Automated Compositional Verification for Robotic State Machines using Isabelle/HOL

Fang Yan, Simon Foster, Ibrahim Habli
Department of Computer Science, University of York, York, UK
firstname.lastname@york.ac.uk

***Abstract*—RoboChart is a graphical language for model-based engineering of robotic systems, in the style of UML and SysML. It contains notations for data structures, system architecture, and the behaviour of individual robotic controllers using state machines. Crucially, RoboChart has a formal semantics in the CSP process algebra, which provides a precise foundation for software engineering and formal verification using model checking. However, due to state explosion, the application of model checking does not scale. In this paper, we contribute a compositional verification technique that uses Isabelle/HOL RoboChart state machines symbolically. Our technique uses state invariants to capture safety requirements over a very large or infinite state, similar to the B method, and is highly automated using Isabelle's sledgehammer tool. We give a model transformation from the RoboTool development environment to Isabelle/HOL and apply this to several verification case studies.**

***Index Terms*—RoboChart, state machine, theorem proving, model transformation, Isabelle/HOL**

I. Introduction

Model-based Engineering (MBE) is widely used in system development. For safety-critical systems, it is crucial to verify system models rigorously. Compared with testing and simulation, formal verification using model checking and theorem proving can explore the whole state space or at least a substantial portion of it. However, the application of formal verification in engineering practice is hindered by the excessive need for formal expertise. As such, the integration of formal verification into MBE has been the subject of intense research [1]–[4].

For example, RoboChart [5] is a UML-like modelling language for robotic systems. It has a denotational semantics [5] based on the CSP process algebra and a probabilistic semantics in PRISM [6]. These formal semantics enable RoboChart to be verified by the model checkers FDR [7] for non-probabilistic properties, and PRISM [8] for probabilistic properties.

Though model checking is a valuable and highly automated technique, it suffers from the state explosion problem [4], and cannot be applied to robotic models containing unbounded data types and real numbers. In contrast, theorem proving represents the state space symbolically, without requiring abstractions, and so avoids these issues.

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 812.788 (MSCA-ETN SAS). This publication reflects only the author's view, exempting the European Union from any liability. Project website: http://etn-sas.eu/.

Although traditionally a manual process, integrating automated proof tools like SMT solvers has the potential to significantly improve theorem proving usability. For example, *sledgehammer* [9] is a tool that applies ATPs and SMT solvers in Isabelle/HOL [10]. However, the integration of theorem proving into MBE workflows remains a major challenge.

So the question is how can we leverage the use of such automated proof techniques for verification in an MBE process? Though work exists on the application of theorem proving to software models, e.g., [11]–[13], the work usually depends on manual translation or proof and so inhibits traceability. Moreover, previous work on RoboChart verification cannot benefit from compositionality, because all transitions are compiled into a monolithic model for checking, meaning that the benefits of theorem proving are only partly gained.

Our contribution is a compositional verification technique for verifying RoboChart state machine models. Each transition of the machine can be independently verified against a set of invariants. Our technique can support seamless integration of theorem proving in supporting MBE. The technique comprises the five activities shown in Fig. 1. We bridge the gap between the domain-specific models and formalization using a set of intermediary notations that balance usability by software engineers with suitable rigour provided by formal semantics.

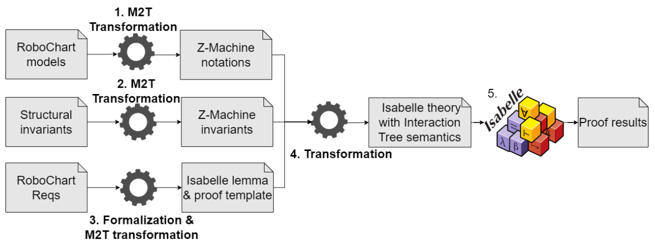

Fig. 1: Pipeline for verifying RoboChart state machine.

The technical contributions of our paper are (1) a pipeline for RoboChart state machine verification by automated theorem proving; (2) the mapping rules from RoboChart to an abstract machine language with verification support of Isabelle/HOL; (3) a requirement proving strategy and proof methods for deadlock freedom; and (4) a case study on an underwater vehicle robot to demonstrate the effectiveness of our method. The transformation algorithm, a running example and the case study can be accessed online[1].

[1]https://github.com/uoy-fangyan/ICECCS2023-Compositional-Verification

The rest of the paper is structured as follows. §II discusses related work on verifying domain-specific models by formal reasoning. §III provides the required background on RoboChart. §IV gives the background knowledge of Z-Machines, the intermediary notation underpinning RoboChart transformation, then begins our contributions with a deadlock freedom verification method. In §V, we present a modelling pattern based on Z-Machines. §VI introduces the mapping rules from RoboChart state machines to Z-Machines according to the modelling pattern. §VII discusses the verification approach, including the strategy for constructing structural invariants, and the type of requirement to be covered. §VIII shows the case study on an autonomous underwater vehicle. We evaluate our method and conclude in §IX.

II. Related work

There is a substantial body of work on verifying models developed in model-driven design languages within a formal reasoning environment. A major part of the results are for verifying the system models with model checking methods by transforming models from a system modelling language (e.g., AADL, SysML, Stateflow) to a formal language (e.g., Lustre, LNT) [1]–[3]. We focus on works in verifying system models by theorem proving using automated theorem provers.

A few works exist to address the model verification by interactive theorem provers, such as Isabelle/HOL, KeYmaera X, etc. Hadad et al. [11] transform a subset of the AADL model into Event-B, then verify the models in the RODIN platform and Atelier B prover. They use UML class diagram to classify the AADL elements and further map the classes to Event-B models. Our intermediary notation, Z-Machines, is conceptually similar to Event-B, but is embedded into Isabelle/HOL. Use of Isabelle/HOL gives us access to an array of verification and modelling facilities, including automated theorem proving, code generation, and real number analysis.

Zou et al. [12], [14] translate Stateflow diagrams to HCSP [15] models which are verified in Isabelle using Hoare logic. We instead use Z-Machines as a higher-level representation, which makes the mapping more direct.

Foster et al. [13] present an automated verification technique for a subset of RoboChart by mechanising an action language for states and transitions in Isabelle/HOL. We take a further step to incorporate Z-Machines to realize the mechanization of RoboChart in Isabelle, and our mapping rules from RoboChart to the intermediary notation are inspired by their dynamic semantics of a state diagram. Their work verifies only the deadlock-free property while ours is capable of verifying different types of properties including deadlock freedom. Also, in [13] the semantics of the whole state machine is generated monolithically. Our semantic translation allows compositional verification.

