Integrating Formal Verification and Assurance: An Inspection Rover Case Study* **

Hamza Bourbouh[1], Marie Farrell[3]***, Anastasia Mavridou[1], Irfan Sljivo[1], Guillaume Brat[2], Louise A. Dennis[4], and Michael Fisher[4]

[1] KBR/NASA Ames Research Center, USA
[2] NASA Ames Research Center, USA
[3] Department of Computer Science, Maynooth University, Ireland
[4] Department of Computer Science, University of Manchester, UK

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 paper 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.

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 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 [40]), the development of a tool/formalism that incorporates

* Work supported by NASA ARMD System-Wide Safety Project, and UK Research and Innovation and EPSRC Hubs for "Robotics and AI in Hazardous Environments": EP/R026092 (FAIR-SPACE).

** The authors thank Dimitra Giannakopoulou for her valuable feedback on this work.

*** Majority of Farrell's work took place at the Universities of Liverpool & Manchester.

multiple others (e.g. Why3 [20]), or the construction of systematic translations between tools/formalizms (e.g. EventB2JML [39]). Recent work argues that, for autonomous robotic systems, the use of multiple formal and non-formal verification techniques is both beneficial and necessary to ensure that such systems behave correctly [19, 29]. Notably, 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) [38]. However, other middlewares such as NASA's core Flight System (cFS) [35] also support the development of similarly complex, modular systems.

In this paper, 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 [16] to perform safety engineering and assurance, FRET [23] to elicit and formalize requirements, and COCOSIM [6] with Kind2 to perform compositional verification of the system-level requirements. Further, we use Event-B [2] and Kind2 for the component-level formal verification. AdvoCATE facilitates the integration of the artifacts/evidence produced from these tools.

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 their limitations (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.

2 Tool Support

Assurance Case Automation Toolset (AdvoCATE) [16]: supports the development and management of assurance cases, which are composed of all of the assurance-related artifacts that are created during system development. To enable automation, AdvoCATE is built with a formal basis where all of the assurance artifacts can be defined and formally related. Some artifacts can be created directly in AdvoCATE (e.g., hazard log, bow tie diagrams), while others, such as formal verification results, can be imported. AdvoCATE uses Goal Structuring Notation (GSN) [1] to document assurance cases in the form of arguments.

Formal Requirements Elicitation Tool (FRET) [23]: is an open source framework [22] for the elicitation, formalization and understanding of requirements. FRET helps understanding and review of semantics by utilizing a variety of forms for each requirement: natural language description, formal logics, and informal diagrams. System requirements are defined in a hierarchical fashion using a structured natural language with precise meaning, and can be exported in a variety of forms to be used by formal analysis tools such as COCOSIM [33, 34].

Contract-based Compositional Verification of Simulink Models (COCOSIM) [6]: is an open source framework [12] for Simulink/Stateflow formal verification. COCOSIM translates a Simulink model into Lustre code [25], which can then be verified using the Kind2 model-checker [9]. COCOSIM annotates the model with assume-guarantee contracts. Verification can then be performed in a compositional way or by checking the contracts against component behavior.

Event-B [2]: is a formal method that is used in the verification of cyber-physical systems [30, 4, 36]. Event-B uses a set-theoretic modelling notation and supports formal refinement. Event-B models are composed of machines, which model the dynamic components of a systems' specification, and contexts, which model the static components. Event-B has tool support via the Rodin Platform, an Eclipse-based IDE, which generates proof obligations for a given specification and provides support for automatic and interactive proof with the Atelier-B prover [3].

3 Assurance-based Formal Methods Integration

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 [18, 26] that can be used in the design and development of mission-critical systems. In particular, existing guidelines [26] 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.

