Formally Verified Animation for RoboChart using Interaction Trees[*]

Kangfeng Ye[a,*], Simon Foster[a], Jim Woodcock[a]

[a] *Department of Computer Science, University of York, Deramore Lane, Heslington, YO10 5GH, York, United Kingdom*

Abstract

RoboChart is a core notation in the RoboStar framework. It is a timed and probabilistic domain-specific and state machine-based language for robotics. RoboChart supports shared variables and communication across entities in its component model. It has formal denotational semantics given in CSP. The semantic technique of Interaction Trees (ITrees) represents behaviours of reactive and concurrent programs interacting with their environments. Recent mechanisation 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 a novel 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 and a patrol robot model additionally exhibiting nondeterminism and using shared variables. With animation, we show two concrete scenarios for the chemical detector when the robot encounters different environmental inputs and three concrete scenarios for the patrol robot when its calibrated position is in different sections of a corridor. We also verify that the animated scenarios are truly trace refinements of the CSP denotational semantics of the RoboChart models using FDR, a refinement model checker for CSP. This supports the soundness of our approach that the use of CSP operators with priority resolves nondeterminism correctly.

Keywords: Interaction trees, CSP, animation, theorem proving, RoboChart, code generation, robot software, operational semantics, nondeterminism.

1. Introduction

The RoboStar[1] framework [1] brings modern modelling and verification technologies into software engineering for robotics. In this framework, models of the platform, environment, design, and simulations are given formal semantics in a unified semantic framework [2]. Correctness of simulation is guaranteed with respect to particular models using a variety of analytical technologies including model checking, theorem proving, and testing. Additionally, modelling, semantics generation, verification, simulation, and testing are automated and integrated into an Eclipse-based tool, RoboTool.[2] The core of RoboStar is RoboChart [3, 4], 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 [3, 5, 4], which enables automated verification using model checking and theorem proving [6].

Previous work [3] gives RoboChart a denotational semantics based on the CSP process algebra [7, 8]. This paper defines direct operational semantics for RoboChart using Interaction Trees (ITrees) [9]. ITrees

[*] This document presents results from the research project CyPhyAssure (`www.cs.york.ac.uk/circus/CyPhyAssure/`) funded by EPSRC.

[*] Corresponding author

Email addresses: `kangfeng.ye@york.ac.uk` (Kangfeng Ye), `simon.foster@york.ac.uk` (Simon Foster), `jim.woodcock@york.ac.uk` (Jim Woodcock)

[1] `robostar.cs.york.ac.uk`.

[2] `robostar.cs.york.ac.uk/robotool/`.

Preprint submitted to Elsevier *March 17, 2023*

are coinductive structures that can model infinite behaviours of a reactive system interacting with its environment. ITrees have been mechanised in both Coq [9] and Isabelle/HOL [10]. ITrees are a powerful semantic technique for the development of formal semantics that can unify trace-based failures-divergences semantics [11, 8] for CSP and transition-based operational semantics, and so unifies verification and animation [10]. In previous work [10], we have proved a formal correspondence between the failures-divergences model and our ITree-based semantics.

The existing implementation of RoboChart's semantics in RoboTool is restricted to machine-readable CSP (or CSP-M) for verification with FDR [12], and so only a subset of RoboChart's rich types and expressions can be supported and quantified predicates cannot be solved. Our contribution here is 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, avoiding the need to generate an explicit transition system. We mechanise the semantics in Isabelle/HOL and then utilise the code generator [13] to produce Haskell code for animation. Isabelle's code generator [13] translates executable definitions in the source HOL logic to target functional languages (such as SML and Haskell) and the translation preserves semantic correctness using high-order rewrite systems [14]. As a result, the semantics of the source logic in Isabelle is preserved during code generation via translation to target functional languages. Our animation, therefore, is formally verified with respect to RoboChart's semantics in ITrees. Thanks to the equational logic, functional algorithms and data refinement are supported in code generation, so less efficient algorithms and data structures, that are used for verification in Isabelle, can be replaced with more efficient ones for animation.

