.
Figure 7.3: The argument fragment for a hazard that leads to the running out of battery hazard.
Figure 7.4: The argument fragment that supports the requirement [R1].
Figure 7.5: An argument fragment for assuring the consistency of the different design modifications.
Figure 7.6: An upper part of the argument for trustworthiness in the formalization of requirement [R1] via FRET.
Figure 7.7: The part of the argument for assuring the results produced by COCOSIM used for verifying requirement [R1].
Figure 7.8: The part of the argument presenting the actual results of the analysis performed through COCOSIM and Kind2 for verification of requirement [R1].
Event-B Model

1 \textbf{CONTEXT} ctx0
2 \textbf{CONSTANTS}
3 \( n \)
4 \( \text{grid} \)
5 \( \text{Obs} \)
6 \( s0 \)
7 \( g \)
8 \( H \)
9 \textbf{AXIOMS}
10 \textbf{axm1:} \( n \in \mathbb{N} \) not theorem
11 \textbf{axm2:} \( n > 0 \) not theorem
12 \textbf{axm3:} \( \text{grid} \subseteq \mathbb{N} \) not theorem
13 \textbf{axm4:} \( \forall x : x \in \text{grid} \Rightarrow x < (n \ast n) - 1 \) not theorem
14 \textbf{axm5:} \( \text{Obs} \subseteq \text{grid} \) not theorem
15 \textbf{axm6:} \( s0 \in \text{grid} \) not theorem
16 \textbf{axm7:} \( g \in \text{grid} \) not theorem
17 \textbf{axm8:} \( H \subseteq \text{grid} \times \mathbb{N} \) not theorem
18 \textbf{axm9:} \( g \notin \text{Obs} \) not theorem
19 \textbf{axm10:} \( s0 \notin \text{Obs} \) not theorem
20 \textbf{axm11:} \( \forall p : p \subseteq \text{grid} \Rightarrow \text{finite}(p) \) not theorem
21 \textbf{axm12:} \( \forall a, b : (a \mapsto b) \in H \Rightarrow a \notin \text{Obs} \) not theorem
22 \textbf{END}

Figure 7.9: Event-B context ctx0.
\textbf{AXIOMS}

\begin{enumerate}
\item\textbf{axm1:} $\text{left} \in \mathbb{N} \rightarrow \mathbb{N}$ \text{not theorem}
\item\textbf{axm2:} $\forall x \cdot x \in \mathbb{N} \Rightarrow \text{left}(x) = (x - 1)$ \text{not theorem}
\item\textbf{axm3:} $\text{right} \in \mathbb{N} \rightarrow \mathbb{N}$ \text{not theorem}
\item\textbf{axm4:} $\forall x \cdot x \in \mathbb{N} \Rightarrow \text{right}(x) = (x + 1)$ \text{not theorem}
\item\textbf{axm5:} $\text{up} \in \mathbb{N} \rightarrow \mathbb{N}$ \text{not theorem}
\item\textbf{axm6:} $\forall x \cdot x \in \mathbb{N} \Rightarrow \text{up}(x) = x - n$ \text{not theorem}
\item\textbf{axm7:} $\text{down} \in \mathbb{N} \rightarrow \mathbb{N}$ \text{not theorem}
\item\textbf{axm8:} $\forall x \cdot x \in \mathbb{N} \Rightarrow \text{down}(x) = x + n$ \text{not theorem}
\item\textbf{axm9:} $\text{adjacent} \in (\text{grid} \times \text{grid}) \rightarrow \mathbb{B}$ \text{not theorem}
\item\textbf{axm10:} $\text{distance} \in (\text{grid} \times \text{grid}) \rightarrow \mathbb{N}$ \text{not theorem}
\item\textbf{axm11:} $\forall x,y \cdot x \in \text{grid} \land y \in \text{grid} \land (\text{distance}(x \rightarrow y) \neq 1)$
\land (\text{distance}(x \rightarrow y) \neq n) \Rightarrow (\text{adjacent}(x \rightarrow y) = \text{FALSE})$ \text{not theorem}
\item\textbf{axm12:} $\forall x \cdot x \in \text{grid} \Rightarrow \text{adjacent}(x \rightarrow x) = \text{TRUE}$ \text{not theorem}
\end{enumerate}

\textbf{END}

\begin{figure}[h!]
\centering
\includegraphics[width=\textwidth]{figure7_10}
\caption{Event-B context ctx1.}
\end{figure}
MACHINE mac0 SEEs ctx1

