Preprint:
Yan, Fang, Foster, Simon, Cavalcanti, Ana et al. (2026) Formal Evidence Generation for Assurance Cases for Robotic Software Models. [Preprint]

https://doi.org/10.48550/arXiv.2602.03550

Noname manuscript No.
(will be inserted by the editor)

Formal Evidence Generation for Assurance Cases for Robotic Software Models

Fang Yan · Simon Foster · Ana Cavalcanti · Ibrahim Habli · James Baxter

Abstract Robotics and Autonomous Systems are increasingly deployed in safety-critical domains, so that demonstrating their safety is essential. Assurance Cases (ACs) provide structured arguments supported by evidence, but generating and maintaining this evidence is labour-intensive, error-prone, and difficult to keep consistent as systems evolve. We present a model-based approach to systematically generating AC evidence by embedding formal verification into the assurance workflow. The approach addresses three challenges: systematically deriving formal assertions from natural language requirements using templates, orchestrating multiple formal verification tools to handle diverse property types, and integrating formal evidence production into the workflow. Leveraging RoboChart, a domain-specific modelling language with formal semantics, we combine model checking and theorem proving in our approach. Structured requirements are automatically transformed into formal assertions using predefined templates, and verification results are automatically integrated as evidence. Case studies demonstrate the effectiveness of our approach.

Keywords Assurance Case, model checking, theorem proving, automated evidence generation, specification formalisation, RoboChart

1 Introduction

As Robotics and Autonomous Systems (RAS) increasingly permeate safety-critical sectors like healthcare, automotive, and aerospace, ensuring the safety of these systems is paramount. Given their complexity and autonomy, RAS must undergo rigorous verification throughout their development and operational lifecycle.

In safety-critical industries, the development of assurance Cases (ACs), commonly required by safety standards such as MOD Def Stan 00-55 [1] and ISO26262 [2], has become common practice. ACs are living documents that evolve alongside the systems they support. A key component of ACs is evidence, which substantiates claims about the system's compliance with key properties, such as the safety requirements. However, generating and updating this evidence is often labour-intensive, prone to errors, and difficult to trace, especially as systems undergo changes or adaptations in uncertain environments. These challenges highlight the need for systematically generating and integrating evidence through automation to ensure ACs remain consistent and aligned with system updates [3–5].

Formal verification techniques, such as model checking and theorem proving, can provide evidence that system models satisfy properties by verifying behaviours against formal specifications. Formal verification can generate rigorous evidence to support AC claims. While various studies have explored different ways of linking verification results to assurance arguments, full automation, providing traceability, maintainability, and adaptability, ensuring consistency of ACs throughout the lifetime of the system, remains elusive and poses significant difficulties.

The first challenge is integrating the production of verification evidence into the workflow for the development of ACs. Traditionally, formal verification is taken into account in ACs through verification results that are referenced at the evidence nodes [6–8]. So, the process of obtaining these results is usually left outside the workflow for the development of ACs. Existing approaches [3, 9] have sought to address this issue, with some integrating code-level formal verification or simulation results directly into the AC process. Further work [10] has demonstrated the integration of model-level formal verification into ACs. However, automation is only partially achieved. Consequently, when the system evolves, formal verification is not automatically triggered, and evidence may therefore fail to be updated, leading to inconsistencies between claims and their supporting evidence.

The second challenge is the description of property specifications as formal tool-ready assertions. In current practice, requirements in safety-critical domains are usually expressed in natural language or semi-structured formats. So, the use of formal verification tools requires the involvement of formal methods experts to formalise assertions. This is a major obstacle to automation.

A substantial body of work has targeted automation of the generation of temporal logic assertions from property specifications written in natural language. Early approaches [10–14] formalise requirements into LTL using templates,

University of York, York, UK
E-mail: {fang.yan, simon.foster, ana.cavalcanti, Ibrahim.habli, james.baxter}@york.ac.uk

This is a preprint. The paper is currently under review at Software and Systems Modeling.

controlled natural languages, or ontology-based parsing. They rely on requirements being rewritten in specialised notations or supported by domain-specific ontologies, and in some cases, can only generate relatively simple assertions. More recent work has used large language models (LLMs) to translate natural language into LTL specifications [15–18], but with limited accuracy.

Very few approaches have addressed probabilistic temporal logics, such as PCTL, which is used in probabilistic model checking via tools like PRISM [19]. For CSP [20], the complexity of specifications, covering concurrency, communication, and refinement, makes the automation of formalisation particularly challenging. Some attempts are based on diagrammatic notations for properties of components of RoboChart [21] models [13, 22], where requirements are manually specified as diagrams and then translated into CSP assertions for FDR [23].

The third challenge is orchestrating multiple formal verification tools. The need for combining different formal methods has long been recognised in the literature on integrated formal methods [24]. Prokhorova et al. [8], for example, demonstrate how different requirement types can be covered in an AC argument by combining different formal verification tools. However, that work does not automate the coordination of these tools. Some approaches embed verification into the AC development process, but usually integrate only one type of formal-verification engine [9]. Consequently, providing a unified framework that can automatically delegate requirements to suitable verification backends, collect their results, and feed them back as assurance evidence remains an open challenge.

In this work, we leverage RoboChart [21], a domain-specific modelling language for robotic systems, to support evidence generation through formal verification. RoboChart's formal semantics, combined with verification tools such as FDR and PRISM, facilitate the automatic production of verification evidence, which can then be systematically integrated into ACs. To address the above challenges, we employ both model checking and theorem proving to generate verification results, and adopt a template-based approach to derive formal assertions from natural language requirements.

Our main contribution in this paper is an approach for creating and evolving AC evidence by embedding formal verification into a model-based workflow. With the level of automation that we achieve, not only we can support the initial development and offline maintenance of ACs. but also their evolution in the context of adaptive systems. For instance, the work in [25] presents an architecture for adaptive safety-critical systems that includes a legitimisation component. Using our work, it is possible to include the update of an AC as part of the functionality of that component to both preclude software adaptations that compromise safety and provide evidence of safety if the adaptation can go ahead. If requirements change, describing those requirements using our controlled language requires manual intervention, but otherwise the update of the AC can be done fully automatically.

In detail, we present: (1) a traceable assurance case evidence generation approach, which embeds formal verification of software models within the assurance case development process and integrates multiple verification techniques to support diverse property types; (2) a template-based technique for systematically structuring requirements and automatically generating formal assertions, bridging informal requirements and tool-ready verification properties while reducing the need for formal methods expertise; (3) an evaluation through multiple robotic case studies that demonstrate the applicability and effectiveness of our approach. Our case studies include an unscrewing robot whose behaviour adapts during operation when new types of screw are found, a safety monitor for an autonomous underwater vehicle that adapts its operation depending on safety-related conditions, a mail delivery robot that runs on a rechargeable battery, and a high-voltage controller with a risk of causing a fire.

Our approach is described in Fig. 1. It is supported by five existing tools: (1) Kapture[0], a software requirement management tool designed to help engineers create precise and unambiguous requirements; (2) RoboTool[1], which supports modelling, validation, and automatic generation of CSP and PRISM semantics for RoboChart models; (3) FDR; (4) PRISM; and (5) Isabelle [26]. We also include a novel assertion generator and an evidence integrator. Development of an AC using our approach starts by identifying the software requirements referenced in the AC claims that require formal verification (model checking or theorem proving). Then, the evidence is generated from the verification results of the requirements to support the claims.

We have established a classification for requirements to be verified using formal verification, and customised and refined requirement templates used by Kapture. Using our Kapture templates, we obtain a structured requirement, written in a controlled natural language. From such requirements, an assertion generator produces formal assertions based on assertion templates we have designed for each class of requirement. These assertions are verified against RoboChart models (or more precisely, their semantics) using the appropriate formal verification tools, and their verification results are transformed into evidence models. These are subsequently integrated into the model-based AC using the evidence integrator.

The remainder of this paper is organised as follows. Sect. 2 briefly introduces ACs, RoboChart, and our formal notations. In Sect. 3, we give an overview of our approach for model-based AC evidence generation. A running example is introduced in Sect. 4. Sect. 5 presents the controlled natural language used for structuring software requirements, namely the requirement templates for RoboChart that we have developed based on the Kapture templates. Sect. 6 discusses the generation of formalised assertions from the structured requirements. Sect. 7 introduces the method for generating and integrating the evidence from verification results. The case studies are presented in Sect. 8. We review related work in Sect. 9. Finally, we conclude and discuss future work in Sect. 10.

[0] https://www.drisq.com/product-kapture

[1] https://robostar.cs.york.ac.uk/robotool/

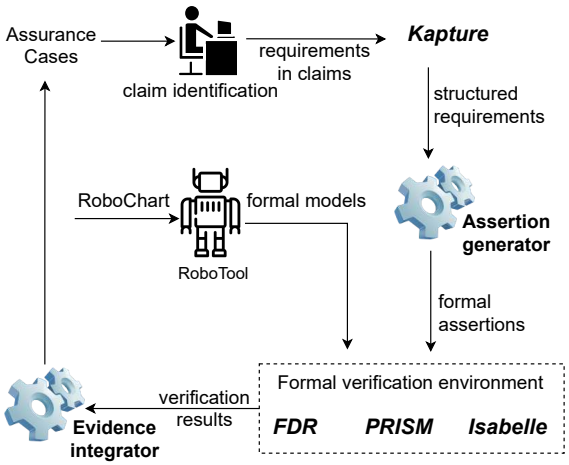


Fig. 1: Approach for evidence generation.

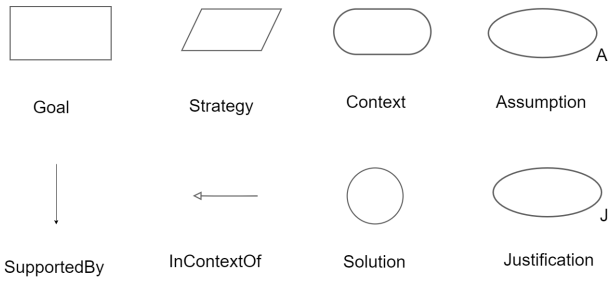


Fig. 2: Main elements of GSN.

2 Background

This section briefly introduces ACs (Sect. 2.1), the RoboChart language (Sect. 2.2), describes the FDR (CSP) and PRISM assertion languages (Sects. 2.3, 2.4), and provides an overview of the Z-Machine notation used in the Isabelle/HOL theorem prover (Sect. 2.5).

2.1 Assurance Cases

An AC is a structured argument that decomposes top-level claims, that is, properties, into supporting sub-claims. A safety property is often addressed through a safety case - a specialised form of assurance case [27]. ACs are crucial for systems operating in high-risk industries like automotive and healthcare. While not mandatory in all safety-related fields, they are commonly used for stakeholder communication and are often required by safety standards, such as ISO26262 [2].

ACs can be represented in different forms, including textual, graphical, and machine-readable models. The most widely used are the structured graphical notations, such as, the Goal Structuring Notation (GSN) [28] and Claims-Arguments-Evidence (CAE) [29]. These notations use various shapes to graphically represent AC elements and their relationships in a structured manner. This helps stakeholders comprehend the range of information necessary to support arguments about system properties.

GSN, developed by Kelly [27] in 1998, is the dominant notation used in engineering and safety-critical industries. In GSN, claims are documented as goals, and items of evidence are cited in solutions. In addition to goals and solutions, the other main elements of GSN are strategy (that is, a description of the approach used to support a goal), context, justification, and assumptions. Two types of linkages between elements are also used: SupportedBy and InContextOf, as shown in Fig. 2. GSN did not originally have a formal metamodel. Therefore, a number of meta-models have been proposed [9,30].

More recently, the Object Management Group released the Structured Assurance Case Meta-Model (SACM) [31], which aims to improve standardisation and interoperability. Compared to other proposed AC metamodels, SACM provides unique features, such as fine-grained modularity, controlled terminology, and traceability from argument to evidence artifacts [32]. SACM can support a variety of notations, including GSN and CAE.

In this work, we use the SACM standard, but adopt the GSN notation to present results graphically, as GSN is generally more familiar to practitioners.

2.2 RoboChart

The core of RoboChart [21] is a subset of the UML state machine notation with a formal semantics. RoboChart offers several features that align with the objectives of this research: (i) it is supported by an EMF metamodel, enabling easy manipulation when implementing model-based techniques, and (ii) it possesses formal semantics and model-checking support, which are advantageous for automating the generation of formal evidence.

A RoboChart model captures the software architecture in a hierarchical format, where a RAS software design is modelled as a RoboChart module. The components of a module are a robotic platform and multiple controllers.

A robotic platform abstracts a physical robot in terms of the functionalities it provides to controllers. These may be of three types: variables, operations, or events. These elements can be either declared directly in the robotic platform, or grouped in interfaces. An interface groups variable lists, operations, events, and clocks. An event may or not have a type [33], depending on whether they communicate data or are just signals.

A controller represents software components that interact with the robotic platform. The behaviour of a controller is defined using state machines. RoboChart's state machines are similar to those of UML, but are enriched with facilities to define time properties, and with probabilistic junctions [34].

Detailed information on the syntax and semantics of RoboChart can be found in [33].

RoboChart's development environment is RoboTool[2]. It calculates a formal semantics for RoboChart models described using the ASCII version of CSP called CSP_M. This semantics supports automatic verification of properties using FDR. RoboTool also converts the RoboChart models into PRISM models for verification using PRISM.

RoboChart is equipped with assertion languages: controlled English to define properties to be verified in FDR and PRISM. RoboTool can transform these assertions into CSP assertions for FDR, or PCTL specifications for PRISM. RoboTool calls the model checkers directly and produces a verification report with results.

2.3 CSP and the assertion language for CSP

RoboChart has an untimed semantics defined using CSP, and a timed semantics defined using its timed dialect, tock-CSP [35]. CSP, a specification language for concurrent systems, has processes and channels as its main constructs. Processes describe interaction patterns, such as choice, deadlock, and termination, and can be composed in parallel, interacting through channels. CSP has three core denotational models: traces, stable failures, and failures/divergences. The traces model represents sequences of observable events, the stable-failures model adds refusal sets, and the failures/divergences model further addresses livelock. More details on CSP semantics and operators can be found in [36].

The RoboChart assertion DSL for CSP is based on CSP_M. The rest of this section provides a brief overview of this assertion language. For more comprehensive information, we refer to Sect. 5.1 of the RoboChart manual [33].

The syntax of an assertion is described below, using an EBNF notation. When defining assertions, the semantics to be checked can be selected by labelling the assertion with `untimed` or `timed`. If no label is given, both semantics are checked by default, leaving it to the user's discretion to decide which result is most relevant.

```
Assertion ::=
   ('timed' | 'untimed')? 'assertion' N ':' SPEC
   ('in' 'the' MODEL)?
   ('with' ('constant' | 'constants') CONSTANTS)?
```

An assertion has a name `N` and a property specification `SPEC`, and optionally the specification of a model `MODEL` (for instance, traces model), and of values for constants used in the RoboChart model. The syntax for `SPEC` is sketched below; we describe just the elements used in this paper.

```
SPEC ::=
   N 'is' ('not')? ('deadlock-free'
                  | 'divergence-free')
   | N ('does' 'not' 'terminate' | 'terminates')
   | N 'is' ('not')? 'reachable' 'in' N
   | N (('does' 'not' 'refine') | ('refines') ) N
   | ...
```

`SPEC` can be unary or binary. Unary assertions describe general properties of RoboChart elements, such as, deadlock freedom, divergence freedom, termination, and state reachability. Binary assertions compare two RoboChart elements. In the definition of `SPEC` above, we show the binary assertion for comparison based on refinement.

The RoboChart assertion language facilitates writing simple assertions, such as deadlock freedom, without requiring detailed knowledge of RoboChart's CSP semantics naming conventions. However, more complex refinement properties necessitate the use of CSP blocks, along with a deeper understanding of the CSP semantics structure, naming conventions, and CSP_M syntax.

[2] www.cs.york.ac.uk/circus/RoboCalc/robotool/.

2.4 PRISM and the assertion language for PRISM

RoboChart also has a formal semantics defined using the reactive modules notation (of PRISM and many other probabilistic model checkers). RoboTool can automatically calculate this semantics as well.

The properties to be checked in the PRISM model checker are specified as formal assertions in PCTL. The RoboChart assertion language for PRISM provides a more readable syntax, sketched below.

```
ProbProperty ::=
    'prob' 'property' N ':' pExpr
    ('with' 'constants' (ConstConfig+ | N))?
    ...
```

A probabilistic property starts with the keyword `prob`. It has a name `N` with a body given by an expression that denotes the property to be verified. In our work, this expression must have a Boolean type. Optionally, an assertion has constant configurations (given by`ConstConfigs`). These configurations can be used to define values of constants declared in the RoboChart model. For details of the full syntax of probabilistic assertions, we refer to Sect. 5.2 of the RoboChart manual [33].

We also consider reward assertions as follows.

```
Reward ::= ('[' pEvent ']')? pExpr ':' pExpr ';'
```

In a reward-based assumption, a reward value is defined using a guard condition `pExpr` of boolean type, an expression `pExpr` defining the value, and an optional event `pEvent`, which is an action label. In PRISM, action labels are names attached to transitions, allowing rewards to be associated with specific types of transitions rather than all transitions satisfying the guard.

Our work automates the generation of assertions from requirements by providing a set of templates for CSP-based and PCTL-based RoboChart assertions.

2.5 Isabelle and Z-Machines

Z-Machine formalism is used to give a simplified semantics to RoboChart state machines [37] in the Isabelle/HOL theorem prover. Z-Machines is a formal modelling language and tool in the style of the Z specification language [38] and B method [39]. It is a form of abstract machine and consists of (1) a state space including invariants; (2) a set of operations acting over that state space; and (3) the machine itself, which initialises the state and groups together the operations. Z-Machines can make use of any data structures available in HOL, such as sets, functions, records, algebraic data types, and real numbers.

Z-Machines is supplied with a proof method called `deadlock_free`, which automates deadlock checking in Isabelle [37]. The method calculates a deadlock-freedom condition for the whole Z-Machine that can typically be discharged using Isabelle's powerful tools that automate the process of finding proofs, such as *auto* or *sledgehammer*.

3 Overview of our approach

This section describes our approach to automating the generation of evidence through formal verification. Fig. 3 provides an overview of the four-step process, from requirement structuring (Step 1 Fig. 3) to the integration of verification results into an AC (Step 4).

We assume that a partially instantiated model-based AC is provided as the starting point, where some of the evidence has not yet been concretised with verification results. The AC may contain claims of various forms; we focus on claims such as "the software requirement R_{ID} has been implemented", which refer to software-related concerns of a broader assurance argument, and for which R_{ID} is a property to be verified using formal methods. The evidence supporting the claim typically takes the form "the result is true," indicating that the software satisfies the requirement. Based on this structure, the automation of evidence generation requires the formalisation of R_{ID}.

The four steps of our process (see Fig. 3) are as follows.