Our technical contributions are as follows: we (1) implement extra CSP operators (generalised choice, interrupt, exception, and renaming) which are required to support the RoboChart semantics, and new CSP operators with priority (hiding with priority and renaming with priority) to resolve nondeterminism in RoboChart based on an order; (2) implement a bounded sequence type for code generation; (3) use the new concepts to define an ITree-based operational semantics for RoboChart; (4) implement the semantics of two RoboChart models for case studies; and (5) apply our animator to explore the behaviour of the models. With our mechanisation and animation, we have detected a number of issues in one RoboChart model, explored the semantics of shared variables in RoboChart, and resolved nondeterminism in a particular way. Specifically, the prioritised renaming operator uses the order in which transitions are given in the renaming mappings to resolve nondeterminism. The benefit of resolving nondeterminism statically in semantics, instead of dynamically in the animator, is that animation becomes more automatic, since the user only needs to resolve external choice and not nondeterministic choice. There is no need to overcome the big τ diamonds [8] as in the animator of ProB [15] and FDR [12]. All definitions and theorems in this paper are mechanised and accompanying icons () link to corresponding repository artefacts.

The remainder of this paper is organised as follows. In Sect. 2, we introduce RoboChart through two examples: an autonomous chemical detector model and a patrol robot model. Section 3 briefly describes the mechanisation of ITrees in Isabelle and presents the extra CSP operators in detail. Then we present the RoboChart semantics in ITrees in Sect. 4, exemplified using the two models, and illustrate several scenarios for the two models in Sect. 5 using animation. We review related work in Sect. 6 and conclude in Sect. 7.

This paper is an extension of [16]. It adds a new generalised choice operator, redefines the CSP external choice operator using the generalised choice, and introduces biased external choice operators in Sect. 3.1. This work also gives the ITrees-based semantics to shared variables in RoboChart and resolves nondeterminism in the semantics using the new CSP operators with priority defined in Sects. 3.5 and 3.6. The semantics for shared variables and nondeterminism are exemplified in a new case study introduced in Sect. 2.2 and illustrated in Sect. 5.2 for its animation. We also add substantial details in Sect. 4 to give a clear understanding of our semantics for RoboChart with examples from the two models, including new subsections: Sect. 4.1 to give an overview of RoboChart semantics, Sect. 4.5 about channels and alphabet transformation, and Sect. 4.7 to give semantics to operations in RoboChart. For the existing subsections, we give more details and examples to illustrate the semantics. In particular, Sect. 4.6 is substantially extended to give semantics to state machines, not only its sketch but also its detailed definitions and examples for memories, different kind of nodes, and their composition in parallel. Similarly, Sects. 4.8 and 4.9 are also extended

