



Deposited via The University of York.

White Rose Research Online URL for this paper:

<https://eprints.whiterose.ac.uk/id/eprint/188566/>

Version: Accepted Version

---

**Proceedings Paper:**

Ye, Kangfeng, Foster, Simon and Woodcock, Jim (2022) Formally Verified Animation for RoboChart Using Interaction Trees. In: Riesco, Adrian and Zhang, Min, (eds.) Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Proceedings. 23rd International Conference on Formal Engineering Methods, ICFEM 2022, 24-27 Oct 2022 Lecture Notes in Computer Science. Springer, ESP, pp. 404-420.

[https://doi.org/10.1007/978-3-031-17244-1\\_24](https://doi.org/10.1007/978-3-031-17244-1_24)

---

**Reuse**

Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

**Takedown**

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing [eprints@whiterose.ac.uk](mailto:eprints@whiterose.ac.uk) including the URL of the record and the reason for the withdrawal request.

# Formally Verified Animation for RoboChart using Interaction Trees

Kangfeng Ye, Simon Foster, and Jim Woodcock

University of York, York, UK

{kangfeng.ye, simon.foster, jim.woodcock}@york.ac.uk

**Abstract.** RoboChart is a core notation in the RoboStar framework. It is a timed and probabilistic domain-specific and state machines based language for robotics. RoboChart supports shared variables and communication across entities in its component model. It has a formal denotational semantics given in CSP. Interaction Trees (ITrees) is a semantic technique to represent behaviours of reactive and concurrent programs interacting with their environments. Recent mechanisations of ITrees along with ITree-based CSP semantics and a Z mathematical toolkit in Isabelle/HOL bring new applications of verification and animation for state-rich process languages, such as RoboChart. In this paper, we use ITrees to give RoboChart the first operational semantics, implement it in Isabelle, and use Isabelle's code generator to generate verified and executable animations. We illustrate our approach using an autonomous chemical detector model. With animation, we show two concrete scenarios when the robot encounters different environmental inputs.

## 1 Introduction

The RoboStar<sup>1</sup> framework [1] brings modern modelling and verification technologies into software engineering for robotics. In this framework, artefacts, including platform, environmental, design, and simulation models, are given formal semantics in a unified semantic framework. So correctness of simulation is guaranteed with respect to a design model in a particular platform and environment using a variety of analysis technologies including model checking, theorem proving, and testing. Additionally, modelling, semantics generation, verification, simulation, and testing are automated and integrated in an Eclipse-based tool, RoboTool.<sup>2</sup> The core of RoboStar is RoboChart [2, 3], a timed and probabilistic domain-specific language for robotics, which provides UML-like architectural and state machine modelling notations. RoboChart is distinctive in its formal semantics [2–4], which enables automated verification using model checking and theorem proving [5].

Previous work [2] gives RoboChart a denotational semantics based on the CSP process algebra [6, 7]. This paper defines a first operational semantics for


---

<sup>1</sup> [robostar.cs.york.ac.uk](http://robostar.cs.york.ac.uk)

<sup>2</sup> [robostar.cs.york.ac.uk/robotool/](http://robostar.cs.york.ac.uk/robotool/)

RoboChart using Interaction Trees (ITrees) [8]. ITrees are coinductive structures, and can model infinite behaviours of a reactive system interacting with its environment. ITrees are a powerful semantic technique for development of formal semantics that allows to unify trace-based failures-divergences semantics [7,9] for CSP and transition-based operational semantics, and so unifies verification and animation [10]. The ITree-based semantics for CSP is also sound with respect to the standard failures-divergences semantics [10].

The existing implementation of RoboChart’s semantics in RoboTool is restricted to machine readable CSP (or CSP-M) for verification with FDR [11], and so only a subset of RoboChart’s rich types and expressions can be supported and quantified predicates cannot be solved. Our contribution is here a richer ITree-based CSP semantics for RoboChart to address these restrictions. Our semantics also allows us to characterise systems with an infinite number of states symbolically. We mechanise the semantics in Isabelle/HOL and then utilise the code generator [12] to produce Haskell code for animation. Our animation is formally verified with respect to RoboChart’s semantics in ITrees.

Our technical contributions are as follows: we (1) implement extra CSP operators (interrupt, exception, and renaming) to deal with the RoboChart semantics and a bounded sequence type for code generation; (2) define an ITree-based operational semantics for RoboChart; (3) implement the semantics of a RoboChart model for a case study; and (4) animate the model. With our mechanisation and the animation, we have detected a number of issues in this RoboChart model. All definitions and theorems in this paper are mechanised and accompanying icons () link to corresponding repository artifacts.

The remainder of this paper is organised as follows. In Sect. 2, we introduce RoboChart through an autonomous chemical detector example. Section 3 briefly describes the mechanisation of ITrees in Isabelle and presents the additional CSP operators in detail. Then we present the RoboChart semantics in ITrees in Sect. 4 and use the chemical detector as an example to illustrate animation in Sect. 5. We review related work in Sect. 6 and conclude in Sect. 7.

## 2 RoboChart

*Modelling* We describe features of RoboChart for modelling controllers of robots using as an example an autonomous chemical detector [2, 13, 14]. The robot is equipped with sensors to (1) analyse the air to detect dangerous gases; (2) detect obstacles; and (3) estimate change in position (using an odometer). The controller of the robot performs a random walk with obstacle avoidance. Upon detection of a chemical with its intensity above a threshold, the robot drops a flag and stops there. This model<sup>3</sup> [2] has been studied and analysed in RoboTool, using FDR4,<sup>4</sup> a CSP refinement checker.