In [16], Foster et al. mechanise an Interaction Tree (ITree) based CSP semantics and a Z mathematical toolkit in Isabelle/HOL. Ye et al. [17] use this ITree-based semantics to give RoboChart an operational semantics in Isabelle, whereas we use Z-Machines as a higher-level one. The intermediary notation we choose for our method (Z-Machines) is also built on top of the ITree CSP semantics. However, our purpose is to verify RoboChart models with theorem proving in Isabelle/HOL but [17] has no attempt at building a verification method and cannot benefit from compositionality.

III. RoboChart

We use an autonomous Chemical detector [5], [18] as a running example to describe features of RoboChart for modelling controllers. The robot performs a random walk with obstacle avoidance and analyses the air to detect dangerous gases. Once a chemical with an intensity above a threshold is detected, the robot drops a flag, reports the location of the gas, and stops the walk.

A RoboChart model's top-level structure is a module containing one robotic platform (an abstracted physical robot indicating available services), and multiple controllers that specify software behaviours. Fig. 2 displays the ChemicalDetector module which contains a platform Vehicle and two controller references (MainController and MicroController). The platform provides several services to the controllers including (1) reading sensor data through gas, obstacle, and odometer events; (2) movement actuation through move, randomWalk, and shortRandomWalk operations grouped in an interface Operations; and (3) dropping a flag when receiving event flag.

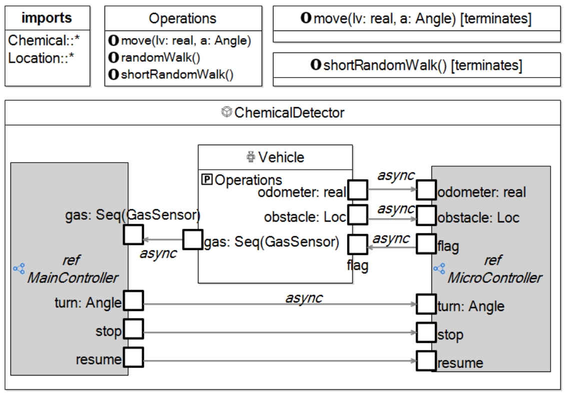

Fig. 2: RoboChart module of Chemical detector.

The types are defined in the two packages and imported as shown in Fig. 2: Chemical and Location. In the two packages, primitive types Chem and Intensity are declared for chemicals to be detected and the intensity of the chemicals, enumerations Status for the detection status, Angle for the turning directions of the robot, and Loc for the chemical locations. The packages declare a record GasSensor containing two fields (c of type Chem and i of type Intensity), and define six functions and an operation changeDirection using a state machine. The behaviour of MicroController is specified using state machine Movement, and MainController using state machine GasAnalysis as shown in Fig. 3.

The machine GasAnalysis declares a constant thr as intensity threshold, and four variables including sts as detection status, gs as the list of sensor readings, ins as the gas intensity, and anl

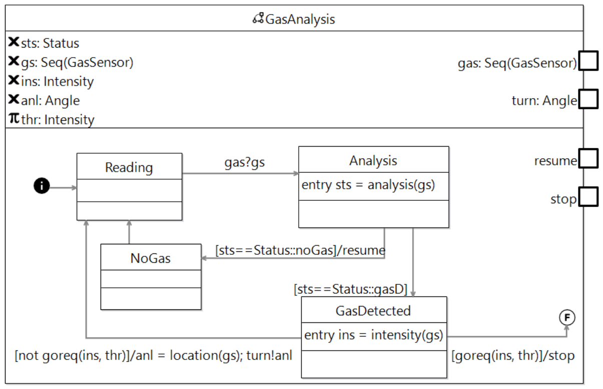

Fig. 3: RoboChart state machine model of Gas Analysis.

as turning direction. The machine has one initial junction, four normal states called Reading, Analysis, NoGas, GasDetected, and a final state. Each state may have an entry action such as sts = analysis(gs), a during action, and an exit action. A Transition is a directed connection from a source node to a target node. It can have a trigger event (e.g., event gas of input type), a guard condition (e.g., "not goreq(ins,thr)"), and an action that is executed during the transition (e.g., the assignment action "anl = location(gs)", and the action of event turn).

GasAnalysis models the gas detection behaviour of the robot. After the machine is initialized, it takes the sensor readings from event gas, then in state Analysis it analyses the gas status according to the readings. If there is no gas detected, a resume command is sent to Movement to let the robot continue its random walk, and GasAnalysis reads the new reading. If there is gas detected, the machine enters GasDetected and calculates the gas intensity. If the intensity reaches the threshold, it is identified as a dangerous gas detected, a stop signal is sent to the machine Movement, and GasAnalysis terminates. Otherwise, it is identified as no danger and a turn signal is sent to Movement to change the direction of the robot, and GasAnalysis enters Reading to take the new readings.

IV. Z-Machines and Deadlock-Freedom

Here, we introduce Z-Machines, which we use to give a semantics to RoboChart, and a novel proof method for checking deadlock-freedom. The Z-Machine language is an abstract machine model, similar to the Z [19] and B [20] methods, which is built as a conservative extension of Isabelle/HOL. A Z-Machine consists of (1) a rich state space including invariants; (2) operations acting over the 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.

As with the B method [20], a Z-Machine is essentially an action system [21], where operations can be called repeatedly depending on their preconditions. We give an operation below taken from the "Birthday Book" example [19], in Isabelle:

```
zoperation AddBirthday =
  params name∈NAME date∈DATE
  pre "name ∉ known"
  update "[ known' = known ∪ {name},
            bday' = bday ⊕ {name ↦ date} ]"
```

Here, NAME and DATE are abstract sets that can be assigned particular values. Each operation consists of (1) a number of parameters (**params**), which model inputs and outputs; (2) a precondition (**pre**), which must be true for the operation to be executed; and (3) an **update**, which simultaneously assigns new values to state variables. In addition to **zoperation**, we provide commands to create state spaces (**zstore**) with invariants, and machines (**zmachine**).

Z-Machines have a semantics in the *Circus* concurrent specification language [22], which combines Z and the CSP process algebra. The main constructs we use in building Z-Machines are (1) external choice $\Box\, e \in E \to P(e)$, where the environment picks an event e in E to chose a successor process $P(e)$; and (2) simultaneous assignment $[x_1' = e_1, \cdots x_n' = e_n]$, where several variables are atomically updated to corresponding expressions.

Z-Machines can be animated using Isabelle's code generator, via a coinductive semantics for *Circus* in the Interaction Tree (ITree) semantic model [16]. Animation consists of first initialising the state, and then checking the precondition of each operation for every possible parameter value. The user can select an enabled operation for execution, which then executes the update and iterates. The animator can be used to explore and evaluate the behaviour of a formal model.