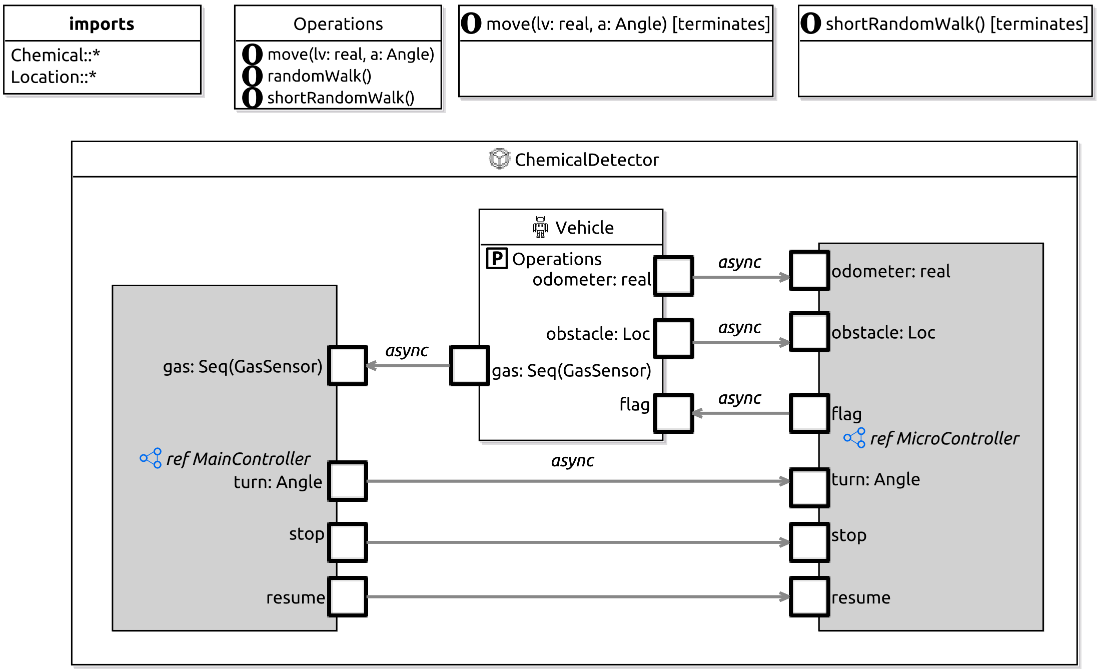

Figure 1: The module of the autonomous chemical detector model.

with examples.

2. RoboChart

RoboChart is a state machine-based notation for the modelling of robotic applications. RoboChart also has a component model with notions of controller and module, in addition to state machines, to foster reuse. In a RoboChart model, physical robots are abstracted into robotic platforms through variables, events, and operations. We describe features of RoboChart for modelling controllers of robots using as two examples: an autonomous chemical detector [17, 3, 18] in Sect. 2.1 and a patrol robot in Sect. 2.2. We refer to the RoboChart reference manual [18] for a complete account of the notation and its semantics.

2.1. Autonomous chemical detector

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 detecting a chemical with its intensity above a threshold, the robot drops a flag and stops there. This model[3] [3] has been studied and analysed in RoboTool, using FDR4,[4] a CSP refinement checker.

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. These services represent observable behaviour or external interaction of the controllers with the physical robot. The controllers MainController is responsible for gas analysis and MicroController accounts for the robot movement with obstacle avoidance.

[3] https://robostar.cs.york.ac.uk/case_studies/autonomous-chemical-detector/autonomous-chemical-detector.html
[4] https://cocotec.io/fdr/

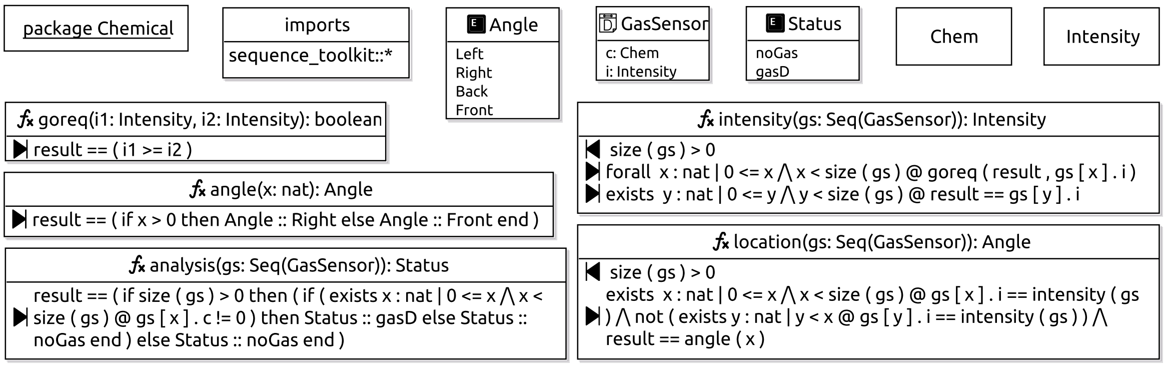
package Chemical
imports
sequence_toolkit::*
Angle
Left
Right
Back
Front
GasSensor
c: Chem
i: Intensity
Status
noGas
gasD
Chem
Intensity
goreq(i1: Intensity, i2: Intensity): boolean
result == (i1 >= i2)
angle(x: nat): Angle
result == (if x > 0 then Angle :: Right else Angle :: Front end)
analysis(gs: Seq(GasSensor)): Status
result == (if size (gs) > 0 then (if (exists x : nat | 0 <= x /\ x < size (gs) @ gs [x] . c != 0) then Status :: gasD else Status :: noGas end) else Status :: noGas end)
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
location(gs: Seq(GasSensor)): Angle
size (gs) > 0
exists x : nat | 0 <= x /\ x < size (gs) @ gs [x] . i == intensity (gs) /\ not (exists y : nat | y < x @ gs [y] . i == intensity (gs)) /\ result == angle (x)

