Formal Model-Based Assurance Cases in Isabelle/SACM

An Autonomous Underwater Vehicle Case Study

Simon Foster
Yakoub Nemouchi
simon.foster@york.ac.uk
yakoub.nemouchi@york.ac.uk
University of York

Colin O'Halloran
Karen Stephenson
Nick Tudor
D-RisQ Software Systems

ABSTRACT

Isabelle/SACM is a tool for automated construction of model-based assurance cases with integrated formal methods, based on the Isabelle proof assistant. Assurance cases show how a system is safe to operate, through a human comprehensible argument demonstrating that the requirements are satisfied, using evidence of various provenances. They are usually required for certification of critical systems, often with evidence that originates from formal methods. Automating assurance cases increases rigour, and helps with maintenance and evolution. In this paper we apply Isabelle/SACM to a fragment of the assurance case for an autonomous underwater vehicle demonstrator. We encode the metric unit system (SI) in Isabelle, to allow modelling requirements and state spaces using physical units. We develop a behavioural model in the graphical RoboChart state machine language, embed the artifacts into Isabelle/SACM, and use it to demonstrate satisfaction of the requirements.

KEYWORDS

Assurance Cases, Verification, Autonomous Systems, Isabelle/HOL

ACM Reference Format:
Simon Foster, Yakoub Nemouchi, Colin O'Halloran, Karen Stephenson, and Nick Tudor. 2020. Formal Model-Based Assurance Cases in Isabelle/SACM: An Autonomous Underwater Vehicle Case Study. In *8th International Conference on Formal Methods in Software Engineering (FormaliSE '20), October 7–8, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3372020.3391559

1 INTRODUCTION

Deployment of autonomous systems into an open operational environment requires a credible argument, underpinned by evidence, for safety; namely an assurance case [27]. Assurance cases demonstrate how hazardous behaviour is mitigated through safety requirements that must be allocated to system components. This often requires refinement and formalisation of the requirements to allow the use of formal methods in providing evidence, particularly for higher assurance levels. The formal requirements provide an implicit system model, which can be refined to an explicit model during the system design phases, and used as an evidential artifact [23]. This process is usually performed in the context of an international certification standard, such as IEC 61508, ISO 26262, or DO-178C.

During refinement and elaboration, it is important to maintain traceability to the original high-level requirements. Otherwise "formalisation gaps" can emerge, where formal models and requirements become detached from the original system specification and therefore dramatically lose value [5, 19, 21]. In particular, when a completed system is evaluated for certification, there must be a clear explication of why design decisions were made, and how they contribute towards satisfying the requirements. In summary, formal models and assurance cases must evolve hand-in-hand [5, 18].

Previously, we have presented Isabelle/SACM [14], an embedding of the Structured Assurance Case Metamodel (SACM) [38] into the Isabelle document model [4, 39]. SACM is a unifying standard for assurance cases to express argumentation, artifact traceability, and terminology [38]. Our implementation provides a machine-checked interactive and hyperlinked assurance language that allows the use of multifarious modelling and verification techniques, provided by Isabelle, to provide evidential artifacts. Due to its extensible and modular architecture, Isabelle is an ideal platform for integrating formal methods [39]. It supports tools for both high level modelling [12], including concurrent [13], hybrid [11, 31], and probabilistic systems [40], and also code verification [20, 37].

At the same time, Isabelle's automated asynchronous document processing [39] supports assurance case evolution, whereby changes to system parameters, models, requirements, and so on, trigger re-checking of the impacted document sections. This means, for example, we can tweak model elements, such as physical dimensions, and see the effect on any verification results. Moreover, Isabelle's document preparation system can be used to generate a hyperlinked PDF that is correct-by-construction, and with all artifacts presented in a human-readable way, possibly for delivery to a regulator [5].

The contribution of this paper is application of Isabelle/SACM to a novel assurance case for the safety controller of an autonomous underwater vehicle (AUV). The safety controller, called the "Last

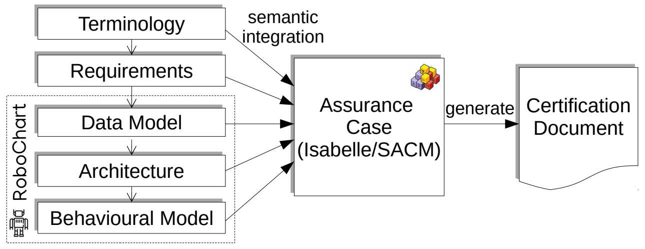

Figure 1: Formalised Assurance Cases

Response Engine" (LRE), acts as a run-time monitor that enforces operational rules, and can engage collision avoidance. It is currently being developed by D-RisQ[1] under the regime of DO-178C. Our end goal is a methodology and tool than can be used to support computer aided certification for a variety of systems.

Our methodology is shown in Figure 1. We give a formal structure to the D-RisQ supplied system requirements document (SRD), starting from terminology definitions, and then use these terms in specifying the requirements in marked up natural language, for which we also extend Isabelle/SACM. We then develop formal data, architectural, and behavioural models for the AUV controller, and use Isabelle/SACM to show how each requirement is mapped to a transition, state, or other model element. Our paper provides evidence that Isabelle can support model-based assurance of critical systems [5, 14] supported by integrated formal analysis tools.

For the data model, we develop a Z-inspired notation [36] with state variables and invariants. The AUV is a cyber-physical system, and so the requirements contain physical quantities, such as its maximum velocity. Consequently, we develop a novel embedding of the SI unit system in Isabelle/HOL, to ensure consistent use of units, and also enable dimension analysis, which allows higher rigour than if this information was omitted. Moreover, the embedding provides access to versatile libraries for Multivariate Analysis [22] and Differential Equations [25, 26], which can be used to symbolically and precisely reason about the AUV's dynamics and geometry.

For modelling the behaviour, we use the RoboChart language [28, 29], which provides block-based architectural and state machine modelling notations. The core of the language is a formalised subset of UML state machines. RoboChart makes the approach accessible to practitioners with little knowledge of formal methods.

RoboChart has a denotational semantics in Hoare's CSP process algebra [2, 28], which allows verification using the FDR refinement checker[2]. However, since the LRE model depends on real numbers and transcendental functions, the state space is uncountable, and so cannot be directly model checked. Previously, we have also embedded a formal semantics for state machines into Isabelle [12] using our mechanisation of Hoare and He's Unifying Theories of Programming (UTP) [24] semantic framework, Isabelle/UTP [16]. The use of UTP allows us to give a unifying semantics to the various formal notations we use for modelling the LRE. We use our state machine implementation to support traceability in Isabelle/SACM, as well as formal verification through theorem proving, which is symbolic and so overcomes the state explosion problem.

The structure of our paper is as follows. In §2 we introduce the LRE, and key requirements. In §3 we introduce Isabelle, and our SACM implementation. In §4, we develop the LRE data model by mechanising the SI unit system, and then using this to describe the state space. In §5 we model the architecture and behaviour of the LRE using RoboChart. We show how we can formally verify the model, and also link the various elements back to the LRE requirements. Finally, in §6 we conclude and highlight related work.

All definitions and theorems in this paper are mechanised in Isabelle[3]. We present them both using screenshots directly from the tool, and also sometimes mathematically for conciseness.

[1]D-RisQ Software Systems. http://www.drisq.com/.
[2]FDR: The CSP Refinement Checker. https://www.cs.ox.ac.uk/projects/fdr/index.html
[3]Supporting materials: https://doi.org/10.5281/zenodo.3739235

2 AUV CASE STUDY

The AUV is a portable (< 200 kg) untethered Remotely Operated Vehicle, equipped with a visual mapping system and verified on-board autonomy. The aim is to make it capable of conducting light intervention tasks, such as, cathodic protection surveys (oil and gas) and simple coring (offshore), with potential to move to more complex interventions in a later phase, such as valve turning. The project brings together the UK expertise from: the National Oceanography Centre and Forth Engineering in Underwater Robotic Development; ROVCO on subsea operation, sensor development and subsea vision perception; the University of Manchester in mixed mode underwater communications; and D-RisQ in Software Verification.

The National Oceanography Centre engages with regulators through their ongoing contribution to the Marine Autonomous Systems regulatory working group to ensure regulatory compliance. To this end the use of a structured assurance case is vital to communicate the evidence of safe operation to non-specialists, especially in the aspect of software controlled autonomous behaviour. In this paper, we focus on development of formal models and an assurance case for the "Last Response Engine" (LRE), which provides run-time safety assurance. We consider the special case of the AUV navigating within an enclosed pond to perform maintenance tasks.

Architecturally, the LRE sits between the operator and autopilot components. The operator, which can be a human or the navigation system, provides instructions to the LRE to support execution of tasks, such as requesting a particular heading and velocity. The autopilot controls the AUV actuators, and takes advice only from the LRE. The LRE's job is to avoid hazardous behaviours, such as getting too close to an obstacle, or entering "object proximity exclusion zones" (OPEZ), and engaging evasive manoeuvres if necessary.