<sup>3</sup> [https://robostar.cs.york.ac.uk/case\\_studies/autonomous-chemical-detector/autonomous-chemical-detector.html](https://robostar.cs.york.ac.uk/case_studies/autonomous-chemical-detector/autonomous-chemical-detector.html)

<sup>4</sup> <https://cocotec.io/fdr/>

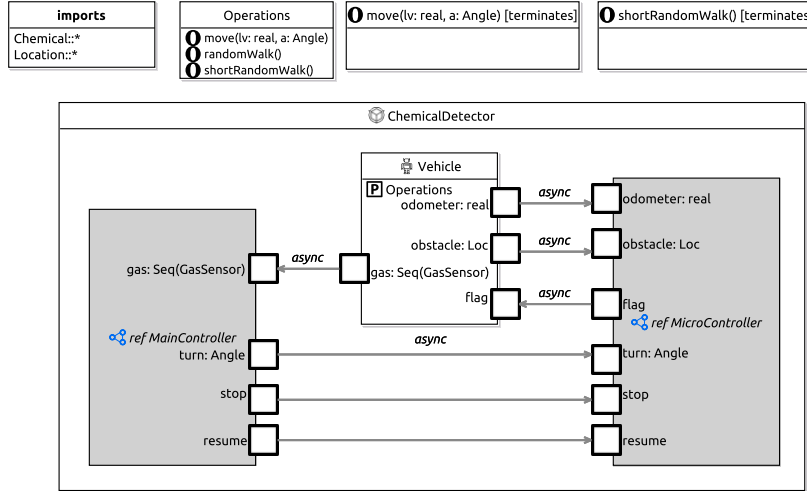


Fig. 1. The module of the autonomous chemical detector model.

The top level structure of a RoboChart model, a module, is shown in Fig. 1. The module `ChemicalDetector` contains a robotic platform `Vehicle` and two controller references `MainController` and `MicroController`. The physical robot is abstracted into the robotic platform through variables, events, and operations. The platform provides the controllers with services (1) to read its sensor data through three events: `gas`, `obstacle`, and `odometer`; (2) for movement through three operations: `move`, `randomWalk`, and `shortRandomWalk` as grouped in an interface `Operations`; and (3) to drop a flag through receiving a `flag` event.

A platform and controllers communicate using directional connections. For example, the platform is linked to `MainController` through an asynchronous connection on event `gas` of type `seq(GasSensor)`, sequences of type `GasSensor`. Furthermore, the `MainController` and `MicroController` interact using the events `turn`, `stop`, and `resume`.

The types used in the module are defined in the two imported packages: `Chemical` and `Location` (whose diagrams are omitted here for simplicity). The two packages declare primitive types `Chem` and `Intensity`, enumerations `Status`, `Angle`, and `Loc`, a record `GasSensor` containing two fields (`c` of type `Chem` and `i` of type `Intensity`), and six functions, of which two are specified using preconditions and postconditions, and four are unspecified. An operation `changeDirction`, with a parameter `l` of type `Loc` and a constant `lv`, is also defined using a state machine.

`MicroController` is implemented using the `Movement` state machine shown in Fig. 2, and `MainController` using another state machine called `GasAnalysis`.

The machine `Movement` declares various constants such as `lv` for linear velocity, four variables (`a`, `d0`, `d1`, and `l`) for preservation of values (angle, odometer readings, and location) carried on communication, and a clock `T`. The machine also contains a variety of nodes: one initial junction (Ⓢ), seven normal states such as `Waiting` and `Going`, and a final state (Ⓣ). A state may have an entry



Interaction trees (ITrees) [8] are a data structure for modelling reactive systems that interact with their environment through events. They are potentially infinite and defined as coinductive trees [15] in Isabelle/HOL.


```
codatatype ('e, 'r) itree =
  Ret 'r | Sil "(('e, 'r) itree)" | Vis "'e  $\rightarrow$  ('e, 'r) itree"
```

ITrees are parameterised over two types: 'e for events ( $E$ ), and 'r for return values or states ( $R$ ). Three possible interactions are provided: (1) *Ret*  $x$ : termination ( $\surd_x$ ) with a value  $x$  of type  $R$  returned; (2) *Sil*  $P$ : an internal silent event ( $\tau P$ , for a successor ITree  $P$ ); or (3) *Vis*  $F$ : a choice among several visible events represented by a partial function  $F$  of type  $E \rightarrow (E, R)\text{itree}$ . Partial functions are part of the Z toolkit<sup>5</sup> [16] which is also mechanised in Isabelle/HOL.

Deterministic CSP processes can be given an executable semantics using ITrees. Previously, the following CSP processes and operators are defined: (1) basic processes: *skip*, *stop*, and *div*; (2) input prefixing  $c?x$ ; (3) output prefixing  $c!v$ ; (4) *guard*  $b$ ; (5) external choice  $P \square Q$ ; (6) parallel composition  $P \parallel_E Q$ ; (7) hiding  $P \setminus A$ ; (8) sequential composition  $P \circledast Q$ ; (9) *loop* and *iterate*.

Here, we give an ITree semantics to three extra CSP operators to allow us to give an ITree-based semantics to RoboChart. The interrupt, exception, and renaming operators are used in the RoboChart's semantics to allow interruption of a during action, termination of a state machine, a controller, or a module, and alphabet transformation of processes. We restrict ourselves to deterministic operators as it makes animation of large models more efficient.

The first operator we introduce is interrupt [6, 7],  $P \triangle Q$ , which behaves like  $P$  except that if at any time  $Q$  performs one of its initial events then it takes over. We present partial functions as sets below. This operator, along with the other two, are defined corecursively, which allows them to operate on the infinite structure of an ITree. In corecursive definitions, every corecursive call on the right-hand side of each equation must be guarded by an ITree constructor.

**Definition 3.1 (Interrupt).** 

$$\begin{aligned} (\text{Sil } P') \triangle Q &= \text{Sil } (P' \triangle Q) & P \triangle (\text{Sil } Q') &= \text{Sil } (P \triangle Q') \\ (\text{Ret } x) \triangle Q &= \text{Ret } x & P \triangle (\text{Ret } x) &= \text{Ret } x \\ (\text{Vis } F) \triangle (\text{Vis } G) &= \text{Vis } (\{e \mapsto (P' \triangle Q) \mid (e \mapsto P') \in (\text{dom}(G) \triangleleft F)\} \oplus G) \end{aligned}$$


In the definition,  $\triangleleft$  is called the domain anti-restriction, and  $A \triangleleft R$  denotes the domain restriction of relation  $R$  to the complement of set  $A$ . The *Sil* cases allow  $\tau$  events to happen independently with priority and without resolving  $\triangle$ . The *Ret* cases terminate with the value  $x$  returned from either left or right side of  $\triangle$ .