1. *Structuring the requirements R_{ID} (labelled (a) in Fig. 3) referenced in the AC claim using predefined templates in the Kapture tool.* Each requirementR_{ID} is described precisely as a property of a RoboChart model of the software (i) using Kapture templates (b). Kapture can export (d) structured requirements (c) into markup formats such as XML (e), thereby enabling their use in assertion generation.
2. *Assertion generation by instantiating (g) novel predefined assertion templates (f) with the markup requirements (e).* This step leads to automatically generated formal assertions (h) traceable to requirements.
3. *Using formal verification (l) based on the RoboChart model (i) to check the assertions (h).* RoboTool is used to calculate (j) a formal semantics (k) for the RoboChart model (i), which is subsequently verified (l) against the assertions (h) from the previous step. This step yields the verification results (m).
4. *Using the verification results (m) to generate an SACM-based evidence model (o).* This is achieved through model-transformation (n), after which the model is integrated (p) into the AC model (q).

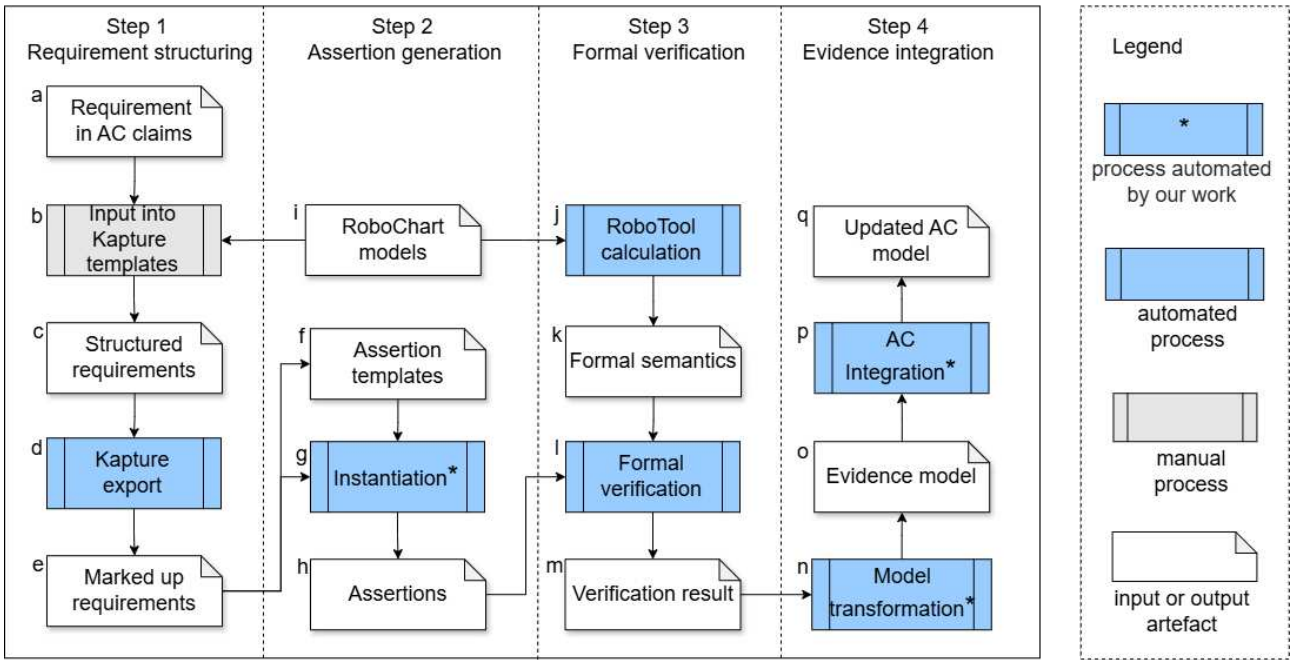


Fig. 3: Overview of our four-step approach to evidence generation by formal verification. Activities and artefacts within these steps are labelled a - q to allow cross-referencing in the text.

The steps of our approach are automated, with the exception only of the input of software requirements into Kapture templates (b). Three key activities (namely, (g), (n), (p)) are automated by the work presented in this paper. They are highlighted in Fig. 3 with a blue background and a star, and described in later sections.

The implementation of the approach is based on the Eclipse EMF framework and relies on three tools: RoboTool, Epsilon[3], and Kapture. Epsilon is employed to facilitate the transformation process, converting XML-encoded claims into RoboChart DSL assertions and translating model checking results into SACM evidence models.

To illustrate the steps and the supporting tools of our approach, we introduce a running example next.

4 Running example

Our example is based on a mobile and autonomous chemical detector presented in [40, 41], which performs a random walk, while avoiding obstacles and analysing the air to detect dangerous gases. Once a dangerous chemical is detected at a high intensity, the robot drops a flag to report the presence of the gas, and stops.

Gas detection is modelled by a RoboChart state machine called GasAnalysis, and the random walk with obstacle avoidance is modelled in another state machine Movement, shown in Fig. 4. We use Movement as the basis of our running example. Movement specifies obstacle avoidance using events obstacle and odometer, specifies the movement behaviours using events turn, stop, and resume, sent from GasAnalysis, and drops a flag using the event flag. All these events represent services, of the robotic platform (a small trolley carrying an artificial nose and a flag) or of the parallel GasAnalysis machine, used by the software implementing Movement.

A RoboChart block for a state machine defines a context of events, variables, and operations used in its actions. For brevity, we omit the definition of that context in Fig. 4.

Movement starts in the state Waiting (as indicated by the black circle with an i) during which the robot performs a random walk. We omit here the definition of the operation randomWalk() and of all other operations used by Movement; they are all declared in the omitted context. When a gas is identified by the machine GasAnalysis, Movement receives a turn event and transitions to the state Going. Movement may also receive a resume event, indicating that no gas has been detected and that random walking should continue. When a dangerous gas is found, Movement receives a stop event and transitions to Found, where a flag is dropped and the robot is stopped, using the move operation, with argument 0 for the linear velocity.

In the state Going, a turn event does not lead to a change of state. The event resume leads Movement back to the state Waiting. The event stop leads to Found. In addition, if obstacles appear along the path while Movement is in the state Going, then an avoidance mechanism is triggered. It uses the event odometer and a clock T to detect situations where the trolley becomes stuck. When an obstacle is first detected, via the event obstacle, guarding the transition from Going to Avoiding, the clock T is reset (#T).

In the entry action of Avoiding, the distance travelled so far is obtained using the event odometer, and recorded in a variable d0. An operation changeDirection() is then called so that the robot moves away from the obstacle. The

[3] http://download.eclipse.org/epsilon/updates/2.2/

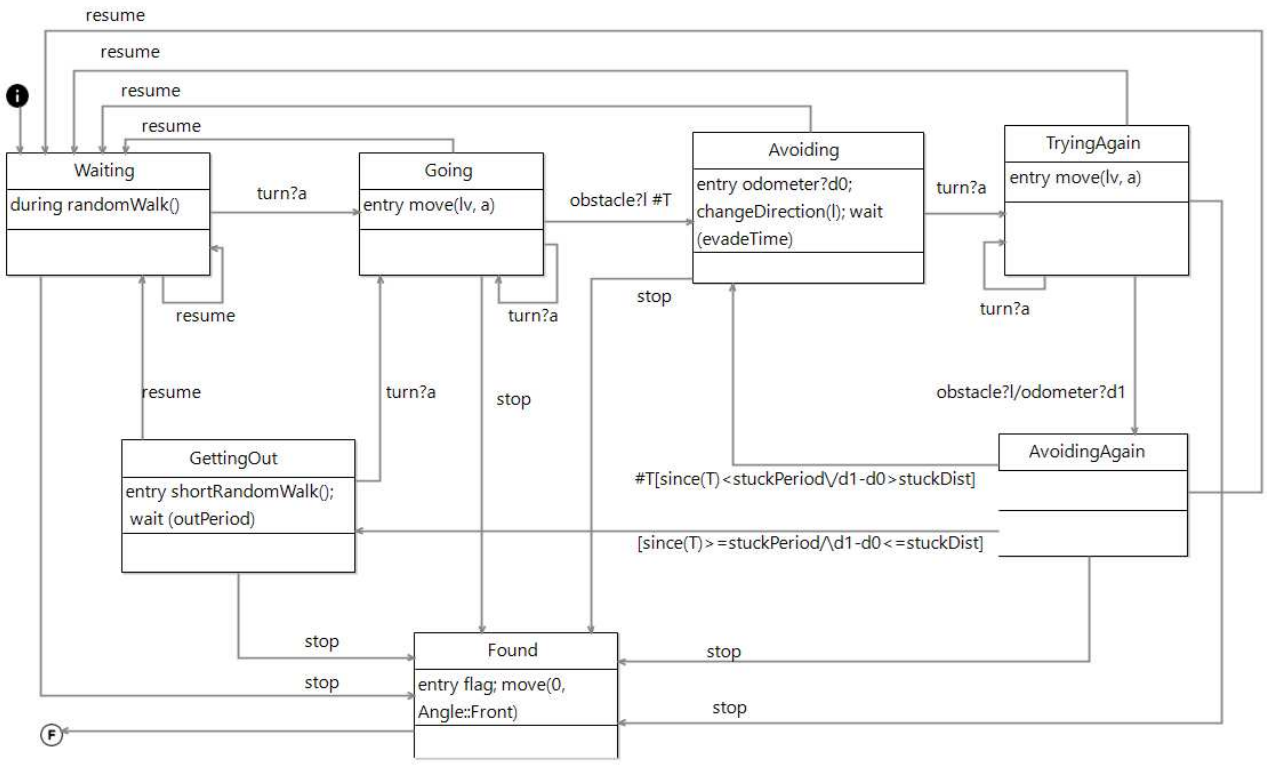


Fig. 4: RoboChart state machine Movement.

Table 1: Requirements for the Chemical Detector

No.	Requirement
R1	If a dangerous gas is detected, the robot shall flag to report the detection.
R2	Once an obstacle is detected, the location of the obstacle shall be recorded within one time unit.
R3	The chemical detector shall be deadlock-free.

Table 2: Example claims for the chemical detector

No.	Claim	Verification Support
C1	Requirement R1 is implemented.	FDR
C2	Requirement R2 is implemented.	FDR
C5	Requirement R3 is implemented.	FDR, Isabelle

wait(evadeTime) action gives the robot an amount of time defined by the constant evadeTime to move away. In Avoiding, a resume event leads back to Waiting, a stop event leads to Found, and turn to the state TryingAgain.

In TryingAgain, the robot, having changed direction, will (attempt to) move again. (The entry action calls the operation move with arguments given by constants lv and a defining a linear velocity and an angle.) If, in this state, another obstacle is detected, Movement enters the state AvoidingAgain, getting a new odometer measure d1.

In AvoidingAgain, occurrence of an event resume leads back to Waiting, and stop leads to Found. Another two transitions have guards that check whether enough time has elapsed since the first obstacle was detected (that is, the value since(T) of the clock T is greater than or equal to a value stuckPeriod) but little progress has been made (that is the difference between d1 and d0 is less than or equal to a value stuckDist). If so, the robot is deemed to be stuck, and Movement transitions to the state GettingOut. Otherwise, it returns to Avoiding to continue the standard avoidance process.

In the entry action of GettingOut, the call to the operation shortRandomWalk() is an attempt to move away from the stuck position, wait(outPeriod) specifying a minimum time budget for this attempt. Similar to other states, events resume, turn, and stop are accepted.

The set of requirements for the chemical detector shown in Table 1 has been identified. They occur in an AC as part of claims summarised in Table 2. Each claim asserts that the corresponding requirement has been implemented, and is linked to a specific verification method. In our case, the FDR model checker and the Isabelle theorem prover provide the required verification support.

Table 3: Requirement types supported by the proposed framework for RoboChart models.

Requirement category	Description
Untimed safety	Event-based safety constraints defined over untimed behaviour.
Timed safety	Safety requirements involving explicit model-level time, such as bounded-response or deadline constraints.
General correctness	Structural and behavioural correctness properties, including state reachability, software component termination, divergence-freedom, and deadlock-freedom.
Probabilistic properties	Quantitative constraints on event likelihoods or performance measures, including probability thresholds and reward-based expectations.
Temporal properties	Qualitative properties expressed in temporal logics, capturing ordering or eventuality relations between events.

In the next sections, we describe the steps of our process (see Fig. 3) in detail, and use this example to illustrate the application of the techniques.

5 Requirement structuring

This section describes Step 1 of our process, in which predefined templates are used to structure requirements written in natural language to enable their formalisation and verification. The remainder of this section is structured as follows. Sect. 5.1 presents a classification of software requirements covered by our framework. This provides a conceptual foundation for our new templates. Sect. 5.2 describes Kapture templates used in the design of our templates, which are then presented in Sect. 5.3. Finally, Sect. 5.4 describes how to use our templates.

5.1 Requirement coverage and classification

The software requirements referenced in the claims, and subject to formal verification, can be classified according to their behavioural characteristics. This classification does not constitute a general taxonomy of robotic software requirements, but rather a coverage-oriented classification of the types of requirements that can be formally verified within our framework. Each category captures an aspect of a software behaviour, reflecting the diversity of properties that can be expressed and verified for RoboChart models.

Table 3 summarises the types of requirements and provides a concise overview of the kinds of properties that can be verified. The classification distinguishes between untimed and timed safety, structural correctness, quantitative probabilistic aspects, and qualitative temporal relationships, each representing a different dimension of behavioural assurance for a software.

For illustration, we map the requirements of the chemical detector in Table 1 to the categories in Table 3. Requirement R1 is an untimed property. Requirement R2 is an example of a timed property, as it constrains the time within which the robot shall record the obstacle location. Finally, Requirement R3 is a general correctness property. Although the chemical detector model does not incorporate probabilistic features, if it were extended with stochastic behaviours, a probabilistic property could specify, for instance, that a dangerous gas is detected with probability at least 0.95, or that the robot successfully escapes from a stuck situation with a probability above a given threshold. A reward-based property could also evaluate the expected number of steps or time units required to recover from being stuck. A temporal property could express that whenever the robot becomes stuck, it eventually resumes normal movement.

In summary, these categories capture the behavioural aspects of a software addressed by our framework and provide a structured basis for associating each type of requirement with suitable formal semantics and verification tool.

5.2 Base templates in Kapture

This section introduces the Kapture templates that form the basis for our requirement-template design. A requirement in Kapture is defined using three components. A *data dictionary*, provides the basic elements of requirements, such as events, time limits, probability, and reward values. For example, an event *obstacle* can be defined as a signal in the data dictionary and later used in requirements.

A *definitions* component specifies functions. For instance, we define `reachable(scope, state)` to check whether a given `state` is reachable within a given `scope`, that is, composite state, machine, controller, and so on.

Finally, *requirements* are constructed using predefined templates for different types of constraints. Kapture provides seven requirement templates. We use three of them: `when`, `trigger on event`, and `every` described below. Each template defines a structured form with optional sections; we describe those we use.

The `when` *template.* This template captures causality between events: when one condition holds, another must eventually be observed until a termination condition occurs. We use `when` templates of the following form.

When *GuardCondition* occurs/is true,
until *UntilCondition* occurs,
RequiredCondition shall be seen/true.

In the templates, we show placeholders, for instance, *GuardCondition*, in italics. These placeholders can be instantiated using the elements defined in the data dictionary. The template captures a causal relationship between events: once the *GuardCondition* becomes true, the software shall satisfy the *RequiredCondition* until the specified *UntilCondition* occurs. The *Until* section is optional.

The `trigger on event` *template.* This template specifies time-bounded responses: if a triggering event occurs, a required condition must hold within a given duration.

If ever *TriggerCondition* occurs,
then at some point, but within *WithinDuration*,
RequiredCondition shall be true.

The `every` *template.* This template expresses invariants, stating that a condition must always or never hold.

The *Condition* shall *AlwaysOrNever* be true.

∗ *AlwaysOrNever* ::= `always` | `never`

Each individual requirement template we have designed is derived from one of these base templates through further specialisation. Our templates and their mapping to requirement categories are presented next.

5.3 Requirement-template design

We have designed nine templates: one for each requirement category in Table 3 and aligned with the corresponding verification tool (FDR, PRISM, or Isabelle). We show here on four representative templates: untimed safety, timed safety with deadlines, deadlock-freedom, and reward-based, which illustrate the main patterns and their instantiations. The remaining templates are in Appendix A.1.

Placeholders and event naming. Templates use placeholders for RoboChart components. The most common placeholder is a RoboChart event, whose names are elements of the syntactic category *fq_ event* defined as follows.

fq_ event ::= *qualified_ scope event_ name* [`.in` | `.out`] [*value_ or_ variable*]

qualified_ scope ::= [*module* `::`] [*controller* `::`] [*state_ machine* `::`]

event_ name ::= *RoboChart_ event_ identifier*

value_ or_ variable ::= `.`*value* | `?`*variable*

For readability, we follow a typographic convention in all grammar-style definitions. Non-terminals appear in *italic black*, while references to non-terminals appear in *blue italic*. Terminals are shown in `monospace`. In addition, we use standard EBNF symbols.

In RoboChart models, events are usually referred to by simple names (such as, flag). For CSP assertions, however, they are expanded into fully qualified events following the syntax above. For instance, the event flag of the state machine Movement of the controller ctrl in the module sys is referred to as `sys::ctrl::Movement::flag.out`.

Here, `::` separates modules, controllers, and state machines in the scope; events may be annotated with `.in` or `.out` to indicate input/output direction, and with `.`*value* or `?`*variable* if data is exchanged.

Instantiating placeholders with fully qualified event names is the first step in producing formal assertions. This requires only knowledge of the structure of the RoboChart model and the names of its components, not of CSP.

Untimed requirements. These requirements typically express that the occurrence of a *guard event* must be eventually followed by the occurrence of a *required event*. We capture this pattern in the untimed template (RT-UNTIMED), which is a specialisation of the Kapture `when` template using `When` and `Required` sections.

RT-UNTIMED

When *guard_ event* occurs,
required_ event shall also be seen.

∗ *guard_ event* ::= *fq_ event*
∗ *required_ event* ::= *fq_ event*

The requirement R1 of our running example (see Table 1) can be expressed using RT-UNTIMED as follows.

When `sys::stop.in` occurs,
`sys::flag.out` shall also be seen.

In this example, "dangerous gas is detected" is represented by the input event stop in module sys, while "the robot shall flag to report the detection" is represented by the output event flag. Here, the placeholders *guard_event* and *required_event* from the template are instantiated with the concrete RoboChart events `sys::stop.in` and `sys::flag.out`, identified by their fully qualified names.

Timed safety requirements A particularly common pattern is the deadline requirement, in which the occurrence of a trigger condition initiates a bounded-time obligation for a required condition to be fulfilled. Specifically, for each instance of the trigger event, the required event must occur within a specified time bound: it may happen earlier, but not later. We capture the deadline pattern in the timed functional template (RT-TIMED), which is based on the `trigger on event` template.