Verification of Z-Machines consists of identifying suitable state invariants to satisfy the critical requirements, and then showing (1) that the initialisation *establishes* the invariants, i.e. $\{true\}\, Init\, \{I\}$; and (2) that each operation *preserves* them, i.e. $\{I\}\, Op_i(x,y,z)\, \{I\}$. We provide the zpog method, which generates a set of proof obligations for such a specification triple, via calculation of the operation's weakest precondition. The resulting proof obligations can then often be discharged automatically using the Isabelle tools *auto* and *sledgehammer*.

In addition to general invariant properties, we contribute a method to prove deadlock-freedom. Generally speaking, a process P is deadlock-free provided that there does not exist a trace tr (a list of events) such that $P \xrightarrow{tr} \mathit{stop}$, where *stop* is the process with no enabled events. More specifically, a Z-Machine is deadlock-free if in any reachable state there is at least one operation that is enabled, which means that no deadlocking trace exists.

We support an automated method for checking deadlock-freedom through a calculus for "deadlock-freedom preconditions" (*dfp*). The *dfp* of a process is a predicate characterising the initial states that do not have any deadlocking transitions, i.e. $dfp(P) = \{s \mid \nexists tr.\, P(s) \xrightarrow{tr} \mathit{stop}\}$. Using our ITree semantics, we prove the following equations of our *dfp* calculus:

Theorem 1 (Deadlock-Freedom Preconditions)**.**

$$\begin{aligned} dfp(\mathsf{stop}) &= false \\ dfp([x_1' = e_1, \cdots x_n' = e_n]) &= true \\ dfp(P;Q) &= (dfp(P) \wedge wlp\ P\ (dfp(Q))) \\ dfp(\Box\, e \in E \to P(e)) &= (E \neq \emptyset \wedge (\forall e \in E. dfp(P(e)))) \end{aligned}$$

The *dfp* of *stop* is *false*, since this process always deadlocks. The *dfp* of a simultaneous assignment for n variables is *true*, since this can never deadlock. The *dfp* of a sequential composition $P;Q$ requires (1) the *dfp* of P; and (2) the final state of P does not cause Q to deadlock, which we characterise with the weakest *liberal* precondition. Finally, the *dfp* of a choice between the events in E requires that E is non-empty, and that every successor process $P(e)$ for $e \in E$ is deadlock-free. We also prove the following theorem:

Theorem 2 (Deadlock-Freedom of Z-Machines)**.** *A Z-Machine with $n \in \mathbb{N}$ operations, each guarded with a parametric event c_i, is deadlock-free provided there exists an invariant I, such that $I \implies dfp\left(\Box_{i \in \{1..n\}}\ c_i(x) \to Op_i(x)\right)$.*

To prove deadlock-freedom we need a suitable invariant of the machine I, such that I implies that the precondition of at least one operation is satisfied. We can verify this by calculating the *dfp* of all the operations, which corresponds to the disjunction of all the operation preconditions. Then, we need to prove that I is sufficient to satisfy one of the preconditions.

We supply a proof method called `deadlock_free`, which automates deadlock checking. It assumes that we have already shown that I is an invariant of all the machine operations, and requires we supply lemmas to this effect. This being the case, the method calculates a deadlock freedom condition predicate for the whole Z-machine, which can typically be discharged using *auto* or *sledgehammer*. Through our Z-Machine semantics for RoboChart, we will apply this method to prove deadlock-freedom of robotic state machines.

V. Semantic Domain

To verify RoboChart models in Isabelle/HOL, we propose a modelling pattern for a RoboChart subset using Z-Machines. We use Z-Machine operations to represent RoboChart transition behaviours, and define observational variables to encode the semantic structure of RoboChart elements. These variables are accompanied by a set of healthiness conditions, following the approach of Unifying Theories of Programming [23]. The healthiness conditions characterise state machines with a coherent semantics, and all operations shall satisfy them.

A. Observational variables

RoboChart states and transitions have separate CSP semantics [5] represented by processes. A transition process starts with an enabled trigger event (if one exists). If the transition condition is met, the process exits the current state, executes any transition actions, and then enters the target state. From here, a state process starts and executes state entry actions and then during actions.

We define two types, *St* and *Evt*, to represent the sets of states and events defined in RoboChart models. A parametric datatype *tag* is defined as follows for an observational variable tr to tag semantic transition information:

```
datatype ('s, 'e) tag = State 's | Event 'e
```

To define tr using the type *tag*, *'s* is instantiated with *St*, and *'e* with *Evt*. The two constructors *State* and *Event* are used to define state-related and event-related requirements. *tag* comes with the built-in functions of *is_State* and *is_Event*, and both are of Boolean type.

We define three observational variables to enrich the structure of the Z-Machine including tr, and to enable a more precise mapping to the RoboChart semantics:

- $st :: St$ type, as the current state the machine is in;
- *triggers* of type "*Evt set*", a set of possible trigger events that can be enabled in the following transitions; and
- $tr :: (St, Evt)\ tag\ list$ as a list representing the sequential trace of the behaviours of each transition.

These variables form part of the Z-Machine store. tr is used to mimic a combined process of states and transitions based on the CSP semantics, but is a simplified version of the CSP process of RoboChart without the process of entering and exiting states. We use tr to record sequentially the trigger event, the exit action of event type in source state, the actions of event type executed, the entry action and during action of event type in target state, and the target state entered. We mainly use tr to define the event-related and state-related safety requirements.

B. Healthiness conditions

With the observational variables defined, we give a set of healthiness conditions to set constraints on the observational variables. For example, when a transition is completed, the entered state in the transition process shall be the current state. These conditions will be the basis for invariant proving. There are four generic healthiness conditions defined as below:

HC1 $\triangleq (\#tr > 0)$. The trace shall not be an empty list. In RoboChart, state machines must contain at least one state, therefore after initialization, tr is not empty.

HC2 $\triangleq (last(tr) = State\ st)$. The last element of tr shall be the current state st the state machine is in after the transition.

HC3 $\triangleq (\forall i < \#tr. tr!i = State\ \ final \longrightarrow i = \#tr - 1)$. If the state machine terminates, the last element of tr shall be one of the termination states.

We collect the healthiness conditions in a predicate called `wf_rcstore`, which is imposed on every state machine store.

VI. Model Transformation

As shown in Fig. 1, our method has five steps. The first activity is to transform the RoboChart state machine automatically into a Z-Machine. The second activity involves the generation of structural invariants for Z-Machine models under analysis. Structural invariants are model-specific constraints on the state components, separate from the healthiness conditions, and are necessary for requirement verification in the next step. The structural invariants should be provided by the users. The third activity is the formalisation of the system properties of interest, and the generation of corresponding lemma templates. The fourth activity is an automatic generation of the ITree-based *Circus* semantics in Isabelle [16]. At last, we use the *auto* proof and the *sledgehammer* interface in Isabelle/HOL to call ATPs for proving the system requirements.