The LRE functions in four modes: Operator Control Mode (OCM), Main Operating Mode (MOM), High Caution Mode (HCM), and Collision Avoidance Mode (CAM). Whilst in OCM, the LRE is effectively inactive, and simply passes through control inputs to the autopilot. MOM is where the LRE takes control for normal behaviour at maximum speed. HCM is for the situation when the AUV is getting close to an obstacle, and so the LRE drops the velocity. Finally, CAM is the mode where a potential collision has been detected, and the AUV is manoeuvring away from the obstacle. The LRE has several high-level safety requirements; here we focus on a small subset:

- R1 The physical dimensions of the AUV shall be $457\,\text{mm} \times 338\,\text{mm} \times 254\,\text{mm}$.
- R2 When, and only when, in OCM, the LRE shall accept operator control inputs and send them to the autopilot.
- R3 The LRE shall enter OCM: (i) when the AUV powers up; (ii) at the end of a task; or (iii) when the operator requests.
- R4 The LRE shall enter MOM from OCM when the following conditions hold: (i) the velocity is less than $0.1\,\text{ms}^{-1}$; (ii) the distance to a static obstacle is $> 300\,\text{mm}$; (iii) the distance to a dynamic obstacle is greater than $7500\,\text{mm}$; (iv) the operator requests it; and (v) the AUV is not in an OPEZ.
- R5 On entering MOM, the LRE shall advise a velocity of $1\,\text{ms}^{-1}$.
- R6 The LRE shall enter HCM from MOM when either:
  - (a) the AUV has a horizontal velocity $> 0.1\,\text{ms}^{-1}$ and is within *StaticObsHorizDist* horizontal distance of a static obstacle;

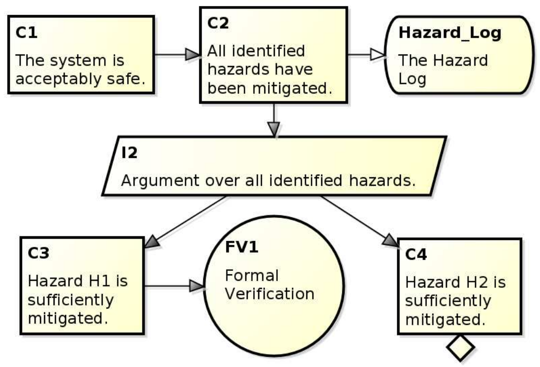
C1
The system is acceptably safe.
C2
All identified hazards have been mitigated.
Hazard_Log
The Hazard Log
I2
Argument over all identified hazards.
C3
Hazard H1 is sufficiently mitigated.
FV1
Formal Verification
C4
Hazard H2 is sufficiently mitigated.

Figure 2: Assurance Cases in Goal Structuring Notation

(b) the AUV has a vertical velocity $> 0.1\,\mathrm{ms}^{-1}$ and is within *StaticObsVertDist* vertical distance of a static obstacle;
(c) the AUV is within *StaticObsDfltVertDist* of a static obstacle;
(d) the operator requests it.

R7 On entering HCM, the LRE shall advise a velocity of $0.1\,\mathrm{ms}^{-1}$.
R8 The LRE shall exit HCM and enter MOM when the AUV is $>$ *StaticObsHorizDist* horizontal distance and $>$ *StaticObsVertDist* vertical distance of a static obstacle.
R9 The LRE shall enter CAM if (i) it is not in OCM and (ii) there is an obstacle with an unsafe trajectory.
R10 The LRE shall never enter a deadlock state.

These requirements depend on a number of constants, including *StaticObsHorizDist* and *LREHorizon*, and several defined terms. Part of requirement satisfaction is to disambiguate and formalise each of these concepts. Consider, for example, that two coordinate systems are used in the requirements: R4 and R5 refer to the AUV "velocity", which corresponds to the overall spherical velocity of the AUV, including its depth component. In contrast, R6 refers to "horizontal" and "vertical" velocities, which are velocity across the ground, and the depth velocity. These distinctions are important for the formalisation. We also emphasise the heavy use of physical quantities, such as in R1, which is used in calculation of safety margins.

3 BACKGROUND: ISABELLE AND SACM

Here, we introduce the Isabelle tool [39], and our implementation of the Structured Assurance Case Metamodel (SACM) [14, 38].

Isabelle/HOL is a proof assistant for Higher Order Logic (HOL). It supports the mechanisation of mathematical theories through a functional specification language and (semi-)automated proof. Isabelle is more than a proof assistant though: it has an extensible architecture with an executable frontend document model and a modular backend supporting an array of formal analysis tools [39]. For this reason, an analogy can be drawn between Isabelle and IDEs like Eclipse. Indeed, Isabelle can provide a frontend and verification facilities for a variety of program languages [1], such as C [37].

Isabelle documents have outer- and inner-syntax levels. Outer-syntax consists of a sequence of commands, using predefined keywords and parsers, that construct and query previously specified formal entities. Each command is implemented in the ML-based backend, which can manipulate a database of axiomatic and definitional entities. For example, the HOL document model includes commands like **datatype**, **function**, and **theorem**, that construct,

```
Claim C1 ‹ The system is acceptably safe. ›

Claim C2 ‹ All identified hazards have
  been sufficiently mitigated. ›

Inference I1 src ‹{@{Claim C2}}› tgt ‹{@{Claim C1}}›
  ‹ @{Claim C1} is supported by @{Claim C2}. ›

Context ac1
  src ‹{@{Artifact Hazard_Log}}› tgt ‹{@{Claim C2}}›
  ‹ The Hazard Log @{HazardLog Hazard_Log}
    is context for @{Claim C2} ›

Claim C3 ‹ @{Hazard H1} is sufficiently mitigated. ›

Evidence E1
  src ‹{@{Artifact FV1}}› tgt "{@{Claim C3}}"
  ‹ Formal verification result @{Artifact FV1}
    is evidence for @{Claim C3}. ›

Claim C4 needsSupport
  ‹ @{Hazard H2} is sufficiently mitigated. ›
```

Figure 3: Assurance Cases in Isabelle/SACM

respectively, algebraic datatypes, recursive functions, and conjectured theorems. New commands can be defined by specifying a keyword, and developing ML code to parse and process the inputs. The top-level command is **theory**, which opens a new named theory context, and can import existing theories. Processing of documents is dynamic, such that edits trigger real-time rechecking of the changed artifacts and any dependants.

Each command can take as parameters formal terms of the logic, which are often enclosed in quotation marks (‹ · · · ›), and referred to as inner-syntax. Inner-syntax has separate parsing, processing, and type-checking layers, which construct terms that must be certified against the axiomatic core. The parser supports unicode characters and prioritised mixfix operator notations. Certified terms can be subjected to various analyses, such as decomposition using theorems to support proof. Consider the following example command:

theorem *t1*: ‹$x + 1 > x$›

This proposes a thereom, using the outer-syntax command **theorem** with the assigned name *t*1. The content is the inner-syntax term, $x + 1 > x$, enclosed by ‹ · · · ›. In the backend, Isabelle first parses the outer-syntax command, and then moves on to parse the term. The operators (+) and (>) must all exist in the theory context, with appropriate syntax defined. Provided this is the case, and the term type checks, Isabelle will construct a formal term which can then be subjected to proof and other analysis.

Commands also exist for structured informal content, such as **text** ‹ · · · ›, that can be used to intersperse formal entities with commentary mark up. The backend can then render a PDF or HTML presentation of the theory content. The content of each **text** command can also contain hyperlinks to previously defined entities using the so-called "antiquotation" notation, @{**cmd** *param*}, where *cmd* is the antiquotation command, and *param* is a parameter. Antiquotation commands include **const** for defined constants, **term** for Isabelle terms, **thm** for theorems, and **typ** for types. For example, if we have proved the theorem $t1$ we can add the text

text ⟨ We make use of @{**thm** $t1$} in the theorem below. ⟩

This checks that $t1$ exists and has been proved, and if so inserts a hyperlink. If an antiquotation reference is invalid, if for example a theorem does not exist or is unproven, then Isabelle raises an error message. We have seen that an Isabelle document also overlays a directed acyclic graph of such linked entities. For structuring of documents, there are also commands like **section** and **subsection**.

We use the outer-syntax layer to develop an embedding of an SACM interactive assurance language [14]. A structured assurance case is typically modelled as a graph of **claims**, that are decomposed into further claims — through argumentation strategies and with reference to defined context and assumptions — down to evidential **artifacts** [27]. For example, claims may exist that each requirement is satisfied, supported by evidence like test reports and formal analysis. The claims are linked together through inferences that assert that certain claims follow from others. Evidence and contextual elements can be presented using SACM artifacts, which can be used as leaf nodes in an argument. The content of claims can be natural language, but can also use formally defined **terms**. Assurance cases are often presented using a notation like GSN (Goal Structuring Notation) [27]; a small example is shown in Figure 2.