Figure 2: The Chemical package of the autonomous chemical detector model.

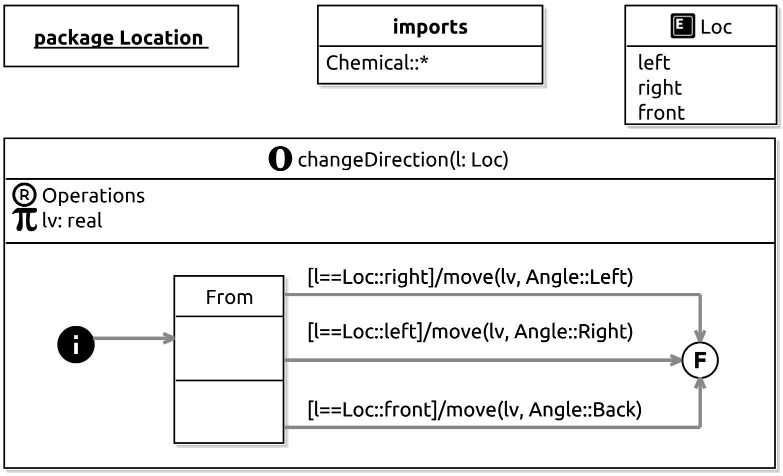
package Location
imports
Chemical::*
Loc
left
right
front
changeDirection(l: Loc)
Operations
lv: real
From
[l==Loc::right]/move(lv, Angle::Left)
[l==Loc::left]/move(lv, Angle::Right)
[l==Loc::front]/move(lv, Angle::Back)
i
F

Figure 3: The Location package of the autonomous chemical detector model.

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 to allow MainController to instruct MicroController to *turn* towards the location where the gas is detected, *stop* the robot if the gas is dangerous, or *resume* its movement behaviour (by ignoring the current movement operation) if no gas is detected. These interactions are the internal behaviour of controllers and are not observable.

The types used in the module are defined in the two imported packages: Chemical and Location shown in Figures 2 and 3. The Chemical package declares primitive types Chem and Intensity, enumerations Status and Angle, a record GasSensor containing two fields (c of type Chem and i of type Intensity), and five functions specified using preconditions and postconditions (in the original model, two are specified and three are unspecified). The Location package declares an enumeration Loc and defines an operation changeDireciton using a state machine. The operation has a parameter l of type Loc and a constant lv. Basically, this operation aims to move the robot in the opposite direction of the currently detected gas location l using a constant linear velocity lv.

MainController, defined in the left diagram of Fig. 4, is implemented using a state machine GasAnalysis (by a contained reference to the machine), presented in the right diagram of Fig. 4. The machine GasAnalysis declares one constant thr of type Intensity for the intensity threshold, and four variables (gas of