For the *Vis* cases, it is also *Vis* constructed from an overriding  $\oplus$  of the further two sets, representing two partial functions. In the partial function,  $(\text{dom}(G) \triangleleft F)$  restricts the domain of  $F$  to the complement of the domain of  $G$ . The first partial function denotes that an initial event  $e$  of  $P$ , that is not the initial event of  $Q$ , can occur independently (without resolving the interrupt) and its continuation

<sup>5</sup> [https://github.com/isabelle-utp/Z\\_Toolkit](https://github.com/isabelle-utp/Z_Toolkit).

is a corecursive call  $P' \triangleleft Q$ . The second function is just  $G$ , which denotes that the initial events of  $Q$  can happen no matter whether they are in  $F$  or not. This means if  $P$  and  $Q$  share events,  $Q$  has priority. This prevents nondeterminism.

Next, we present the exception operator,  $P \llbracket A \triangleright Q$ , which behaves like  $P$  initially, but if  $P$  ever performs an event from the set  $A$ , then  $Q$  takes over.

**Definition 3.2 (Exception).** 

$$\begin{aligned} (Ret\ x) \llbracket A \triangleright Q &= Ret\ x & (Sil\ P') \llbracket A \triangleright Q &= Sil\ (P' \llbracket A \triangleright Q) \\ (Vis\ F) \llbracket A \triangleright Q &= Vis\ \left( \begin{aligned} &\{e \mapsto (P' \llbracket A \triangleright Q) \mid (e \mapsto P') \in (A \triangleleft F)\} \oplus \\ &\{e \mapsto Q \mid e \in (A \cap \text{dom}(F))\} \end{aligned} \right) \end{aligned}$$


The *Ret* case terminates immediately with the value  $x$  returned, and  $Q$  will not be performed. The *Sil* case allows the  $\tau$  event to be consumed.

Similar to Definition 3.1, the *Vis* case is also represented by the overriding of two partial functions. The first partial function represents that an initial event  $e$  of  $P$ , that is not in  $A$  (that is,  $e \in \text{dom}(A \triangleleft F)$ ), can occur independently and its continuation is a corecursive call  $P' \llbracket A \triangleright Q$ . Following execution of an initial event  $e$  of  $P$  that is in  $A$  (that is,  $e \in (A \cap \text{dom}(F))$ ), the exception behaves like  $Q$ , which is expressed by the second partial function.

The last operator we define for this work is renaming,  $P \llbracket \rho$ , which renames events of  $P$  according to the renaming relation  $\rho : E_1 \leftrightarrow E_2$ . We note this relation is possibly heterogeneous, and so  $E_1$  and  $E_2$  are different types of events. First, we define an auxiliary function for making a relation functional by removing any pairs that have duplicate distinct values. This is the case when the renaming relation, restricted to the initial events of  $P$ , is functional.

$$mk\_functional(R) = \{(x, y) \in R. \forall y'. (x, y') \in R \Rightarrow y = y'\}$$

This produces the minimal functional relation that is consistent with  $R$ . For example,  $mk\_functional(\{e_1 \mapsto e_2, e_1 \mapsto e_3, e_2 \mapsto e_3\}) = \{e_2 \mapsto e_3\}$ . This function is used to avoid nondeterminism introduced by renaming multiple events to a same event. We use this function to define the renaming operator.

**Definition 3.3 (Renaming).** 

$$\begin{aligned} (Ret\ x) \llbracket \rho &= Ret\ x & (Sil\ P') \llbracket \rho &= Sil\ (P' \llbracket \rho) \\ (Vis\ F) \llbracket \rho &= \left( \begin{aligned} &\mathbf{let}\ G = F \circ mk\_functional\ ((\text{dom}(F) \triangleleft \rho) \sim) \\ &\bullet\ Vis\ (\{e_2 \mapsto (P' \llbracket \rho) \mid (e_2 \mapsto P') \in G\}) \end{aligned} \right) \end{aligned}$$

The *Ret* case behaves like  $P$  and the renaming has no effect on it. The *Sil* case allows  $\tau$  events to be consumed, since they are not subject to renaming.


In the *Vis* case,  $G$  is a partial function  $(E_2 \rightarrow (E_1, R)itree)$  that is the backward partial function composition  $\circ$  of  $F$  and a partial function made using  $mk\_functional$  from the inverse  $\sim$  of the relation  $(\text{dom}(F) \triangleleft \rho)$  which is the domain restriction  $\triangleleft$  of  $\rho$  to the domain  $\text{dom}(F)$  of  $F$ . Basically, the multiple events of  $E_1$  that are mapped to a same event of  $E_2$  in  $\rho$  and also are the initial events of  $P$ , or in  $\text{dom}(F)$ , are removed in  $G$ . The renaming result is a partial function in which each event  $e_2$  in the domain of  $G$  is mapped to a renamed process by a corecursive call  $P' \llbracket \rho$  where  $(e_2 \mapsto P') \in G$ .


## 4 RoboChart semantics in interaction trees


In this section, we describe how we give a semantics to RoboChart in terms of ITrees in Isabelle/HOL. These include types, instantiations, functions, state machines, controllers, and modules. In the implementation of RoboChart's semantics, we also take into account the practical details of the CSP semantics generation in RoboTool, such as naming and bounded primitive types.

*Types.* RoboChart has its type system based on that of the Z notation [17]. It supports basic types: `PrimitiveType`, `Enumeration`, records (or schema types), and other additional types from the mathematical toolkits of Z.