Isabelle/SACM uses an Isabelle plugin called DOF (Document Ontology Framework) [3–5], to develop an ontology for SACM. DOF provides an enriched version of the **text** command of the form

$$\textbf{text}^{*}[x : c, a_1 = v_2, \cdots a_n = v_n]\langle \cdots \rangle$$

which creates an instance x of a predefined document class c, and assigns a value to each of the n attributes in c. DOF also provides a command **doc_class** $c \triangleq a_1 : t_1 \cdots a_n : t_n$, to create new document classes with a name and set of attributes. We extend the document model with classes for the various SACM entities, and commands for defining instances, such as **Claim**, **Artifact**, and **Inference**. Moreover, every SACM class has a corresponding antiquotation, provided by DOF, to support links in the assurance case graph. The meta-model instances can be used to analyse the assurance case structure, for example, to check that every claim is supported.

An example assurance case fragment is shown in Figure 3, that corresponds to the GSN in Figure 2. It has a top-level claim, *C1*, claiming that a system is acceptably safe. The means to show this is that all identified hazards have been mitigated, which is claim *C2*. An inference shows that *C1* is supported by *C2*, by assigning these as the target (**tgt**) and source (**src**), respectively. The identified hazards are given by the context element *ac1* that uses a predefined hazard log as context for *C2*. Claim *C3* then is made that one of the identified hazards, *H1*, is mitigated, which is evidenced by a predefined formal verification result, *FV1*, which in this case is a theorem proved in Isabelle/HOL. Claim *C4*, asserting that a second hazard, *H2*, is mitigated is yet to be developed into an argument, and so it is marked with the keyword **needsSupport**.

Figure 4 shows the definition of several AUV terms and acronyms in Isabelle/SACM, some of which refer to other terms using antiquotations. Each **Definition** command adds a new term into the theory context. The list of terms can be used to provide a glossary for the certification document output. We particularly draw attention to the definition of OPEZ, which refers to several Isabelle terms to

```
Definition AUV ‹ Autonomous Underwater Vehicle ›

Definition CPA ‹ Closes Point of Approach: a point
  where the closest approach will be. ›

Definition CDA ‹ Closest Distance of Approach:
  separation at the @{Term CPA}. ›

Definition Collision ‹ An event where the @{Term AUV}
  danger area and the obstacle danger area intersects. ›

Definition OPEZ ‹ ❙‹Object Proximity Exclusion Zone.›
  There are regions of the pond where the @{Term AUV}
  should not operate autonomously. These regions are
  referred to as Object Proximity Exclusion Zones (OPEZ).
  These regions include:
  ▪ the surface to a depth of @{term "300·milli·meter"}
  ▪ @{term "300·milli·meter"} from pond walls
  ▪ @{term "300·milli·meter"} from all other objects. ›
```

Figure 4: Nomenclature in Isabelle/SACM

```
subsection ‹ Certification ›

Requirement DQ_AUV_LRE_SRD_010_00 ‹ ❙‹DO-178C›
  The software for the @{Term LRE} shall be developed using
  @{Artifact DO_178C} and associated documentation. ›

text ‹ Comment: @{Artifact DO_178C} is the de facto standard
  for aerospace development. There are associated documents
  such as the Formal Methods Supplement @{Artifact DO_333}
  and the Tool Qualification Supplement @{Artifact DO_330}
  that will also be used. ›

subsection ‹ Timing ›

abbreviation "MAX_LRE_EXEC_TIME ≡ 50·milli·second"

Requirement DQ_AUV_LRE_SRD_020_00 ‹ ❙‹Execution Time›
  The maximum execution time for the @{Term LRE}
  shall be less than @{const MAX_LRE_EXEC_TIME}. ›
```

Figure 5: LRE Requirements in Isabelle/SACM

specify key dimensions, including their SI units, each of which is parsed and type-checked before being accepted into the document.

Similarly, in Figure 5 we specify two requirements for the LRE in two subsections. For this, we have created the **Requirement** command, that creates an SACM artifact. The first requirement states that the LRE shall be developed according to DO-178C, with an explanatory note. The second sets a timing upper bound of 50ms. This second requirement uses a HOL constant, MAX_LRE_EXEC_TIME, that carries an SI quantity. The DOF document preparation system can render each of the terms, requirements, and other artifacts in a PDF. Requirements are rendered, for example, as follows:

> **Requirement 2.** ***Execution Time.*** The maximum execution time for the LRE shall be less than 50·*milli*·*second*.

with a hyperlink for "LRE" pointing to the location of its definition.

In the next section we develop the data model for the LRE, which includes an mechanisation of the SI unit system.

4 DATA MODEL

Here, we describe the AUV data model, including SI unit types, and state spaces with invariants. The results from this section support the LRE architectural model, and requirement formalisation.

4.1 SI Units and Quantities in Isabelle

The International System of Units (SI)[4] is a standard for expression of physical quantities. It defines seven base units – including meters, seconds, kilograms, and amperes – that can be combined to derive all the quantities required in science and engineering. Typically in a computer program or system model, quantities are represented only by a numeric value, such as an integer (*int*) or real number (*real*). However, for specifying a physical system, such as the AUV, it is important to both represent the units, and use them consistently.

We embed SI units into the Isabelle type system through a type of the form $n[\boldsymbol{u}]$, where n is a numeric type, such as *int*, *rat*, and *real*, and $\boldsymbol{u}$ is a unit type. This allows us to enforce consistent use: for example, $x + y$ and $x - y$ are well-formed only when x and y have both the same numeric type and unit. However, implementing the SI unit system at the type level in Isabelle has some challenges; notably Isabelle lacks dependent types and so units must be types rather than values. This issue has previously been overcome in a Haskell SI implementation [32], from which we take inspiration.

Isabelle's type system supports parametric polymorphism, where types carry parameters, such as α *list*, where α can be instantiated with a type like *int*. It also supports overloading using the type class mechanism, where functions and their properties can be polymorphic, and instantiated for concrete types. For example, the *semigroup* class characterises a polymorphic function $\_ \cdot \_ : \alpha \to \alpha \to \alpha$, and the associativity property $x \cdot (y \cdot z) = (x \cdot y) \cdot z$. We can then instantiate this with *int*, using $\cdot \triangleq +$, by proving that $+$ is associative.

Our approach uses the type class mechanism to characterise a family of types that correspond to unit tags, which have a singleton carrier set and exist only for type annotation. Every unit tag associates to a particular SI unit, which we encode through a data structure. We therefore encode the SI system at both the value- and type-level. We begin with datatypes to describe value-level units:

$$\textbf{datatype}\ \mathit{SIBase} \triangleq \mathit{Second} \mid \mathit{Meter} \mid \mathit{Kilogram} \mid \mathit{Ampere} \mid \mathit{Kelvin} \mid \mathit{Mole} \mid \mathit{Candela}$$

$$\textbf{type-synonym}\ \mathit{SIUnit} \triangleq (\mathit{SIBase} \to \mathit{int})$$

SIBase enumerates the seven base units. A derived *SIUnit* is a total function from *SIBase* to *int*, which defines a power for base unit. For example, ms^{-1} is encoded with a function that assigns 1 to *Meter*, -1 to *Second*, and 0 to every other unit. We characterise each of the base units as the subset of $\mathit{SIBase} \to \mathit{int}$ where precisely one domain element is assigned 1, and so define *seconds*, *meters*, *kilograms*, ... : *SIUnit* for every such case. We define arithmetic operators for units:

$$\begin{aligned} 1 &\triangleq (\lambda\, b \bullet 0) \\ u_1 \cdot u_2 &\triangleq (\lambda\, b \bullet u_1(b) + u_2(b)) \\ u^{-1} &\triangleq (\lambda\, b \bullet -u(b)) \end{aligned}$$

Here, 1 corresponds to the dimensionless unit, where every base unit has power 0. The product, $u_1 \cdot u_2$ sums up all of the base unit powers, and the inverse negates all of the powers. Together with the base units, we can use these operators to algebraically specify any derived unit, for example $\mathit{meters} \cdot \mathit{seconds}^{-1}$. Moreover, we can prove the following theorem demonstrating the algebraic properties:

[4]SI Brochure, 9th edition. https://www.bipm.org/utils/common/pdf/si-brochure/SI-Brochure-9.pdf. BIPM, 2019.

Theorem 4.1. *($\mathit{SIUnit}, \cdot$) forms an abelian group: $\cdot$ is associative, commutative, and has identity 1. Moreover, for any u : SIUnit, the inverse element is u^{-1}, that is $u \cdot u^{-1} = 1$.*

We also define division, $x/y \triangleq x \cdot y^{-1}$, which satisfies the usual laws. With this algebraic structure, we support dimension analysis in Isabelle; for example, we can prove $\mathit{meter} \cdot \mathit{second}^{-1} \cdot \mathit{second} = \mathit{meter}$.