In this section, we discuss Step 1 for mapping RoboChart state machines to the semantic domain of Z-Machines. We group the elements of RoboChart meta-models into six categories: Type Declarations, Context, Nodes, Transitions, Statements and Expressions, and State machines. The mapping for the Statements and Expressions is straightforward and is omitted in the paper. Thus, we present five groups of mapping rules in this section. We illustrate the whole process with the running example of the GasAnalysis state machine of the Chemical Detector from §III.

A. Types

Rule 1: **Primitive datatypes** of RoboChart are mapped to Isabelle/HOL primitive types including nat to `nat`, int to `integer`, boolean to `bool`, real to `real`, and char to `string`.

Rule 2: **Type declarations** in RoboChart are mapped to HOL type declarations. These include abstract types without a definition (i.e. given sets), record types, enumerations, Cartesian products, and sets. HOL has similar elements for these types. An example mapping for GasAnalysis is shown in Fig. 4. We use the text format of RoboChart here to compare it with the models in Isabelle/HOL.

```
datatype GasSensor {
    c : Chem
    i : Intensity }
type Chem
type Intensity
enumeration Status {noGas gasD}
```

(a) RoboChart types

```
type_synonym Chem= "nat"
type_synonym Intensity= "nat"
record GasSensor =
  c :: Chem
  i :: Intensity
enumtype Status = noGas | gasD
definition "Status= {noGas, gasD}"
```

(b) Isabelle types

Fig. 4: Gas Analysis example for mapping **Rule** 2.

Rule 3: **Types in the Z mathematical toolkit** including functions, relations and sequences, are mapped to Isabelle/HOL functions, relations, and lists, using our implementation of the Z mathematical toolkit[2]. RoboChart functions are partial, so have a precondition and postcondition pattern. We have developed a corresponding command in the Z toolkit library with a **`precondition`** and **`postcondition`** to be transformed automatically. However, for the purpose of theorem proving, an abstract function declaring the type of the domain and the range is often sufficient.

B. RoboChart context

RoboChart context includes Variables, Constants, Operations, and Events. These elements can either be declared directly in the robotic platform, or grouped in interfaces [5]. Operations are omitted in our transformation as they represent the hardware platform behaviours.

Rule 4: **Constants** are mapped to polymorphic constants in HOL using the **`consts`** keyword. Such constants are uninterpreted symbols of a given type, which can be later assigned a value using the **`def_consts`** command.

Rule 5: **Variables** are mapped to store variables. We use the **`zstore`** command to collect all the variables.

Rule 6: **Events** are collected and defined as an enumeration type `Evt` with all the events as the constructors in Isabelle/HOL. A corresponding set with all events as its elements is defined. Events may be used as a transition trigger, a transition action, or a state action.

The transformation examples using Rule 4-6 is as below:

```
consts thr::"Intensity"
consts SeqGasSensor::"GasSensor list set"
enumtype Evt = gas | turn | resume | stop
definition "Evt = {gas, turn, resume, stop}"

zstore GasAnalysis =
  sts::"Status"
  gs:: "GasSensor list"
  ins::"Intensity"
  anl::"Angle"
where invs...
```

The **`where`** section contains the healthiness conditions and structural invariants, which will be explained in detail in §VII.

C. Nodes

A RoboChart node can either be a state or a junction, with Final States and Initial Junctions as special types of States and Junctions [5]. We do not need to differentiate the initial junction from other states for transformation.

Rule 7: **Nodes** are collected and defined as an enumeration type `St` with all the nodes as the constructors. A corresponding set with all nodes as its elements is also defined.

In GasAnalysis there are six nodes as below.

```
enumtype St = initial | NoGas | GasDetected |
    Analysis  | Reading  | final
definition "St = {initial, NoGas, GasDetected,
    Analysis, Reading, final}"
```

D. RoboChart transition behaviour

Rule 8: **Transition behaviour** of RoboChart (entering and exiting states, and transitions between the states) is mapped to **`zoperation`**s. One transition matches one **`zoperation`**. This means that we retain traceability to the original state machine model. Moreover, it makes our technique compositional, since we can independently verify whether each individual transition preserves the invariants, without affecting any of the other transitions. Thus we harness the incremental nature of the Isabelle document model, whereby individual commands are processed independently [24]. The mapping to RoboChart elements is according to the following template.

```
zoperation TransN =
  params <parameter_input> ∈ "<PARAMETER>"
```

[2]https://isabelle-utp.york.ac.uk/theories/z-toolkit

```
  pre "st=<src_st> (∧ <tran_cond>)*"
  update "[<state_var>′ = <new_val_state_var>
          ,st′ = <tgt_st>
          ,triggers′={<possible_trg_events>}
          ,tr′= tr @ [<trg_event>,
              <src_exit_action>, <tran_action>,
              <tgt_action>, <tgt_st>]
          ,<parameter>′= <parameter_input> ]"
```

We now explain each of the components of this template with the example of GasAnalysis shown in Fig. 5.

```
state Reading { }
state Analysis {
    entry st = analysis (gs)}
state NoGas { }
transition t0 {
    from Reading
    to Analysis
    trigger gas ? gs
    }
transition t3 {
    from Analysis
    to NoGas
    condition st == Status::noGas
    action send resume
    }
```

(a) RoboChart transitions

```
zoperation ReadingToAnalysis =
  params g∈"SeqGasSensor"
  pre "st= Reading "
  update "[ sts′ =analysis(g)
          ,gs′ = g
          ,st′= Analysis
          ,tr′=tr @ [Event gas, State
          Analysis]
          ,triggers′={}]"
zoperation AnalysisToNoGas =
  pre "st= Analysis ∧ sts=(noGas)"
  update "[ st′= NoGas
          ,tr′=tr @ [Event resume,
          State NoGas]
          ,triggers′={}]"
```

(b) Z-Machine operations

Fig. 5: Examples for mapping Rule 8, 8.1, 8.2, 8.3.

Rule 8.1: For RoboChart events of input and synchronization types that communicate a value to a variable, we use **params** section to let the variable parameter first read a value from a set PARAMETER that represents the possible input values. The variable to be updated will then be assigned the value of parameter in the **update** section.

For example, in transition t0 of Fig. 5, gs is to be updated through event gas of input type. After transformation, the input value is first read by g from the set SeqGasSensor. Then gs is assigned with g for an update.

Rule 8.2: For each operation, there are two possible sources of precondition including transition conditions (e.g., sts = Status::noGas), and the source state of the transition (e.g., st = Reading).