The core package of RoboTool provides five primitive types: boolean, naturals, integers, real numbers, and string. We map integers, naturals, and strings onto the corresponding types in Isabelle/HOL, but with support for code generation to target language types. This improve efficiency of evaluation and thus animation. RoboChart models can also have abstract primitive types with no explicit constructors, such as `Chem` and `Intensity` in the chemical detector model presented in Sect. 2. We map primitive types to finite enumerations for the purpose of code generation. We define a finite type `PrimType` parametrised over two types: `'t` for specialisation and a numeral type `'a` for the number of elements.

`datatype ('t, 'a::finite) PrimType = PrimTypeC 'a` 

For example, `Chem` is implemented as a generic type `(ChemT, 'a) PrimType` . An example for a finite type `'a` is the numeral type in Isabelle, such as type `2` which contains two elements: zero (`0::2`) and one (`1::2`).

For enumerations and records, we use `datatype` and `record` in Isabelle. For finite sequences such as `Seq(GasSensor)`, we also bound the length of each sequence in this type. We define bounded lists or sequences `( 'a, 'n::finite) blist`  over two parametrised types: `'a` for the type of elements and a finite type `'n` for the maximum length of each list. For `Seq(GasSensor)`, its bounded type in Isabelle is `(GasSensor, 2) blist` with length bounded to 2.

For other types, we have their counterparts in the Z toolkit.

*Instantiations.* The `instantiation.csp` file of the CSP semantics contains common definitions used by all models for verification using FDR. These include the definitions of bounded core types (named types in CSP) and arithmetic operators under which these bounded types are closed. We show below one example for type `core_int` and one for the plus operator, closed under a bounded type `T`.

```
nametype core_int = {-2 .. 2}
Plus(e1, e2, T) = if member(e1 + e2, T) then (e1 + e2) else e1
```

That is, if `e1+e2` is within `T` then this is the result, otherwise it is `e1`. We use `locale` [18] in Isabelle to define these for reuse in all models. Locales allow us to characterise abstract parameters (such as `min_int` and `max_int`, to define bounded core type `core_int`) and assumptions in a local theory context.

$f$ : intensity(gs: Seq(GasSensor)): Intensity
◀ size(gs)>0
▶ forall x: nat   0<=x/\x<size(gs) @ goreq(result, gs[x].i)
▶ exists y: nat   0<=y/\y<size(gs) @ result==gs[y].i

**Fig. 3.** The specification of function intensity.

*Functions.* Functions in RoboChart benefit from the rich expressions in Isabelle and the Z toolkit in Isabelle. The expressions that are not supported in CSP-M such as logical quantifications are naturally present in Isabelle. Using the code generator, the preconditions and postconditions of a function definition can be solved effectively, while this is not possible in CSP-M and FDR.

An example function is `intensity`, shown in Fig. 3, defined in the chemical detector model, whose two postconditions (▶) involve universal and existential quantifications where  $\textcircled{C}$  separates constraint and predicate parts, and `goreq` is a  $\geq$  relation on intensities. The result of the function is the largest intensity in `gs`. The precondition (◀) is that the length (`size`) of the parameter `gs` is larger than 0. For verification with FDR in RoboTool, an explicit implementation of this function must be supplied. Our definition of this function in Isabelle, however, is directly from its specification and is shown below.

```

definition "pre_Chemical_intensity gs = (blength gs > 0)"
definition "Chemical_intensity gs = (THE result.
  (∀x::nat<blength gs. Chemical_goreq(result, gs_i (bnth gs x)))∧
  (∃x::nat<blength gs. result = gs_i (bnth gs x)))"

```

In the definitions, `blength gs` gives the length of a bounded sequence `gs`, `bnth gs n` gives the `n`th element in `gs`, and `gs_i` returns the field value in a record of type `GasSensor`. The two definitions are straightforward except that a definite description (`THE result`, denoting the unique `result` such that the predicate holds) is used to return the `result`. We have two definitions here corresponding to the definition of `intensity`: one for its precondition and one for its postconditions. This is due to the semantics of such a function `f` in RoboChart: a boolean guard (`pre(f) & P`) where `pre(f)` is the preconditions of `f` and `f` is called in process `P`, and so if the preconditions are not satisfied, the semantics deadlocks.

We note that there is an error in the definition of `intensity` in the original model where  $\leq$  (instead of  $<$ ) is used for comparison between `x` (and `y`) and `size(gs)`. This is because sequences are zero-indexed. Our animation detects this error and so we have fixed it. Similarly, we also found another error in the postcondition of the function `location`: the postcondition is not strong enough to identify a unique result of the function.

*State machine definitions.* The RoboChart semantics of a state machine is a parallel composition of memory processes for its variables (`MemoryVar`) and transitions (`MemoryTrans`), and a process (`STM`) for its behaviour with internal events hidden and also catering for its termination using the exception operator.

`STM` is a parallel composition of the behaviour (`STM_I`) for its initial junction and the restricted behaviour (`S_R`) for each state `S` synchronising on state enter-

ing and exiting events. A state's behaviour  $S$  involves the entering of this state, the execution of its during action, and the execution of one of its transitions. The execution of a transition exits the state, executes the action of the transition, and enters the target state of the transition. Not all transitions are available for  $S$ , such as the transitions from sibling states of  $S$  and substates of  $S$ . These transitions are excluded in the restricted behaviour  $S\_R$ .

The state machine semantics uses a general type `InOut` for the direction of an event in a connection, two data types for state and transition identifiers (`SIDS` and `TIDS`), and an event alphabet ( $E$ ) for the process of this state machine. The event alphabet is represented by the parametrised type  $E$  of events which is declared through the `chantype` command.  $E$  is expressed by a finite set of channels declared in the command. We show below an example for the `Movement` machine in Fig. 2.

```

datatype InOut = din | dout
datatype SIDS_Movement = SID_Movement|SID_Movement_Waiting|...
datatype TIDS_Movement = TID_Movement_t1|TID_Movement_t2|...
chantype Chan_Movement = internal_Movement :: TIDS_Movement
  terminate_Movement::unit
  enter_Movement::"SIDS_Movement×SIDS_Movement" ...
  get_l_Movement::"Location_Loc"  set_l_Movement::"Location_Loc"
  obstacle__Movement::"TIDS_Movement×InOut×Location_Loc"
  obstacle_Movement::"InOut×Location_Loc" ...
  moveCall_Movement::"core_real×Chemical_Angle" ...