RT-TIMED

If ever *trigger_event* occurs,
then at some point, but within *deadline*,
target_event shall be true.

- *trigger_event* ::= *fq_event*
- *target_event* ::= *fq_event*
- *deadline* ::= (*nat*) `rounds`

The requirement R2 (in Table 1) can be expressed using the RT-TIMED template as follows.

If ever `sys::obstacle.in` occurs,
then at some point, but within `1 rounds`,
`sys::odometer.in` shall be true.

In this instantiation, the placeholders *trigger_event*, *deadline*, and *target_event* are replaced by the concrete RoboChart events `sys::obstacle.in`, `1 rounds`, and `sys::odometer.in`, respectively. Since Kapture provides *rounds* as a time unit, we adopt `1 rounds` as a practical encoding of one time unit.

We recall that general requirements encompass properties such as deadlock freedom, divergence freedom, reachability, and termination, which are verified using FDR or Isabelle. To develop a structured template for these general properties, the base template `every` is employed.

Deadlock-freedom requirements We capture this pattern in the deadlock-freedom template (RT-DDLK).

RT-DDLK

The property *function*(*scope*) shall *bool_statement* be true.

- *function* ::= `deadlock_free_untimed` | `deadlock_free_timed` | `deadlock_free` | `deadlock_free_isa`
- *scope* ::= *RoboChart_component_identifier*
- *bool_statement* ::= `always` | `never`

As discussed in Sect. 2.3, compiling a RoboChart model into CSP_M produces both an untimed and a timed semantics. When verifying assertions in FDR, the intended semantics can be selected by labelling the assertion as `untimed` or `timed`; if no label is provided, both variants are checked by default. Accordingly, the template provides three functions for FDR: `deadlock_free_untimed` checks use the untimed semantics, `deadlock_free_timed` the timed semantics, and `deadlock_free` leaves the semantics unspecified. In addition, `deadlock_free_isa` indicates verification in Isabelle/HOL, using the Z-Machine mechanisation of RoboChart state machines (Sect. 2.5). This method [37] automatically generates a deadlock-freedom condition and discharges it using proof tactics, providing an alternative that mitigates state explosion.

Requirement R3 from Tabel 1 can be encoded as follows. The placeholder *scope* is instantiated to the state machine Movement. The chosen function is `deadlock_free`, and the Boolean statement is `always`.

The property `deadlock_free`(`Movement`) shall `always` be true.

Reward-based requirements As said, they capture quantitative objectives over execution paths, and are verified using PRISM. They typically state that a reward target must be achieved for given constant values and reward specifications. We have designed the reward-based template (RT-RWD) using the Kapture `when` template as follows.

RT-RWD

When *constants_ configs* and *reward_ event* and *reward_ value* is true,
Until *path_ formula* becomes true,
reward_ target shall also be true.

∗ *path_ formula* ::= `pathFormula_`*RPathFormula*
∗ *RPathFormula* ::= (`Reachable` | `LTL` | `Cumul` | `Total`) *pExpr*[a]
∗ *reward_ target* ::= `reward_target_`*op Expr*
∗ *op* ::= `>` | `>=` | `<` | `<=` | `==`
∗ *reward_ event* ::= `reward_event_`*fq_ event*
∗ *reward_ value* ::= `reward_value_`*Expr*

[a] The full syntax of *pExpr* is given in [33]. One of its forms, *QualifiedNameToElement*, denotes a RoboChart element.

Although RT-RWD is written using the `when` template skeleton, the semantics of its three sections differ from those of functional requirements, where the `When`, `Until`, and `Required` sections express a guard-scope-obligation pattern. For reward-based requirements, we adopt the `when` skeleton, interpreting its three sections in a specialised way: the `When` section sets constant configurations and reward specifications (*constants_ configs*, *reward_ event*, *reward_ value*); the `Until` section encodes the *path_ formula*; and the `Required` section gives the quantitative bound (*reward_ target*) to be satisfied.

Since this requirement is to be formalised as an assertion in the PRISM reactive-modules language, to facilitate the assertion generation, the placeholders follow the terminology of PRISM. We describe each of the sections next.

When section This corresponds to the line of the template that starts with the keyword When. It sets constant values for PRISM verification and introduces reward specifications. Instantiation of the placeholder *constants_ configs* specifies one or more RoboChart constants with assigned values (single assignments or ranges). Its full grammar is given in Appendix A.1. Instantiation of the placeholder *reward_ event* identifies the event contributing to the reward and its name must be prefixed with "`reward_event_`". Instantiation of *reward_ value* gives the amount added per occurrence of *reward_ event*. That expression must be labelled with the prefix "`reward_value_`".

Until section. This corresponds to the line of the template starting with Until. It specifies the path condition (described by a *path_ formula*) under which the reward is accumulated. Here, *path_ formula* is labelled with the prefix "`pathFormula_`" and given by a term of the syntactic category *RPathFormula* from the RoboChart assertion language (see Sect. 2.4 and [33]).

Required section. This corresponds to the line of the template starting with a quantitative bound (*reward_ target*) to be satisfied by the accumulated reward, whose definition must be labelled with the prefix "`reward_target_`".

We illustrate the use of this template in Section 8.

5.4 Requirement structuring

Writing a requirement in Kapture is a manual activity. The user is expected to know the RoboChart model under consideration and to follow the naming conventions of RoboChart events and variables, and also be familiar with the syntax of the RoboChart assertion languages. Importantly, however, no expertise in the underlying formal languages (CSP or PRISM) is required, as the templates shield the user from these details.

For illustration, we consider again the requirement template R-UNTIMED. The process of obtaining the encoding of a requirement using this template begins by declaring *guard_ event* and *required_ event* in Kapture's *data dictionary*, using RoboChart events as their identifiers. These declared events are then referenced in the *requirements* section, where the `when` template is instantiated.

Kapture allows each requirement to carry traceability metadata. Backward traceability links the requirement to its corresponding claim in the AC, while forward traceability links it to the lower-level claim that specifies the verification method or tool to be used. This lower-level claim is the one that will be directly supported by our verification evidence. In our work, we use such links to integrate verification evidence into the SACM AC model automatically.

Kapture supports multiple output formats, including XML, HTML, and CSV. In our process, the XML export is used. The result of Step 1 is therefore a semi-formal XML file that encodes: (i) the requirements instantiated from templates, and (ii) their associated traceability links. This XML file serves as the structured input for the assertion generation of Step 2 (see Fig. 3).

6 Assertion generation

This section presents our approach in Step 2 of Fig. 3, the generation of assertions. The outputs of this step are assertions in the RoboChart assertion language, for model checking, and in the Isar language, for theorem proving. They are obtained by instantiating assertion templates with the structured requirements produced in Step 1.

For model checking, the generated RoboChart assertions are translated by RoboTool into CSP_M or PRISM assertions, and verified against the RoboChart models using FDR and PRISM. For theorem proving, RoboChart models are transformed into Z-Machines using the approach described in our previous work [37]. These Z-Machines are then verified against Isar assertions in Isabelle/HOL.

The remainder of this section is structured as follows. Sect. 6.1 introduces the assertion templates defined for different requirement categories. Section 6.2 describes our assertion generation process, including template selection and template instantiation.

6.1 Assertion-template design

We define a set of assertion templates (ATs) that are derived from, and aligned with, the requirement templates introduced in Sect. 5.3. In total, eleven assertion templates have been designed, covering all the requirement categories listed in Table 3. Some requirement templates map to more than one assertion template, so the number of ATs is larger than the number of requirement templates. Table 4 summarises the mapping between requirement templates and assertion templates, and also indicates the verification tool used in each case. This section presents five representative templates in detail, while the remaining ones are provided in Appendix A.2.

Table 4: Mapping between requirement templates and the assertion requirements

Requirement Template	Section	Assertion Template	Section	Tool
RT-UNTIMED	Sect. 5.3	AT-UTG AT-UTL	Sect. 6.1 Appendix A.2	FDR
RT-TIMED	Sect. 5.3	AT-DLINE	Sect. 6.1	FDR
RT-REACH	Appendix A.1	AT-REACH	Appendix. A.2	FDR
RT-DDLK	Sect. 5.3	AT-DDLK-1 AT-DDLK-2	Sect. 6.1	FDR Isabelle
RT-DIV	Appendix A.1	AT-DIV	Appendix A.2	FDR
RT-TERM	Appendix A.1	AT-TERM	Appendix A.2	FDR
RT-PROB	Appendix A.1	AT-PROB	Appendix A.2	PRISM
RT-RWD	Sect. 5.3	AT-RWD	Sect. 6.1	PRISM
RT-TEMP	Appendix A.1	AT-TEMP	Appendix A.2	PRISM

As explained in Sect. 2.3 and Sect. 2.4, RoboChart has a domain-specific assertion language [33] that is designed to simplify the construction of model checking assertions for FDR and PRISM. This language is the basis of our assertion templates: each template is expressed in the RoboChart assertion language.

Untimed refinement assertions For untimed requirements specified by the RT-UNTIMED template, we define two assertion templates: AT-UTG and AT-UTL. AT-UTG handles requirements that apply globally, without any scoping restrictions. By contrast, AT-UTL is used when the requirement applies only when a state machine is in a particular state. The assertions written using these templates are verified in the FDR model checker by refinement checking. Therefore, each assertion template has three parts: (1) the specification process expressing the required behaviour; (2) the implementation process, corresponding to the RoboChart model; and (3) the refinement clause, written in the RoboChart assertion language, which states that the implementation refines the specification. We check refinement in the *traces* model, as our focus is on the safety aspects of the properties.

The specification and implementation sections are expressed as CSP_M blocks delimited by the keywords **csp-begin** and **csp-end**. The requirement itself provides the content for the specification section. The AT-UTG template is shown below. Keywords of the RoboChart assertion language are displayed in normal font. Placeholders are written in black italics. Fixed parts of a template. are written in teletype black font.

AT-UTG:

untimed csp Spec
csp-begin
`Spec` = `CHAOS(Events) [|{|` *guard_ event* `|}|> (RUN({|` *guard_ event* `|}) /\(` *required_ event* `->` *Spec* `))`
csp-end

untimed csp Spec_impl **associated to** *module_ name*
csp-begin
`spec_impl` = *module_ name*
csp-end

untimed assertion A_ *Claim_ ID* :
Spec_impl **refines** Spec **in the traces model**

The identifier of the assertion, that is, *A_ Claim_ ID*, is derived from the claim identifier in the AC models, providing traceability between the assertion and the evidence.

In its specification section, AT-UTG uses the CSP_M operators `[| |>` (Exception) and `/\` (Interrupt) [42], together with the processes `CHAOS(A)` and `RUN(A)`. The process defined as `P [|A|> Q` behaves as the process `P`, but if `P` performs an event from `A` (the set of exception events), it then behaves as the process `Q`. The process defined as `P/\Q` behaves as the process `P`, but if any action of `Q` is performed, `P` is abandoned and the execution proceeds with `Q`. `CHAOS(A)` offers a nondeterministic choice over all events in `A` but may deadlock at any point, while `RUN(A)` offers a perpetual choice over all events in `A`.

So, in AT-UTG, the specification *Spec* is the process that may behave nondeterministically over all events in scope, which are those in the set "Events". If the *guard_ event* occurs, however, then further occurrences of *guard_ event* may still take place until *required_ event* occurs, and then *Spec* recurses. The implementation part of AT-UTG refers to the CSP_M semantics of the RoboChart model under verification. For this reason, it directly uses the identifiers of the corresponding components. The third part of AT-UTG is the assertion section itself, written in the RoboChart assertion language for CSP.

Timed refinement assertion As mentioned, RoboChart's timed semantics is given using *tock*-CSP [43]. This variant of CSP models discrete time using an event `tock` to represent the passage of one unit of time, which is an abstraction for an arbitrary amount of time: 1 ms or 1 s, for instance. CSP_M provides a **Timed** section for *tock*-CSP processes, enabling verification in FDR.

AT-DLINE is the assertion template for timed refinement. It captures requirements expressed using RT-TIMED. The specification *Spec* is defined inside a **Timed** section. (The OneStep parameter defines that events are instantaneous, like in RoboChart.) After the *trigger_ event* occurs, *Spec* enters a timed monitoring phase in which any events may occur, but time is constrained by an interrupting waiting process `WAIT`(*deadline*); $\texttt{STOP}_\texttt{U}$. `WAIT(t)` is a process that engages in `t` occurrences of `tock` before terminating. $\texttt{STOP}_\texttt{U}$ represents a *timelock* state in which no event is offered and time cannot progress. The interrupt `/\` in *Spec* therefore ensures that if the waiting period expires, time cannot advance. So, the exception takes over `[|{|`*target_ event*`|}|>SKIP`, and so *target_ event* has to happen and then the monitoring phase terminates (`SKIP`). The recursion ensures that subsequent *trigger_ event*s are handled in the same way.

Finally, the wrapper `timed_priority` is needed due to a technicality of FDR: it enforces the *maximal progress principle*. In other words, `timed_priority` gives precedence to internal events over `tock`, ensuring that time passes only when no further internal actions are possible.

AT-DLINE:

timed csp Spec
csp-begin
```
Timed(OneStep) {
  Spec = timed_priority(
    CHAOS(Events) [| {| trigger_ event |} |> SKIP;
    ((CHAOS(Events) /\(WAIT( deadline ); STOPU))
    [| {| target_ event |} |> SKIP); Spec);
  }
```
csp-end

timed csp Spec_impl **associated to** *module_ name*
csp-begin
```
Timed(OneStep) {
  spec_impl = timed_priority(
    module_ name
    |\{| trigger_ event, target_ event, tock |});
  }
```
csp-end

timed assertion A_ *Claim_ ID* :
Spec_impl **refines** Spec **in the traces model**

The implementation section of AT-DLINE uses the *Project* operator (|\) to hide, in the process under verification (that, the process that defines the RoboChart semantics) all events, except *trigger_event* and *target_event*, as well as `tock`, so that passage of time can be observed.

General assertions Five assertion templates are provided for general software properties: AT-REACH (for reachability), AT-DDLK-1 and AT-DDLK-2 (for deadlock freedom), AT-DIV (for divergence freedom), and AT-TERM (for termination). AT-REACH, AT-DIV, and AT-TERM are expressed in the RoboChart assertion DSL for FDR. AT-DDLK-1 is for model checking in FDR and AT-DDLK-2 for theorem proving in Isabelle/HOL.

Each template is explicitly labelled as `untimed` or `timed` to indicate which semantics it uses. If no label is given, the assertion is interpreted against both timed and untimed semantics. AT-DDLK-1 and AT-DDLK-2 are shown below; the others are provided in Appendix A.2.

AT-DDLK-1 (Deadlock-freedom, FDR):

[**untimed** | **timed**] **assertion** A_*Claim_ID* :
scope *bool_term* **deadlock-free**.

∗ *bool_term* ::= **is** | **is not**

AT-DDLK-1 uses CSP_M, and AT-DDLK-2 uses Isar syntax to pose a lemma that claims deadlock-freedom.

```
AT-DDLK-2 (Deadlock-freedom, Isabelle)
lemma Claim_ID_deadlock_free:
"deadlock_free scope"
  apply deadlock_free
```

In both AT-DDLK-1 and AT-DDLK-2, the placeholder *scope* are as defined in the specification of RT-DDLK.

Reward assertion Three assertion templates, AT-PROB, AT-RWD, and AT-TEMP, are provided for probabilistic and temporal requirements. These templates are expressed in the RoboChart assertion language for PRISM. AT-RWD is presented below; the others in Appendix A.2. AT-RWD is used to generate assertions for requirements written with the RT-RWD template.

AT-RWD:

rewards reward_*Claim_ID*
= [*reward_event*] (true | false) : *reward_value*
endrewards

prob property R_*Claim_ID*:
Reward reward_*Claim_ID* *reward_target* **of**
[*path_formula*]

[**with constant** { *constant*} { *multi_constant* }]

AT-RWD is written in the RoboChart assertion language for PRISM. Accordingly, its structure follows PRISM's reward and property declarations.

AT-RWD has three sections. The first section, introduced by **rewards**, specifies the event that generates a reward, together with its corresponding reward value. The second section, introduced by **prob property**, defines the actual property to be checked. Here, the identifier *R_Claim_ID* is derived from the AC claim identification, ensuring traceability. The property asserts that the accumulated reward, as recorded by *reward_Claim_ID*, satisfies the required bound (*reward_target*) along the paths described by *path_formula*. The third section introduces optional constant values. The syntax for the placeholders *constant* and *multi_constant* is given in Appendix A.1.

6.2 Assertion generation

An assertion is produced in two steps: (1) selection of the appropriate assertion template for each requirement, and (2) instantiation of that template with concrete values taken from the requirement. Algorithm 1 describes both steps, and is implemented using the Epsilon model-transformation engine, and in particular its domain-specific language EGL [44]. The input of Algorithm 1 is an XML document *kapture_xml* exported from Kapture containing requirements, and the output is a set of well-formed assertion instances, one for each requirement.

Template selection. Before an assertion can be created, the correct assertion template must be identified. The XML file exported from Kapture may contain multiple requirements, each represented as an entry r (line 1). Each entry includes its claim identifier (*backwardsTr*, line 2) and requirement type (*templ*, line 3). For requirements defined using the *when* Kapture template (line 4), an entry r also has a *requiredCondition* (line 5) and a *guardCondition* (line 6), which record the instantiation of the placeholders of the Kapture template discussed in Section 5.2.

Algorithm 1: Assertion generation

Input: *kapture_xml*
Output: A set of instantiated assertions

```
1  for r ∈ kapture_xml.reqs do
2      claim_ID ← r.backwardsTr;
3      templ ← r.template;
4      if templ = "when" then
5          rCond ← r.requiredCondition;
6          gCond ← r.guardCondition;
7          prefix ← head(rCond);
8          if prefix = "prob_target" then
9              GenerateProbAsst(r, claim_ID);
10         else if prefix = "reward_target" then
11             GenerateRewardAsst(r, claim_ID);                    // reward branch
12         end
13         else if prefix = "term" then
14             GenerateTempAsst(r, claim_ID);
15         end
16         C ← {"constant", "multi_constants"};
17         for cond ∈ guardCond.conds do
18             if cond.prefix ∈ C then
19                 GenConstant(...);
20             end
21         end
22         if prefix ∉ {"prob_target", "reward_target", "term"} then
23             GenerateUntimedAsst(r, claim_ID);
24         end
25     end
26     else if templ = "trigger on event" then
27         GenerateTimedAsst(r, claim_ID);
28     end
29     else if templ = "every" then
30         GenerateGeneralAsst(r, claim_ID);
31     end
32 end
```

We recall that probabilistic, reward-based, temporal, and untimed requirements use the *when* Kapture template. As discussed in Section 5.3, we add prefixes to their *requiredCondition* placeholder to explicitly identify the requirement type defined. Algorithm 1 uses this *prefix* (line 7) to determine whether the requirement matches a probabilistic (line 8), reward-based (line 10), temporal (line 13), or untimed assertion if there is no specific prefix (line 22). For example, if the placeholder *requiredCondition* is prefixed with "*reward_target_*" (line 10), then this is a reward-based requirement, and we select AT-RWD.