In our methodology we focus on the application of formal methods and connect it to the parts of a greater system safety assurance methodology [13] needed to perform and assure 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 in Simulink.
Step 2: Perform preliminary hazard analysis.
Step 3: Define mitigations and safety requirements.
Step 4: Refine Simulink 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. 1 presents a detailed view of our methodology. The upper part of Fig. 1 shows the system-level concept, design, and assurance steps that are mainly performed by the AdvoCATE tool, while the lower part shows the formal methods application steps performed by the FRET, COCOSIM, and Event-B tools. In the analysis phase (step 6) we perform two types of analysis. We use COCOSIM to

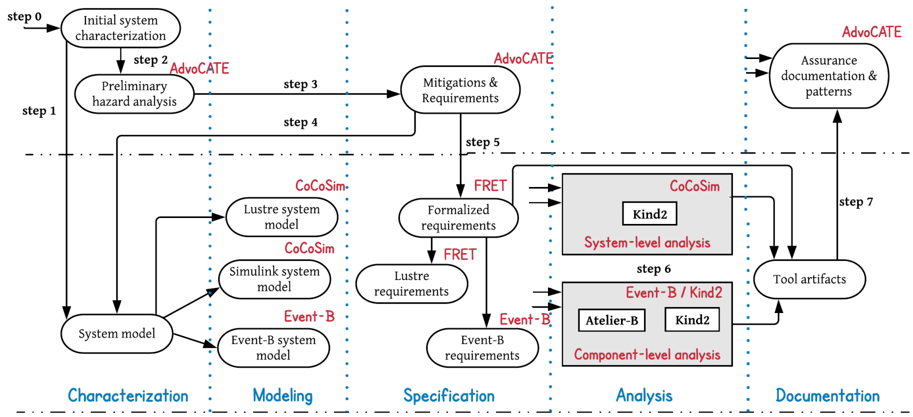

Fig. 1. Our methodology for integrating formal verification results via AdvoCATE. The incoming arrows without a source represent all relevant artifacts from previous phases. For system-level analysis these comprise the Lustre requirements and the Simulink system model, while for component-level analysis these comprise the Lustre and Event-B system models and requirements. In the documentation phase we input all artifacts.

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, 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 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. To 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 [5], 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[5]. 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

[5] UKRI and EPSRC Hubs for "Robotics and AI in Hazardous Environments".

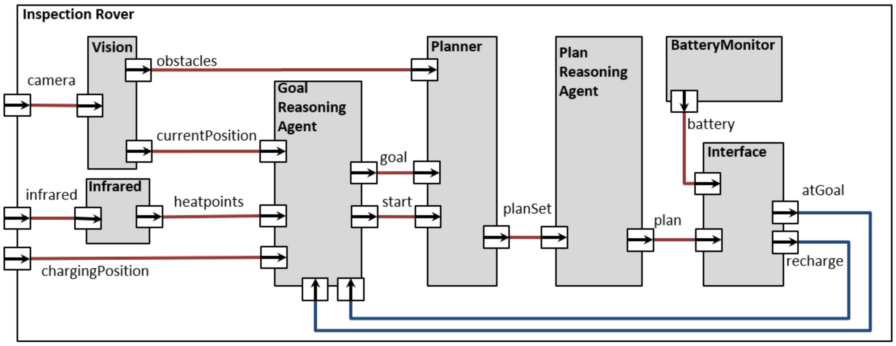

Fig. 2. Preliminary inspection rover system architecture.

and domain experts. Steps 2 and 3 were performed by the safety expert. Steps 4 and 6 were performed by the safety, domain and Simulink experts. Step 5 was performed by the requirements and domain experts, and finally step 7 was performed mainly by the safety expert with contributions from all other experts.

4 The Case Study Step-by-Step

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.

4.2 Step 1: Create Initial Simulink Model

Simulink is usually used by engineers for developing an executable system specification, 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 implementations 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).