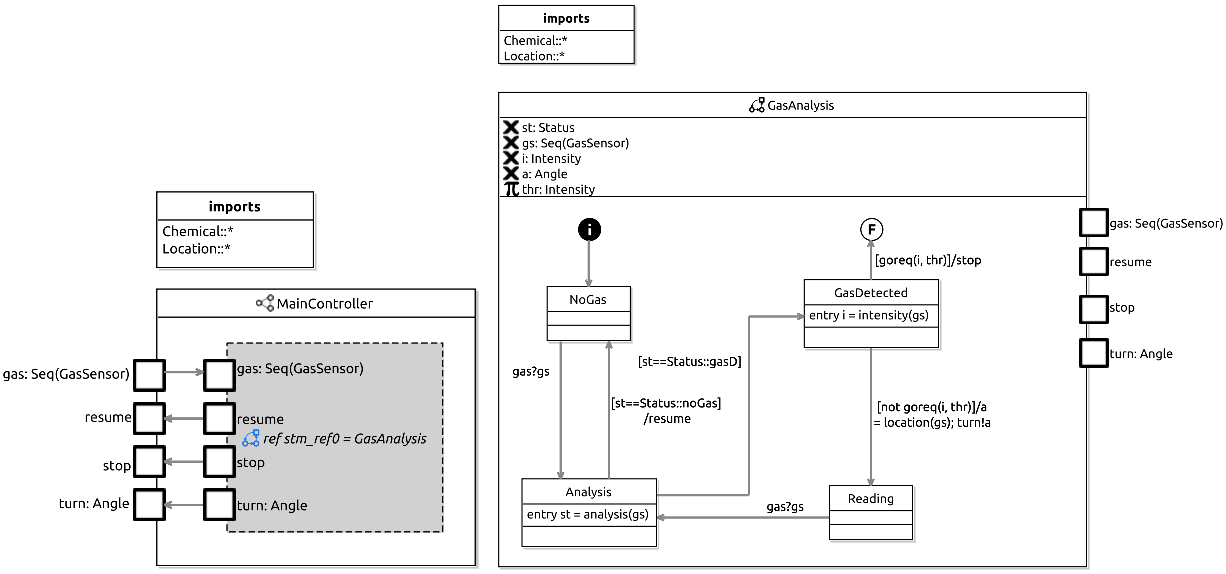

Figure 4: MainController and GasAnalysis state machine of the autonomous chemical detector model.

Seq(GasSensor), st of Status, i of Intensity, and a of Angle) for a sequence of gas sensor readings (from the platform), and the gas analysis results including its status (either no gas NoGas or a gas gasD detected), intensity and angle. The machine also contains a variety of nodes: one initial junction (ⓘ), seven normal states such as NoGas and Analysis, and a final state (Ⓕ). A state may have an entry action such as an assignment of st from an application of function analysis to gs in the state Analysis, an exit action, or a during action.

In state machines, transitions connect states and junctions. Transitions have a label with optional features: a trigger event, a clock reset, a guard condition, and an action. For example, the transition of GasAnalysis from NoGas to Analysis has an input trigger gas?gs enabling the machine to receive sensor readings from the channel gs and store the value in the variable gs, and the transition from GasDetected to Reading has a guard (not goreq(i, thr)) and an action (a=location(gs); turn!a) which is a sequential composition of an assignment and an output communication (turn!a) enabling the machine to send the angle a of the detected gas over the channel turn.

This machine gives the behaviour of the robot's gas analysis: (1) enter the state NoGas after the transition from the initial junction is taken; (2) wait for the gas sensor to be ready on channel gs, then receive readings, recorded in gs, on the channel from the platform (via connections from MainController), and at the same time the transition to state Analysis is taken; (3) upon entering the state Analysis whose entry action is executed first to analyse the sensor readings by the function analysis and to record the result in variable st; (4) signal the event resume if no gas is detected (guard [st == noGas]) and return to state NoGas; (5) go to state GasDetected otherwise (guard [st == gasD]); (6) upon entering the state GasDetected whose entry action is executed to determine the intensity by the function intensity and to record the result in variable i; (7) signal stop if the intensity i is larger than or equal to (implemented in function goreq) the intensity threshold thr, and terminate by going to the final state; (8) take the transition to state Reading otherwise with the action of the transition being executed to determine the angle a of the detected gas and signal turn towards the angle; (9) try to read the gas sensor again at state Reading with its only outgoing transition, and if the transitions is taken, the machine is back to state Analysis.