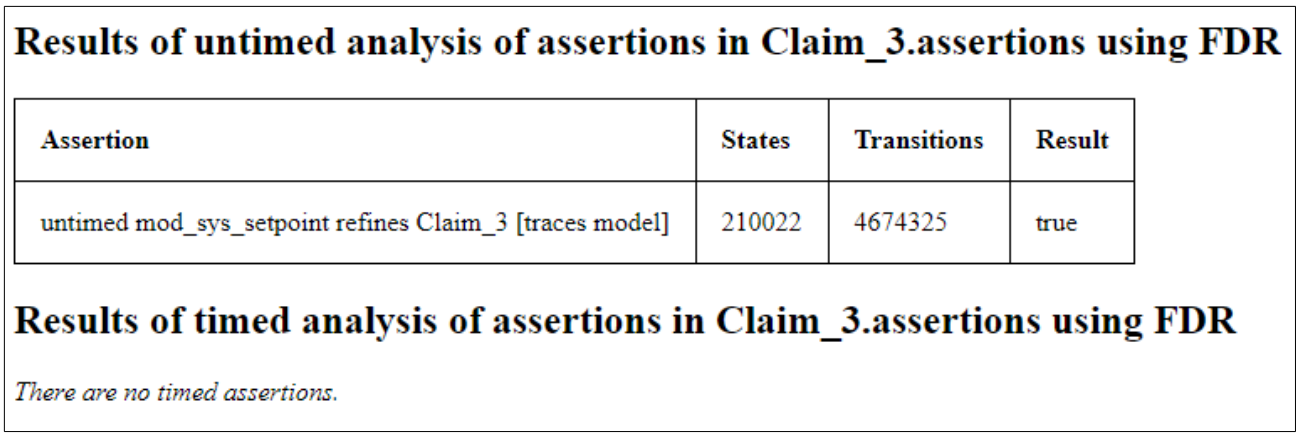
The assertions using AT-PROB, AT-RWD, and AT-TEMP may have a section for setting the constant values, which are recorded in the*guardCondition* placeholder with prefix "constant" or "multi_constants". Each constant appears as a separate element in *guardCondition.conds*. Algorithm 1 uses a local variable C (line 16) to store these two prefixes. Algorithm 1 iterates over all the elements of *guardCondition.conds* (lines 17–21) to identify the elements whose prefixes fall into C and to generate constant configurations for the chosen assertion template accordingly. This is achieved by a call to a procedure GenConstant(...). We omit the details of the call here, which can be found, together with the definition of GenConstant, in Appendix A.3.

The requirements defined using the *when* template that have no prefix are untimed requirements. Requirements defined using the templates *trigger_on_event* (line 26) and *every* (line 29) also do not require a prefix analysis.

For each kind of template, we have a Gen procedure, further described below, that defines an assertion generator. The definitions for each of the assertion generators called in Algorithm 1 are provided in Appendix A.3.

As an example, we consider a requirement defined using the Kapture template RT-UNTIMED. Its XML entry records *templ* = *when*, and its *requiredCondition* begins with the prefix "required_event". Since this prefix does not match the specialised branches for probabilistic, reward, or temporal assertions (line 22), Algorithm 1 follows the untimed branch (line 23), and calls the function GenerateUntimedAsst to generate the assertion by instantiating the AT-UTG assertion template.

Instantiation. Once a template is selected, its placeholders are instantiated with concrete values extracted from the XML requirement. This is the role of the Gen procedures called in Algorithm 1. Our assertion templates (as described in Sect. 6.1) are encoded in EGL, and the Gen procedures use values extracted from the Kapture templates to instantiate the assertion templates automatically. For illustration, we consider the following instance of our Kapture template RT-UNTIMED.



Results of untimed analysis of assertions in Claim_3.assertions using FDR

Assertion	States	Transitions	Result
untimed mod_sys_setpoint refines Claim_3 [traces model]	210022	4674325	true

Results of timed analysis of assertions in Claim_3.assertions using FDR

There are no timed assertions.

Fig. 5: Excerpt of a RoboTool FDR model-checking report, showing untimed and timed analysis results.

When `sys::stop.in` occurs,
`sys::flag.out` shall also be seen.

Here the placeholders *guard_ event* and *required_ event* are instantiated with `sys::stop.in` and `sys::flag.out`. Since AT-UTG was selected, the instantiation yields the following CSP_M specification and refinement assertion.

untimed csp Spec
csp-begin
`Spec = CHAOS(Events) [| {| sys::stop.in |} |>  (RUN({| sys::stop.in |}) /\ sys::flag.out -> Spec)`
csp-end

untimed csp Spec_impl **associated to** *module_ name*
csp-begin
`spec_impl = sys::O__(0)`
csp-end

untimed assertion A_1 :
Spec_impl **refines** Spec **in the traces model**

The assertion A_1 can be translated to CSP_M by RoboTool and verified using FDR. The result contributes to AC evidence generation, as discussed in Sect. 7.

Complexity and correctness. Let R be the number of requirements and E the total number of guard elements. The algorithm performs one pass over all requirements and, for each requirement, a linear scan of its guard elements. Thus, the running time is $O(R+E)$, and the auxiliary space consumption is $O(1)$, since only a constant number of temporary variables (namely, *claim_ID*, *templ*, *rCond*, *gCond*, *prefix*, and C) are stored.

With well-formed XML entries, each Kapture requirement matches exactly one case of the selection logic, ensuring that one assertion template is chosen unambiguously. The chosen template's placeholders are instantiated using the structured fields of the XML, guaranteeing that the generated assertions remain well-formed, consistent with Table 4, and traceable via the claim identifier.

7 Evidence model generation and integration

We recall from Sect. 3 that we assume the existence of a preliminary model-based AC created with predefined SACM patterns, where the evidence elements either remain as placeholders, since verification has not yet been performed, or need updating. Building on this assumption, our process (Fig. 3) generates assertions for the claims (Steps 1–2), obtains the corresponding verification results (Step 3), and then automatically generates and integrates evidence models into the AC (Step 4). Each evidence model carries explicit traceability information linking the verification outcome to its supported claim. Any earlier evidence models are replaced with new ones derived from updated verification results. This section focuses on Steps 3 and 4.

With the assertions generated from Step 2 and the formal semantics of RoboChart generated by RoboTool, we can conduct the verification using FDR, PRISM or Isabelle within RoboTool. During model checking, RoboTool outputs verification results as HTML reports, where each report corresponds to the evaluation of a single assertion file. These reports record whether the checked assertions hold. Examples of reports of FDR and PRISM results are shown in Figs. 5 and 6. During theorem proving, Isabelle can be invoked in RoboTool by communicating with the Isabelle server to discharge the proof, and the resulting proof log, containing the requirement identification and the result, can be reported.

With the verification results available, Algorithm 2 formalises the workflow for generating evidence from those results and integrating it into the assurance case. The algorithm takes as input the verification report *report*, the

Results of probabilistic analysis of assertions in Claim_4.assertions using PRISM

Assertion: P_Claim_4

Assertion	Const	Const	Const	states:	transitions:	result:	checkTime:
P_Claim_4	x=1	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	43.522 seconds
P_Claim_4	x=2	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	42.446 seconds
P_Claim_4	x=3	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	40.292 seconds

Fig. 6: Excerpt of a RoboTool PRISM model-checking report, showing probabilistic analysis results.

Algorithm 2: AC evidence generation and integration

Input: *report*, *kapture_xml*, *ac*
Output: updated *ac* with integrated evidence

```
1  reqId ← getClaimId(report)
2  reqEntry ← lookupBackTr(kapture_xml,reqId)
3  acClaimId ← reqEntry.forwardTr
4  tool ← getToolType(report)
5  result ← true
6  if tool = PRISM then
7  |   for r ∈ report.tables.first.rows do
8  |   |   result ← result ∧ getResult(r)
9  |   end
10 end
11 if tool = FDR then
12 |   for t ∈ report.tables do
13 |   |   result ← result ∧ getResult(t)
14 |   end
15 end
16 if tool = Isabelle then
17 |   result ← getResult(report)
18 end
19 evidence ← genEvidence(result,acClaimId,tool)
20 evidenceLink ← findByTgt(acClaimId)
21 clear(evidenceLink.source)
22 evidenceLink.source ← {evidence}
```

requirements *kapture_xml* exported from Kapture, and the current AC model *ac*. Its output is an updated version of *ac* with integrated evidence.

Algorithm 2 is defined based on the structure of the verification report; therefore, we briefly outline this structure. It differs slightly depending on which tool is used to generate the report: for FDR, a report may contain one table (untimed or timed) or two tables if no label for a model is specified. In each table, there is a row per assertion. Verification succeeds when the Result column entries for all rows are "true" in the single-table case, and only if both tables record "true" otherwise. For PRISM, a report always contains a single table with multiple rows and again verification succeeds only if all rows record "true". As shown in Figs. 5 and 6, the title of each table encodes both the identifier of the requirement and the verification tool used, which facilitates the evidence generation. For example, a title for a report can be as follows.

Results of probabilistic analysis of assertions in SR1_1_1.assertions using PRISM

Here, the tool is PRISM and the requirement is *SR1_1_1*. For Isabelle, the proof log returned can be converted into an EMF report, which contains the requirement identification, the verification tool, and the result.

The *ac* input of Algorithm 2 is assumed to adopt the AC pattern defined in [45], where each software requirement corresponds to a requirement-level claim that is further refined into one or more verification-method claims. These sub-level claims describe how the requirement is demonstrated (for instance, by model checking, theorem proving, or testing), and that they are the claims that the evidence provided directly supports. It is this evidence that our work generates and integrates or updates automatically.

Consistent with this structure, the Kapture template records two traceability links: the backwards traceability link refers to the requirement, and the forward traceability link $forwardsTr$ identifies the verification-method claim. Algorithm 2 first uses the backwards trace to locate the requirement and then the forward trace to obtain the identifier of the AC claim supported by the evidence.

Algorithm 2 operates in three stages. First, it identifies the claim to be supported by the evidence (lines 1–3). Second, it generates the evidence (lines 4–19). Finally, it updates the AC model (lines 20–22). In the first stage,

Table 5: Summary of the case studies and automation overhead. Model size is given in terms of the number of states and transitions of the RoboChart model. Times are averages over 10 runs in a 1.6 GHz Core i5 computer with 8GB of RAM.

Case study	# Assertions	Model size (states / transitions)	Assertion generation (ms)	Evidence generation (ms)
Mail robot [46]	3	8 / 11	39	56
Painting robot [47]	4	7 / 10	277	87
Underwater vehicle [37]	1	5 / 18	580	92
Maintenance robot	4	25 / 35	67	112

Table 6: Requirement templates (RT) and assertion templates (AT) used in the case studies.

Case study	Requirement Templates (RT)	Assertion Templates (AT)
Mail robot	RT-PROB, RT-RWD, RT-TEMP	AT-PROB, AT-RWD, AT-TEMP
Painting robot	RT-UNTIMED, RT-REACH	AT-UTG, AT-UTL, AT-REACH
Underwater vehicle	RT-DDLK	AT-DDLK-2
Maintenance robot	RT-UNTIMED, RT-REACH, RT-DDLK, RT-DIV	AT-UTG, AT-REACH, AT-DDLK-1, AT-DIV

Algorithm 2 extracts the requirement identifier *reqId* from the *report* (line 1). For that, we use a simple projection function GETCLAIMID; its definition and that of other simple projection functions are omitted. Next, Algorithm 2 retrieves the matching requirement entry *reqEntry* from *kapture_xml* using the backwards trace (line 2), and obtains the corresponding verification-method claim identifier *acClaimId* via the forward trace (line 3).

To generate the evidence artefact, Algorithm 2 first determines the verification tool (PRISM, FDR or Isabelle) from the report metadata (line 4). Afterwards, the aggregated verification result is recorded in a local variable *result* (lines 5–17). For PRISM, all rows must evaluate to *true* (lines 6–10); for FDR, every table must yield *true* (lines 11–15). As we only check deadlock freedom using Isabelle, its verification outcome is already a single Boolean result, and therefore, no aggregation is required (line 17). The result is then used to construct an evidence artefact using the function GENEVIDENCE (line 19).

Given a Boolean *result*, a claim identifier *acClaimId*, and a verification *tool* used, GENEVIDENCE produces an evidence element that records the verification method, the associated claim, and the outcome. Evidence is generated for both successful and failed verification outcomes, allowing the AC to faithfully reflect the current verification status of each claim. A result *true* indicates that the verification-method claim is supported, whereas a result of *false* records a verification failure and indicates that the claim is not satisfied. We assume that the resulting AC is reviewed by an engineer prior to submission to the certification authority. In our implementation, this evidence artefact is realised as an EMF model conforming to SACM, but this representation choice is not essential.

The final stage updates the AC model so that the verification-method claim identified by *acClaimId* is supported by the newly generated *evidence*. In the SACM metamodel, the relationship between a claim and the evidence that supports it is represented by a link, whose source refers to the evidence artefact and whose target refers to the supported claim. So, Algorithm 2 first locates the *evidenceLink* whose target matches *acClaimId* by invoking FINDBYTGT (line 20). Once that link has been identified, its existing source is cleared by removing either a placeholder evidence element or outdated verification results (line 21). Finally, the new *evidence* artefact is assigned as the source of the link (line 22), thereby updating the AC so that the verification-method claim is supported by the most recent verification result.

The correctness of Algorithm 2 is straightforward, as it propagates verification outcomes into evidence models while preserving traceability links, and this behaviour is validated in our case studies described next.

8 Evaluation

In this section, four case studies are presented to demonstrate the application of our algorithms for evidence model generation through FDR, PRISM, and Isabelle (Sect. 8.2). Sect. 8.3 uses these case studies to answer the research questions described in Sect. 8.1. In Sect. 8.4 we discuss threats to the validity of our results.

8.1 Research Questions

We have used four case studies to answer the following four research questions and evaluate our process.

RQ1 (Generality): To what extent can our automated evidence-generation approach be applied across different robotic domains, and how dependent is it on the choice of modelling language and verification tools?

RQ2 (Coverage): Do the requirement classification and the associated templates provide sufficient coverage for the kinds of software requirements that typically arise in robotic and autonomous systems, and can the scheme be extended when new requirement categories are encountered?

RQ3 (Completeness of integration): Does the approach ensure that all formalised requirements are systematically transformed into verification results and that these results are consistently integrated into the assurance-case argument structure?

RQ4 (Scalability and efficiency): Is the approach scalable and efficient in practice, both in terms of handling larger models and requirement sets, and in reducing the manual effort required for maintaining evolving assurance cases?

One assumption has been made for the approach: the software's modelling language has a formal semantics that enables proper formal verification. Therefore, the formalisation of the software models is outside the scope of our approach and is not evaluated.

The four selected case studies provide a diverse and representative evaluation of the proposed approach across different robotic domains, verification back-ends, and classes of software requirements. A mail delivery robot and a High Voltage Controller demonstrate evidence generation via PRISM and FDR model checking for probabilistic and discrete-event control systems, respectively. An Autonomous Underwater Vehicle case study focuses on deadlock-freedom verification using Isabelle theorem proving, illustrating applicability beyond model checking and to systems with real-valued parameters. Finally, a maintenance robot involves a larger and more structurally complex RoboChart model. Together, these case studies provide a strong basis to answer our research questions.

8.2 Case studies description

We describe each of our case studies below. The workflow implementation and case studies presented in this paper are maintained in a private GitHub repository. Public release is under consideration, and the implementation can be made available upon reasonable request.

8.2.1 Mail delivery robot

A case study of a mail-delivering robot [46] has been conducted to demonstrate the application of our work for generating evidence models using PRISM model checking.

The robot delivers mail to eight offices (1 to 8), with Office 0 serving as the charging station. An operator can request mail delivery between offices, excluding the charging station. The robot handles one task at a time: it picks up mail from the source office and delivers it to the destination. After completing a task, the robot becomes available for the next request. The robot may either remain in its current office or move to one of the offices that are adjacent according to a fixed connectivity relation; all available options are chosen with equal probability. For instance, from Office 3, it can stay, move to Office 2, or move to Office 5, each with a $1/3$ chance. The robot has a finite battery capacity, which is recharged at the charging station in discrete steps with a fixed charging rate.

The RoboChart models for the mail delivery robot can be found online[4]. A top-level system requirement considered in this case study is as follows.

TR1 The robot shall deliver the mail to the destination.

One possible cause for violation of this requirement is the robot running out of battery while away from the charging station. The robot starts with a battery capacity of 20 units, each movement consumes one unit of battery charge, and the battery is recharged in discrete steps of 4 units per update. This scenario leads to a set of software-level requirements as follows.

1. SR1_1_1: The probability of the robot running out of power at each non-charging position shall be less than 0.5 when the starting battery capacity and the charge step are 20 and 4.
2. SR1_1_2: The average number of moves before running out of power shall be greater than 8 when the starting battery capacity and the charge step are 20 and 4.
3. SR1_1_3: There is a path through which the robot will not get stuck.

It is these requirements that we handle.

Initial AC generation For the system-level mission requirement TR1, an initial AC argumentation structure is constructed. The resulting AC is shown in Fig. 7 using GSN notation. (For the sake of simplicity, context elements are omitted from the diagram.) In GSN, AC claims are referred to as goals.

In this AC, the three software-level goals "Goal for SR1_1_1", "Goal for SR1_1_2", and "Goal for SR1_1_3" are supported by the corresponding verification-level goals "Goal for VR1_1_1", "Goal for VR1_1_2", and "Goal for VR1_1_3", all of which are intended to be discharged using PRISM model checking of a RoboChart model.

We assume that, at the time this initial AC is developed, the verification activities have not been performed. As a result, no concrete verification results are yet available to be referenced as evidence. Instead, the corresponding evidence nodes in the AC are represented by placeholders ("result 1", "result 2", and "result 3"). So, the next step is to prepare for evidence generation as described below.