VARIABLES PlanSet, currentplan, currentloc

INVARIANTS

inv1: ∀ p · p ∈ PlanSet ⇒ p ⊆ grid ∧ (card(p) ≥ 2 ⇒ (g ∈ p ∧ s0 ∈ p)) not theorem

inv3: ∀ p,x · p ∈ PlanSet ∧ x ∈ p ⇒ x ̸∈ Obs not theorem

inv4: currentplan ⊆ grid not theorem

inv5: currentloc ∈ grid not theorem

inv6: ∀ p,x · p ∈ PlanSet ⇒ p ⊆ grid ∧ (card(p) ≥ 2 ⇒ (g ∈ p ∧ r ∈ p ∧ (adjacent(p0 → r) = TRUE) ∧ (adjacent(r → q) = TRUE)) not theorem

inv7: ∀ p,x · p ∈ PlanSet ∧ x ∈ p ⇒ x ̸∈ grid ∧ p0 ∈ grid ∧ (card(p) ≥ 3 ∧ p0 ̸= g ∧ p0 ̸= s0 ⇒ (g ∈ p ∧ (adjacent(p0 → q) = TRUE) ∧ (adjacent(r → p0) = TRUE)) not theorem

inv8: ∀ p · p ∈ PlanSet ⇒ p ⊆ grid not theorem

EVENTS

Initialisation

then act1: PlanSet := {∅} not theorem

act2: currentplan := ∅ not theorem

act3: currentloc := s0 not theorem

Event addstart = ordinary

when grd1: s0 ̸= g not theorem

grd2: currentloc = s0 not theorem

then act1: currentplan := currentplan ∪ {s0} not theorem

Event addcurrentplan = ordinary

when grd1: g ∈ currentplan not theorem

grd2: currentplan ̸∈ PlanSet not theorem

grd3: (g ∈ currentplan ∧ (adjacent(s0 → q) = TRUE) ∧ (adjacent(q → r) = TRUE)) not theorem

grd4: currentplan ⊆ grid not theorem

grd5: g ̸= s0 not theorem

grd6: ∀ p,x · p ∈ PlanSet ∧ p0 ̸= g ∧ p0 ̸= s0 ⇒ (g ∈ p ∧ (adjacent(p0 → q) = TRUE) ∧ (adjacent(r → p0) = TRUE)) not theorem

grd7: s0 ∈ currentplan not theorem

grd8: ∀ x · x ∈ currentplan ⇒ x ̸∈ Obs not theorem

then act1: PlanSet := PlanSet ∪ {currentplan} not theorem

act2: currentloc := s0 not theorem

Event moveLeft = ordinary

when grd1: left(currentloc) ∈ grid not theorem

grd3: adjacent(currentloc → left(currentloc)) = TRUE not theorem

grd4: left(currentloc) ̸∈ Obs not theorem

then act1: currentplan := currentplan ∪ {currentloc} not theorem

act2: currentloc := left(currentloc) not theorem

Event moveRight = ordinary

when grd1: right(currentloc) ∈ grid not theorem

grd2: right(currentloc) ̸∈ Obs not theorem

grd3: adjacent(currentloc → right(currentloc)) = TRUE not theorem

then act1: currentplan := currentplan ∪ {currentloc} not theorem

act2: currentloc := right(currentloc) not theorem

Event moveUp = ordinary

when grd1: up(currentloc) ∈ grid not theorem

grd2: up(currentloc) ̸∈ Obs not theorem

grd3: adjacent(currentloc → up(currentloc)) = TRUE not theorem

then act1: currentplan := currentplan ∪ {currentloc} not theorem

act2: currentloc := up(currentloc) not theorem

Event moveDown = ordinary

when grd1: down(currentloc) ∈ grid not theorem

grd2: down(currentloc) ̸∈ Obs not theorem

grd3: adjacent(currentloc → down(currentloc)) = TRUE not theorem

then act1: currentplan := currentplan ∪ {currentloc} not theorem

act2: currentloc := down(currentloc) not theorem

END

Figure 7.11: Event-B machine mac0.
CONTEXT ctx2
extends ctx1
CONSTANTS
planSet_sizeMax
AXIOMS
axm1: planSet_sizeMax ∈ N not theorem
END

Figure 7.12: Event-B context ctx2.
1 **MACHINE** mac1 **refines** mac0 **SEES** ctx2
2 **VARIABLES** PlanSet, currentplan, currentloc, chosenplan, planningCompleted, returnplan

3 **INVARIANTS**
4 inv1: chosenplan ∈ PlanSet not theorem
5 inv2: finite(chosenplan) not theorem
6 inv3: (planningCompleted = TRUE) ∧ (returnplan = TRUE) ⇒ (∀x. x ∈ PlanSet ⇒ card(chosenplan) ≤ card(x)) not theorem
7 inv4: planningCompleted ∈ BOOL not theorem
8 inv5: finite(PlanSet) not theorem
9 inv6: returnplan ∈ BOOL not theorem