```

The channel type of `Movement` includes four kinds of channels. Firstly, flow control channels include (a) `internal`<sup>6</sup> for transitions without a trigger; (b) `enter`, `entered`, `exit`, and `exited` for the entering and exiting of a state; and (c) `terminate` for the termination of the machine. Secondly, variable channels contain a `set` and a `get` channel for each variable with an additional `set_EXT` for each shared variable to accept an external update. Thirdly, event channels include two channels for each event of the machine, one such as `obstacle` for the event `obstacle` used in actions of RoboChart and another such as `obstacle_` (with an additional `TIDS` for its type) for the event `obstacle` used as triggers of transitions. The distinction of two event channels (`obstacle` and `obstacle_`) for each event (`obstacle`) is necessary because the guard of a transition is evaluated in `MemoryTrans`, and so only the trigger event (not action event) of the transition is subject to the guard evaluation, and, therefore, has a new channel (`obstacle_`) with a transition id. We note, however, that events `obstacle_.tid` of this new channel are eventually renamed to the event channel `obstacle` in the process for the machine. Fourthly, operation call channels include a channel such as `moveCall` for each call to the operation `move` provided by the platform.

The memory process `Memory_x` for a shared variable  $x$  is shown below.

$$\text{loop } (\lambda v. \text{get\_x!}v \rightarrow \checkmark_v \square \text{set\_x?}x \rightarrow \checkmark_x \square \text{set\_EXT\_x?}x \rightarrow \checkmark_x)$$

<sup>6</sup> In the Isabelle code, we include suffixes to ensure that names do not collide, but omit them here

The process is an infinite loop. It provides three choices: output the value  $v$  on `get_x` without updating the variable and accept a local (or external) update of the variable through `set_x` (or `set_EXT_x`).

The memory process for transitions of the state machine `Movement` in Fig. 2 is partially (3 in 24 transitions) illustrated below.

```
loop (λid. internal!TID_t1 → ✓id □ resume!(TID_t0,din) → ✓id □
      (get_d1?d1 → get_d0?d0 →
        (d1-d0>stuckDist)&(internal!TID_t12 → ✓id) □ ...)
```

The state of this loop process is a constant `id` (used to identify a RoboChart module). Each choice corresponds to a transition: (1) the first choice for the default transition `t1` from the initial junction to state `Waiting` which has no trigger (hence `internal`); (2) the second for the self transition of state `Waiting` whose trigger is `resume` (an input so `din`); and (3) the third for the transition from `AvoidingAgain` to `Avoiding` whose guard is `(d1-d0>stuckDist)` (time semantics is ignored) and evaluated in this memory transition process.

The process  $S$  for the behaviour of a state  $S$  is sketched below.

$$\begin{aligned}
S(id) &= \text{enter?sd} : OSIDS \rightarrow S\_exec(id, \text{fst } sd) \\
S\_exec(id, s) &= S\_entry \ ; \ \text{entered}!(s, SID\_S) \ ; \\
&\left( \left( S\_during \ ; \ stop \right) \Delta \right. \\
&\quad \left( \left( \begin{array}{l} \square t : sTrans \bullet \left( \begin{array}{l} e\_t?(TID\_t, \_) \rightarrow \text{exit!pSID\_S} \rightarrow S\_exit \ ; \\ \text{exited!pSID\_S} \rightarrow \text{enter!pSID\_S} \rightarrow \\ S\_exec(id, SID\_S) \end{array} \right) \square \\ \square t : oTrans \bullet \left( \begin{array}{l} \dots \ ; \ \text{exited!pSID\_S} \rightarrow \text{enter}!(SID\_S, SID\_td) \\ \rightarrow \text{entered}!(SID\_S, SID\_td) \rightarrow S(id) \end{array} \right) \square \\ \square e\_ : EvtChs \bullet \left( \begin{array}{l} e\_?(TID\_t, \_) \rightarrow \text{exit?sd} : OSIDS \rightarrow \\ (S\_exit \ ; \ \text{exited}?(fst\ sd, SID\_S) \rightarrow S(id)) \end{array} \right) \square \end{array} \right) \right)
\end{aligned}$$

$S$  and  $S\_exec$  are defined by mutual recursion. Initially,  $S$  accepts *entering* from other nodes of the state machine containing  $S$  where  $OSIDS$  denotes a set of pairs  $(oSID, SID\_S)$  ( $oSID$  is  $SID$  for one of other nodes and  $SID\_S$  is  $SID$  for  $S$ ). Afterwards, the behaviour of  $S$  is given by  $S\_exec$  with its second argument being the other node (the first element of  $sd$ ).


$S\_exec$ , with a parameter  $s$  denoting the node entering  $S$ , executes the entry action of  $S$  first, if any, denoted by  $S\_entry$ . Then  $S$  is *entered*. After that, the behaviour is given by an interrupt. The during action of  $S$  ( $S\_during$ ) can be executed if none of the initial events of the right side (external choices) of the interrupt is performed, that is, none of the self transitions  $sTrans$  of  $S$  or other transitions  $oTrans$  from  $S$  is taken, or none of trigger events  $EvtChs$  of the state machine containing  $S$  is signalled. If, however, any of these transitions is taken or any of these trigger events is signalled, then the during action is interrupted. A *stop* process after  $S\_during$  prevents the interrupt from being terminated and so interruption is always possible.