[4] https://robostar.cs.york.ac.uk/prob_case_studies/mail_delivery_robot/index.html

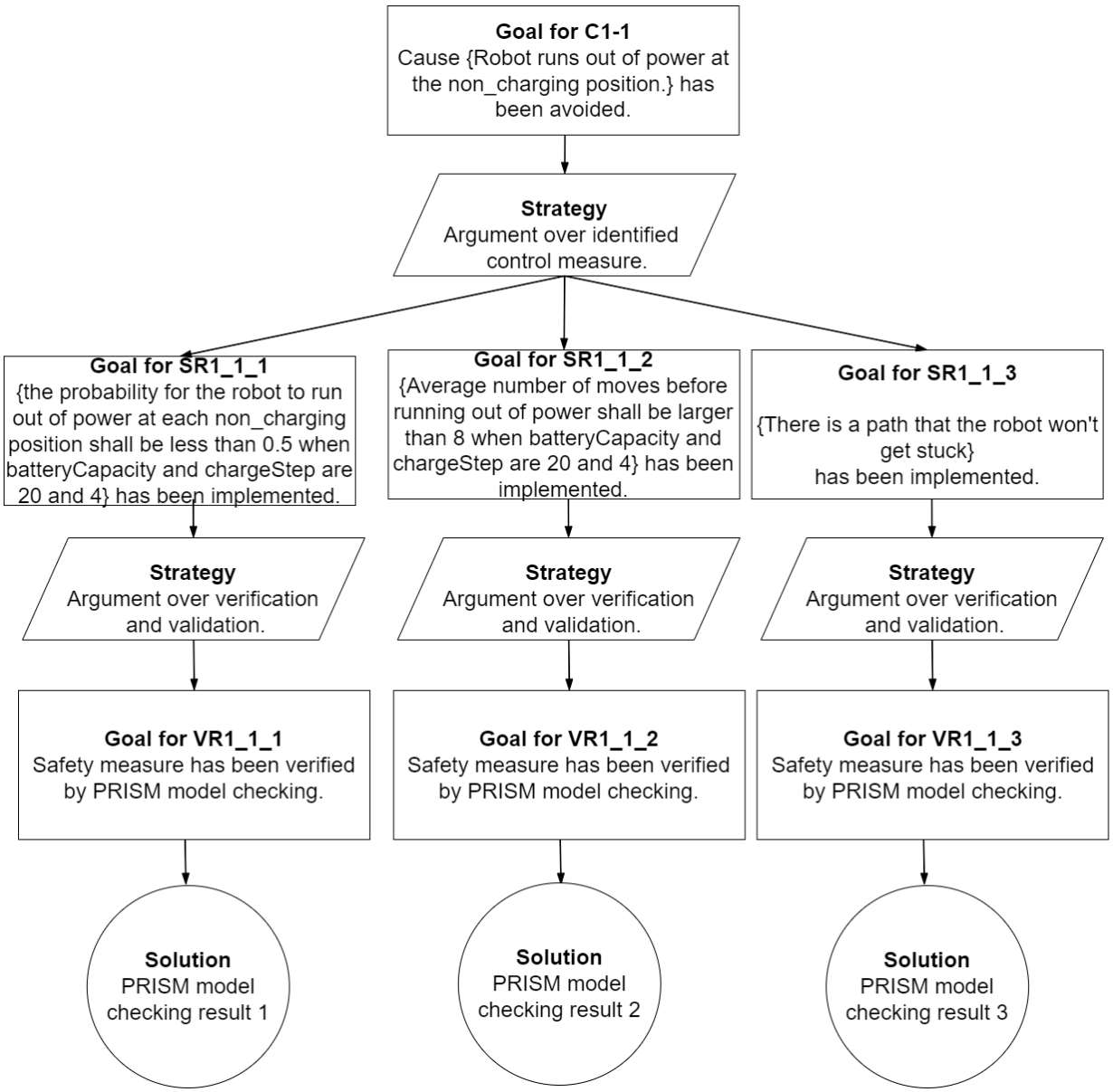


Fig. 7: AC in GSN notation for the mail-delivering robot.

Requirement structuring SR1_1_1 is a probabilistic requirement, SR1_1_2 is a reward-based requirement, and SR1_1_3 is a temporal requirement. Accordingly, the requirement templates RT-PROB, RT-RWD, and RT-TEMP have been used to describe them. Based on the RoboChart model, the requirements referenced by the AC claims are structured in Kapture as illustrated in Figs. 8–10. For clarity, we do not use fully qualified names here.

The software behaviour of the robot is modelled in a RoboChart module deliverMOD. The robotic platform rp_ref0 provides the software controller with two variables, recording the robot's position p and the current battery level c, as well as two constants, recording the starting battery capacity batteryCapacity and the amount of battery charged per update chargeStep.

The movement control behaviour of the robot software is modelled by the state machine stm_ref0. This machine includes a state Stuck, which denotes that the robot becomes stuck and is unable to move further. The event move is used to trigger movement when the current position differs from the desired goal. Battery management is modelled by the state machine stm_ref1, which includes a state batteryState representing battery charging behaviour.

For SR1_1_1 (see Fig. 8), the `When` clause defines values for the RoboChart constants in accordance with the state of the requirement: `batteryCapacity` and `chargeStep` are set as 20 and 4. The `When` clause also defines a variable `x` (prefixed with "multi_constant_" in Kapture), ranging over values 1 to 8, needed for the specification of the path formula. The scenario in which the robot runs out of power is characterised by the battery capacity `c` being zero (`c == 0`). The robot being located at a non-charging position is represented by `p == x`, where, as said above, `x` ranges from 1 to 8. Accordingly, the placeholder *path_formula* is instantiated as the conjunction of `p == x` and `c == 0`, using the temporal operator `Finally`. The probability target is less than 0.5.

For each requirement (Figs. 8–10), Kapture provides two traceability labels: a `Backwards Traceability` label that identifies the corresponding software requirement, and a `Forwards Traceability` label that identifies the

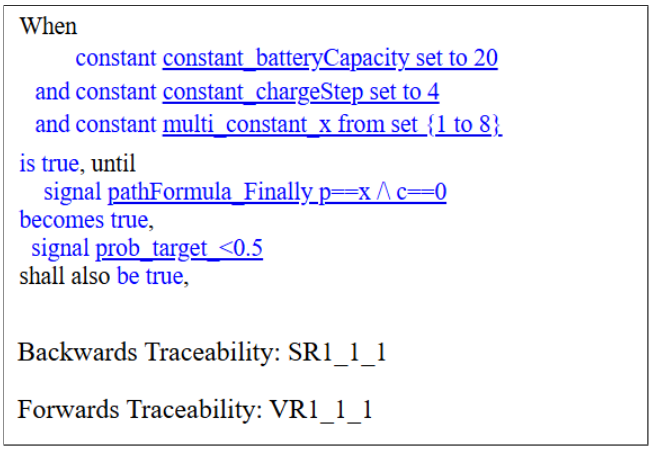


Fig. 8: SR1_1_1 exported from Kapture.

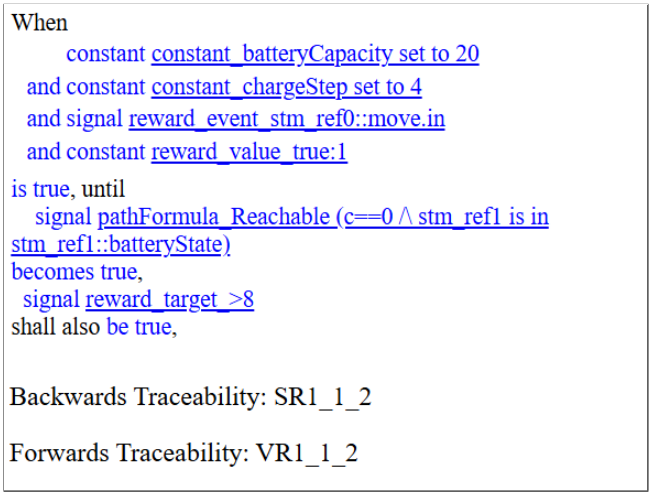


Fig. 9: SR1_1_2 exported from Kapture.

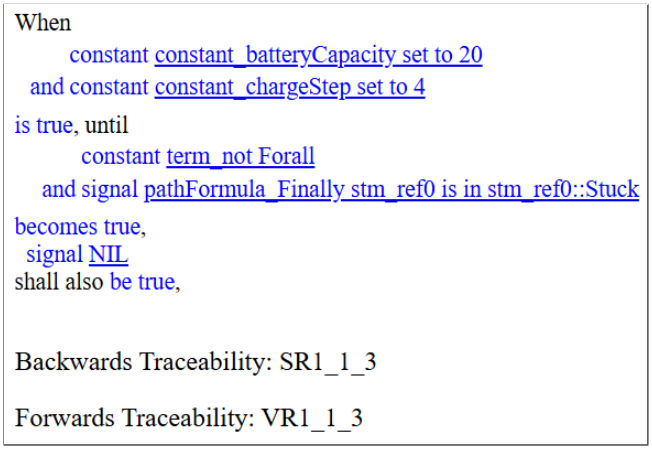


Fig. 10: SR1_1_3 exported from Kapture.

verification-method claim to which the generated evidence will be attached. This traceability information is used to integrate the verification evidence back into the AC model.

SR1_1_2 characterises the number of movements the robot can perform before battery exhaustion. The `When` clause gives values to RoboChart constants and defines the reward event as `move` with a reward value of 1, indicating that each movement consumes one unit of battery capacity. The resulting reward expression therefore measures the expected number of movements performed before the battery is depleted (`c == 0`) and the battery is ready to be charged (defined in RoboChart as `stem_ref1 is in stm_ref1::batteryState`).

SR1_1_3 requires the robot to avoid entering a stuck condition, which is represented by the state machine `stm_ref0` not being in the `Stuck` state. The `When` clause just defines the values of the constants `batteryCapacity` and `chargeStep`.

Results of probabilistic analysis of assertions in SR1_1_1.assertions using PRISM

Assertion: P_SR1_1_1

Assertion	Const	Const	Const	states:	transitions:	result:	checkTime:
P_SR1_1_1	x=1	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	43.522 seconds
P_SR1_1_1	x=2	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	42.446 seconds
P_SR1_1_1	x=3	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	40.292 seconds
P_SR1_1_1	x=4	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	41.461 seconds
P_SR1_1_1	x=5	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	40.97 seconds
P_SR1_1_1	x=6	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	43.843 seconds
P_SR1_1_1	x=7	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	42.66 seconds
P_SR1_1_1	x=8	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	39.676 seconds

Fig. 11: Model checking result for SR1_1_1.

The placeholder *path_formula* is instantiated as `stm_ref0 is in stm_ref0::Stuck`, combined with the temporal operator `not Forall`, requiring that there exists at least one execution path along which the robot does not become stuck.

With the Kapture templates, we can now generate assertions automatically as explained next.

Assertion and evidence generation Assertion generation for this case study follows the generic procedure defined in Algorithm 1, which takes as input the XML file exported from Kapture and produces a set of instantiated RoboChart assertions. For the mail delivery robot, the Kapture XML contains three requirement entries corresponding to SR1_1_1, SR1_1_2, and SR1_1_3. Each entry records the claim identifier to be supported, the Kapture template used, and the instantiated placeholders of the requirement.

Taking SR1_1_1 as an example, its Kapture entry uses the **when** template, and its *requiredCondition* is prefixed with `prob_target`. Accordingly, Algorithm 1 follows the probabilistic branch and invokes GenerateProbAsst, which instantiates the AT-PROB assertion template. The constant configurations specified in the *guardCondition* placeholder (e.g. battery capacity and charge step) are then processed to complete the assertion instantiation. The same mechanism is applied to SR1_1_2 and SR1_1_3, which are identified as reward-based and temporal requirements, respectively, and handled using the AT-RWD and AT-TEMP templates.

As a result, three RoboChart DSL assertions are automatically generated for this case study using the templates AT-PROB, AT-RWD, and AT-TEMP. RoboTool then automatically translates the RoboChart models into PRISM models. By invoking the PRISM model checker within RoboTool, the generated assertions are translated into PRISM specifications and verified. The verification results are reported in HTML format, using fully qualified names. An example verification result for SR1_1_1 is shown in Fig. 11.

Using the verification result produced by PRISM together with the traceability information recorded in the Kapture requirement entries, the algorithm identifies the verification-method claim to be supported and generates the corresponding evidence artefact. Using the verification results produced by PRISM together with the traceability information recorded in the Kapture requirement entries, Algorithm 2 is applied to generate the corresponding evidence artefact. For example, for SR1_1_1, the verification report shown in Fig. 11 encodes both the requirement identifier and the verification tool used. Algorithm 2 extracts this identifier, resolves the associated verification-method claim via the forward traceability link, and generates an evidence model from the aggregated verification result. This evidence is then integrated into the AC by replacing the placeholder solution element supporting the verification-method claim VR1_1_1_1.

As a result, the preliminary AC shown in Fig. 7 is updated with concrete evidence derived from model checking, yielding the AC shown in Fig. 12.

With this example, we demonstrate the application of our workflow and tools from start to end. In the next section, we describe our additional examples, covering different tools (FDR and Isabelle) and focusing on the extra lessons learned in those case studies.

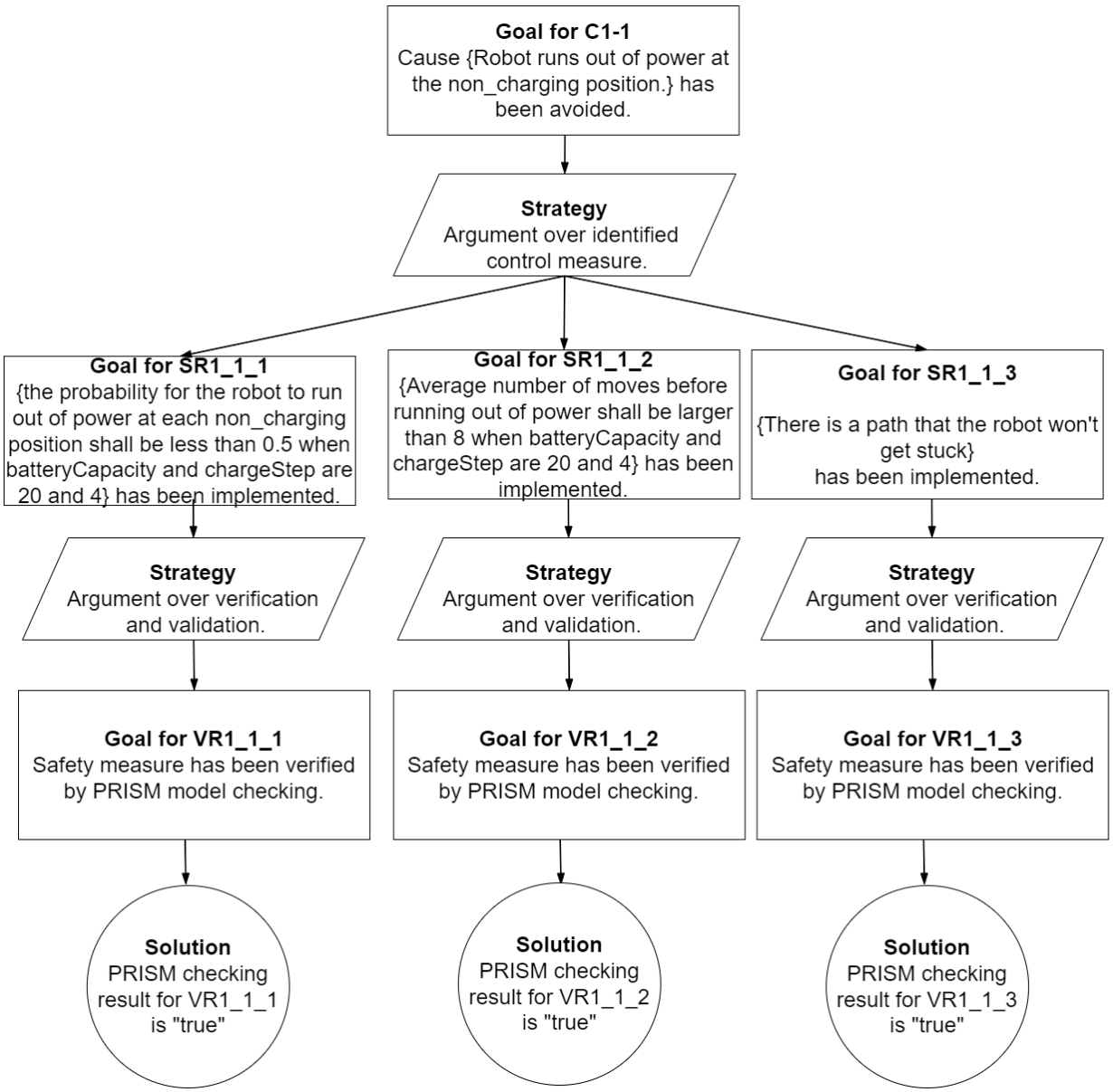


Fig. 12: AC module for mail delivery robot updated with formal evidence.

8.2.2 Painting robot

The industrial painting robot [47] uses high voltage to charge paint particles, improving efficiency and quality. However, this poses safety risks, as high voltage can cause discharge sparks, potentially igniting paint particles in the air and leading to a fire. To ensure safety, a High Voltage Controller (HVC) has been developed.

The system is powered by a 24V power signal, which supplies electricity to the HVC. The HVC uses the desired voltage setting to calculate a Pulse Width Modulation (PWM) value. This PWM value, ranging from 0% to 100%, corresponds to an analogue voltage signal between 0 and 10V, which is amplified by a transformer. The amplified signal serves as the input to the HVC, which adjusts the actual high voltage output. The system continuously measures the actual voltage and the current in real time, providing feedback to the HVC. These measurements are used to dynamically adjust the PWM output to ensure that the output voltage matches the desired set point.

At the software level, three scenarios are identified that could lead to hazardous high-voltage behaviour:

- the HVC does not respond to changes of the voltage set point, resulting in a constant voltage output;
- the voltage set point is not disabled when the 24V power supply fails, causing the HVC to continue outputting high voltage;
- the PWM output is not disabled when the 24V power supply is off, leading to continued high-voltage output.

The software behaviour of the painting robot is modelled as a RoboChart module `mod_sys`. Here, we describe only the elements of that module necessary to understand the example requirements; the complete RoboChart model is available online[5]. The event `ext_pow24VStatus` has an enumerated type `Power` with two values, `ON` and `OFF`, representing the

[5] https://robostar.cs.york.ac.uk/case_studies/hvc/index.html

When
signal mod_sys::ext_pow24VStatus.in.Power_Off
is true
signal mod_sys::ext_setPoint.out.0
shall also be true.

Fig. 13: HVC Claim for SR1_2_1.

The property
reachable_untimed(signal State_machine :: ClosedLoop, signal State_machine)
shall always be true.

Fig. 14: HVC Claim for SR1_1_1 validation.

```
1 untimed csp SR1_2_1 csp-begin
2 SR1_2_1 = CHAOS(Events) [| {|
  mod_sys::ext_pow24VStatus.in.Power_Off|} |> (RUN
  ({|mod_sys::ext_pow24VStatus.in.Power_Off|}) /\
  mod_sys::ext_setPoint.out.0 -> SR1_3_1)
3 csp-end
4
5 untimed csp SR1_2_1_implement associated to
  mod_sys csp-begin
6 SR1_2_1_implement = mod_sys::O__(0)
7 csp-end
8
9 untimed assertion A_SR1_2_1 : SR1_2_1_implement
  refines SR1_2_1  in the traces model
```

Fig. 15: HVC assertion for SR1_2_1 verification.

status of the external 24V power supply. The event ext_setPoint is used to communicate the desired voltage set point. The behaviour of the HVC is modelled by the state machine called State_machine. Its ClosedLoop state specifies the normal operation, where the controller regulates the output voltage with respect to the set point. If the voltage exceeds predefined upper or lower bounds, the state machine transitions from ClosedLoop to another state ErrorMode.