10 **EVENTS**
11 **Initialisation**
12 then act1: PlanSet := {∅}
13 act2: currentplan := ∅
14 act3: currentloc := s0
15 act4: chosenplan := ∅
16 act5: planningCompleted := FALSE
17 act6: returnplan := FALSE
18 Event addstart :=
19 refines addstart
20 when grd1: s0 ≠ g not theorem
21 grd2: currentloc = s0 not theorem
22 then act1: currentplan := currentplan ∪ {s0}
23 Event addcurrentplan :=
24 refines addcurrentplan
25 when grd1: g ∈ currentplan not theorem
26 grd2: currentplan ∈ PlanSet not theorem
27 grd3: (∃p,q. p ∈ currentplan ∧ q ∈ currentplan ∧ (adjacent(p → q) = TRUE)) not theorem
28 grd4: currentplan ⊆ grid not theorem
29 grd5: g ≠ s0 false
30 grd6: ∀p0. p0 ∈ currentplan ∧ p0 ≠ g ∧ p0 ≠ s0 ⇒ (∃q. q ∈ currentplan ∧ r ∈ currentplan ∧ (adjacent(p0 → q) = TRUE) ∧ (adjacent(r → p0) = TRUE)) not theorem
31 grd7: s0 ∈ currentplan not theorem
32 grd8: currentplan := currentplan ∪ {currentloc}
33 then act1: PlanSet := PlanSet ∪ {currentplan}
34 act2: currentloc := s0
35 Event moveLeft := ordinary
36 refines moveLeft
37 when grd1: left(currentloc) ∈ grid not theorem
38 grd2: adjacent(left(currentloc) ⇒ left(currentloc)) = TRUE not theorem
39 grd3: left(currentloc) ∉ Obs not theorem
40 then act1: currentplan := currentplan ∪ {currentloc}
41 act2: currentloc := left(currentloc)
42 Event moveRight := ordinary
43 refines moveRight
44 when grd1: right(currentloc) ∈ grid not theorem
45 grd2: right(currentloc) ∉ Obs not theorem
46 grd3: adjacent(right(currentloc) ⇒ right(currentloc)) = TRUE not theorem
47 grd4: planningCompleted = FALSE not theorem
48 then act1: currentplan := currentplan ∪ {currentloc}
49 act2: currentloc := right(currentloc)

Figure 7.13: Event-B machine mac1 part 1/2.
Event moveUp ≡ ordinary
refine moveUp
when grd1: up(currentloc) ∈ grid not theorem
grd2: up(currentloc) ∉ Obs not theorem
grd3: adjacent(currentloc → up(currentloc)) = TRUE false
grd4: planningCompleted = FALSE not theorem
then act1: currentplan := currentplan ∪ {currentloc}
act2: currentloc := up(currentloc)

Event moveDown ≡ ordinary
refine moveDown
when grd1: down(currentloc) ∈ grid not theorem
grd2: down(currentloc) ∉ Obs not theorem
grd3: adjacent(currentloc → down(currentloc)) = TRUE not theorem
grd4: planningCompleted = FALSE not theorem
then act1: currentplan := currentplan ∪ {currentloc}
act2: currentloc := down(currentloc)

Event PRA ≡ ordinary
any
p
when grd1: p ∈ PlanSet not theorem
grd2: \( \forall q \cdot q \in \text{PlanSet} \Rightarrow \text{card}(p) \leq \text{card}(q) \) not theorem
grd3: planningCompleted = TRUE not theorem
then act1: chosenplan := p
act2: returnplan := TRUE

Event stopPlanning ≡ ordinary
when grd1: \( \text{card}(\text{PlanSet}) \geq \text{planSet_sizeMax} \) not theorem
grd2: returnplan = FALSE not theorem
then act1: planningCompleted := TRUE

END

Figure 7.14: Event-B machine mac1 part 2/2.
**Integration and Evaluation of the AdvoCATE, FRET, CoCoSim, and Event-B Tools on the Inspection Rover Case Study**

**Author(s):** Hamza Bourbouh, Marie Farrell, Anastasia Mavridou, Irfan Sljivo

**Performing Organization:** NASA Ames Research Center, Moffett Field, CA 94035

**Abstract:** The complexity and flexibility of autonomous robotic systems necessitates a range of distinct verification tools. This presents new challenges not only for design verification but also for assurance approaches. Combining the distinct formal verification tools, while maintaining sufficient formal coherence to provide compelling assurance evidence is difficult, often being abandoned for less formal approaches. In this technical memorandum, we demonstrate, through a case study, how a variety of distinct formal techniques can be brought together in order to develop a justifiable assurance case. We use the AdvoCATE assurance case tool to guide our analyses and to integrate the artifacts from the formal methods that we use, namely: FRET, CoCoSim and Event-B. While we present our methodology as applied to a specific Inspection Rover case study, we believe that this combination provides benefits in maintaining coherent formal links across development and assurance processes for a wide range of autonomous robotic systems.

This technical report provides a more detailed overview of the work presented in [1]. In addition to thorough and detailed descriptions, it contains initial design models and describes some of the artifacts that were omitted from discussion in [1].

For full source code and models please contact the authors.