From this foundation, we characterise type-level units. Though we cannot have an algebraic data structure at the type level, we can effectively declare an isomorphism between a certain set of types and the value-level *SIUnit*. We define a type class, *siunit*, that imposes the following constraints on any member type $\boldsymbol{u}$: (1) its carrier set has a cardinality of 1 (it is a tag, and its members have no meaning), and (2) it associates with an element of *SIUnit* with the function $\mathit{siunit\text{-}of} : \boldsymbol{u} \to \mathit{SIUnit}$. Every type instantiating *siunit* therefore corresponds to a value-level unit. We also define the type class *sibaseunit*, that extends *siunit* and identifies base unit types.

We then define seven unitary tag types for each base unit, and instantiate *sibaseunit* with each of them, such that *siunit-of* returns the corresponding value-level unit. We also define a binary type constructor, $\boldsymbol{u_1} \boldsymbol{\cdot} \boldsymbol{u_2}$, where $\boldsymbol{u_1}$ and $\boldsymbol{u_2}$ are both members of *siunit*, which is type-level product, such that

$$\mathit{siunit\text{-}of}(\boldsymbol{u_1} \boldsymbol{\cdot} \boldsymbol{u_2}) = \mathit{siunit\text{-}of}(\boldsymbol{u_1}) \cdot \mathit{siunit\text{-}of}(\boldsymbol{u_2})$$

Similarly, we define unary type constructor for inverse u^{-1}. With these type constructors, we can also construct every unit at the type-level in Isabelle. However, we cannot easily perform dimension analysis at the type level since every syntactically distinct construction is a unique type: for example $\mathrm{ms}^{-1}\mathrm{s}$ and m are different types. There is no type equality in Isabelle, and so we cannot automatically substitute these values, but must perform coercions, which is another reason for also having the value-level units.

We can now define the type for SI quantities:

$$\textbf{typedef}\ \alpha[\boldsymbol{u}] \triangleq \{(x : \alpha, u : \mathit{SIUnit}) \mid \mathit{sitype\text{-}of}(\boldsymbol{u}) = u\}$$

A quantity is parameterised by a numeric type α and a unit type $\boldsymbol{u}$. An element is a pair consisting of a factor x, and an SI value-level unit u that corresponds with the type-level unit $\boldsymbol{u}$; this how the value- and type-level units fit together. We can define the arithmetic operators + and − easily, since they do not change the unit and so simply apply to the factors. Technically, we can also define multiplication, but due to restriction in the type system, it has the monomorphic type $\alpha[\boldsymbol{u}] \to \alpha[\boldsymbol{u}] \to \alpha[\boldsymbol{u}]$ and so does not account for units. However, it is still useful, because it can be applied when multiplying a dimensionless quantity and a unit. Consider the following unitary and prefix quantities:

$$\begin{aligned} \mathit{second} &\triangleq (1 : \alpha[\mathit{second}]) & \mathit{kilo} &\triangleq (1000 : \alpha[\boldsymbol{u}]) \\ \mathit{meter} &\triangleq (1 : \alpha[\mathit{meter}]) & \mathit{centi} &\triangleq (100 : \alpha[\boldsymbol{u}]) \\ \mathit{kilogram} &\triangleq (1 : \alpha[\mathit{kilogram}]) & \mathit{milli} &\triangleq (1/1000 : \alpha[\boldsymbol{u}]) \end{aligned}$$

Here, *second* corresponds to the quantity of 1 second. It is polymorphic in its numeric type, but its unit type is fixed to *second* (note the different namespaces). We also define SI prefixes, including *milli* and *kilo*. These prefixes are dimensionless, or more precisely dimension-polymorphic, since they can possess any unit. Then, using the monomorphic multiplication operator we can construct quantities like $50 \cdot \mathit{milli} \cdot \mathit{second}$, as demonstrated in Figure 5. Both

```
schema AUV_Inp_Par =
  — ‹ Depth below water surface ›
  depth        :: "real[meter]"
  — ‹ North-South Velocity; North is positive ›
  ns_vel       :: "real[meter/second]"
  — ‹ East-West Velocity; East is positive ›
  ew_vel       :: "real[meter/second]"
  — ‹ Up is positive ›
  rate_of_climb :: "real[meter/second]"
where "depth ∈ {-10..0}" "ns_vel ∈ {-5..5}"
      "ew_vel ∈ {-5..5}" "rate_of_climb ∈ {-5..5}"

Requirement DQ_AUV_LRE_SRC_030_00 ‹ ❙‹AUV Input Parameters›
  The @{Term AUV} sensors are formalised by the state space
  @{typ AUV_Inp_Par} and predicate @{const AUV_Inp_Par_inv}. ›
```

Figure 6: AUV Input Parameters Data Model

the numeral 50 and prefix *milli* are dimension polymorphic, but *second* causes the whole expression to have unit type *second*.

For the general case of quantities with specific units, we define specialised arithmetic operators that account for units:

$$qmult : \alpha[\boldsymbol{u_1}] \to \alpha[\boldsymbol{u_2}] \to \alpha[\boldsymbol{u_1} \cdot \boldsymbol{u_2}]$$
$$qsq : \alpha[\boldsymbol{u}] \to \alpha[\boldsymbol{u}^2]$$
$$qinv : \alpha[\boldsymbol{u}] \to \alpha[\boldsymbol{u}^{-1}]$$
$$qsqrt : real[\boldsymbol{u}^2] \to real[\boldsymbol{u}]$$

Function *qmult* multiplies together two quantities, which composes the two units at the type level, and *qinv* takes the inverse. The effect on the underlying factors is the same as the usual operators. We define square root, *qsqrt* since this is needed for calculating distances. Square root can only be defined precisely for real numbers, and so the numeric type is fixed. It can only be applied when the unit has the correct form, such as m^2. We give these four functions the usual syntax: $x \cdot y$, x^2, x^{-1}, and $\sqrt{x}$, respectively. They are used for the distance and velocity calculations in the next section.

4.2 LRE State Space

In order to specify the data model of the LRE, we need state spaces with invariants. Isabelle/HOL does not have a high-level command for constructing these, and so we implement a Z-like schema command [36] for state space types with invariants attached:

$$\textbf{schema}\, S \triangleq x_1 : t_1 \cdots x_n : t_n \,\textbf{where}\, P(x_1, \cdots, x_n)$$

This creates a type S, with n variables, each with an assigned type. An invariant P constrains the variables in a valid state.

The AUV input parameter data model is described using the schema shown in Figure 6. The schema uses our SI quantity type to specify the units for each of the variables. Each variable also has an explanatory textual comment. The inputs include the current depth of the AUV, its horizontal velocity in north-south and east-west components, and the vertical velocity (rate of climb). We model the velocity in this way, as opposed to using an explicit heading angle, because it makes calculation of relative velocities for collision straightforward. We don't include a global position for the AUV, as we need only consider relative distances to obstacles for safety monitoring and collision avoidance. There are also several invariants included in the state schema, which assign permissible ranges to each of the variables. We also encode a requirement stating that the LRE's inputs are described by the variables in the AUV_Inp_Par

```
schema Obstacle =
  — ‹ Relative distance North of AUV ›
  ns_rel_dist :: "real[meter]"
  — ‹ Relative distance East of AUV ›
  ew_rel_dist :: "real[meter]"
  — ‹ Absolute depth below water surface ›
  obs_depth   :: "real[meter]"
  obs_ns_vel  :: "real[meter/second]"
  obs_ew_vel  :: "real[meter/second]"
  obs_roc     :: "real[meter/second]"
where
  "ns_rel_dist ∈ {-50..50}"
  "ew_rel_dist ∈ {-50..50}"
  "obs_depth ∈ {-10..0}" "obs_ns_vel ∈ {-5..5}"
  "obs_ew_vel ∈ {-5..5}" "obs_roc ∈ {-5..5}"

Requirement DQ_AUV_LRE_SRD_050_00 ‹ ❙‹Obstacle Parameters›
  @{typ Obstacle} lists the sensor information that shall
  be provided to the @{Term LRE} for every obstacle. ›

schema Sensors =
  auv :: "AUV_Inp_Par"  obs :: "nat ⇀ Obstacle"
where "finite(dom(obs))"
```

Figure 7: Obstacle Inputs and Sensor Data Model

type. To satisfy this requirement, it is necessary for the AUV to supply these inputs at the LRE interface, with the given ranges.