In step 1, we created the initial Simulink model, which comprises a preliminary architecture of our rover (Fig. 2). The rover must navigate to all heat positions on a 2D grid map of known size. The *Vision* system detects obstacles to be avoided. The *Infrared* component identifies grid locations that are hotter than expected. From 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* computes obstacle-free plans for navigating from the current position to the goal. The autonomous

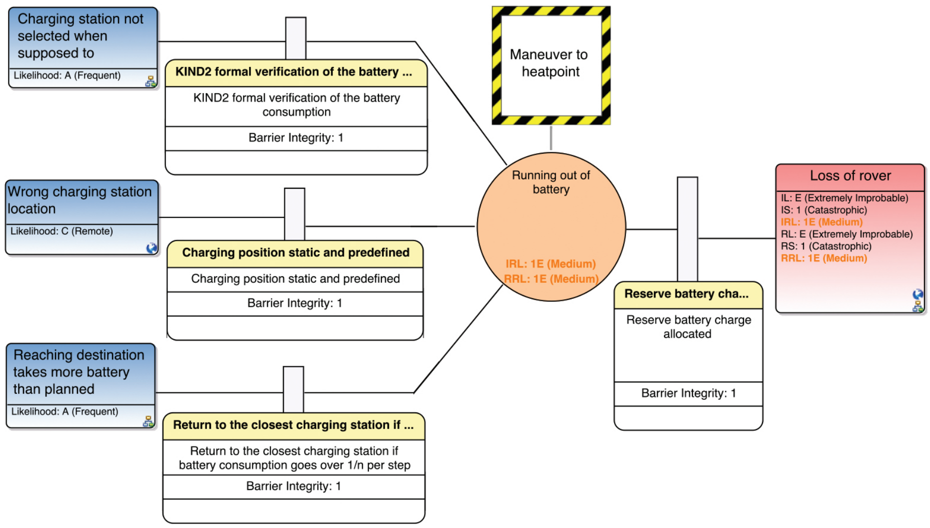

Fig. 3. Bow Tie Diagram presenting the *running out of battery* hazard (orange circle), its causes (blue rectangles to the left) and consequence (red rectangle to the right).

Plan Reasoning Agent selects the shortest plan. Finally, the *Interface* translates the navigation actions of the plan into the instructions for 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.

4.3 Step 2: Perform Preliminary Hazard Analysis

To perform the preliminary hazard analysis in AdvoCATE as part of the safety assurance methodology [13], we first defined a functional decomposition of the Inspection Rover based on Fig. 2. Then, we performed the traditional hazard analysis (FMEA [42]) in the AdvoCATE hazard log. We 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. E.g., 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 [15]. A single BTD shown in Fig. 3 details the causes and consequences of the *running out of battery* hazard.

4.4 Step 3: Define Mitigations and Safety Requirements

After preliminary hazard analysis, we conducted a risk analysis 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 performed mitigation planning using BTDs. For example, in order to minimize the risk of *running out of battery* shown in Fig. 3: (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 case the hazard still occurs.

For each of the two top-level hazards, *loss of rover* and *inspection finished before visiting 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. 3 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. The full list of child requirements for these system-level requirements is presented in [5].

4.5 Step 4: Refine Simulink Model According to Mitigations

Some of the identified mitigations required design modifications resulting in a refined system architecture (Fig. 4), which was reassessed in terms of hazards and mitigations. For brevity, we present this as a single step but there are iterations between these steps in practice. We consider the initial rover position and the charging position 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 by adding *MapValidator* to check that the initial position, charging position, obstacles and heat points are mutually exclusive. Furthermore, *MapValidator* checks that the initial position, as recognized by *Vision*, is equal to the pre-defined *initialPosition*.

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, current rover position, heat points and the charger position. It outputs: (1) a plan from the current position to the goal (*plan2D*), (2) a plan from the goal to the charger location (*plan2C*), and (3) the list of *visited* locations. Within the *ReasoningAgent*, the goal reasoning agent (*GRA*) chooses the next goal as either the hottest heat point not visited before or as the charger if the *recharge* flag is set to true by *Battery_Interface*. The *GRA* updates the *visited* locations.