The following software requirements are identified.

1. SR1_1_1: The actual voltage shall follow the set-point during normal operation.
2. SR1_2_1: The voltage set point shall be set to 0 when the 24V power supply is switched off.
3. SR1_3_1: The PWM output shall be set to 0 when the 24V power supply is off.
4. SR1_1_1_VA: The ClosedLoop state shall be reachable in the state machine State_machine.

An initial AC has been constructed to argue that the software-level causes of unintended high-voltage behaviour are mitigated by the HVC software. The AC structure focuses on the satisfaction of the software requirements identified above. For brevity, the AC diagram is omitted. In this AC, the four goals corresponding to SR1_1_1, SR1_1_2, SR1_1_3, and SR1_1_1_VA are each supported by a verification-method goal. All verification activities for these goals are performed using FDR.

Requirement structuring To structure the requirements in Kapture, for the requirements SR1_1_1, SR1_2_1, and SR1_3_1 we have used the RT-UNTIMED template, while for SR1_1_1_VA we have used RT-REACH. We use SR1_2_1 and SR1_1_1_VA as examples. Figs. 13 and 14 show their requirement specifications in Kapture. The complete set of structured requirements and generated assertions is available in [48].

For SR1_2_1, the placeholder `guard_event` is instantiated with the event ext_pow24Vstatus (identified by its fully qualified name) carrying the value Power_Off, and the `required_event` is instantiated with the event ext_setPoint producing the output value 0.

For SR1_1_1_VA, the `function` placeholder is instantiated with the predefined `reachable_untimed` function, and constraining the analysis to the state machine State_machine via the `scope` placeholder.

Assertion generation and evidence generation The assertion for SR1_2_1, shown in Fig. 15, is generated using the assertion template AT-UTG. We refer to the explanation of AT-UTG in Section 6 for a detailed explanation of how

the CSP definitions in Fig. 15 capture our requirement. We highlight that its definition requires considerable expertise in CSP, which is obviated by the use of our approach and tool.

In RoboTool, the RoboChart models are automatically translated into CSP models for verification. The FDR model checker then checks the generated assertions against these models under the CSP_M semantics.

8.2.3 Autonomous Underwater Vehicle

The Autonomous Underwater Vehicle (AUV) can operate under human supervision or autonomously to perform underwater maintenance and intervention tasks. A key safety issue for such systems is the risk of collision with subsea infrastructure, which may arise from both operator actions and autonomous behaviour.

The core safety component of the AUV is the Last Response Engine (LRE), which monitors system behaviour and enforces safety constraints. In this case study, we apply our approach to verify deadlock-freedom of the LRE, which is captured as a verification-method claim in the AC. The property is verified using Isabelle. We use the requirement template RT-DDLK and the assertion template AT-DDLK-2 to generate the assertion. The generated assertion and the proof are as follows.

```
lemma LRE_deadlock_free: "deadlock_free LREMachine"
  apply deadlock_free
  by (metis St.exhaust_disc)
```

The formalised assertion is `deadlock_free LREMachine`, where `LREMachine` denotes the identifier of the LRE model in the Z-Machine notation. The property is automatically verified in Isabelle by applying the `deadlock_free` proof method and invoking the sledgehammer [49] Isabelle tool, which generates the proof line `by (metis St.exhaust_disc)` automatically. Details of the RoboChart formalisation using Z-Machine and the proof method definition are in [37].

Because the LRE model includes real number parameters, it cannot be directly verified using the FDR model checker. This case study therefore illustrates the complementary role of theorem proving within our approach, enabling the use of verification of properties beyond the scope of model checking in AC development. The generation and integration of the corresponding evidence into the AC follow the same process as in the previous case studies.

8.2.4 Maintenance robot

The fourth case study is a maintenance robot that unscrews laptop screens by identifying the screws on the laptop and applying the driver force to the screws automatically. The way used to model the robot in RoboChart is in two steps via the method provided in [50]. The first step defined a RoboArch [51] model for the robot; then, that model was translated automatically to RoboChart. This RoboChart model represents the architecture of the robot software, and application details were subsequently added to describe the complete behaviour of the robot.

The RoboChart model of this robot adopts a variant of the MAPE-K [52] design structure. The model contains an Adaptation controller that specifies the main MAPE-K loop. The controller has six state machines to represent the behaviour of each MAPE-K component, among which we discuss two of them: the state machine Adaptation_Knowledge and the state machine Adaptation_Plan. Adaptation_Knowledge stores all the variables shared between components of the Adaptation controller. The image variable records the image received from the managed system. The robot is equipped with a camera that takes images of the laptop screws. Every time Adaptation_Knowledge receives an image request through the event get_image, it should send out the stored image through the event image. The state machine Adaptation_Plan creates a plan to adapt to an identified anomaly and the key state of the machine is the MakePlan state, where the plan is created.

An AC has been built based on the structural RoboChart model to justify the safety goal of the robot. A set of software requirements referenced in the AC claims that require model checking is given below.

- SR1: The Adaptation_Knowledge state machine shall always respond via the event image when the event get_image is received.
- SR2: The MakePlan step of the plan phase shall be reachable in the Adaptation_Plan state machine.
- SR3: The Adaptation_Plan state machine shall be deadlock-free.
- SR4: The Adaptation_Plan state machine shall be divergence-free.

We first used the Kapture templates RT-UNTIMED, RT-REACH, RT-DDLK, and RT-DIV to create requirements. Secondly, we have used the assertion templates AT-UTG, AT-REACH/DDLK/DIV to generate assertions. Both the requirements exported from Kapture and the CSP assertions generated are in Appendix A.4 and [53].

8.3 Evaluation

This section answers our research questions proposed in Sect. 8.1.

RQ1 (Generality) Our approach generates formal evidence through formal verification in support of AC claims. Its applicability, therefore, depends primarily on the availability of suitable modelling languages with formal semantics, and on the verification tools employed, rather than on the application domain itself.

This is illustrated by our case studies: the mail delivery robot, the high-voltage controller, the underwater vehicle, and the unscrew robot originate from different domains of application, yet those domains do not affect the evidence model

generation process. What is essential is the use of RoboChart, whose formal semantics described in CSP and PRISM for model checking and in Z-Machine notations for theorem proving enable the automatic generation of verification models. In principle, other modelling languages with comparable formal semantics could also be adopted in place of RoboChart.

Our approach incorporates the FDR and PRISM model checkers and the Isabelle theorem prover. Accordingly, the assertion templates are defined in terms of CSP_M, PRISM, and Isar, but the core workflow is independent of these particular tools. The approach can be adapted to other verification backends by developing suitable assertion templates aligned with their formal semantics needs.

RQ2 (Coverage) The coverage of our approach stems from the support for multiple formal semantic models of RoboChart and their associated verification tools. Unlike single-method frameworks that rely on one formalism or verification tool, our workflow combines model checking and theorem proving across the CSP, Z-Machine, and PRISM semantics. This multi-semantics design enables different classes of properties to be verified within a unified framework, thereby mitigating the limitations of individual tools and techniques. Table 3 summarises the requirement types currently supported across these domains.

Within this framework, the CSP semantics is used to address behavioural properties, such as untimed and timed safety and general behavioural correctness (reachability, divergence-freedom, deadlock-freedom, and termination). The Z-Machine semantics is used to deal with scalable state-based reasoning, supporting theorem proving of deadlock-freedom as a complementary form of verification. Finally, with the PRISM semantics, we extend the coverage to both probabilistic (quantitative) and non-probabilistic (qualitative temporal) properties, allowing the analysis of reliability, performance, and liveness.

Each requirement category is realised by one or more requirement templates, which define the syntactic and semantic patterns used to generate corresponding assertion templates. The current templates deliberately focus on common and well-defined verification patterns within each semantic domain. In the CSP domain, the templates are event-based and trace-oriented, each designed for a specific verification pattern such as the occurrence of a guard event followed by either the same guard or a required event. These templates address the most common forms of safety requirements over event traces, but they do not yet cover all possible behavioural combinations. For example, cases where a guard must be immediately followed by a required event would require a separate template. Furthermore, because the CSP semantics focuses on event interactions rather than explicit states, they are less suited to capturing data-dependent or condition-based requirements directly. Such cases can instead be expressed through the PRISM domain's qualitative temporal templates, which provide higher-level expressiveness via temporal operators over state predicates.

For timed requirements, the framework currently supports deadline constraints, where a trigger event must be followed by a target event within a bounded time interval. This represents a common real-time software requirement pattern in embedded and robotic systems [54]. However, time-related properties are inherently more complex and may require additional specialised templates to address case-specific needs. Our approach is designed to accommodate such extensions by allowing new requirements templates and their corresponding assertion templates to be incorporated into the workflow and tool.

In the PRISM domain, the templates for probabilistic and temporal requirements adopt a deliberately general structure, with their expressive power concentrated in the `path_formula` placeholder. This design follows the semantics of PRISM, where temporal and probabilistic behaviours are captured by logic formulas over state predicates. As a result, even though the template structure itself is simple, it can accommodate properties of varying complexity, ranging from basic reachability or safety checks to nested temporal and reward-based expressions, without requiring multiple specialised templates.

Overall, the current set of templates achieves practical coverage across the behavioural, state-based, and probabilistic aspects of RoboChart, and has proved sufficient for the range of requirements encountered in our case studies. While the expressive scope is intentionally conservative, the framework is designed to support incremental extension of the template library to accommodate additional behavioural and timing patterns when required.

Limitations of template coverage. While the current templates cover common requirement patterns encountered in our case studies, certain patterns are not yet supported and would require additional templates in future, and we briefly discuss here.

- **Immediate-next relations:** Requirements of the form "event A must be *immediately* followed by event B" (without any intervening events) cannot be expressed by the current CSP templates, which focus on *eventual* occurrence patterns.
- **Complex data-dependent conditions:** Requirements involving arithmetic constraints or predicates over multiple variables that cannot be naturally encoded as RoboChart events are better suited to the PRISM domain's state-based templates.
- **Nested probabilistic properties:** Properties requiring multiple reward structures or nested probabilistic operators are not covered by the current PRISM templates, though they could be added if needed.

The framework is designed to accommodate such extensions: new requirement and assertion templates can be defined following the patterns established in Sections 5.3 and 6.1, and integrated into the approach with minimal modification to the core workflow.

RQ3 (Completeness of Integration) To answer this research question, we evaluate the completeness of integration in terms of the structure of the resulting ACs. Specifically, we consider two aspects of structural completeness. First, end-to-end traceability is maintained throughout the workflow: from the claim to be supported, to the structured requirement referenced in the claim, to the assertion automatically generated from the requirement, and finally to the verification

result produced by the verification tool. This traceability ensures that every claim linked to a formal requirement is systematically discharged with corresponding evidence, and that the evidence is correctly integrated into the AC modules.

Second, the instantiated evidence models follow a predefined AC pattern introduced in [45], implemented by Algorithm 2. This pattern explicitly defines the expected structure of an AC fragment, including the relationships between claims, verification-method claims, and evidence links. As evidence integration is performed by construction according to this pattern, the resulting AC fragments are structurally complete by design. Across the case studies, we inspected the generated AC fragments and confirmed that they conform to the intended pattern, with each verification-method claim being associated with a corresponding evidence link. This provides confidence that the approach consistently produces AC fragments that are structurally complete with respect to their claims and evidence links.

While the overall completeness of an AC also depends on the adequacy of the claims and requirements themselves, our workflow guarantees structural completeness of integration: once the links between claims and requirements are provided, every linked requirement is (automatically) formalised, verified, and integrated into the AC with its corresponding evidence.

RQ4 (Scalability and Practical Efficiency) To evaluate efficiency, we focus on the steps that are automated by our approach: the generation of assertions from structured requirements, and the instantiation and integration of evidence models from verification results.

All case studies were executed on a 1.6 GHz Core i5 computer with 8GB of RAM. Table 5 reports the average execution time for the case studies, calculated over 10 runs. The automated steps for assertion generation and evidence integration completed within milliseconds, with the largest example (the underwater vehicle) requiring less than 0.7 seconds in total. These results confirm that the scalability of our workflow is determined by the verification back-ends, and that our work does not introduce any significant additional overhead.

This low computational cost is because both assertion generation and evidence instantiation are lightweight transformations. Their complexity grows linearly with the number of requirements: assertion generation has complexity $O(R+E)$ where R is the number of requirements and E is the total number of guard elements (Algorithm 1), and evidence integration has complexity $O(R)$ (Algorithm 2), where R is the number of requirements. The true scalability bottleneck lies in the verification backends (FDR, PRISM, Isabelle), which are well-studied and widely used in industry. Our contribution of assertion generation and evidence integration operates independently of model size, with verification time dominating the workflow based on model complexity and tool performance rather than our approach.

In engineering practice, formal verification is typically applied to a focused set of safety-critical requirements, usually in the order of tens. This aligns with the scale of our case studies and indicates that the automation overhead remains negligible even for realistic verification scenarios with larger models.

In contrast, a manual workflow requires engineers to write each assertion and link each verification result to an AC module by hand. Based on our experience, writing one assertion typically takes 3–10 minutes depending on its nature (refinement-style properties tending towards the upper end), plus around 1–2 minutes to link the corresponding verification result into the AC. For case studies with tens of assertions, this accumulates to hours of manual effort, whereas our automation reduces human effort to triggering the workflow; the runtime overhead then grows roughly linearly with the number of assertions and, based on our measurements reported in Table 5, is expected to remain in the order of seconds. In addition, the automation also eliminates many of the manual editing steps, thereby reducing the likelihood of human error in evidence generation and integration.

Our evaluation based on the four case studies involves relatively small RoboChart models and only a few assertions. These are sufficient to demonstrate feasibility and illustrate different requirement categories, but they do not yet represent industrial-scale applications. As future work, we plan to apply the workflow to larger models with more complex requirements, in order to further validate its scalability and practical efficiency.

8.4 Threats to validity

Construct validity threats may arise from the assumptions about the validity of the software requirements and design models. To mitigate this, the safety requirements and RoboChart models used in the case studies have been mainly taken from previously validated examples [34, 47, 55]. This increases our confidence in their suitability.

External validity threats concern the restricted use of RoboChart as the modelling language and FDR, PRISM, and Isabelle as the verification back-ends. Although other languages and tools have not been considered, the core concept of our workflow is not limited to RoboChart: it can be applied to any modelling notation with formal semantics and associated verification support. Extending the evaluation to additional notations and tools will further improve the validity and generalisability of the findings.

Internal validity threats may arise from the implementation of the approach, in particular, the accuracy of generating assertions from requirements. To address this, we relied on three complementary measures. First, the correctness of several key assertion templates has been analysed during their design, as discussed in Sect. 6.1, providing assurance that the generated assertions reflect the intended semantics. Second, formalised assertions from the literature have been used for comparison, ensuring that the automatically generated assertions are consistent with prior work. Finally, extensive testing has been carried out on the evidence model generation, comparing the produced outputs with expected results, which increases confidence in the reliability of the generated evidence models.

9 Related work

A common approach to applying formal verification in the AC process is referencing existing verification results within the AC evidence component [6–8]. Thus, the process of obtaining verification results to support claims is usually not addressed. Furthermore, previous research typically focuses on generating evidence from formal design models, whereas RAS systems are more commonly developed using MDE techniques and semi-formal modelling languages. A comprehensive solution is therefore needed to effectively incorporate formal verification into ACs for evidence generation. Our work proposes an integrated approach to derive formal evidence from argument claims for RAS systems modelled in RoboChart, thereby streamlining the AC process with formal verification support.

Early efforts towards integration include the work of Denney and Pai [9], who integrated formal verification into the Assurance Case Automation Toolset (AdvoCATE). AdvoCATE invokes AutoCert, a static code analysis tool that verifies source code generated from system models against formally specified properties and produces machine-checkable certificates, which are subsequently integrated into the assurance case. Prokhorova et al. [8] demonstrated how different requirement types can be addressed by combining multiple verification methods, but their workflow relied on manual steps and did not achieve automation. Bourbouh et al. [10, 56] investigated the integration of heterogeneous formal methods into ACs, combining the contracts with Simulink-to-Lustre translation and Event-B refinement proofs. The contracts are written in the restricted natural language of the Formal Requirements Elicitation Tool (FRET) [57], referred to as FRETISH. However, at the component verification level, the Lustre and Event-B models were created from Simulink models manually, so the consistency between these models and the design artefacts required expert review.

For requirement formalisation, several approaches translate natural language or semi-structured requirements into temporal logic. The FORM-L approach [11] employed ontology and templates to resolve ambiguities, producing semi-formal requirements for Modelica models, but its expressiveness is limited. Bolton et al. [12] generated LTL assertions from predefined templates to verify Human-Automation Interaction behaviour, but only for a restricted set of properties. ARSENAL [14] applied parsing and domain ontologies to translate natural language into LTL, providing more flexibility at the cost of manual ontology development. A prominent representative of structured requirement formalisation is FRET [57], an open-source framework developed by NASA. FRET allows requirements to be written in FRETISH, which requires users to manually populate six fields, and automatically translated into future-time and past-time LTL.Recent work by Mavridou et al. [58] extends FRET with explicit support for probabilistic requirements by enriching the FRETISH language and providing an automated translation into PCTL*, together with validation of the generated formulas.

More recently, large language models (LLMs) were explored as a means of automating requirement formalisation. Zhao et al. proposed NL2CTL [15], which translates requirements into CTL/LTL specifications; Cosler et al. presented nl2spec [17], which supports interactive correction of sub-formula translations; Li et al. introduced a self-supervised framework for safety-compliant LTL [16]; and Xu et al. improved robustness by learning from failed translations [18]. These works showed the promise of LLMs but remain limited to propositional temporal logics (LTL/CTL) and achieve only partial accuracy as reported in the literature. By contrast, our approach covers CSP refinement, and CTL and PCTL properties.

The work of Denney and Pai [9] also addressed the formalisation of properties for formal verification within assurance cases. Among its practical functions for AC construction, AdvoCATE provides a claim formalisation (CF) pattern, through which users interactively supply the values of the pattern parameters based on informal claims and the language and logic adopted for formalisation. One of the parameters is the claim itself, directly expressed as a logical formula. In this way, informal claims are formalised and represented as structured sub-claims within the instantiated pattern. The formula is placed into an atomic slot of the pattern. As a result, the internal structure of the formula is not constrained by the pattern itself. This design offers greater expressive freedom but requires users to possess substantial expertise in the underlying formal languages and logics. In this respect, the CF pattern is conceptually analogous to our RT templates in that both aim to bridge informal statements and formal verification artefacts. However, while CF supports the interactive formalisation of assurance case claims by allowing users to directly provide logical formulae, our RT templates focus on the systematic structuring of software requirements and constrain the form of the resulting assertions to reduce the need for user-level knowledge of specific formal languages.