Rule 8.3: The **update** section updates the store variables if involved in this transition. Depending on the transition to be executed, the updated variables vary. However, the observational variables are always updated in each operation:

- st is updated to the target state of the transition.
- triggers is updated to a set of events that may be enabled while the machine is at the target state.
- tr is updated by appending to the list sequentially the trigger event, the exit action of the event type in the source state, the transition action of the event type, the entry action and the during the action of the event type in the target state as appropriate, and the target state.
- For the variable to be updated through an input event, it is assigned with the value of parameter from the **params** section. Besides, if there is an assignment in the same operation using this variable as a function parameter, the assignment shall use parameter instead of the variable itself. For example, variable sts is updated through the assignment sts' = analysis(gs_input) where gs_input is the input parameter for gs.

We explain the rules using examples in Fig. 5. Transition t0 goes from source state Reading to target state Analysis and is transformed into operation ReadingToAnalysis. This transition is triggered by gas; which assigns a value to variable gs; then the source state is exited and state Analysis is entered; and an entry action assigns a new value to sts. Accordingly, in ReadingToAnalysis, **pre** uses the source state as a precondition, i.e., st = Reading. The event gas is of input type, so its value is communicated through the **params** section. In the **update** section, st is updated to Analysis, tr gains two new elements of trigger event Event gas and target state State Analysis. Since there are no trigger events in the following transitions with Analysis as the source state, trigger is an empty set.

E. RoboChart state machine

Rule 9: **State machine** is mapped to a **zmachine** which is defined using the initialisation, invariants and operations. The initialisation assigns uniquely defined values to all the store variables to provide a starting point for the operations. For example, tr is initialized to [State initial]. The values of the state variables are provided by users. Keyword **invariant** points to the set of invariants (including healthiness conditions and the structural invariants) defined in the **zstore** using the naming form of "Machine_name_inv". The invariants are grouped in GasAnalysis_inv. The **zmachine** for GasAnalysis is as follows:

```
definition Init::"GasAnalysis subst" where
"Init = "[gs⇝[]
         ,ins⇝0
         ,sts⇝noGas
         ,anl⇝Front
         ,st ⇝initial
         ,tr ⇝[State initial]
         ,triggers⇝{}]"

zmachine GasAnalysisMachine =
    over GasAnalysis
    init Init
    invariant GasAnalysis_inv
    operations InitialToReading AnalysisToNoGas
        AnalysisToGasDetected GasDetectedToFinal
        GasDetectedToReading ReadingToAnalysis
        NoGasToReading
```

VII. Verification Approach

In this section, we provide the principles to define invariants and requirement specifications, and a method to verify them.

After a RoboChart Z-Machine is generated in Step 1, the developer needs to provide a set of structural invariants and requirement specifications, and then use Isabelle to verify them. The verification strategy is shown in Fig. 6. We first define a set of structural invariants for the Z-Machine as the baseline for requirement proof, and then specify the requirements and prove (or refute) them with the help of the zpog method (see §IV). zpog may verify a property directly, or else provide a set of residual proof obligations. We can then call *sledgehammer* to complete the proof, or *nitpick* to find counterexample traces.

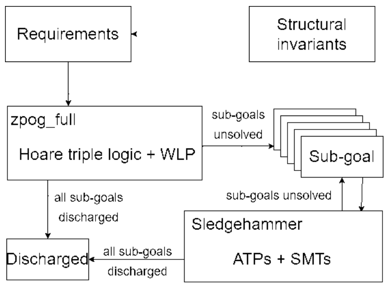

Fig. 6: Z-Machine verification strategy in Isabelle/HOL.

A. *Structural invariants*

We require structural invariants to impose constraints on the state space, which are typically supplied by the user. However, we also implement some simple heuristics to infer invariants for state variables according to some common patterns employed in RoboChart.

A variable can be updated with an assignment of a function value. If the variable is not updated with this assignment in each operation, we can infer an invariant to record the relation between the variable and the function parameters and to impose this relation to all operations so that those operations where the variable is not updated know their relation's existence. Effectively, we are using the operation's frame to infer these invariants.

There are two possible scenarios when a variable is updated using the value of a function. **Case 1**: the variable and the function parameters are updated during the same transition and not updated anywhere else, an invariant shall state that the variable equals the function value for all operations. **Case 2**: The operation updating the variable differs from the one updating the function parameter. An invariant should state that the variable equals the function value in certain states, except where only the parameters are updated.

The invariants are defined in the **where** section of the **zstore**, as shown below:

```
zstore GasAnalysis =
 ...
 where
  hc:"wf_rcstore tr st (Some final)"
  inv_1:"sts = analysis(gs)"
  inv_2:"st=GasDetected⟶ ins=intensity(gs)"
```

We first require that the state space satisfies all the healthiness conditions of §V by the inclusion of the `wf_rcstore` predicate. We then state the structural invariants.

Invariant `inv_1` is an example of Case 1. In operation `ReadingToAnalysis`, the input event `gas` updates the sensor reading list `gs`. The detection status `sts` is updated by `analysis(gs)`. The variable `sts` and the parameter `gs` are updated in the same operation, and at nowhere else. Consequently, `sts = analysis(gs)` is a structural invariant.

Invariant `inv_2` is an example of Case 2. `ins` is updated using `intensity(gs)` in operation `AnalysisToGasDetected`, while `gs` is updated in operation `ReadingToAnalysis`. Thus, the invariant "`ins=intensity(gs)`" shall hold at state `GasDetected`.

We prove that each operation preserves the structural invariants using `zpog`, which is a fully automated process.

B. *Requirements*

After the structural invariants have been established we use a command keyword **zexpr** to define the requirement expressions over the store. These are also specified as a form of state invariant. We can construct the requirements related to events, states, and state variables based on the available store variables. For example, we can use the trace variable `tr` to impose constraints about sequencing of events. Some example requirements from GasAnalysis, and their formalisation as invariant expressions, are given below:

Req1: If no gas is detected, the machine does not terminate. “$sts = noGas \longrightarrow \forall i < \#tr.tr\ !\ i \neq State\ final$”

Req2: The machine shall terminate only on arrival of a stop event. “$\forall\ i < \#tr.tr\ !\ i = State\ final \longrightarrow tr!(i-1) = Event\ stop$”

Req3: Immediately before entering the state Analysis, the event gas happens. “$\forall i < (\#tr - 1).tr\ !(\ i{+}1) = State\ Analysis \longrightarrow tr\ !\ i\ = Event\ gas$”

Req4: Every time the robot changes direction, new readings are taken from sensors. “$\forall\ i < \#tr - 1.tr\ !\ i = Event\ turn \longrightarrow tr\ !\ (i{+}1) = State\ Reading$”

Req5: The readings from the sensors shall be analyzed before the gas detection status is identified. “$\forall i < \#(states\ tr) - 1.\ (states\ tr)\ !\ i = State\ Reading \longrightarrow (states\ tr)\ !\ (i{+}1) = State\ Analysis$”

Req6: If intensity exceeds the threshold, gas detected shall be indicated. “$intensity(gs) \geq thr \longrightarrow sts = gasD$ ”

Here, $\#xs$ denotes the length of list xs and $xs!i$ returns the ith element of xs. $states\ tr$ is defined as $(filter\ is\_State\ tr)$ using functions of $filter$ and is_State. Five types of claims are verified against the state machine in Isabelle including (1) the state reachability driven by the variables, e.g., Req1; (2) the sequential relationship between states and events, e.g., Req2-Req4; (3) the sequential relationship of states, i.e., the correctness of the transition source and target, e.g., Req5; (4) the relationship between state variables, e.g., Req6; and (5) deadlock freedom, which is discussed at the end of this section.