MicroController, defined in Fig. 5, is implemented using a state machine Movement (by a contained reference), presented in Fig. 6, and a reference to the operation changeDireciton. The machine Movement declares various constants such as lv for linear velocity, four variables (a, d0, d1, and l) for the preservation of values (angle, odometer readings, and location) carried on communication, and a clock T. The machine also contains one initial junction, seven normal states such as Waiting and Going, and a final state. Notably, the state Waiting has a during action, an operation call randomWalk(), which provides parallelism in a machine

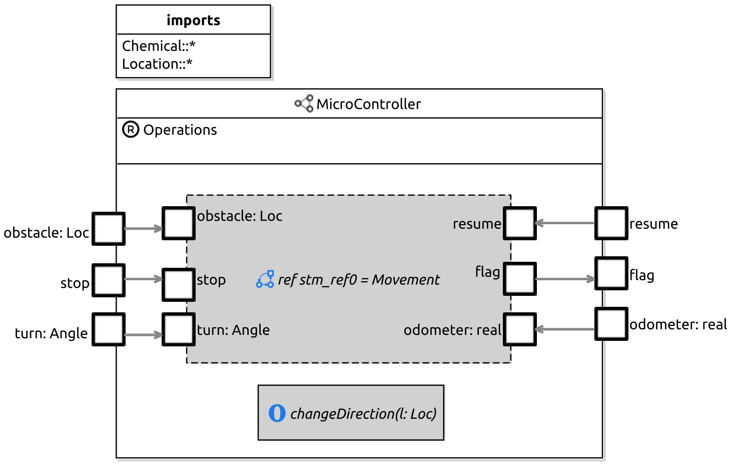

Figure 5: MicroController of the autonomous chemical detector model.

and means the robot is doing a random walk at the state. This operation can be interrupted at any time as long as a transition from the state is taken. The transitions of this machine have labels with various features. The transition from Going to Avoiding has an input trigger obstable?l and a clock reset #T, and the transition from TryingAgain to AvoidingAgain has an input trigger and an action odometer?d1 (an input communication). The transition from AvoidingAgain to Avoiding has a clock reset #T and a disjunctive guard in which since(T) counts the elapsed time since the last reset of T.

This machine gives the behaviour of the robot's response to outcomes of the chemical analysis: (1) resume to state Waiting if no gas is detected (implemented in GasAnalysis); (2) stop to state Found and then terminate , if a gas above the threshold is detected; (3) turn to the direction, where a gas is detected but not above the threshold, with obstacle avoidance in state Going; (4) upon the first detection of an obstacle, reset T and start Avoiding with an initial odometer reading and the movement direction changed (software waits for evadeTime for the effect of that change); (5) if a gas is still detected after the changed direction, TryingAgain to turn and move to the gas direction; (6) if another obstacle is detected during avoidance, AvoidingAgain by reading the odometer to check the distance of two obstacles; (7) if the robot has moved far enough between the two obstacles or not got stuck long enough, go back to continue Avoiding; (8) otherwise, the robot has got stuck in a corner, use a shortRandomWalk for GettingOut of the area, then resume normal activities.

2.2. One-dimensional patrol robot

In addition to the features introduced in the chemical detector example, RoboChart also supports abstraction via nondeterministic choice and interaction via shared variables, which is exemplified in the model for a one-dimensional patrol robot on a corridor. The corridor, as shown in Fig. 7, is split into three sections by four boundaries, denoted by their coordinates: -MAX_INT, -MAX, MAX, and MAX_INT. The left and right boundaries are hard and represent the walls, and the other two are soft and represent controlled limits. Section S2, soft boundaries exclusive, is the central working area. The sections S1 and S3, soft boundaries inclusive, are limited sections and the robot will tend to move to S2 if it is in these sections.