The *ReasoningAgent* contains *ComputePlan2Charging* and *ComputePlan2Destination* which both have a *Planner* and plan reasoning agent (*PRA*). These re-

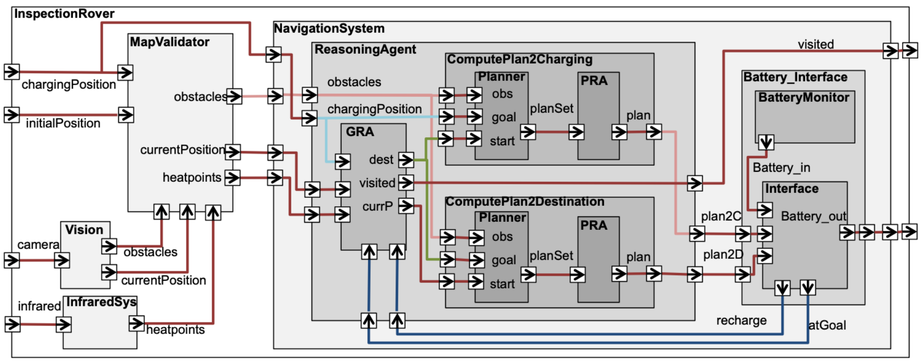

Fig. 4. Upgraded inspection rover architecture with additional components and data.

turn the shortest plan from the goal to the charger (*ComputePlan2Charging*) and the shortest plan from the current position to the goal (*ComputePlan2Destination*).

Battery_Interface contains a *BatteryMonitor* and a hardware *Interface*. The *Interface* takes the plans from *NavigationSystem* and the battery status from the *BatteryMonitor* as input, and returns two flags indicating whether the rover has reached the goal (*atGoal*) and the status of the battery charge (*recharge*). The *recharge* flag becomes true if the current battery charge is insufficient to follow the plan to the goal (*plan2D*) and return to the charging station (*plan2C*).

If the *recharge* flag is false, then the *Interface* executes the plan and returns *atGoal* as true once it reaches the goal. However, if *recharge* is true, then *atGoal* is set to false. We note that we do not need both of these outputs since we have always *recharge* = *not 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 *NavigationSystem* and *Interface* have a similar execution frequency.

4.6 Step 5: Formalize Requirements and Create Formal Specifications

We manually wrote the requirements in the restricted natural language of FRET, i.e., FRETISH, which has a precise, unambiguous meaning. FRETISH requirements contain up to six fields: scope, condition, component*, shall*, timing, and response*, with mandatory fields indicated by '*'. 'component' specifies the component that the requirement refers to and 'shall' expresses 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 example, 'scope' can specify system behavior *after* a mode ends, or when the system is *in* a mode. 'condition' defines a Boolean expression that triggers a 'response'. When triggered, the response must occur as specified by the timing, e.g., *immediately*, *always*, *for/within N time units*.

For each FRETISH requirement, FRET produces natural language and diagrammatic explanations of its exact meaning, and formalizes the requirement in temporal logics. 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.
Notice that 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"*.

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 formulas. 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 formulas use exclusively past-time temporal operators, i.e., `Y, O, 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 formulas for **[R1]** and **[R1.2]** are:
[R1]: `H(battery>0);`
[R1.2]: `H((recharge & (Y(!recharge) | FTP))`$\Rightarrow$`(goal=chargePosition));`
where `FTP` means First Time Point of execution (equivalent to $\neg$ `Y TRUE`). From the pmLTL formulas we automatically generated Lustre-based verification contracts that can be directly fed into COCOSIM for verification with the Kind2 model checker using the COCOGEN tool, which is integrated in FRET. COCOGEN is described in [32]. For example, below is the generated Lustre code for **[R1]**:

```
guarantee "R1.2" (battery > 0);
```

If requirements were based only on model inputs, e.g., **[R1.2]**, then CoCoGen generates assumptions (instead of guarantees):

```
assume "R1.2" ((recharge and ((pre (not recharge)) or FTP)) => (goal =
    chargePosition));
```

where `FTP = true` $\rightarrow$ `false`. As mentioned earlier, some requirements used first-order operators such as **[R3.3]** which was generated as follows:

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

Notice that the `forAll_i` placeholder was replaced by `forall (i:int)`, and `i_inGrid` was replaced by `(0 <= i and i < width)` during generation.