Domain-specific solutions have also been proposed for RoboChart. Lindoso et al. [13] have developed a diagrammatic notation mixing UML activity elements with RoboChart constructs, from which CSP specifications can be derived and verified with FDR. Their diagrams must be created manually, in contrast to our template-based automatic assertion generation. Use of Lindoso's diagrams can, however, be an alternative to Kapture templates.

The Isabelle/SACM framework [55, 59, 60] formalised ACs by embedding the SACM metamodel into Isabelle, yielding machine-checkable cases that can be verified for logical consistency. Since cases are represented in Isabelle/Isar, results from formal proofs, such as deadlock freedom of RoboChart models, can be linked directly as evidence. The emphasis of this work is on AC formalisation and consistency checking, with limited automation support.

Building on SACM-based foundations, Wei et al. [61] developed the ACME tool for AC management. One strand of this work integrates model validation into the AC process by allowing claim nodes to be annotated with Constrained Natural Language (CNL) rules, which are automatically translated into Epsilon EVL and executed on EMF-based RoboChart models, with outcomes recorded as evidence. The ACCESS methodology extends this vision by positioning ACs as central artefacts in system development, integrating Isabelle/SACM for formalised cases, supporting model validation with Epsilon rules, and introducing dynamic ACs that evolve at runtime with system data [62]. Together, these

Table 7: Coverage of assurance case workflow stages across related lines of work.

Reference (line of work)	Req. struct.	Assert. gen.	Model transf.	Verif. exec.	Evid. integ.	Trace-ability
Denney & Pai – AutoCert / AdvoCATE [9]	✗	✓	✓	✓	✓	✓
Prokhorova et al. – Multi-tool FV (Event-B-centred) [8]	✗	✗	✗	✓	✓	✗
Bourbouh et al. – Heterogeneous FM (FRET/Lustre/Event-B) [10, 57, 58]	✓	✓	✓	✓	✓	✗
Bouskela et al. – FORM-L [11]	✓	✓	✗	✗	✗	✗
Bolton et al. – Template-based LTL [12]	✓	✓	✗	✓	✗	✗
Ghosh et al. – ARSENAL [14]	✓	✓	✗	✗	✗	✗
Zhao et al.; Cosler et al.; Li et al.; Xu et al. – LLM-based NL2LTL [15–18]	✗	✓	✗	✗	✗	✗
Lindoso et al. – Diagrammatic CSP (RoboChart) [13]	✓	✓	✓	✓	✗	✗
Nemouchi et al.; Foster et al. – Isabelle/SACM [55, 59, 60]	✗	✗	✗	✓	✓	✓
Wei et al. – CNL / ACME [61]	✓	✓	✗	✓	✓	✓
Wei et al. – ACCESS [62]	✗	✗	✗	✓	✓	✓
Sorokin et al. – ETB [3]	✗	✗	✗	✗	✓	✓
Sljivo et al. – Tool Confidence Patterns [63]	✗	✗	✗	✗	✓	✓
This work	✓	✓	✓	✓	✓	✓

Notation: ✓ the stage is explicitly addressed in the work; ✗ the stage is not addressed.

Note: Some lines of work (e.g., AdvoCATE [9], FORM-L [11], ARSENAL [14], and ETB [3]) are explicitly named in the original publications. Others have no explicit name; we use short descriptive labels (e.g., "Template-based LTL" [12], "LLM-based L2LTL" [15–18], and "Diagrammatic CSP (RoboChart)" [13]) for clarity, while citations always point to the original sources. CNL/ACME and ACCESS are listed separately as they represent different strands of work by the same authors, addressing distinct stages of the assurance case workflow.

approaches focus on formalisation, validation, and lifecycle management of ACs. In contrast, our work does not formalise the AC itself, but integrates formal verification results from FDR, PRISM, and Isabelle into the assurance workflow.

Sorokin et al. [3] employed the Evidential Tool Bus (ETB) to orchestrate testing and simulation tools for incremental AC maintenance. Their focus, however, is on non-formal verification artefacts, whereas our work integrates formal verification results into the AC workflow.

Sljivo et al. [63] proposed reusable tool confidence patterns to strengthen the justification of verification results in assurance cases. These patterns explicitly capture the assumptions, guarantees, and applicability conditions of verification tools, making the use of tool-generated evidence more transparent and credible. The patterns are realised within the AdvoCATE framework, but the contribution primarily concerns the structuring of arguments about tool trustworthiness rather than the automation of verification or evidence generation. Our automatically generated verification evidence can complement this line of work by providing formally verified results that can be embedded within such patterns.

Table 7 summarises which stages of the assurance case workflow are explicitly addressed by different lines of work. The workflow stages used in the table are defined with respect to the scope of our approach. As a result, our work naturally addresses all listed stages, while other works cover different subsets according to their respective research focus. For conciseness, we do not describe each stage individually for every work; instead, the table supports the discussion of how our approach relates to existing work in terms of workflow focus. Different strands of work address different aspects of an AC development: requirement formalisation into temporal logics (FORM-L, Template-based LTL, ARSENAL, LLM-based NL2LTL), model validation integrated with ACs (CNL/ACME), AC formalisation and consistency checking (Isabelle/SACM, ACCESS), and evidence orchestration or tool patterns (ETB, AdvoCATE). AdvoCATE is marked as not addressing requirement structuring but as supporting assertion generation. While AdvoCATE does not provide mechanisms for structuring prior to formalisation, it does support the formalisation of informal statements into verification properties through its claim formalisation patterns. Our work concentrates on the formal verification dimension: we automate the generation of CSP and PCTL assertions from structured requirements, verify them with heterogeneous backends (FDR, PRISM, Isabelle), and inject the results as evidence into ACs. We acknowledge that requirement structuring still requires manual input and that traceability is only partially supported via assurance patterns, but our contribution lies in advancing automation for formal verification evidence within AC workflows.

10 Conclusions and future work

This paper has presented a model-based approach to automating evidence generation for ACs in RAS. By embedding model checking and theorem proving into the assurance workflow, our approach closes the automation gap that left evidence generation outside. Requirements referenced in AC claims are classified and linked to customised templates, enabling the automatic derivation of tool-ready assertions. Verification is then delegated to suitable backends (FDR, PRISM, Isabelle/HOL), and the results are integrated as assurance evidence. Collectively, these contributions establish a unified framework for AC engineering supported by formal verification.

Our case studies demonstrate that our approach reduces manual effort from hours to seconds, avoids human errors in evidence handling, and ensures that assurance evidence evolves consistently with software changes. Among the assertion templates we defined, those for CSP refinement stand out as particularly valuable: they encapsulate some of the most challenging aspects of formalisation, where a deep understanding of CSP syntax and semantics would otherwise be required. By lowering this learning barrier, the templates directly support the wider adoption of formal

verification in engineering practice. Other templates, while associated with a less steep learning curve, also contribute to easing the transition from natural language requirements to verification-ready assertions.

Looking ahead, we see several promising directions for future work. One line of extension is to integrate additional verification and validation techniques, such as testing and simulation, to complement the current focus on formal verification. Secondly, we currently check refinement in the traces model to focus on safety; an extension is to broaden this to include richer refinement models, such as the failures model.

Another important direction concerns the automation of theorem-proving-based evidence generation. At present, deadlock-freedom properties are verified in Isabelle by generating the Z-Machine semantics and proof obligations within a single Isabelle theory file, while the subsequent identification of the supported AC claim and the integration of the resulting evidence follow the same workflow but are carried out manually. Future work will focus on fully automating this process, aligning theorem-proving-based verification with the level of automation already achieved for model checking.

Another promising direction concerns requirement formalisation. Our current approach reduces the need for formal expertise through structured templates, but it still requires semi-structured input via Kapture. Advances in Natural Language Processing (NLP) offer opportunities to further automate this step. Traditional NLP pipelines, such as semantic parsing, have been applied (e.g., NL2CTL [15], nl2spec [17]) to generate temporal logic from natural language, but they rely heavily on domain-specific annotations. Recent progress in Large Language models (LLMs) provides new possibilities. LLMs such as GPT-4 have demonstrated strong capabilities in generating structured representations from free text and could assist in translating natural language requirements into formal assertions. Unlike earlier machine learning approaches, LLMs benefit from large-scale pretraining and therefore require less task-specific data. Exploring the integration of LLMs with our assertion-generation process is a promising avenue of work to improve both automation and usability.

Finally, enhancing tool support and integration into AC development environments will be crucial to improve usability and encourage adoption.

Acknowledgements

This work was partially funded by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 812788 (MSCA-ETN SAS). It was also supported in part by the UK EPSRC under grants EP/R025479/1 and EP/V02 6801/2, and by the Royal Academy of Engineering under grants CiET1719/45 and IF2122\183. Additional support was provided by the RoboSAPIENS project, funded by the European Commission's Horizon programme under grant agreement No. 101133807. We would like to thank Nick Tudor and Colin O'Halloran from D-RisQ for kindly providing free access to the Kapture tool used in this work.

A Appendix

Appendices A.1 and A.2 provide the additional requirement and assertion templates that complement those presented in Sect. 5.3 and Sect. 6. Appendix A.3 presents the algorithms corresponding to the auxiliary procedures invoked in Algorithm 1 (e.g. GenerateGeneralAsst, GenerateTimedAsst, etc.). Appendix A.4 presents the set of generated assertions for the unscrew robot.

A.1 Requirement templates

This section provides the software requirement templates for the requirement categories of *reachability*, *divergence-freedom*, *termination*, and *probability*, and *temporal*.

Reachability requirements Reachability requirements specify whether a given state shall or shall not be reachable in a RoboChart component. We capture this pattern in the reachability template (RT-REACH).

RT-REACH

The property *function*(*state*, *scope*) shall *bool_stmt* be true.

- *function* ::= `reachable_untimed` | `reachable_timed` | `reachable`
- *state* ::= *RoboChart_state_identifier*
- *scope* ::= *RoboChart_component_identifier*
- *bool_stmt* ::= `always` | `never`

Divergence-freedom requirements The divergence-freedom requirement is captured in the template (RT-DIV).

RT-DIV

The property *function*(*scope*) shall *bool_stmt* be true.

* *function* ::= `divergence_free_untimed` | `divergence_free_timed` | `divergence_free`
* *scope* ::= *RoboChart_component_identifier*
* *bool_stmt* ::= `always` | `never`

Termination requirements The template for the termination requirements is as follows.

RT-TERM

The property *function*(*scope*) shall *bool_stmt* be true.

* *function* ::= `terminate_untimed` | `terminate_timed` | `terminate`
* *scope* ::= *RoboChart_component_identifier*
* *bool_stmt* ::= `always` | `never`

Constants configuration for reward/probability/temporal templates. The placeholder *constants_configs* is used in RT-RWD, RT-PROB, and RT-TEMP. It specifies how constants are assigned for PRISM-based verification. Its grammar is as follows.

Constants configuration (for RT-RWD/RT-PROB/RT-TEMP)

constants_configs ::= [*constants*] [`and`] [*multi_constants*]
constants ::= *const_config* { `and` *const_config* }
const_config ::= `constant_`*RoboChart_constant* `set to` *Expr*
multi_constants ::= *const_multi_config* { `and` *const_multi_config* }
const_multi_config ::= `multi_constant_x from set` *Expr*

Probability requirements. These requirements request that the probability of satisfying a path property shall reach a given bound under fixed constants. The template (RT-PROB) is as follows.

RT-PROB

When *constants_configs* is true,
Until *path_formula* becomes true,
prob_target shall also be true.

* *constants_configs*: see above for the grammar
* *path_formula* ::= `pathFormula_`*pExpr*
* *prob_target* ::= `prob_target_`*op Expr*
* *op* ::= `>` | `>=` | `<` | `<=` | `==`

Temporal requirements. The temporal requirements specify qualitative temporal properties in PRISM, with no numeric bounds. The template (RT-TEMP) is as follows.

RT-TEMP

When *constants_configs* is true,
term_op path_formula shall also be true.

* *constants_configs*: see above for the grammar
* *term_op* ::= `term_`[`not`](`forall` | `exists`)
* *path_formula* ::= `pathFormula_`*pExpr*

A.2 Assertion templates

This appendix provides the additional templates for the assertion templates not covered in Sect. 6.1, namely AT-UTL, AT-REACH, AT-DIV, AT-TERM, AT-PROB, and AT-TEMP.

Untimed refinement assertion template for local invariants (AT-UTL). AT-UTL is used when the requirement applies only when a state machine is in a particular state.

AT-UTL:

untimed csp Spec csp-begin
Spec = CHAOS(Events) [|{*guard_event_1*}|>
  LocalBehaviour

LocalBehaviour = CHAOS(Events)[|{*guard_event_2*}|>
  (RUN({*guard_event_2*})/\(*required_event* -> Spec))
csp-end

untimed csp Spec_impl **associated to** *module_name*
csp-begin
Spec_impl = *module_name*
csp-end

untimed assertion A_*Claim_ID* :
Spec_impl **refines** Spec **in the traces model**

In the specification section, AT-UTL follows a structure similar to AT-UTG, using the CSP_M exception operator [| |> together with the process CHAOS(A) to introduce a scope, and the interrupt operator /\ inside the local behaviour. The principal difference from AT-UTG is the introduction of an explicit scope, denoted by *guard_event_1*. This event acts as a scope delimiter, restricting the requirement to apply when the state machine is in a particular state. For example, the condition that a state machine is in a particular state s_1 can be modelled using the event module::currentState.s1, where currentState is an enumeration over the states of the machine.

Within this scope, the process *LocalBehaviour* enforces the same safety pattern as in AT-UTG, but relative to *guard_event_2* (the local guard): once *guard_event_2* occurs, only further *guard_event_2* events may appear until the next *required_event*; when *required_event* occurs, the behaviour recurs by returning to *Spec*. Note that this is a safety property and does not enforce that *required_event* must eventually occur; infinite continuations of *guard_event_2* are permitted.

Reachability, divergence-free, termination Now we introduce the three assertion templates for reachability, divergence-free, and termination.

AT-REACH:

(**untimed** | **timed**)? **assertion** A_*Claim_ID*:
state bool_term **reachable in** *scope*.

∗ *bool_term* ::= **is** | **is not**

AT-DIV:

(**untimed** | **timed**)? **assertion** A_*Claim_ID*:
scope bool_term **divergence-free**.

∗ *bool_termd* ::= **is** | **is not**

AT-TERM:

(**untimed** | **timed**)? **assertion** A_*Claim_ID*:
scope bool_term.

∗ *bool_term* ::= **terminates** | **does not terminate**

In the above templates, the placeholders *state* and *scope* will be replaced with the elements of the same placeholders from the corresponding requirements, which are structured using RT-REACH, RT-DIV, and RT-TERM.

Probability assertion AT-PROB "Probability assertion template" is designed as follows:

AT-PROB:

(**const** *cname*: *type*)∗

prob property P_*Claim_ID*:
Prob *prob_target* **of** [*path_formula*]
(**with constant**
(*constant_i*)∗ ,
(*multi_constant_j*)∗)∗

The first line of the template declares any constants that the assertion may depend on. The assertion section begins with the keywords **prob property**. The assertion includes the property name and content, which verifies whether

the probability of *path_formula* satisfies the quantitative bound specified in *prob_target*. The identification of the assertion, *P_Claim_ID*, is defined based on the claim identification in AC models, ensuring traceability between the assertions and AC models.

The second section 'with constant' is used for the constant configuration. *constant_i* is employed to assign a single value to the constant, while *multi_constant_j* is used to assign a set of values to the constant.

The placeholders within AT-PROB will be substituted with elements of the corresponding type found in requirements written using RT-PROB.

Temporal assertion AT-TEMP "Temporal assertion template" features two sections similar to AT-PROB. These assertions assess whether the property specified in the *path_formula* is satisfied by either all paths or some paths. The placeholders *temporal_operator* can have the values of *forall* or *exists*. Both placeholders will be substituted with the corresponding elements found in the requirements documented in RT-TEMP.

AT-TEMP:

prob property T_*Claim_ID*:
temporal_operator [*path_formula*]
(**with constant**
(*constant_i*)∗ ,
(*multi_constant_j*)∗)∗

The assertion templates and the requirement templates utilise the identical set of placeholders, which simplifies the process of generating assertions by instantiating the assertion placeholders with the content from the requirement placeholders.

A.3 Assertion generation algorithms

This appendix presents six auxiliary algorithms that instantiate assertion templates from structured requirements. They implement the mapping rules described in Sect. 6.1 and are invoked as helper functions by the integrated procedure in Algorithm 1. Since their behaviour is subsumed by the correctness proof of the integrated procedure, their correctness is guaranteed accordingly.

The `when` template has three instantiable sections `GuardCondition`, `UntilCondition`, and `RequiredCon- dition`. Depending on the types of claims, these three sections map to different sets of elements.

Algorithm 3: GeneratePROBAsst

```
1 Procedure GeneratePROBAsst(r, claim_ID):
2     prob_target ← r.RequiredCondition.removePrefix()
3     path_formula ← r.UntilCondition.removePrefix()
4     InstProbAsst(claim_ID, prob_target, path_formula)
```

Algorithm 3 handles probability claims, identified by the prefix "prob_target" in the *RequiredCondition*. The algorithm extracts three elements of *prob_target*, *path_formula*, and *constant_i* by de-prefixing the corresponding sections of the requirement. These are then used to instantiate template AT-PROB, partly within this algorithm (lines 2–4) and partly in the main procedure (line 29 of Algorithm 1).

Algorithm 4 is similar to Algorithm 3 but has two more elements "reward_event" and "reward_value" in the assertion template AT-RWD. These two are mapped back to the "GuardCondition" section of the claim.

Algorithm 4: GenerateRewardAsst

```
1 Procedure GenerateRewardAsst(r, claim_ID):
2     reward_target ← r.requiredCondition.removePrefix()
3     path_formula ← r.untilCondition.removePrefix()
4     reward_event ←
        r.guardCondition.elems.select(m | m.prefixed("reward_event")).removePrefix()
5     reward_value ←
        r.guardCondition.all.elems.select(m | m.prefixed("reward_value")).removePrefix()
6     InstRewardAsst(claim_ID, reward_target, path_formula, reward_event, reward_value)
```

Algorithm 5 follows the same pattern as the other two presented above to generate assertions using AT-TEMP.