For tracking obstacles, we specify a further data schema shown in Figure 7. It specifies the sensed information about each obstacle that the AUV is aware of, including the relative distance in north-south and east-west components, and the velocities. As for the LRE input parameters, we also encode a requirement that the AUV can supply this obstacle information to the AUV – effectively we are assuming adequate sensing capabilities. From this schema, we can describe a useful function for distinguishing static obstacles:

Definition 4.2 (Static Obstacles).

$$is\_static(ob : Obstacle) \triangleq (ob{:}obs\_ns\_vel = 0 \wedge ob{:}obs\_ew\_vel = 0)$$

In this context, the syntax $x{:}a$ means selection of attribute a in x. An obstacle is static if both of its velocity components are zero.

Finally, the LRE and obstacle sensed inputs are combined in the type Sensors, which describes the complete set of sensor inputs. The obstacle register is represented by a partial function ($A \rightharpoonup B$) from natural number identifiers to obstacles, with a finite domain.

Using the Sensor state type, we define several derived velocity quantities for horizontal, vertical, and the overall (spherical) velocity, which are needed to implement requirement R6, for example.

$$hvel \triangleq \sqrt{(auv{:}ns\_vel^2 + auv{:}ew\_vel^2)}$$
$$vvel \triangleq auv{:}rate\_of\_climb$$
$$vel \triangleq \sqrt{(hvel^2 + vvel^2)}$$

The horizontal velocity combines the north-south and east-west components using a vector calculation employing square root. In Isabelle, the type system ensures correctness of the units: $auv{:}ns\_vel^2$ and $auv{:}ew\_vel^2$ both have type $real[(meter \cdot second^{-1})^2]$, and so they can be added together. Then, taking the square root results in a value of type $real[meter \cdot second^{-1}]$, as expected for a velocity. The vertical velocity is really a synonym for the rate of climb, and so we declare it as such. The overall spherical velocity, including the vertical component, can again be obtained by a vector calculation. In

Isabelle/SACM we link each of these variables to the corresponding term definition, to allow traceability back to the requirements.

Similarly, we also define some functions for calculating distances between the AUV and an obstacle identified in the register:

$$hdist(ob:nat) \triangleq \sqrt{\left(\begin{array}{l} obs(ob){:}ns\_rel\_dist^2 \\ +obs(ob){:}ew\_rel\_dist^2) \end{array}\right)}$$

$$vdist(ob:nat) \triangleq |auv{:}depth - obs(ob){:}obs\_depth|$$

$$odist(ob:nat) \triangleq \sqrt{(hdist(ob)^2 + vdist(ob)^2)}$$

The horizontal distance is, again, obtained by combination of the components, though with different units to the velocities. The vertical distance is the absolute difference between the depth of the AUV and obstacle. Finally, the overall distance is calculated by combining the horizontal and vertical distances. We can use the *odist* function to formalise the notion of the AUV being in an OPEZ:

Definition 4.3 (Object Proximity Exclusion Zones).

$$inOPEZ \triangleq \left(\begin{array}{l} (\exists\, ob \in \mathrm{dom}(obs) \bullet odist(ob) \le 300{\cdot}milli{\cdot}meter) \\ \vee\ auv{:}depth \le 300{\cdot}milli{\cdot}meter) \end{array}\right)$$

This formalises the term definition of OPEZ in Figure 4. The assumption is that the pond walls are registered as static obstacles. The definition therefore checks whether there is either an obstacle within 300 mm, or the AUV's depth is less than 300 mm. We use the functions defined in this section for the behavioural model in §5.

5 BEHAVIOURAL MODEL

Here, we model the LRE using RoboChart [28, 29], a formal graphical modelling language for robotic controllers. It includes both a block-based architectural notation, and a statechart-like language. Modelling in RoboChart is supported by the Eclipse-based RoboTool[5], with which we created the diagrams in this section.

A typical RoboChart model (a "module" ⚙) consists of a robotic platform (🤖), and one or more controllers (∝). The robotic platform acts as an abstraction layer for the hardware, and can provide shared variables, which often represent sampled continuous variables for sensors and actuators, and events. The platform is controlled by one or more controllers, which are modelled using the state machine notation (∝). Each controller has access to variables in the robotic platform, and can also have its own private variables. Controllers can also communicate with one another, and the robotic platform, using events that allow the communication of commands and data.

Shared variables and events can be grouped together in interfaces which can be provided (🅿), required (Ⓡ), or defined (ⓘ). For example, a controller may require shared variables provided by the robotic platform. Defined interfaces are usually used to assign a set of events to a controller or state machine. In this work, we model interfaces using data schemas, like those defined in §4.

5.1 AUV Architecture

The overall AUV architecture in Figure 8 uses the RoboChart block notation in the module AUV_Module. The LRE is modelled using the controller LRE_Ctrl. It uses inputs both from the physical sensors of the AUV, and also digital operator inputs.

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

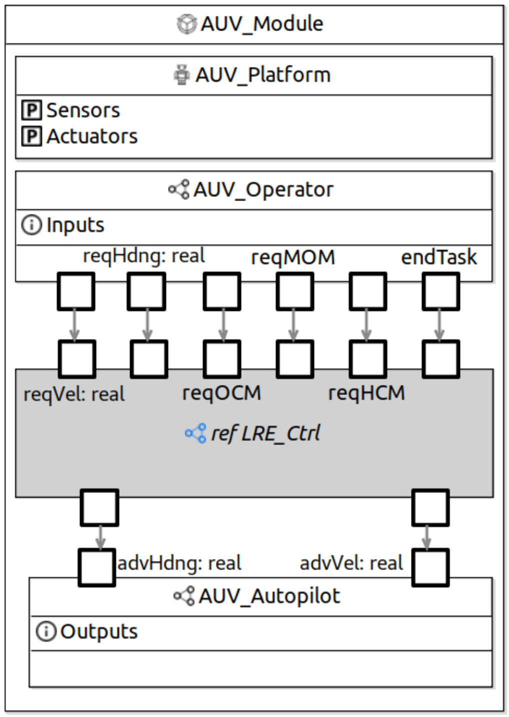

Figure 8: Overall Architecture of the AUV and LRE

The physical inputs are modelled as shared variables in the robotic platform AUV_Platform, through two provided interfaces: Sensors and Actuators. The former corresponds to the state space specified in Figure 7, and the latter is unused, as only the autopilot control the actuators, so its definition is elided for now.

The digital inputs are modelled using RoboChart events, that are represented by the squares on the controller borders. Events consist of a name and an optional list of typed parameters, to specify any data the event may carry. They are analogous to channels in the CSP process algebra [2], which is also used to give a semantics to RoboChart [28]. Events can either be synchronous or asynchronous.

The LRE takes its digital inputs from the controller AUV_Operator, and gives outputs to the controller AUV_Autopilot. The LRE does not directly control the actuators, but rather communicates advice to the autopilot, which in turns controls the actuators. There are six event inputs, which are collected in the defined interface Inputs:

- reqVel, with which the operator can request a new velocity;
- reqHdng, to request a new heading;
- reqOCM, reqMOM, and reqHCM, with which the operator can request a particular mode;
- endTask, with which the operator can delineate a task.

The two output events are advVel and advHdng, with which the LRE can sends instructions to change velocity or heading to the autopilot, which are collected in the defined interface Outputs.

5.2 LRE Controller

The state space of the LRE controller is specified in Figure 9. In addition to the sensor variables, which are included by extending Sensors, the state space includes variables to track the closest dynamic and static obstacles. For simplicity in the model, we assume

```
schema LRE_State = Sensors +
  — ‹ Closest dynamic obstacle ›
  cdyn    :: nat
  — ‹ Closest static obstacle ›
  cstc    :: nat
where
  "cdyn ∈ dom(obs)" "cstc ∈ dom(obs)" "is_static(cstc)"
```

Figure 9: LRE state space

that it is sufficient to consider only the closest obstacles in decision making, though this can be generalised. The specified invariants ensure that the closest obstacles are both in the obstacle register, and that the static obstacle is indeed static (see Definition 4.2).

The assumption of our model is that in each behavioural cycle, the LRE will calculate the closest obstacles, and for a dynamic obstacle will calculate the time at which it will reach its closest point of approach (CPA). From this information, the LRE will be able to determine whether it is currently on an unsafe trajectory, and apply evasive manoeuvres. Moreover, the LRE will also use the obstacle information to determine whether it needs to switch into high caution mode (HCM). We first specify the operations for calculating the closest static and dynamic obstacles below:

Definition 5.1 (Closest Obstacle Calculation Operations).

$$sobs \triangleq \{obs \in \mathrm{dom}(obs) \mid is\_static(obs)\}$$
$$dobs \triangleq \mathrm{dom}(obs) \setminus sobs$$
$$CalcCStc \triangleq cstc{:}\left[sobs \neq \emptyset,\ \begin{array}{c} cstc \in sobs \land \\ \left(\begin{array}{l} \forall x \in sobs \bullet \\ \quad odist(x) \geq odist(cstc) \end{array}\right)\end{array}\right]$$
$$CalcCDyn \triangleq cdyn{:}\left[dobs \neq \emptyset,\ \begin{array}{c} cdyn \in dobs \land \\ \left(\begin{array}{l} \forall x \in dobs \bullet \\ \quad odist(x) \geq odist(cdyn) \end{array}\right)\end{array}\right]$$

We first define the sets of static and dynamic obstacles in the register: *sobs* and *dobs*. We specify the two operations for calculating static and dynamic obstacles using Morgan's specification statement operator [30]. It has the form $a{:}[pre, post]$, with a variable frame (a): the set of variables permitted to change, precondition (*pre*), and a postcondition (*post*). They are both in theory non-deterministic as there could be several obstacles that are close by. Both operations have a similar form: they select a static or dynamic obstacle, so that this is the closest such obstacle in the register. We have automatically proved, using Isabelle/UTP, that both of these operations preserve the state invariants of *LRE_State*, that is

$$\Sigma{:}[LRE\_State, LRE\_State] \sqsubseteq CalcCStc, CalcCDyn$$

where $\sqsubseteq$ means "is refined by", and Σ is the set of all variables.