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 (planningCompleted & returnPlan) then (if (forAll_x & x_inPlanSet) then (card(chosenPlan) <= card(x))), where the `card()` function computes the length of a path. The corresponding Event-B invariant was based on the FRETISH version:

$(planningCompleted = TRUE) \wedge (returnplan = TRUE) \Rightarrow (\forall x \cdot x \in PlanSet \Rightarrow card(chosenplan) \leq card(x))$

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

$\forall p, x \cdot p \in PlanSet \wedge x \in p \Rightarrow x \notin Obs$
where every element of `PlanSet` is a set of grid locations.

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

Compositional Verification in CoCoSim: Our objective was to attach the component-level child requirements to the relevant component(s) and then, using COCOSIM, compositionally verify the system-level parent requirements. We were not able to model/verify all requirements, e.g., *The current position as recognized by the rover is its current physical position* should be physically tested.

Compositional verification in COCOSIM involves defining a top system node with associated system-level contract. During verification, the model checker attempts to show that these system-level properties can be successfully derived from the component-level contracts. Using compositional reasoning in COCOSIM, we were able to verify system-level requirements **[R1]** and **[R3]**, defined in §4.4. However, we could not verify **[R2]** which involves the *Vision* and the *Planner* components, because there is no COCOSIM model for the *Vision* component.

Compositional verification of **[R1]** was achieved quite quickly ($<$ 20 secs), as the model checker only had to analyse two components: the *Interface* and *BatteryMonitor* to verify **[R1]**. **[R3]** was more complex since it involved a loop between the *Interface* and *ReasoningAgent*. Kind2 had to carry out a lot of unrolling to adequately assess this property and deal with more complex contracts including quantifiers and arrays. Thus, we were only able to prove **[R3]** for specific grid widths (minutes for 3×3, hours for 4×4, and larger grids timed out).

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 here [19, 29]. 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 constructed a simple Lustre implementation of the *GRA* that we verified using Kind2. Full details can be found in [5]. The *GRA* computes the `start`, `goal` and the `visited` cells. The `start` is initialized as the `currentPosition`, if the goal was reached during the last execution (`atGoal` is true) then the start is the previous goal (`pre_goal`), if the `recharge` flag is true then the `start` is the previous start position since the rover did not move. The `goal` is set to `chargingPosition` if the `recharge` flag is active. Otherwise we choose the hottest heat point, computed using the `hottestPoint` local array that keeps track of the hottest heat point. We used Kind2 to verify all of the properties specified in the specification. We were able to verify most properties in less than 1 second. Due to state space explosion, there were some properties, e.g., requirement [**R3.3**] that were only provable for specific grid sizes. E.g., we verified [**R3.3**] for a grid size up to 4x4.

Specification and Verification of ComputePlan using Event-B: Our Event-B model contains three contexts (modelling static aspects) and two machines (modelling dynamic aspects). Event-B supports formal refinement, so our contexts extend one another and our machines indicate refinement steps. Our most primitive context, `ctx0`, specifies basic details such as 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. This is extended via `ctx1` which specifies functions that capture the behavior of the planning component.

The abstract machine, `mac0`, models a simple search-based planning algorithm that produces a set of plans containing the start and goal. Event-B uses sets as primitive so we ensure that these plans, encoded as sets, can be linearized using the `adjacent` function specified in `ctx1`. The refinement, `mac1`, incorporates a plan reasoning agent and chooses the shortest plan from `PlanSet`. Another context, `ctx2`, defines a constant to limit the number of generated plans.