For each  $t$  of  $sTrans$ , it behaves as follows: (1) the corresponding event channel  $e\_t$  for its trigger event, such as `obstacle_`, synchronises on  $t$  (iden-

tified by  $TID\_t$ ) only; (2)  $S$  starts to *exit* by itself where  $pSID\_S$  denotes  $(SID\_S, SID\_S)$ ; (3) the exit action of  $S$ , denoted by  $S\_exit$ , is executed; (4)  $S$  is *exited*; (5)  $t$  starts to *enter*  $S$  again because it is a self transition; and (6) finally  $S\_exec$  is recursively called with the source state  $s$  being  $SID\_S$ .

For each  $t$  of  $oTrans$ , the early behaviour is the same as above and so it is omitted (...). After  $S$  is *exited*,  $t$  starts to *enter* its target from  $S$ , identified by  $SID\_td$ , and then the target is *entered*. Finally,  $S$  returns to its initial state ( $S(id)$  is called) and accepts a further *enter* request.

For each trigger event  $e$  of the state machine containing  $S$ , there is a corresponding additional event channel  $e\_$  declared in the channel type of the machine. The set of these channels are denoted by  $EvtChs$ . If this event  $e$  of a transition  $t$  is signalled ( $e?(TID\_t, \_)$ ),  $S$  accepts an *exit* from one of other nodes. Then its exit action, denoted by  $S\_exit$ , is executed. Afterwards,  $S$  is *exited* from the node ( $fst\ sd$ ). Finally,  $S$  returns to its initial state ( $S(id)$  is called) and accepts a further *enter* request.

$S$  and  $S\_exec$  are implemented in ITrees through nested iterations: the outer iteration, corresponding to  $S$ , is an infinite *loop* and the inner, corresponding to  $S\_exec$ , is a conditional iteration by the *iterate* constructor. The condition is true for self transitions and false otherwise. An example of the process for state *Waiting in Movement* can be found online ()

*Controllers.* The event alphabet of the process for a controller contains termination, event and operation call channels. The event channels include not only the events of the controller, but also those in connections between its state machines.

Parallel composition of the heterogeneous state machine processes for a controller requires they all share a common event type  $E$ , and so we rename them. The events are renamed to the corresponding events in the controller alphabet, according to the connections between the controller and its state machines. For *MicroController* in Fig. 1, the `terminate` channel, the event channels, and the operation call channels of the process for *Movement* are mapped to the corresponding controller channels.

In particular, for a connection  $c$  from an event  $e$  of a state machine  $stm1$  to an event  $e$  of another state machine  $stm2$  of a controller, we declare a channel  $e\_ctrl$  in the event alphabet of the controller. The channel in the process for  $stm1$  is renamed to that of the controller with the same direction  $dout$ . Then the channel in the process for  $stm2$  is renamed to that of the controller but with the opposite direction ( $din$  to  $dout$ ). Finally, the processes ( $D\_stm1$  and  $D\_stm2$ ) for both  $stm1$  and  $stm2$  synchronise on the channels of the controller with direction  $dout$ , which is sketched below.

$$\begin{array}{l} D\_stm1[\{(e\_stm1\ dout, e\_ctrl\ dout), \dots\}] \\ \parallel_{\{e\_ctrl\ dout, \dots\}} \\ D\_stm2[\{(e\_stm2\ din, e\_ctrl\ dout), \dots\}] \end{array}$$

Here  $\parallel_E$  is parallel composition over event synchronisation set  $E$ . In this way, the output of  $e$  in  $stm1$  synchronises with the input of  $e$  in  $stm2$ , which is the semantics of connection  $c$ .

```

1 Starting ITree animation...
2 Events: (1) RandomWalkCall (); (2) Gas (Din, []); ...;
3 [Choose: 1-22]: 1
4 Events: (1) Gas []; (2) Gas [(0,0)]; (3) Gas [(0,1)]; (4) Gas [(1,0)];
5 (5) Gas [(1,1)]; (6) Gas [(0,0),(0,0)]; (7) Gas [(0,0),(0,1)]; (8) Gas
6 [(0,0),(1,0)]; (9) Gas [(0,0),(1,1)]; ...; (21) Gas [(1,1),(1,1)];
7 [Choose: 1-21]: 9
8 Events: (1) MoveCall (0,Chemical_Angle_Front);
9 [Choose: 1-1]: 1
10 Events: (1) Flag Dout;
11 [Choose: 1-1]: 1
12 Terminated: ()

```

**Fig. 4.** Animation of the example when dangerous chemical detected.

*Modules.* Similar to the event alphabet of the process for a controller, that of the process for a module also contains a termination channel, event channels, and operation call channels. The event channels include not only the events of its platform, but also the events in connections between its controllers for the same reason. The process for a module is a parallel composition of the renamed processes for its controllers, memory processes, and buffer processes for asynchronous connections between its controllers such as the connection on event turn from MainController to MicroController in Fig. 1.

## 5 Code generation, animation, and case studies

As discussed previously in [10, Sect. 5], the animation of ITrees is achieved through code generation [12] in Isabelle. Infinite corecursive definitions over ITrees are implemented using lazy evaluation in Haskell. Associative lists are used as an implementation for partial functions in ITrees and a simple animator in Haskell is presented.

We illustrate two scenarios of the animation of the autonomous chemical detector in Figs. 4 and 5. Here, we instantiate **Chem** and **Intensity** to be a numeral type 2 and the sequence of **GasSensor** is bounded to 2, which is the same as the instantiations for the verification with FDR4. An animation scenario represents the interaction of the model with its environment: the lines starting with **Events** are produced by the model and represents all enabled events; and the lines starting with **[Choose: 1-n]** represents a user's choice of enabled events from number 1 to n. In Fig. 5, we omit the lines for enabled events and append the chosen event to the chosen number for simplicity.