Once the closest obstacles have been determined, the next step is to work out whether the current trajectory of the AUV, with respect to the closest dynamic obstacle, is safe. For this, we need to calculate the closest distance of approach (CDA), which is minimum separation that the obstacle and AUV will have if they both remain on their current course. If the CDA is too low, it means a collision could occur, and therefore the AUV needs to enact collision avoidance. To determine the CPA, we need (1) functions to calculate the relative distance at time t, and (2) calculate the time at closest point of approach (TCPA). We first define relative velocity and distance:

Definition 5.2 (Relative Velocity and Distance).

$$ns\_rvel \triangleq obs(cdyn){:}obs\_ns\_vel - auv{:}ns\_vel$$
$$ew\_rvel \triangleq obs(cdyn){:}obs\_ew\_vel - auv{:}ew\_vel$$
$$ns\_rdist(t) \triangleq obs(cdyn){:}ns\_rel\_dist + ns\_rvel{\cdot}t$$
$$ew\_rdist(t) \triangleq obs(cdyn){:}ew\_rel\_dist + ew\_rvel{\cdot}t$$

Functions *ns_rvel* and *ew_rvel* calculate the relative velocity between the AUV and the obstacle. Using these, and the relative position of the obstacle, we can determine whether the obstacle is approaching or retreating from the AUV. For example, if the obstacle is 5 m due north of the AUV, the AUV has a velocity of 1 ms^{-1} north ($ns\_vel = 1$), and the obstacle has a velocity of 0.5 ms^{-1} due south ($ns\_vel = -0.5$), then the relative velocity is $(-0.5) - 1 = -1.5\,\mathrm{ms}^{-1}$ (south) and so the obstacle is approaching from the north.

Functions $ns\_rdist(t)$ and $ew\_rdist(t)$ calculate the relative distance from the obstacle at time t. The definitions simply subtract the velocity component mulitplied by t from the current distance component. Continuing the example above, we have $ns\_rdist(t) = 5 - 1.5{\cdot}t$, so the AUV and the obstacle will be on top of each other at $t = 3.34$ s. We can now give the formula for the TCPA:

Definition 5.3 (Timed to Closest Point of Approach).

$$TCPA \triangleq \frac{\left(\begin{array}{l} -obs(cdyn){:}ew\_rel\_dist \cdot ew\_rel\_vel + \\ -obs(cdyn){:}ns\_rel\_dist \cdot ns\_rel\_vel \end{array}\right)}{ew\_rel\_dist^2 + ns\_rel\_vel^2}$$
$$CDA \triangleq \sqrt{(ns\_rdist(TCPA)^2 + ew\_rdist(TCPA)^2)}$$

This formula obtains the t that gives the minimal combined distance from the two distance components. In the example above, the east-west component is zero, and so we have $TCPA = -(5 \cdot -1.5)/(-1.5)^2 = 3.5/2.25 = 3.34$, as expected. Finally, we can obtain the CDA by simply plugging the TCPA into the two relative distance formulas, and calculate the overall distance.

With these functions, we model the LRE's behaviour in the RoboChart state machine shown in Figure 10. Its goal is to implement the requirements in Section 2. The transitions use an action language with a similar syntax to CSP [2]: $a?v$ receives a value over channel a and places it into variable v, and $b!e$ sends e over channel b. The state machine has four states (OCM, MOM, HCM, CAM) and transitions between them. Each transition is decorated with an expression with the general form of $trigger[condition]/action$. The trigger denotes an event required for execution of the transition, and the condition is a predicate on the variables. The action is executed if the transition executes, and following the trigger. Each part of the general transition form can be omitted. For example, there is a transition from OCM to MOM that has the form

$$reqMOM\left[\begin{array}{l} vel \leq 0.1 \land odist(cdyn) > 7.5 \\ \land\ odist(cstc) > 0.3 \land \neg inOPEZ \end{array}\right]$$

It states that the LRE can move from OCM to MOM when the trigger event *reqMOM* is received from the operator, and the set of conjoined conditions specified in requirement R4 hold. The state *MOM* has an entry action, *advVel*!1, that is executed when the state is activated from any transition, and advises the autopilot to set the velocity to the maximum 1 ms^{-1}. The top-most transition from

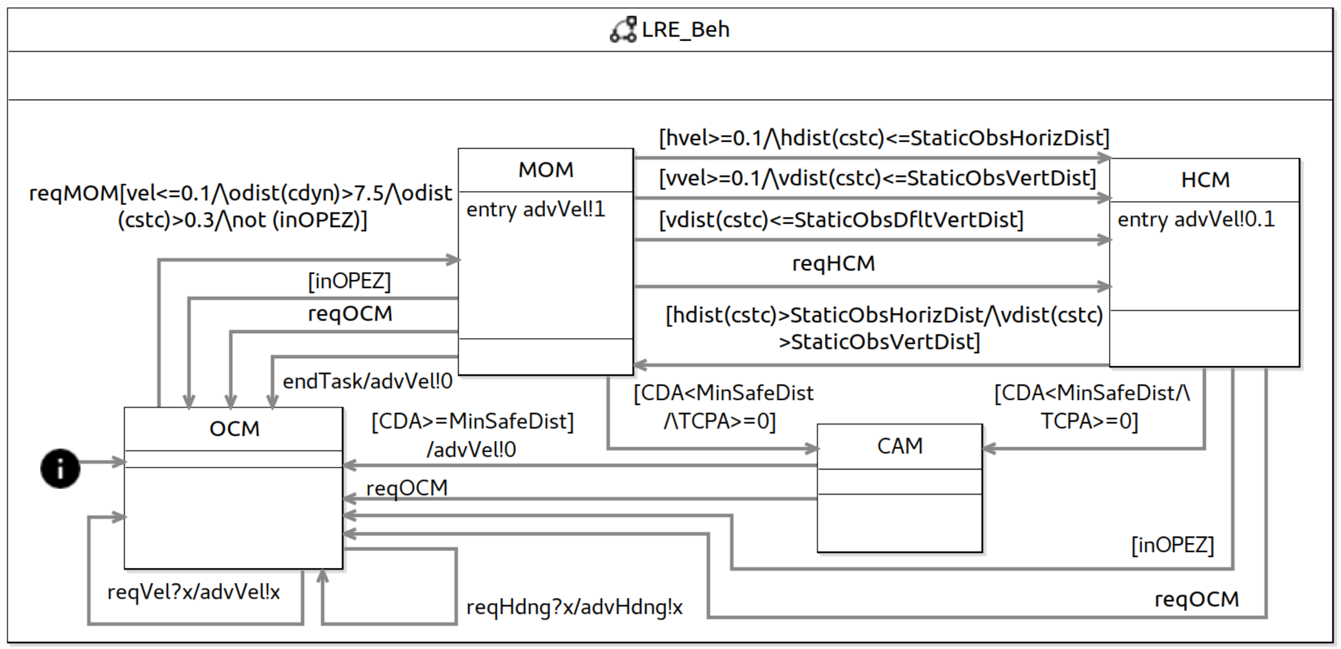
LRE_Beh
MOM
entry advVel!1
HCM
entry advVel!0.1
OCM
CAM
reqMOM[vel<=0.1/\odist(cdyn)>7.5/\odist (cstc)>0.3/\not (inOPEZ)]
[hvel>=0.1/\hdist(cstc)<=StaticObsHorizDist]
[vvel>=0.1/\vdist(cstc)<=StaticObsVertDist]
[vdist(cstc)<=StaticObsDfltVertDist]
reqHCM
[hdist(cstc)>StaticObsHorizDist/\vdist(cstc) >StaticObsVertDist]
[inOPEZ]
reqOCM
endTask/advVel!0
[CDA<MinSafeDist /\TCPA>=0]
[CDA<MinSafeDist/\ TCPA>=0]
[CDA>=MinSafeDist] /advVel!0
reqOCM
[inOPEZ]
reqVel?x/advVel!x
reqHdng?x/advHdng!x
reqOCM