We encoded [**R2.1**], [**R2.4.1**], [**R2.4.3**], [**R2.4.4**], [**R2.5**] and [**R3.4**] in our Event-B model. We could not verify [**R2**] compositionally but its child requirements feature in our Event-B model (e.g. [**R.2.5**]). This ensures that the planning components do not accidentally cause the rover to collide with an obstacle. Most of the Event-B proof obligations were proven automatically by Atelier-B in Rodin. Those requiring interactive proof were relatively straightforward.

Event-B was not limited by the state space explosion that was causing Kind2 to time out. We specified more complex component-level properties that would have been difficult to verify for a model-checker. [5] contains the Event-B model.

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. 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

Fig. 5. 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).

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. 5 presents an argument fragment about mitigating the *running out of battery* hazard that causes *loss of rover*. Similar arguments exist for other causes of *loss of rover* and the other hazards. For brevity, Fig. 5 only contains a fragment of the existing argument. For example, 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). Full details can be found in [5].

The goal *G14* focuses on **[R1]** that was verified using COCOSIM. We have created a similar argument for each of the system-level requirements verified compositionally with COCOSIM. For each, we extended the automatically generated arguments based on a combination of existing argumentation patterns [14, 41] to argue application-specific goals (base of Fig. 5): (**G3-A1**) the formalisation of the natural language requirement is correct; (**G4-A1**) the results from

COCOSIM are trustworthy; (**G5-A1**) the different design representations are consistent; (**G6-A1**) the COCOSIM verification result for [**R1**] is valid.

To ensure that the different design representations were consistent across the tools, we performed manual reviews where automated consistency validation was not available. E.g., we used manual reviews to verify that the design as specified in AdvoCATE was consistent with the Simulink, Kind2 and Event-B models.

The goal *G3-A1* relates to the correct specification of [**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* presents the rationale behind the COCOSIM results that [**R1**] is valid in the design. 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 in the goal *G4-A1*. Since COCOSIM relies on some model transformations and external tools for verification, the correctness of these has to be established. For example, we argued the correctness of the translation from Simulink to Lustre code that is used by Kind2.

5 Discussion

Using the given formal verification tools we were able to 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.

Although in most cases, the initial natural language requirements looked relatively straightforward, a closer study revealed many questions regarding their precise meaning. Translating the natural language requirements into FRETISH was not always straightforward. To this end, the semantic explanations and simulation capabilities offered by FRET were instrumental in ensuring that the FRETISH requirements captured our intended semantics. Notice that we could not directly encode first-order logic requirements in FRETISH. We tackled this problem using auxiliary variables as placeholders for the quantifiers at requirement-level, but a FRETISH-level solution is desirable. Finally, we noticed that most of the Inspection Rover requirements follow a small number of patterns, a characteristic that we have observed in other studies within our organization.

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/or required physical testing. This supports the claim that the robotics domain requires both formal and informal verification processes [19]. E.g., everything depends on the accuracy of the rover's current position - a property that we could not formally verify in this case. However, by formalizing the requirements to be verified via testing, we can potentially incorporate run-time analysis. Specifically, the formalized properties can be used to generate formal run-time monitors to help with fault management during operation. These might 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.

Approaches to integrating formal methods often rely on bespoke translations between languages/tools. However, these translations can be difficult and sometimes impossible to correctly formalize/implement. Further, if used in an assurance case then the translations themselves must be assured, as for our translation from FRET to COCOSIM [5]. Although the use of tightly integrated formal methods is desirable, our approach, using an assurance case as the point of integration, incorporates tools for which such systematic translations do not exist by providing arguments demonstrating how to link models in distinct formalizms.

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

Although our methodology (Fig. 1) uses specific tools, we believe that this approach can be followed irrespective of the choice of specific tools. In particular, the phases at the base of Fig. 1 relate to existing development guidelines [26].

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 [7].

Other approaches to compositional verification include AGREE [37] and OCRA [11]. 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 [17], TLA+ [43] and Dafny [28].

Isabelle/SACM [21, 24] 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 paper, we have illustrated the benefits of using various formal verification techniques. Related to this, [10, 27, 31] 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.