To verify that the Z-Machine model satisfies a requirement, we need to prove that (1) the initialisation establishes the requirement; and (2) each operation preserves the requirement. An excerpt of **Req1** verification is given in Fig. 7.

The initialisation `Init` establishing **Req1** is proved automatically by `zpog` method. The operation `InitialToReading` preserving **Req1** is proved by applying `zpog` method and calling *sledgehammer*. The operation `GasDetectedToFinal` is proved by the `zpog` method but with the support of the invariants `GasAnalysis_inv`. This `GasAnalysis_inv` includes `inv_1`, `inv_2`, and `hc` listed in §VII-A. The lemma for `GasdetectedToFinal` states that

```
{Req1 ∧ GasAnalysis_inv} GasdetectedToFinal() {Req1}
```

```
zexpr Req1 is
"sts=noGas⟶(∀i <(length tr). tr ! i ≠ State final)"

lemma "Init establishes Req1 "
  by zpog_full

lemma "InitialToReading() preserves Req1"
  apply zpog_full
  by (metis St.distinct(29) less_SucE nth_append nth_append_length
  tag.inject(1))

lemma "GasDetectedToFinal() preserves Req1 under GasAnalysis_inv"
  by zpog_full
```

Fig. 7: An excerpt of requirement **Req1** proof.

that is, when the operation is started in a state where both the structural invariants and **Req1** are satisfied, then following execution the requirement still holds.

Although we need to interact with Isabelle/HOL to give the commands and instructions, the proof itself is completely automated. As for the automation of the requirement generation, we generate the verification template, i.e., the **`zexpr`** section with the requirement content left blank and the lemmas for the initialization and each operation. The users need to fill in the requirements manually.

We also verify deadlock freedom of the Z-Machine using the automated method in §IV. For RoboChart, this means that each state has at least one enabled transition when the structural invariants are satisfied. For GasAnalysis, we want to check the state machine only deadlocks when it enters `final` state, i.e., it is deadlock-free when not entering `final` state. We introduce an operation to bypass `final` state and use the `deadlock_free` proof method defined in §IV to verify the deadlock freedom. The bypass operation and the deadlock freedom requirement of GasAnalysis are as follows.

```
zoperation Bypass =
  pre "st= final"

lemma GasAnalysis_deadlock_free:
"deadlock_free GasAnalysisMachine"
    apply deadlock_free
    sledgehammer ...
```

The proof is done by first **`unfolding`** the Z-Machine definition `GasAnalysisMachine`, then applying the `deadlock_free` method to provide the sub-goals, and completing the proof by calling *sledgehammer*.

VIII. Case Study

We implemented all the transformation rules of §V in Eclipse using the Epsilon EGL language [26], and carried out a case study on an Autonomous Underwater Vehicle (AUV) introduced in [25] to illustrate our approach.

A. System description and safety requirements

The AUV can be operated by humans or by the system automatically. Its mission is to perform underwater maintenance and intervention tasks. The main hazards involve potential collisions with subsea system components and infrastructure, which can result from the actions of either the operator or the AUV system. While local path planning utilizes machine learning techniques, the "Last Response Engine" (LRE) safety monitoring component is developed without artificial intelligence to allow safety assurance through formal verification. We apply our RoboChart transformation and verification technique to the LRE.

Fig. 8 shows the RoboChart state machine model of the LRE. The LRE's function is to switch between operating modes of the system based on certain safety conditions. There are four modes: (i) Operator Control Mode (OCM), a manual mode, (ii) Main Operating Mode (MOM), the automatic mode in safe conditions, (iii) High Caution Mode (HCM), the automatic mode used when the collision risk is to be lowered by reducing speed, (iv) Collision Avoidance Mode (CAM), the emergency automatic mode used when the collision risk is too high and needs to be reduced by evasive manoeuvres. Several transitions model the possible moves from one mode to another. For example, the LRE can move from MOM to HCM when the horizontal velocity is greater than or equal to 3, and the distance to a static obstacle is less than a given constant. Moreover, the operator can command the LRE to switch modes using the events reqOCM/MOM/HCM. A detailed description of the system operation can be referred to in [25].

There are two store variables including `pos` for the position of the robot, and `vel` for the velocity. Four constants are defined including `Obsts` as a list of static obstacles, `HCMVel` as the advisory velocity for HCM mode, `MOMVel` as the advisory velocity for MOM mode, `StaticObsDist` as the acceptable distance to the obstacles, `MinSafeDist` as the minimum safe distance to the obstacles, which is assumed to be non-negative.

We also define five functions. Function `inOPEZ` determines whether the robot has entered one of the forbidden areas, called "Object Proximity Exclusion Zones" (OPEZ) with a Boolean value. Function `dist` calculates the distance from the robot to the closest obstacle based on their respective positions. Function `CDA` (Closest Distance of Approach) calculates the closest distance the robot will have with an approaching obstacle in the near future, calculated using both the position and velocity. Function `maneuv` can be used to change the direction of the robot by 90 degrees to avoid collision with an obstacle. Function `setVel` sets the velocity to a new value.

The machine communicates with other components through six events consisting of five input events and one output event.