Algorithm 5: GenerateTempAsst

```
1 Procedure GenerateTempAsst(r, claim_ID):
2     temporal_operator ← r.requiredCondition.elems.select(m | m.prefixed("term_")).removePrefix()
3     path_formula ← r.requiredCondition.elems.select(m | m.prefixed("path_formula")).removePrefix()
4     InstTempAsst(claim_ID, temporal_operator, path_formula)
```

Algorithm 6: GenerateUntimedAsst

```
1  Procedure GenerateUntimedAsst(r, claim_ID):
2      if r.guardCondition.elem1Defined() then
3          guard_event_1 ← r.guardCondition.elems.at(0)
4          guard_event_2 ← r.guardCondition.elems.at(1)
5          required_event ← r.requiredCondition
6          required_event_wo ← required_event.removeValue()
7          instLocUntimedAsst(claim_ID,
               guard_event_1, guard_event_2, required_event, required_event_wo)
8      end
9      else
10         guard_event ← r.guardCondition
11         required_event ← r.requiredCondition
12         required_event_wo ← required_event.removeValue()
13         instGlbUntimedAsst(claim_ID, guard_event, required_event, required_event_wo)
14     end
```

Algorithm 6 generates the assertions related to untimed refinement, using the assertion template AT-UTG for the global invariant type, AT-UTL for the local invariant type. The algorithm first decides whether the requirement is of a global or local type. If there are two elements defined in the "GuardCondition" section of the requirement (line 2), the claim is of type "local invariant". Then, the first element of "GuardCondition" is the scope constraint mapped to "guard_event_1" in the assertion template, the second is the local guard condition mapped to "guard_event_2", and "required_event" in the assertion is assigned the value of "requiredCondition" (lines 3-5).

Next, the assertion is generated by instantiating the template AT-2 (line 10). If there is one element defined in "GuardCondition" (line 12), the requirement is a global type, and the assertion is generated by instantiating AT-1 (line 16).

Algorithms 7 and 8 also follow the same pattern: fetching the elements to instantiate template. Algorithm 8 is for general assertions: the claim uses the **every** Kapture template whose sections of "Condition" and "AlwaysOrNever" are used to identify and generate the elements of the assertion template.

Algorithm 7: GenerateTimedAsst

```
1 Procedure GenerateTimedAsst(r, claim_ID):
2     target_event ← r.requiredCondition
3     trigger_event ← r.triggerCondition
4     deadline ← r.duration.getTimeLimit()
5     instTimedAsst(claim_ID, target_event, trigger_event, deadline)
```

A.4 Assertions generated for unscrew robot

This appendix shows the Kapture requirements for the unscrew robot case study listed in Sect. 8.2.4 and the corresponding assertions automatically generated by our toolchain. SR1 is an untimed requirement that fits in the template RT-UNTIMED. The general requirements SR2-SR4 fit in the templates RT-REACH/DDLK/DIV. Their Kapture requirements are shown in Fig. 16-19.

Algorithm 8: GENERATEGENERALASST

```
Procedure GENERATEGENERALASST(r, claim_ID):
    asst_type ← r.condition.getType()
    asst_time_feature ← asst_type.time
    if asst_type = reachable then
        scope ← r.condition.getScope()
        state ← r.condition.getState()
        bool_term ← r.alwaysOrNever.generateBool()
        instReachAsst(claim_ID, scope, state, bool_term, asst_time_feature)
    else
        if asst_type = deadlock_free then
            scope ← r.condition.getScope()
            bool_term ← r.alwaysOrNever.generateBool()
            instDdlkfAsst(claim_ID, scope, bool_term, asst_time_feature)
        else
            if asst_type = divergence_free then
                scope ← r.condition.getScope()
                bool_term ← r.alwaysOrNever.generateBool()
                instDivfAsst(claim_ID, scope, bool_term, asst_time_feature)
            else
                if asst_type = terminate then
                    scope ← r.condition.getScope()
                    bool_term ← r.alwaysOrNever.generateBool()
                    instTmntAsst(claim_ID, scope, bool_term, asst_time_feature)
                end
            end
        end
    end
```

D-RisQ/UC/SRS/1.00: knowledge phase always responds

When
signal Adaptation_Knowledge::Adaptation_Knowledge::get_images?io__
occurs
signal Adaptation_Knowledge::Adaptation_Knowledge::images?io__?x__
shall also be seen.

Derived Requirement:	no
Safety Related:	yes
Security Related:	no
Verify by:	Other

Fig. 16: DTI_SR1 requirement in Kapture template

D-RisQ/UC/SRS/2.00: MakePlan Reachable

The property
reachable(constant Adaptation_Plan::Adaptation_Plan::MakePlan, constant Adaptation_Plan::Adaptation_Plan)
shall always be true.

Derived Requirement:	no
Safety Related:	yes
Security Related:	no
Verify by:	Other

Fig. 17: DTI_SR2 requirement in Kapture template

D-RisQ/UC/SRS/3.00: Plan phase is deadlock-free

The property
deadlock_free(constant Adaptation_Plan::Adaptation_Plan)
shall always be true.

Derived Requirement:	no
Safety Related:	yes
Security Related:	no
Verify by:	Other

Fig. 18: DTI_SR3 requirement in Kapture template

D-RisQ/UC/SRS/4.00: Plan phase is divergence-free

The property
divergence_free(constant Adaptation_Plan::Adaptation_Plan)
shall always be true.

Derived Requirement:	no
Safety Related:	yes
Security Related:	no
Verify by:	Other

Fig. 19: DTI_SR4 requirement in Kapture template

The assertions automatically generated from the four Kapture requirements are as follows.

untimed csp *Spec* csp-begin
Spec = CHAOS(Events) [|
{| *Adaptation_Knowledge::Adaptation_Knowledge::*
get_image.in |} |>
(RUN(
{| *Adaptation_Knowledge::Adaptation_Knowledge::*
get_images.in |}) /\(*Adaptation_Knowledge::Adaptation_Knowledge::*
images.out?x__ -> *Spec*))
csp-end

untimed csp *Spec*_impl **associated to**
Adaptation_Knowledge::Adaptation_Knowledge csp-begin
*spec*_impl =
Adaptation_Knowledge::Adaptation_Knowledge::O__(0)
csp-end

untimed assertion A_*DTI-1* :
*Spec*_impl **refines** *Spec* **in the traces model**

assertion A_*DTI-2*:
Adaptation_Plan::Adaptation_Plan::MakePlan is **reachable in** *Adaptation_Plan::Adaptation_Plan.*

assertion A_*DTI-3*:
Adaptation_Plan::Adaptation_Plan is **deadlock-free**.

assertion A_*DTI-4*:
Adaptation_Plan::Adaptation_Plan is **divergence-free**.

References

1. Standard, D.: Standard 00-55 part 1 (1997). Requirements for safety related software in defence equipment, Part **1** (1997)

2. ISO: ISO 26262 Road vehicles–Functional Safety, Version 1 (2011)
3. Sorokin, L., Bouchekir, R., Beyene, T.A., Liao, B.H.C., Molin, A.: Towards continuous assurance case creation for ads with the evidential tool bus. In: European Dependable Computing Conference, pp. 49–61. Springer (2024)
4. Meng, B., Paul, S., Moitra, A., Siu, K., Durling, M.: Automating the assembly of security assurance case fragments. In: International Conference on Computer Safety, Reliability, and Security, pp. 101–114. Springer (2021)
5. Odu, O., Belle, A.B., Wang, S., Kpodjedo, S., Lethbridge, T.C., Hemmati, H.: Automatic instantiation of assurance cases from patterns using large language models. Journal of Systems and Software **222**, 112,353 (2025)
6. Calinescu, R., Weyns, D., Gerasimou, S., Iftikhar, M.U., Habli, I., Kelly, T.: Engineering trustworthy self-adaptive software with dynamic assurance cases. IEEE Transactions on Software Engineering **44**(11), 1039–1069 (2018). DOI 10.1109/TSE.2017.2738640
7. Jee, E., Lee, I., Sokolsky, O.: Assurance cases in model-driven development of the pacemaker software. In: Leveraging Applications of Formal Methods, Verification, and Validation: 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part II 4, pp. 343–356. Springer (2010)
8. Prokhorova, Y., Laibinis, L., Troubitsyna, E.: Facilitating construction of safety cases from formal models in Event-B. Information and Software Technology **60**, 51–76 (2015). DOI 10.1016/j.infsof.2015.01.001
9. Denney, E., Pai, G.: Tool support for assurance case development. Automated Software Engineering **25**(3), 435–499 (2018). DOI 10.1007/s10515-017-0230-5
10. Bourbouh, H., Farrell, M., Mavridou, A., Sljivo, I., Brat, G., Dennis, L.A., Fisher, M.: Integrating Formal Verification and Assurance: An Inspection Rover Case Study. In: NASA Formal Methods Symposium, vol. 12673 LNCS, pp. 53–71. Springer (2021). DOI 10.1007/978-3-030-76384-8_4
11. Bouskela, D., Falcone, A., Garro, A., Jardin, A., Otter, M., Thuy, N., Tundis, A.: Formal requirements modeling for cyber-physical systems engineering: An integrated solution based on form-l and modelica. Requirements Engineering **27**(1), 1–30 (2022)
12. Bolton, M.L., Jiménez, N., van Paassen, M.M., Trujillo, M.: Automatically generating specification properties from task models for the formal verification of human–automation interaction. IEEE Transactions on Human-Machine Systems **44**(5), 561–575 (2014)
13. Lindoso, W., Nogueira, S.C., Domingues, R., Lima, L.: Visual Specification of Properties for Robotic Designs. In: Brazilian Symposium on Formal Methods, pp. 34–52. Springer (2021). DOI 10.1007/978-3-030-92137-8_3
14. Ghosh, S., Elenius, D., Li, W., Lincoln, P., Shankar, N., Steiner, W.: Arsenal: automatic requirements specification extraction from natural language. In: NASA Formal Methods: 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings 8, pp. 41–46. Springer (2016)
15. Zhao, M., Tao, R., Huang, Y., Shi, J., Qin, S., Yang, Y.: Nl2ctl: automatic generation of formal requirements specifications via large language models. In: International Conference on Formal Engineering Methods, pp. 1–17. Springer (2024)
16. Li, J., Tian, M., Zhong, B.: Automatic generation of safety-compliant linear temporal logic via large language model: A self-supervised framework. arXiv preprint arXiv:2503.15840 (2025)
17. Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., Trippel, C.: nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In: International Conference on Computer Aided Verification, pp. 383–396. Springer (2023)
18. Xu, Y., Feng, J., Miao, W.: Learning from failures: Translation of natural language requirements into linear temporal logic with large language models. In: 2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS), pp. 204–215. IEEE (2024)
19. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, pp. 200–204. Springer (2002)
20. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall Series in Computer Science. Prentice-Hall (1998)
21. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J., Woodcock, J.: RoboChart: modelling and verification of the functional behaviour of robotic applications. Software and Systems Modeling **18**(5), 3097–3149 (2019). DOI 10.1007/s10270-018-00710-z. URL https://doi.org/10.1007/s10270-018-00710-z
22. Windsor, M., Cavalcanti, A.L.C.: RoboCert: Property Specification in Robotics. In: A. Riesco, M. Zhang (eds.) International Conference on Formal Engineering Methods, *Lecture Notes in Computer Science*, vol. 13478. Springer (2022)
23. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3—a modern refinement checker for CSP. In: International conference on tools and algorithms for the construction and analysis of systems, pp. 187–201. Springer (2014)
24. Gleirscher, M., Foster, S., Woodcock, J.: New opportunities for integrated formal methods. ACM Computing Surveys (CSUR) **52**(6), 1–36 (2019)
25. Larsen, P.G., Ali, S., Behrens, R., Cavalcanti, A.L.C., Gomes, C., Li, G., Meulenaere, P., Olsen, M.L., Passalis, N., Peyrucain, T., Tapia, J., Tefas, A., Zhang, H.: Robotic safe adaptation in unprecedented situations: the RoboSAPIENS project. Research Directions: Cyber-Physical Systems **2** (2024). DOI 10.1017/cbp.2024.4
26. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
27. Kelly, T.P.: Arguing safety: a systematic approach to managing safety cases. Ph.D. thesis, University of York York, UK (1999)
28. OMG: Goal Structuring Notation Community Standard. Version 3. Tech. rep., OMG (2021). URL https://scsc.uk/r141C:1?t=1
29. Bloomfield, R.E., Bishop, P.G., Jones, C., Froome, P.K.D.: Ascad—adelard safety case development manual. Adelard, 1998. ISBN 0-9533771-0 **5** (1998). https://usermanual.wiki/Document/EUROCONTROL20Safetycasedevelopmentmanual.687924513/html
30. Hawkins, R., Habli, I., Kolovos, D., Paige, R., Kelly, T.: Weaving an Assurance Case from Design: A Model-Based Approach. In: 2015 IEEE 16th International Symposium on High Assurance Systems Engineering, pp. 110–117. IEEE (2015). DOI 10.1109/HASE.2015.25
31. OMG: Structured Assurance Case Metamodel (SACM), Version 2.1 beta (2020)
32. Wei, R., Kelly, T.P., Dai, X., Zhao, S., Hawkins, R.: Model based system assurance using the structured assurance case metamodel. Journal of Systems and Software **154**, 211–233 (2019). DOI 10.1016/j.jss.2019.05.013
33. Miyazawa, A., Ribeiro, P., Ye, K., Cavalcanti, A., Li, W., Woodcock, J., Timmis, J.: RoboChart Reference Manual (2021)
34. Ye, K., Cavalcanti, A., Foster, S., Miyazawa, A., Woodcock, J.: Probabilistic modelling and verification using RoboChart and PRISM. Software and Systems Modeling **21**(2), 667–716 (2022). DOI 10.1007/s10270-021-00916-8
35. Baxter, J., Ribeiro, P., Cavalcanti, A.L.C.: Sound reasoning in tock-CSP. Acta Informatica **59**, 125–162 (2022). DOI 10.1007/s00236-020-00394-3
36. Roscoe, A.W.: Understanding concurrent systems. Springer Science & Business Media (2010)
37. Yan, F., Foster, S.D., Habli, I.: Automated Compositional Verification for Robotic State Machines using Isabelle/HOL. In: 27th International Conference on Engineering of Complex Computer Systems. IEEE (2023)
38. Spivey, M.: The Z-Notation - A Reference Manual. Prentice Hall, Englewood Cliffs, N. J. (1989)
39. Abrial, J.R.: The B-Book: assigning programs to meanings. Cambridge University Press (1996)
40. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J., Woodcock, J.: RoboChart: modelling and verification of the functional behaviour of robotic applications. Software & Systems Modeling **18**(5), 3097–3149 (2019)
41. Hilder, J.A., Owens, N.D., Neal, M.J., Hickey, P.J., Cairns, S.N., Kilgour, D.P., Timmis, J., Tyrrell, A.M.: Chemical detection using the receptor density algorithm. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) **42**(6), 1730–1741 (2012)
42. FDR 4.2 Documentation. https://cocotec.io/fdr/manual/index.html (2020). Online; accessed 04th May, 2022
43. Roscoe, A.W.: The theory and practice of concurrency. Prentice Hall (1998)
44. Kolovos, D.S., Paige, R.F., Polack, F.A.: The Epsilon transformation language. In: International Conference on Theory and Practice of Model Transformations, pp. 46–60. Springer (2008)

45. Yan, F., Foster, S.D., Habli, I., Wei, R.: Model-based generation of hazard-driven arguments and formal verification evidence for assurance cases. In: 10th International Conference on Model-Driven Engineering and Software Development, pp. 252–263. SciTePress (2022)
46. PRISM Lab Session, Part B: Mail Delivery Robot. http://www.prismmodelchecker.org/courses/aims1617/deliveryRobot.php (2017). Online; accessed 5th May, 2022
47. Murray, Y., Sirevåg, M., Ribeiro, P., Anisi, D.A., Mossige, M.: Safety assurance of an industrial robotic control system using hardware/software co-verification. Science of Computer Programming p. 102766 (2022)
48. Yan, F.: Assurance case generation using model-based engineering and formal verification. Ph.D. thesis, University of York (2023)
49. Böhme, S., Nipkow, T.: Sledgehammer: judgement day. In: International Joint Conference on Automated Reasoning, pp. 107–121. Springer (2010)
50. Baxter, J., Van Acker, B., Kristensen, M., Wright, T., Cavalcanti, A., Gomes, C.: Formal architectural patterns for adaptive robotic software. In: International Conference on Fundamental Approaches to Software Engineering, pp. 145–165. Springer Nature Switzerland Cham (2025)
51. Barnett, W., Cavalcanti, A.L.C., Miyazawa, A.: Architectural Modelling for Robotics: RoboArch and the CorteX example. Frontiers of Robotics and AI (2022). DOI https://doi.org/10.3389/frobt.2022.991637
52. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
53. Baxter, J., Cavalcanti, A., Yan, F.: Operational Requirements for Case Studies. RoboSapiens Project Deliverable, University of York (2025). https://drive.google.com/file/d/1TkCEECTpOhxoKph_dM2neyYIyaiydjKa/view
54. Akesson, B., Nasri, M., Nelissen, G., Altmeyer, S., Davis, R.I.: A comprehensive survey of industry practice in real-time systems. Real-Time Systems **58**(3), 358–398 (2022)
55. Foster, S., Nemouchi, Y., O'Halloran, C., Stephenson, K., Tudor, N.: Formal model-based assurance cases in Isabelle/SACM: An autonomous underwater vehicle case study. In: Proceedings of the 8th International Conference on Formal Methods in Software Engineering, pp. 11–21 (2020)
56. Bourbouh, H., Farrell, M., Mavridou, A., Sljivo, I.: Integration and evaluation of the advocate, fret, cocosim, and event-b tools on the inspection rover case study. Tech. rep. (2020)
57. Giannakopoulou, D., Mavridou, A., Rhein, J., Pressburger, T., Schumann, J., Shi, N.: Formal requirements elicitation with FRET. In: International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020) (2020)
58. Mavridou, A., Farrell, M., Vázquez, G., Pressburger, T., Wang, T.E., Calinescu, R., Fisher, M.: Automated formalization of probabilistic requirements from structured natural language. arXiv preprint arXiv:2512.15788 (2025). Preprint
59. Nemouchi, Y., Foster, S., Gleirscher, M., Kelly, T.: Isabelle/SACM: Computer-Assisted Assurance Cases with Integrated Formal Methods. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **11918 LNCS**, 379–398 (2019). DOI 10.1007/978-3-030-34968-4_21
60. Foster, S., Nemouchi, Y., Gleirscher, M., Wei, R., Kelly, T.: Integration of formal proof into unified assurance cases with Isabelle/SACM. Formal Aspects of Computing **33**(6), 855–884 (2021)
61. Wei, R., Jiang, Z., Mei, H., Barmpis, K., Foster, S., Kelly, T., Zhuang, Y.: Automated model-based assurance case management using constrained natural language. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **43**(1), 291–304 (2023)
62. Wei, R., Foster, S., Mei, H., Yan, F., Yang, R., Habli, I., O'Halloran, C., Tudor, N., Kelly, T., Nemouchi, Y.: Access: Assurance case centric engineering of safety–critical systems. Journal of Systems and Software **213**, 112,034 (2024)
63. Sljivo, I., Denney, E., Menzies, J.: Guided integration of formal verification in assurance cases. In: International Conference on Formal Engineering Methods, pp. 172–190. Springer (2023)