7 Conclusions and Future Work

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 methods can impact system design and discussed how a heterogeneous set of verification results can be linked during assurance with AdvoCATE. Further, we made our case study artifacts publicly available to fuel discussion in the research community.

This work has opened up a number of avenues for future research. In particular, we would like to support the definition of probabilistic requirements in FRET, since such requirements are increasingly used in complex robotic systems. Additionally, we intend to develop a DSL to facilitate the integration of AdvoCATE with different verification tools. Furthermore, 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.

References

1. GSN Community Standard Version 2. Technical report, Assurance Case Working Group of The Safety-Critical Systems Club, Jan. 2018.
2. J.-R. Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.
3. J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010.
4. R. Banach. Hemodialysis machine in hybrid event-b. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 376–393. Springer, 2016.
5. H. Bourbouh, M. Farrell, A. Mavridou, and I. Sljivo. Integration and evaluation of the Advocate, FRET, CoCoSim, and Event-B tools on the Inspection Rover Case Study. Technical report, TM–2020–5011049, NASA, 2021.
6. H. Bourbouh, P.-L. Garoche, T. Loquen, É. Noulard, and C. Pagetti. Cocosim, a code generation framework for control/command applications an overview of cocosim for multi-periodic discrete simulink models. In *European Congress on Embedded Real Time Software and Systems*, 2020.
7. R. C. Cardoso, L. A. Dennis, M. Farrell, M. Fisher, and M. Luckcuck. Towards compositional verification for modular robotic systems. In *Workshop on Formal Methods for Autonomous Systems*, pages 15–22. Electronic Proceedings in Theoretical Computer Science, 2020.
8. R. C. Cardoso, M. Farrell, M. Luckcuck, A. Ferrando, and M. Fisher. Heterogeneous verification of an autonomous curiosity rover. In *NASA Formal Methods Symposium*, pages 353–360. Springer, 2020.
9. A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. The kind 2 model checker. In *International Conference on Computer Aided Verification*, pages 510–517. Springer, 2016.
10. M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *International Symposium on Formal Methods*, pages 132–146. Springer, 2012.
11. A. Cimatti, M. Dorigatti, and S. Tonetta. Ocra: A tool for checking the refinement of temporal contracts. In *International Conference on Automated Software Engineering*, pages 702–705. IEEE, 2013.
12. CoCoSim-Team. CoCoSim – automated analysis framework for simulink. `https://github.com/NASA-SW-VnV/CoCoSim`.
13. E. Denney and G. Pai. Automating the assembly of aviation safety cases. *IEEE Transactions on Reliability*, 63(4):830–849, 2014.
14. E. Denney and G. Pai. Safety case patterns: theory and applications. *NASA/TM2015218492*, 2015.
15. E. Denney and G. Pai. Architecting a safety case for uas flight operations. In *34th international system safety conference*, volume 12, 2016.
16. E. Denney and G. Pai. Tool support for assurance case development. *Automated Software Engineering*, 25(3):435–499, 2018.
17. L. A. Dennis and B. Farwer. Gwendolen: A BDI Language for Verifiable Agents. In *Workshop on Logic and the Simulation of Interaction and Reasoning*, pages 16–23. AISB, 2008.
18. H. Dezfuli, A. Benjamin, C. Everett, M. Feather, P. Rutledge, D. Sen, and R. Youngblood. Nasa system safety handbook. volume 2: System safety concepts, guidelines, and implementation examples. 2015.