B. Z-Machine model

The LRE store includes two state variables, the three observational variables and the healthiness conditions. The definitions of the constants, functions, and operations for each transition are omitted here for reasons of space.

```
zstore LRE =
  pos::   "real×real"
  vel::   "real×real"
  st::    "St"
  tr::    "(St, Evt) tag list"
  triggers:: "Evt set"
where
  hc:"wf_rcstore tr st None"
```

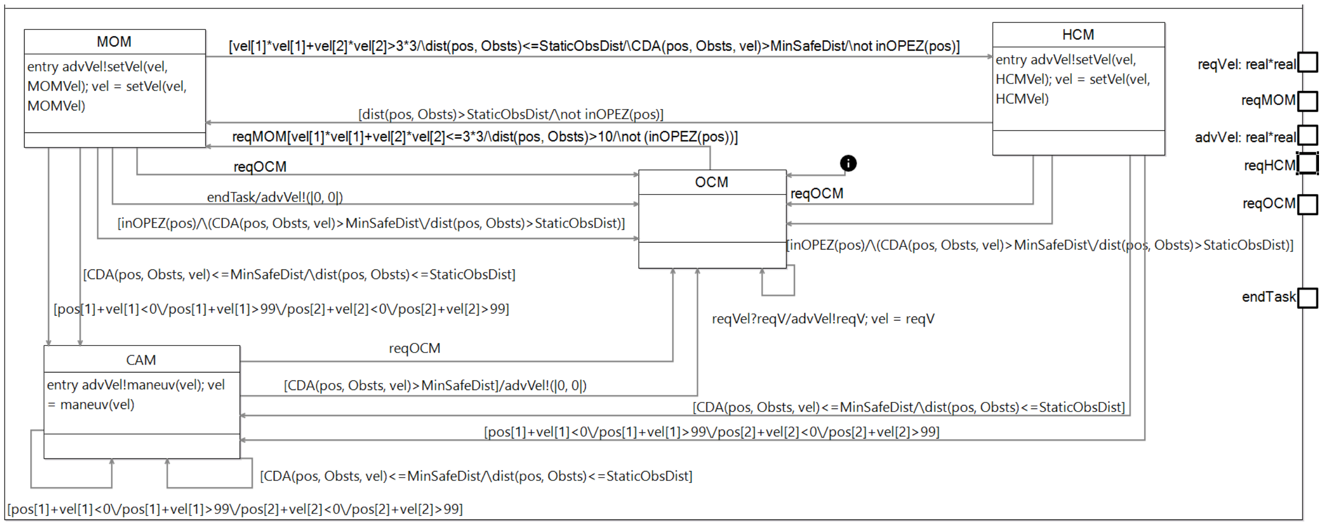
MOM
[vel[1]*vel[1]+vel[2]*vel[2]>3*3/\dist(pos, Obsts)<=StaticObsDist/\CDA(pos, Obsts, vel)>MinSafeDist/\not inOPEZ(pos)]
HCM
entry advVel!setVel(vel, MOMVel); vel = setVel(vel, MOMVel)
entry advVel!setVel(vel, HCMVel); vel = setVel(vel, HCMVel)
reqVel: real*real
reqMOM
[dist(pos, Obsts)>StaticObsDist/\not inOPEZ(pos)]
reqMOM[vel[1]*vel[1]+vel[2]*vel[2]<=3*3/\dist(pos, Obsts)>10/\not (inOPEZ(pos))]
advVel: real*real
reqHCM
reqOCM
OCM
reqOCM
endTask/advVel!(|0, 0|)
reqOCM
[inOPEZ(pos)/\(CDA(pos, Obsts, vel)>MinSafeDist\/dist(pos, Obsts)>StaticObsDist)]
[inOPEZ(pos)/\(CDA(pos, Obsts, vel)>MinSafeDist\/dist(pos, Obsts)>StaticObsDist)]
[CDA(pos, Obsts, vel)<=MinSafeDist/\dist(pos, Obsts)<=StaticObsDist]
endTask
[pos[1]+vel[1]<0\/pos[1]+vel[1]>99\/pos[2]+vel[2]<0\/pos[2]+vel[2]>99]
reqVel?reqV/advVel!reqV; vel = reqV
CAM
reqOCM
entry advVel!maneuv(vel); vel = maneuv(vel)
[CDA(pos, Obsts, vel)>MinSafeDist]/advVel!(|0, 0|)
[CDA(pos, Obsts, vel)<=MinSafeDist/\dist(pos, Obsts)<=StaticObsDist]
[pos[1]+vel[1]<0\/pos[1]+vel[1]>99\/pos[2]+vel[2]<0\/pos[2]+vel[2]>99]
[CDA(pos, Obsts, vel)<=MinSafeDist/\dist(pos, Obsts)<=StaticObsDist]
[pos[1]+vel[1]<0\/pos[1]+vel[1]>99\/pos[2]+vel[2]<0\/pos[2]+vel[2]>99]

Fig. 8: RoboChart state machine of LRE adapted from [25].

According to Fig. 8, the machine needs an input of the current position provided by the robot platform in each state. The position is used as a parameter in several functions. Though we do not model the platform behaviour or the communication with the platform, we construct an operation `Move` to represent the position update as below.

```
zoperation Move =
  update "[pos′ = (fst(pos)+fst(vel),
      snd(pos)+snd(vel))]"
```

This operation simply updates the position using the current velocity. A possible initialization is as below.

```
definition Init::"LRE subset" where
"Init = "[pos⇝(0,0)
        , vel⇝(0,0)
        , st⇝ initial
        , tr⇝ [State initial]
        , triggers⇝{reqOCM}]"
```

C. Safety requirements and Verification

We formalised and verified four safety requirements for the LRE in Isabelle/HOL.

R1: When the LRE is in CAM, it can only change to OCM or re-enter CAM. "$\forall i < \#(states\ \ tr) - 1.(states\ \ tr)\ !\ \ i = State\ CAM \longrightarrow (states\ \ tr)\ !\ \ (i+1) \in \{State\ OCM,\ State\ CAM\}$"

R2: On entering MOM, the LRE shall advise a new velocity with a value between 0.4 m/s and 0.6 m/s for MOM mode. "$st = MOM \longrightarrow (tr\ !\ (\#tr - 2) = Event\ advVel \wedge (xvel^2 + yvel^2 > 0.16) \wedge (xvel^2 + yvel^2 < 0.36))$"

R3: When LRE is not in OCM mode, the trigger *reqOCM* can always be enabled. "$st \neq OCM \longrightarrow reqOCM \in triggers$"

For each requirement of **R1**-**R3**, we prove 18 lemmas including one for initialisation establishing the requirement, and 17 lemmas for operations generated from 16 transitions to preserve the requirement.

During the verification of the LRE, some requirements could be successfully verified but took a long period of more than 20 minutes, and some requirements failed the proving due to timing out. As a result, we optimised the Z-Machine library to improve the proof efficiency. As an example, we introduced an additional lemma stating that a pair-typed state variable x can be rewritten as (x.1, x.2), which allows for the efficient evaluation of function `CDA`. This optimization enabled the discharge of a requirement property that had originally timed out, now taking only around 90 seconds to verify.

The fourth requirement **R4** is to check deadlock freedom of the LRE state machine. The requirement is formalized as below and verified by applying the *deadlock_free* method and calling *sledgehammer*. The proof process took only 20 seconds or so. We tried to verify R4 using the FDR model checker, but FDR does not support the analysis of real numbers but only the abstractions, so the verification failed.

```
lemma R4_LRE_Beh_deadlock_free:
"deadlock_free LREMachine"
  apply deadlock_free
  by (metis St.exhaust_disc)
```

IX. Conclusion

In this paper, we have developed a compositional and automated verification technique for RoboChart state machines using a Z-Machine-based semantics. Our technique overcomes the state explosion problem by representing state machines symbolically in Isabelle/HOL. We view our technique as complementary to model checking, which remains very valuable as a push-button analysis to support early-stage prototyping, with theorem proving following to provide more exhaustive verification.