Figure 4 illustrates the behaviour of the model when detecting a dangerous chemical: (1) initially the controller calls the platform to perform a random walk: the number 1 event is chosen on line #3, which corresponds to the call of the during action `randomWalk()` of state `Waiting` in Fig. 2; (2) then a sequence of gas sensor readings is received through the `gas` event, and we choose number 9

```

1 [Choose: 1-22]: 1 RandomWalkCall ()
2 [Choose: 1-21]: 4 Gas (Din,[(1, 0)])
3 [Choose: 1-22]: 1 MoveCall (1,Chemical_Angle_Front)
4 [Choose: 1-24]: 2 Obstacle (Din,Location_Loc_right)
5 [Choose: 1-23]: 1 Odometer (Din,0)
6 [Choose: 1-22]: 1 MoveCall (1,Chemical_Angle_Left)
7 [Choose: 1-21]: 8 Gas (Din,[(0, 0),(1, 0)])
8 [Choose: 1-22]: 1 MoveCall (1,Chemical_Angle_Front)
9 [Choose: 1-24]: 1 Obstacle (Din, Location_Loc_left)
10 [Choose: 1-23]: 2 Odometer (Din,1)
11 [Choose: 1-23]: 1 Odometer (Din,0)
12 [Choose: 1-22]: ...

```

**Fig. 5.** Animation of the example when chemical detected with low intensity.

(among 21 enabled `gas` events shown on lines #4-6 where the first element `Din` of each event is omitted) on line #7: `Gas [(0,0),(1,1)]`, representing a chemical being detected and its intensity is high in the second pair of the sequence; (3) the controllers call the `move` operation with speed 0 (on line #9), provided by the platform, to stop the robot; (4) the controllers indicate the platform to drop a flag (on line #11); and finally (5) the controllers terminate (on line #12).

In Fig. 5, we illustrate another scenario: a chemical is detected but its intensity is low for the two readings on lines #2 and #7. The model behaves as follows: (1) the initial behaviour is the same: calling the platform to request a random walk; (2) a sequence of gas sensor readings is received (on line #2); (3) the controllers call the `move` operation (the entry action of state `Going` in Fig. 2) to request the robot to move forward at speed 1 (on line #3); (4) an obstacle on its right is encountered (on line #4); (5) the odometer reading is 0 (on line #5); (6) the controllers call `move` (the action of a transition in the defined operation `changeDirection`) to request the robot to move towards its opposite direction (left here) to the obstacle at speed 1 (on line #6); (7) another reading of the gas sensor shows there is still a chemical detected with low intensity (on line #7); (8) the controllers call `move` (the entry action of state `TryingAgain` in machine `Movement`) to request the robot to move towards its front at speed 1 (on line #8); (9) an obstacle on its left is encountered (on line #9); (10) the odometer reading (the action of the transition from state `TryingAgain` to state `AvoidingAgain`) is 1 (on line #10); (11) there is another odometer reading (0) on line #11, which corresponds to the entry action of state `Avoiding` (the entering of this state is resulted from the taken transition from state `AvoidingAgain` to state `Avoiding` due to its guard `d1-d0>stuckDist` is true where the values of `d0` and `d1` are the previous two odometer readings 0 and 1, and the value of `stuckDist` is set 0 in this animation); (12) we omit further interactions.

Based on the animation, we also observe that if no chemical is detected, the model returns to its initial state. If low intensity chemical is detected, even without progress of `MicroController`, the model can continuously read through the `gas` event without blocking. This is due to the connection between the controllers

on event `turn` being asynchronous, and so `MainController` can continuously send a `turn` event without waiting for the synchronisation of `MicroController`.

## 6 Related work

Animation is a lightweight formal method. Kazmierczak et al. [19] describe the advantages of using animation to verify models. It is highly automated and cheap to perform. It provides an insight into the specification and its implicit assumptions and is very suitable for demonstrating the system. It is a form of interactive testing of the model and its properties. It requires little expertise: less than model checking and much less than theorem proving. But its biggest drawback is that it cannot prove consistency, correctness, or completeness.

Animation can be tailored to specific application domains. For example, Boichut et al. [20] report on using animation to improve the formal specifications of security protocols. They animate these specifications to draw diagrams of typical executions of the protocols. They use this to visualise protocol termination and understand interleaved execution. They experiment with the animation to detect unwanted side effects. Finally, they use visualisation to simulate intruders to find attacks not detected by other protocol analysis tools.

We use ITrees to implement a framework for animation of formal specifications. The ProB animator and model checker provides a different framework [21]. ProB contains a model checker and a constraint-based checker, both of which can be used to detect various errors in B specifications. It implements a back-end in a framework for a variety of different specification languages, including the B language, Event-B, CSP-M, TLA+, and Z.

De Souza [22] provides another framework: Joker. This is a tool for producing animators for formal languages. The application is based on general labelled transition systems and provides graphical animation, supporting B, CSP, and Z.

## 7 Conclusions

This work gives RoboChart an ITrees-based operational semantics and enables animation of RoboChart using code generation in Isabelle/HOL. To provide animation support, we extend ITree-based CSP with three operators and present their definitions. We describe how the semantics of RoboChart is implemented in ITree-based CSP, and illustrate it with an autonomous chemical detector example. With the semantics of a RoboChart model in Isabelle, we generate Haskell code and animate it using a simple simulator. We show two concrete scenarios of the example using animation.

This work targets at deterministic RoboChart models and covers a large part of RoboChart features (but not all). Our immediate future work is to investigate support of nondeterminism and give a semantics to more features such as hierarchical state machines and timed semantics.

In this paper, we manually translate the RoboChart semantics to Isabelle. This process can be automated and our work brings insights on how RoboChart

semantics in ITrees can be automatically generated. The simple animator will be improved to directly allow visualisation of RoboChart models in RoboTool.

With the RoboChart semantics in ITrees, we can also conduct verification in Isabelle/HOL, in addition to animation in this paper. We will investigate the use of temporal logics as a property language for verification of ITrees. We note that verification can also capitalise on the contributions of this work.

ITrees can also be extended to other semantic domains. Further work would be of great help in extending ITrees with probability and linking them to discrete-time Markov chains (DTMCs) [23, 24], which will allow us give a ITree-based probabilistic semantics to RoboChart.

Our work has many potential applications in robotics. Further research could investigate the development of verified ROS nodes using code generation here for a concrete implementation of RoboChart controllers.