19. M. Farrell, M. Luckcuck, and M. Fisher. Robotics and Integrated Formal Methods: Necessity meets Opportunity. In *International Conference on Integrated Formal Methods*, pages 161–171. Springer, 2018.
20. J.-C. Filliâtre and A. Paskevich. Why3—where programs meet provers. In *European symposium on programming*, pages 125–128. Springer, 2013.
21. S. D. Foster, Y. Nemouchi, C. O'Halloran, N. Tudor, and K. Stephenson. Formal model-based assurance cases in isabelle/sacm: An autonomous underwater vehicle case study. In *Formal Methods in Software Engineering*. ACM, 2020.
22. FRET-Team. FRET – formal requirements elicitation tool. `https://github.com/NASA-SW-VnV/FRET`.
23. D. Giannakopoulou, A. Mavridou, T. Pressburger, J. Rhein, J. Schumann, and N. Shi. Formal requirements elicitation with FRET. In *Requirements Engineering: Foundation for Software Quality*, 2020.
24. M. Gleirscher, S. Foster, and Y. Nemouchi. Evolution of formal model-based assurance cases for autonomous robots. In *International Conference on Software Engineering and Formal Methods*, pages 87–104. Springer, 2019.
25. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
26. J. C. Kelly. Formal methods specification and analysis guidebook for the verification of software and computer systems volume ii: A practitioner's companion. 1997.
27. V. H. Le, L. Correnson, J. Signoles, and V. Wiels. Verification coverage for combining test and proof. In *International Conference on Tests and Proofs*, pages 120–138. Springer, 2018.
28. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.
29. M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher. Formal Specification and Verification of Autonomous Robotic Systems: A Survey. *ACM Computing Surveys (CSUR)*, 52(5):1–41, 2019.
30. A. Mammar and R. Laleau. Modeling a landing gear system in event-b. *International Journal on Software Tools for Technology Transfer*, 19(2):167–186, 2017.
31. F. Maurica, D. R. Cok, and J. Signoles. Runtime assertion checking and static verification: collaborative partners. In *International Symposium on Leveraging Applications of Formal Methods*, pages 75–91. Springer, 2018.
32. A. Mavridou, H. Bourbouh, P.-L. Garoche, D. Giannakopoulou, T. Pressburger, and J. Schumann. Bridging the gap between requirements and Simulink model analysis. In *26th Intl Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020, Poster)*, 2020.
33. A. Mavridou, H. Bourbouh, P.-L. Garoche, and M. Hejase. Evaluation of the FRET and CoCoSim tools on the ten Lockheed Martin Cyber-Physical challenge problems. Technical report, TM-2019-220374, NASA, 2019.
34. A. Mavridou, H. Bourbouh, D. Giannakopoulou, T. Pressburger, M. Hejase, P. L. Garoche, and J. Schumann. The Ten Lockheed Martin Cyber-Physical Challenges: Formalized, Analyzed, and Explained. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*, pages 300–310, 2020.
35. D. McComas. Nasa/gsfc's flight software core flight system. In *Flight Software Workshop*, volume 11, 2012.
36. D. Méry and N. K. Singh. Formal development and automatic code generation: Cardiac pacemaker. In *International Conference on Computers and Advanced Technology in Education*, 2011.

37. A. Murugesan, M. W. Whalen, S. Rayadurgam, and M. P. Heimdahl. Compositional verification of a medical device system. In *SIGAda annual conference on High integrity language technology*, pages 51–64, 2013.
38. M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5, 2009.
39. V. Rivera and N. Cataño. Translating event-b to jml-specified java programs. In *ACM Symposium on Applied Computing*, pages 1264–1271, 2014.
40. S. Schneider, H. Treharne, and H. Wehrheim. A csp approach to control in event-b. In *International Conference on Integrated Formal Methods*, pages 260–274. Springer, 2010.
41. I. Sljivo, B. Gallina, J. Carlson, H. Hansson, and S. Puri. Tool-supported safety-relevant component reuse: From specification to argumentation. In A. Casimiro and P. M. Ferreira, editors, *Reliable Software Technologies – Ada-Europe 2018*, pages 19–33, Cham, 2018. Springer International Publishing.
42. D. H. Stamatis. *Failure mode and effect analysis: FMEA from theory to execution*. Quality Press, 2003.
43. Y. Yu, P. Manolios, and L. Lamport. Model checking tla+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.