Figure 10: State machine for LRE behaviour

```
statemachine LRE_Beh =
uses LRE_State
vars v::"real[meter/second]" h::"real[radian]"
events reqMOM reqOCM reqHCM
  reqVel::"real[meter/second]"
  advVel::"real[meter/second]"
  reqHdng::"real[radian]" advHdng::"real[radian]"
states OCM MOM:"entry advVel!(1.0)"
  HCM:"entry advVel!(0.1)" CAM
initial OCM
transitions
  t1: "from OCM to MOM trigger reqMOM
       condition vel ≤ 0.1 ∧ odist(cdyn) > 7.5·meter
               ∧ odist(cstc) > 0.3·meter ∧ ¬ inOPEZ"
  t2: "from MOM to OCM condition inOPEZ"
  t3: "from MOM to OCM trigger reqOCM"
  t4: "from MOM to HCM
       condition hvel ≥ 0.1
                 ∧ hdist(cstc) ≤ StaticObsHorizDist"
  t5: "from MOM to HCM
       condition vvel ≥ 0.1
               ∧ vdist(cstc) ≤ StaticObsVertDist"
  t6: "from MOM to HCM
       condition vdist(cstc) ≤ StaticObsDfltVertDist"
  t7: "from MOM to HCM trigger reqHCM"
  t8: "from HCM to MOM
       condition hdist(cstc) > StaticObsHorizDist
               ∧ vdist(cstc) > StaticObsDfltVertDist"
  t9: "from HCM to OCM condition inOPEZ"
  t10: "from HCM to OCM trigger reqOCM"
  t11: "from MOM to CAM
        condition CDA < MinSafeDist ∧ TCPA ≥ 0"
```

Figure 11: LRE state machine in Isabelle

MOM to HCM has no trigger action, and only the condition

$$[hvel \geq 0.1 \wedge hdist(cstc) \leq StaticObsHorizDist]$$

attached, meaning that it will activate as soon as the sensor values enter the characterised range. Requirements R2 to R9 can be straightforwardly implemented in RoboChart using transitions and entry actions, since they largely deal with mode switching, and what the LRE can do when in a particular mode.

The LRE initially enters OCM, indicated by the initial junction (ⓘ). Whilst in OCM, the LRE is not directing the AUV, and instead accepts velocity and heading commands using *reqVel* and *reqHdng*, which are passed on to the autopilot using the corresponding advice events (cf. R2). Once a suitable trajectory has been selected, the operator can request MOM, which can be entered provided the AUV is not close to obstacles (R4). Whilst in MOM, and indeed any other state, the operator can request control using the *reqOCM* event (R3). Control is also handed back when a task ends, triggered by *endTask*, at which point the velocity is also set to 0. The LRE moves from MOM to HCM either when the AUV is close to a static obstacle, or when the operator requests it (R6). CAM is entered when an unsafe trajectory is detected using the *CDA* calculation from Definition 5.3 (R9). For now, the behaviour in CAM is unspecified, as the collision avoidance algorithm is under development. The LRE exits CAM once it is no longer on an unsafe trajectory, and drops back to OCM setting the velocity to 0, to await further instructions.

An Isabelle representation is shown in Figure 11. This uses the command **statemachine**, which we developed previously [12], and has been extended and improved for this paper. It provides a textual language and theorem proving facilities for RoboChart. The LRE state machine was manually translated from Figure 10, but this can be automated. The command creates a state machine called *LRE_Beh*, using the state space defined in Figure 9. We add two

```
Claim LRE_Reqs ‹ All LRE requirements are implemented
  in the statemachine @{const LRE_Beh.machine}. ›

Claim LREC1 ‹ @{Requirement DQ_AUV_LRE_SRD_140_00}
  is implemented by transition @{const LRE_Beh.t1}. ›

Claim LREC2 ‹ @{Requirement DQ_AUV_LRE_SRD_170_00}
  is implemented by the entry action of state
  @{const LRE_Beh.MOM}. ›

Claim LREC3 ‹ @{Requirement DQ_AUV_LRE_SRD_180_00}
  is implemented by the transitions
  @{const LRE_Beh.t4} -- @{const LRE_Beh.t7}. ›

Inference AUV_S1 needsSupport
src "{@{Claim LREC1}, @{Claim LREC2}, @{Claim LREC3}}"
tgt "{@{Claim LRE_Reqs}}"
```

Figure 12: LRE Requirement Allocation Claims

variables, v and h, that are used to store velocities and headings. Most variables have physical units, and so expressions must use these consistently (§4.1). We also create the events, with the types as specified in Figure 8. We then specify each state, with entry actions when needed, and specify that OCM is the initial state. Finally, we specify each transition, with source and target states, and an identifier. Due to space constraints, we only show 11 transitions.

Underneath, the **statemachine** command automatically checks well-formedness of the state machine, and generates a denotational semantics, assigned to a definition called *action*, which targets a formal modelling language called *Circus* [33]. *Circus* combines the concurrency primitives of CSP, with state modelling primitives from Z [36], refinement calculus [30], and Dijkstra's guarded command language [10]. We previously mechanised *Circus* in Isabelle [13, 15], which allows us to perform various verifications, including checking for deadlock. The semantics depends on previously defined constants like *StaticObsHorizDist*. If these are changed, the entire model is automatically recompiled and rechecked by Isabelle.

Though the LRE has an uncountable state space, due to the presence of real transcendental manipulations, we can still verify the model due to the symbolic nature of our verification technique [15]. Below is a theorem and proof of deadlock freedom:

```
theorem LRE_deadlock_free: "dlockf ⊑ local.action"
  by ((sm_induct wf:Wf simps: action_def inters
    , sm_calc simps: nmap tmap semantics simps)
    ; (simp add: action_rep_eq, rdes_refine; blast))
```

This states that *action*, which gives the semantics to *LRE_Beh*, refines a specification of deadlock freedom, *dlockf* [12, 15]. As can be seen, this process is completely automated through several tactics we have developed for state machine reasoning [12], and can generally be applied to any state machine. Deadlock freedom means that *LRE_Beh* never enters a state where no behaviour is possible. It follows because there is always an enabled event.

Now, using Isabelle/SACM we can link each of the state machine elements to one of the requirements, which allow traceability for each behaviour and would form part of the case for DO-178C compliance. An argument for allocation of three requirements to *LRE_Beh* is shown in Figure 12. The top-level claim is *LRE_Reqs*, stating that all of the requirements are implemented in the generated state machine *LRE_Beh*. There are then three subclaims, *LREC1-LREC3*, that provide part of the argument for *LRE_Reqs*. *LREC1* states that requirement R4 is implemented by transition $t1$. *LREC2* states that requirement R5 is implemented by the entry action of MOM. *LREC3* states that R6 is implemented by transitions $t4 - t7$. Finally, the inference *AUV_S1* shows that the three subclaims all provide support for *LRE_Reqs*. However, not all requirements have yet been allocated in the argument (although in reality they have), and so the inference is annoted with the the **needsSupport** keyword, meaning that the argument is to be completed. In a similar way, we can link our deadlock freedom theorem to R10.

6 CONCLUSIONS

We applied Isabelle/SACM in mechanising a fragment of a model-based assurance case for an AUV safety controller. This is supported by several formal artifacts mechanised in Isabelle, including terminology and requirements, a data model with SI unit support, several functions and operations, and a high-level behavioural model in the graphical RoboChart language. All these heterogeneous artifacts have a unifying semantics in the Isabelle/UTP semantic framework [16], which also provides verification support through Isabelle's powerful automated reasoning capabilities.

The overarching goal of Isabelle/SACM is to support certification of critical systems, by conveying assurance arguments and verification results to regulators and other stakeholders. Mechanised support increases confidence through machine-checking and traceability, and eases the burden of maintaining and evolving an assurance case. This is of particular importance for autonomous systems, which must evolve to match an open environment and changing requirements [18]. Our paper adds weight to the evidence [3, 5, 14, 39] that Isabelle is an ideal platform for assurance cases, utilising a variety of formal methods, to support certification.

In related work, Denney et al. [9] have developed a sophisticated graphical tool for assurance cases, called AdvoCATE, that includes management of hazards, requirements, arguments, and other artifacts. This, and Denney's pioneering work on formal semantics for assurance cases [7, 8], is a strong inspiration. Wei et al. [38] have developed an SACM-based tool called ACME, supporting graphical arguments and integration with model-based engineering. Rushby [35] proposes an evidential tool bus as for managing results from several verification tools, for assurance cases, an idea that was implemented by Cruanes et al. [6], and shares several characteristic with Isabelle. Resolute [17] is a tool for automating assurance case generation from an AADL architectural model that we are exploring links with. Brucker and Wolff have applied DOF to a CENELEC 50128 safety case for an odometric case study [5].