**Acknowledgements.** This work is funded by the EPSRC projects CyPhyAssure<sup>7</sup> (Grant EP/S001190/1), RoboCalc (Grant EP/M025756/1), and RoboTest (Grant EP/R025479/1). The icons used in RoboChart have been made by Sarfraz Shoukat, Freepik, Google, Icomoon and Madebyoliver from [www.flaticon.com](http://www.flaticon.com), and are licensed under CC 3.0 BY.

## References

1. Cavalcanti, A., Barnett, W., Baxter, J., Carvalho, G., Filho, M.C., Miyazawa, A., Ribeiro, P., Sampaio, A. In: *RoboStar Technology: A Robotist's Toolbox for Combined Proof, Simulation, and Testing*. Springer International Publishing, Cham (2021) 249–293
2. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J., Woodcock, J.: *RoboChart: modelling and verification of the functional behaviour of robotic applications*. *Softw. Syst. Model.* **18**(5) (2019) 3097–3149
3. Ye, K., Cavalcanti, A., Foster, S., Miyazawa, A., Woodcock, J.: *Probabilistic modelling and verification using RoboChart and PRISM*. *Software and Systems Modeling* (Oct 2021)
4. Woodcock, J., Cavalcanti, A., Foster, S., Mota, A., Ye, K.: *Probabilistic Semantics for RoboChart*. In Ribeiro, P., Sampaio, A., eds.: *Unifying Theories of Programming*, Cham, Springer International Publishing (2019) 80–105
5. Ye, K., Foster, S., Woodcock, J.: *Automated Reasoning for Probabilistic Sequential Programs with Theorem Proving*. In Fahrenberg, U., Gehrke, M., Santocanale, L., Winter, M., eds.: *Relational and Algebraic Methods in Computer Science*, Cham, Springer International Publishing (2021) 465–482
6. Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice-Hall Int. (1985)
7. Roscoe, A.W.: *Understanding Concurrent Systems*. Texts in Computer Science. Springer (2011)
8. Xia, L.y., Zakowski, Y., He, P., Hur, C.K., Malecha, G., Pierce, B.C., Zdancewic, S.: *Interaction Trees: Representing Recursive and Impure Programs in Coq*. *Proc. ACM Program. Lang.* **4**(POPL) (December 2019)
9. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: *A Theory of Communicating Sequential Processes*. 560–599

<sup>7</sup> CyPhyAssure Project: <https://www.cs.york.ac.uk/circus/CyPhyAssure/>

10. Foster, S., Hur, C.K., Woodcock, J.: Formally Verified Simulations of State-Rich Processes Using Interaction Trees in Isabelle/HOL. In Haddad, S., Varacca, D., eds.: 32nd International Conference on Concurrency Theory (CONCUR 2021). Volume 203 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2021) 20:1–20:18
11. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3 - A Modern Refinement Checker for CSP. In: Tools and Algorithms for the Construction and Analysis of Systems. (2014) 187–201
12. Haftmann, F., Nipkow, T.: Code Generation via Higher-Order Rewrite Systems. In Blume, M., Kobayashi, N., Vidal, G., eds.: Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings. Volume 6009 of Lecture Notes in Computer Science., Springer (2010) 103–117
13. Hilder, J.A., Owens, N.D.L., Neal, M.J., Hickey, P.J., Cairns, S.N., Kilgour, D.P.A., Timmis, J., Tyrrell, A.M.: Chemical Detection Using the Receptor Density Algorithm. IEEE Trans. Syst. Man Cybern. Part C **42**(6) (2012) 1730–1741
14. Miyazawa, A., Cavalcanti, A., Ribeiro, P., Ye, K., Li, W., Woodcock, J., Timmis, J.: RoboChart Reference Manual. Technical report, University of York (2020) [www.cs.york.ac.uk/circus/publications/techreports/reports/robochart-reference.pdf](http://www.cs.york.ac.uk/circus/publications/techreports/reports/robochart-reference.pdf).
15. Blanchette, J.C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A., Traytel, D.: Truly Modular (Co)datatypes for Isabelle/HOL. In Klein, G., Gamboa, R., eds.: Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings. Volume 8558 of Lecture Notes in Computer Science., Springer (2014) 93–110
16. Spivey, J.M.: The Z Notation: A Reference Manual. 2nd. Prentice-Hall (1992)
17. Toyn, I., ed.: Information Technology — Z Formal Specification Notation — Syntax, Type System and Semantics. ISO (July 2002) ISO/IEC 13568:2002(E).
18. Ballarin, C.: Locales and Locale Expressions in Isabelle/Isar. In Berardi, S., Coppo, M., Damiani, F., eds.: Types for Proofs and Programs, Berlin, Heidelberg, Springer Berlin Heidelberg (2004) 34–50
19. Kazmierczak, E., Winikoff, M., Dart, P.W.: Verifying Model Oriented Specifications through Animation. In: 5th Asia-Pacific Software Engineering Conference (APSEC '98), 2-4 December 1998, Taipei, Taiwan, ROC, IEEE Computer Society 254–261
20. Boichut, Y., Genet, T., Glouche, Y., Heen, O.: Using Animation to Improve Formal Specifications of Security Protocols. In: 2nd Conference on Security in Network Architectures and Information Systems (SARSSI 2007). (2007) 169–182
21. Leuschel, M., Butler, M.J.: ProB: A Model Checker for B. In Araki, K., Gnesi, S., Mandrioli, D., eds.: FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings. Volume 2805 of Lecture Notes in Computer Science., Springer (2003) 855–874
22. de Souza, D.H.O.: Joker: An Animator for Formal Languages. PhD thesis, Departamento de Informática e Matemática Aplicada, Universidade Federal do Rio Grande do Norte (2011)
23. Kemeny, J.G., Snell, J.L., Knapp, A.W.: Denumerable Markov Chains. (1976)
24. Kemeny, J.G., Snell, J.L.: Finite Markov Chains: With a New Appendix "Generalization of a Fundamental Matrix" (Undergraduate Texts in Mathematics). Springer (1983)