We give a direct mapping from one transition in RoboChart to one operation in Z-Machines. This means our verification technique is compositional, since we prove that each transition preserves the invariants individually. As a result, when a single transition or state is modified, we need only reverify the corresponding results, with the others remaining untouched. We can also more easily trace back from proof errors in Isabelle to state machine elements, which provides better integration into a software engineering workflow. Moreover, compositionality can potentially support incremental development and verification, where models are developed and verified one component at a time.

Most of our results are applicable to any EMF-based language with a formal semantics that can be mechanised in Isabelle/HOL. For example, a new version of SysML will soon be released with a formal semantics[3], and we could readily apply our pipeline to this with suitable model transformations.

Our transformation in this work supports only basic state machines of RoboChart. Our future work will investigate the incorporation of communication between RoboChart components and hierarchical state machines in the approach. We also plan to verify the transformation using the baseline CSP semantics, to ensure its correctness. The trace variable, tr, currently covers only the state and events, and so we will explore extending this data structure to cover assignments and operations. We are also interested in realising a bidirectional communication between Eclipse-based development tools, such as RoboTool, and Isabelle/HOL in order to facilitate incremental MBE development and to avoid users' direct interaction with Isabelle/HOL.

Acknowledgment

We thank Colin O'Halloran and Nick Tudor from D-RisQ Software Systems for providing the LRE case study.

References

[1] H. Bourbouh, M. Farrell, A. Mavridou, I. Sljivo, G. Brat, L. A. Dennis, and M. Fisher, "Integrating formal verification and assurance: an inspection rover case study," in *NASA Formal Methods Symposium*. Springer, 2021, pp. 53–71.

[2] H. Mkaouar, B. Zalila, J. Hugues, and M. Jmaiel, "A formal approach to AADL model-based software engineering," *International Journal on Software Tools for Technology Transfer*, vol. 22, no. 2, pp. 219–247, 2020.

[3] A. Johnsen, K. Lundqvist, P. Pettersson, and O. Jaradat, "Automated verification of AADL-specifications using UPPAAL," in *IEEE 14th International Symposium on High-Assurance Systems Engineering (HASE), 25-27 Oct. 2012, Omaha, NE, USA*, 2012, pp. 130–138.

[4] H. Kausch, M. Pfeiffer, D. Raco, and B. Rumpe, "Model-based design of correct safety-critical systems using Dataflow languages on the example of SysML architecture and behavior diagrams," in *Software Engineering (Satellite Events)*, 2021.

[5] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, and J. Woodcock, "RoboChart: modelling and verification of the functional behaviour of robotic applications," *Software & Systems Modeling*, vol. 18, no. 5, pp. 3097–3149, 2019.

[6] K. Ye, A. Cavalcanti, S. Foster, A. Miyazawa, and J. Woodcock, "Probabilistic modelling and verification using RoboChart and PRISM," *Software and Systems Modeling*, vol. 21, no. 2, pp. 667–716, 2022.

[7] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe, "FDR3: a parallel refinement checker for CSP," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 2, pp. 149–167, 2016.

[8] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: Probabilistic symbolic model checker," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 2002, pp. 200–204.

[9] S. Böhme and T. Nipkow, "Sledgehammer: judgement day," in *International Joint Conference on Automated Reasoning*. Springer, 2010, pp. 107–121.

[10] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[11] A. S. A. Hadad, C. Ma, and A. A. O. Ahmed, "Formal verification of AADL models by Event-B," *IEEE Access*, vol. 8, pp. 72 814–72 834, 2020.

[12] L. Zou, N. Zhany, S. Wang, M. Fränzle, and S. Qin, "Verifying Simulink diagrams via a hybrid hoare logic prover," in *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*. IEEE, 2013, pp. 1–10.

[13] S. Foster, J. Baxter, A. Cavalcanti, A. Miyazawa, and J. Woodcock, "Automating verification of state machines with reactive designs and Isabelle/UTP," in *International Conference on Formal Aspects of Component Software*. Springer, 2018, pp. 137–155.

[14] L. Zou, N. Zhan, S. Wang, and M. Fränzle, "Formal verification of Simulink/Stateflow diagrams," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2015, pp. 464–481.

[15] J. Liu, J. Lv, Z. Quan, N. Zhan, H. Zhao, C. Zhou, and L. Zou, "A calculus for hybrid CSP," in *Asian Symposium on Programming Languages and Systems*. Springer, 2010, pp. 1–15.

[16] S. Foster, C.-K. Hur, and J. Woodcock, "Formally verified simulations of state-rich processes using Interaction Trees in Isabelle/HOL," in *32nd Intl. Conf. on Concurrency Theory (CONCUR)*, ser. LIPIcs, vol. 203. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

[17] K. Ye, S. Foster, and J. Woodcock, "Formally verified animation for RoboChart using Interaction Trees," in *International Conference on Formal Engineering Methods*. Springer, 2022, pp. 404–420.

[18] J. A. Hilder, N. D. Owens, M. J. Neal, P. J. Hickey, S. N. Cairns, D. P. Kilgour, J. Timmis, and A. M. Tyrrell, "Chemical detection using the receptor density algorithm," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 6, pp. 1730–1741, 2012.

[19] M. Spivey, *The Z-Notation - A Reference Manual*. Englewood Cliffs, N. J.: Prentice Hall, 1989.

[20] J.-R. Abrial, *The B-Book: assigning programs to meanings*. Cambridge University Press, 1996.

[21] R.-J. Back and R. Kurki-Suonio, "Decentralization of process nets with centralized control," *Distributed Computing*, vol. 3, pp. 73–87, June 1989.

[22] J. C. P. Woodcock and A. L. C. Cavalcanti, "A Concurrent Language for Refinement," in *IWFM'01: 5th Irish Workshop in Formal Methods*, ser. BCS Electronic Workshops in Computing, A. Butterfield and C. Pahl, Eds., Dublin, Ireland, July 2001.

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

[24] M. Wenzel, "Interaction with formal mathematical documents in Isabelle/PIDE," in *CICM*, ser. LNCS 11617. Springer, 2019, pp. 1–15.

[25] S. Foster, Y. Nemouchi, C. O'Halloran, K. Stephenson, and N. Tudor, "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*, 2020, pp. 11–21.

[26] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The epsilon transformation language," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2008, pp. 46–60.

[3] https://github.com/Systems-Modeling/SysML-v2-Release