In future work, we aim to use Isabelle/SACM as a backend for tools like AdvoCATE [9], and are currently integrating it with Eclipse for use with ACME [38] and RoboTool [28]. We will also further verify the LRE model, in particular its timing requirements and continuous dynamics, for which we will utilise our Isabelle implementation of Differential Dynamic Logic [11, 31, 34]. The ultimate goal is a automated assurance cases that go from hazard analysis and safety requirements, right down to executable code.

Acknowledgements. We thank Prof. Burkhart Wolff and Prof. Achim Brucker, for their helpful advice on the use of DOF and feedback on our SI unit formalisation. We thank Dr. Alvaro Miyazawa for his support in the use of RoboTool and its development. This work is supported by the EPSRC UKRI project CyPhyAssure (Grant EP/S001190/1, https://www.cs.york.ac.uk/circus/CyPhyAssure/).

REFERENCES

[1] A. Armstrong, V. Gomes, and G. Struth. 2015. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing* 28, 2 (2015).

[2] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. 1984. A Theory of Communicating Sequential Processes. *J. ACM* 31, 3 (1984), 560–599.

[3] A. Brucker, I. Aït-Sadoune, P. Crisafulli, and B. Wolff. 2018. Using the Isabelle Ontology Framework – Linking the Formal with the Informal. In *Intelligent Computer Mathematics (CICM) (LNCS)*, Vol. 11006. Springer, 23–38.

[4] A. Brucker and B. Wolff. 2019. Isabelle/DOF: Design and Implementation. In *SEFM (LNCS 11724)*. Springer, 279–292.

[5] A. Brucker and B. Wolff. 2019. Using Ontologies in Formal Developments Targeting Certification. In *Integrated Formal Methods (iFM) (LNCS)*, Vol. 11918. Springer, 65–82.

[6] S. Cruanes, G. Hamon, S. Owre, and N. Shankar. 2013. Tool Integration with the Evidential Tool Bus. In *VMCAI (LNCS)*, Vol. 7737. Springer, 275–294.

[7] E. Denney and G. Pai. 2013. A Formal Basis for Safety Case Patterns. In *SAFECOMP (LNCS)*, Vol. 8153. Springer, 21–32.

[8] E. Denney and G. Pai. 2015. Towards a Formal Basis for Modular Safety Cases. In *Computer Safety, Reliability, and Security (SAFECOMP) (LNCS)*, Vol. 9337. Springer, 328–343.

[9] E. Denney and G. Pai. 2018. Tool support for assurance case development. *Automated Software Engineering* 25 (2018), 435–499.

[10] E. W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457.

[11] S. Foster. 2019. Hybrid Relations in Isabelle/UTP. In *7th Intl. Symp. on Unifying Theories of Programming (UTP) (LNCS)*, Vol. 11885. Springer, 130–153.

[12] S. Foster, J. Baxter, A. Cavalcanti, A. Miyazawa, and J. Woodcock. 2018. Automating Verification of State Machines with Reactive Designs and Isabelle/UTP. In *FACS (LNCS 11222)*. Springer, 137–155.

[13] S. Foster, A. Cavalcanti, S. Canham, J. Woodcock, and F. Zeyda. 2020. Unifying Theories of Reactive Design Contracts. *Theoretical Computer Science* 802 (January 2020), 105–140.

[14] S. Foster, Y. Nemouchi, M. Gleirscher, and T. Kelly. 2019. Isabelle/SACM: Computer-Assisted Assurance Cases with Integrated Formal Methods. In *iFM (LNCS 11918)*. Springer, 379–398.

[15] S. Foster, K. Ye, A. Cavalcanti, and J. Woodcock. 2018. Calculational Verification of Reactive Programs with Reactive Relations and Kleene Algebra. In *Proc. 17th Intl. Conf. on Relational and Algebraic Methods in Computer Science (RAMICS) (LNCS)*, Vol. 11194. Springer, 205–224.

[16] S. Foster, F. Zeyda, and J. Woodcock. 2016. Unifying heterogeneous state-spaces with lenses. In *ICTAC (LNCS 9965)*. Springer, 295–314.

[17] A. Gacek, J. Backes, D. Cofer, K. Slind, and M. Whalen. 2014. Resolute: An Assurance Case Language for Architecture Models. In *Proc. 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT)*. ACM, 19–28. https://doi.org/10.1145/2663171.2663177

[18] M. Gleirscher, S. Foster, and Y. Nemouchi. 2019. Evolution of Formal Model-Based Assurance Cases for Autonomous Robots. In *SEFM (LNCS 11724)*. Springer, 87–104.

[19] M. Gleirscher, S. Foster, and J. Woodcock. 2019. New Opportunities for Integrated Formal Methods. *Comput. Surveys* 52, 6 (2019), 36.

[20] G. Greenaway, J. Lim, J. Andronick, and G. Klein. 2014. Don't sweat the small stuff: Formal verification of C code without the pain. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 429–439.

[21] I. Habli and T. Kelly. 2014. Balancing the Formal and Informal in Safety Case Arguments. In *VeriSure Workshop, colocated with CAV*.

[22] J. Harrison. 2005. A HOL Theory of Euclidean space. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005* (August 2005) *(LNCS)*, Joe Hurd and Tom Melham (Eds.), Vol. 3603. Springer, Oxford, UK.

[23] R. Hawkins, I. Habli, D. Kolovos, R. Paige, and T. Kelly. 2015. Weaving and Assurance Case from Design: A Model-Based Approach. In *Proc. 16th Intl. Symp. on High Assurance Systems Engineering*. IEEE.

[24] C. A. R. Hoare and J. He. 1998. *Unifying Theories of Programming*. Prentice-Hall.

[25] F. Immler. 2014. Formally Verified Computation of Enclosures of Solutions of Ordinary Differential Equations. In *Proc. 6th NASA Formal Methods Symposium (NFM) (LNCS)*, Vol. 8430. Springer.

[26] F. Immler and J. Hölzl. 2012. Numerical Analysis of Ordinary Differential Equations in Isabelle/HOL. In *3rd Intl. Conf. on Interactive Theorem Proving (ITP) (LNCS)*, Vol. 7406. Springer, 377 – 392.

[27] T. Kelly. 1998. *Arguing Safety – A Systematic Approach to Safety Case Management*. Ph.D. Dissertation. University of York.

[28] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, and J. Woodcock. 2019. RoboChart: modelling and verification of the functional behaviour of robotic applications. *Software and Systems Modelling* 18 (January 2019), 3097–3149. Issue 5.

[29] A. Miyazawa, P. Ribieiro, W. Li, A. Cavalcanti, and J. Timmis. 2017. Automatic Property Checking of Robotic Applications. In *Intl. Conf. on Intelligent Robots and Systems (IROS)*. IEEE, 3869–3876.

[30] C. Morgan. 1996. *Programming from Specifications*. Prentice-Hall.

[31] J. H. Y. Munive, G. Struth, and S. Foster. 2020. Differential Hoare Logics and Refinement Calculi for Hybrid Systems with Isabelle/HOL. In *18th Intl. Conf on Relational and Algebraic Methods in Computer Science (RAMiCS) (LNCS)*, Vol. 12062. Springer, 169–186.

[32] T. Muranushi and R. A. Eisenberg. 2014. Experience Report: Type-Checking Polymorphic Units for Astrophysics Research in Haskell. In *Proc. 2014 Haskell Symposium*. ACM, New York, NY, USA, 31–38.

[33] M. Oliveira, A. Cavalcanti, and J. Woodcock. 2009. A UTP semantics for Circus. *Formal Aspects of Computing* 21 (2009), 3–32. Issue 1-2.

[34] André Platzer. 2018. *Logical Foundations of Cyber-Physical Systems*. Springer.

[35] J. Rushby. 2005. An Evidential Tool Bus. In *Formal Methods and Software Engineering (ICFEM) (LNCS)*, Vol. 3785. Springer.

[36] M. Spivey. 1989. *The Z-Notation - A Reference Manual*. Prentice Hall.

[37] F. Tuong and B. Wolff. 2019. Deeply Integrating C11 Code Support into Isabelle/PIDE. In *Formal Integrated Development Environment (F-IDE) (EPTCS)*, Vol. 310. 13–28.

[38] R. Wei, T. Kelly, X. Dai, S. Zhao, and R. Hawkins. 2019. Model based system assurance using the Structured Assurance Case Metamodel. *Systems and Software* 154 (2019).

[39] M. Wenzel. 2019. Interaction with Formal Mathematical Documents in Isabelle/PIDE. In *CICM (LNCS 11617)*. Springer, 1–15.

[40] J. Woodcock, A. Cavalcanti, S. Foster, A. Mota, and K. Ye. 2019. Probabilistic Semantics for RoboChart. In *7th Intl. Symp. on Unifying Theories of Programming (UTP) (LNCS)*, Vol. 11885. Springer.