Basically, the control software of this patrol robot behaves as follows: (1) it maintains a belief state (x) of the robot, default at 0 (denoting the centre of the corridor) and able to reset to 0 by an event reset; (2) when x is 0, it can be calibrated to the actual position of the robot from its sensor by a cal event; (3) when x is within S2, it can be either increased or decreased by 1, denoting the robot moves either to the left or to the right nondeterministically; (4) when x is within S1, it can only be increased by 1, denoting the robot moves to the right; and (5) when x is within S3, it can only be decreased by 1, denoting the robot moves to the left.

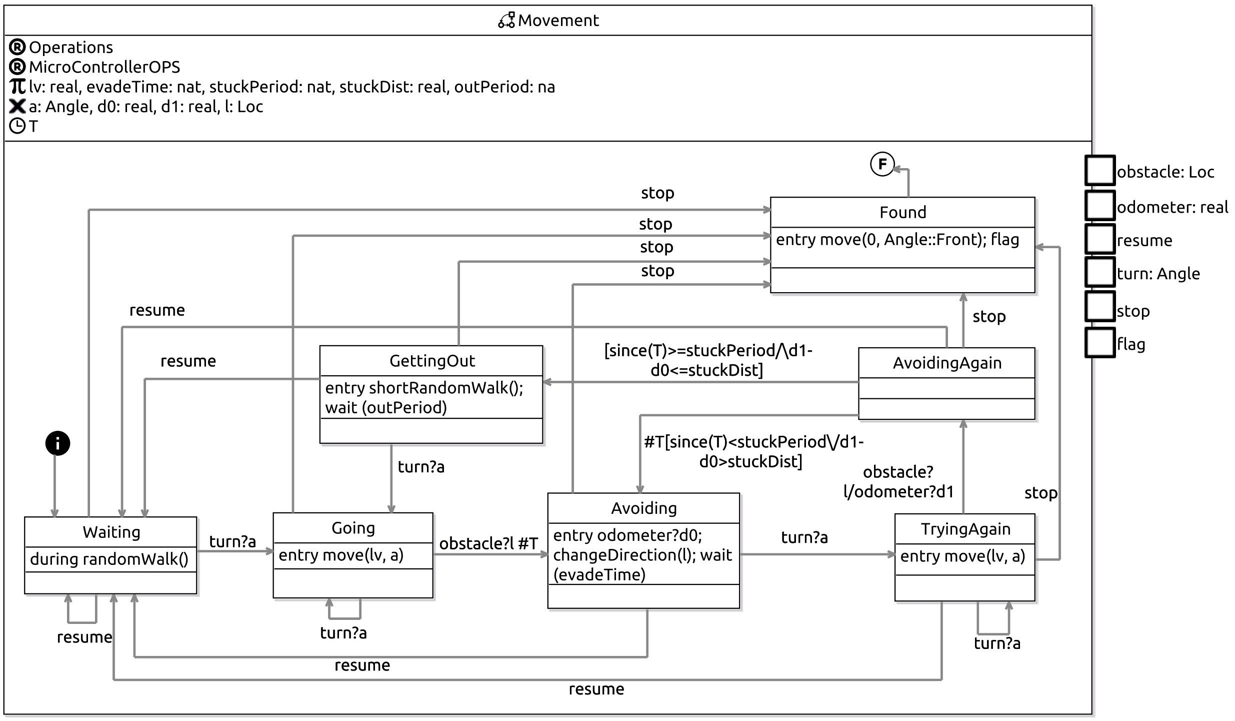

Figure 6: Movement state machine of the autonomous chemical detector model.

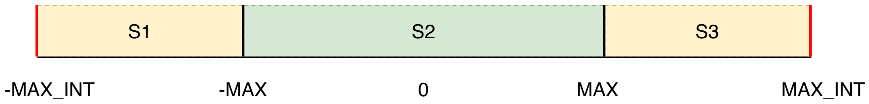

Figure 7: Sections for the patrol robot: S1 - adjacent to the left boundary; S2 - central area; S3 - adjacent to the right boundary.

We illustrate in Fig. 8 the RoboChart model for this patrol robot. The module PatrolMod contains a robotic platform RP and a controller Ctrl composed of two state machines CalSTM and MoveSTM. RP declares two output events cal of type int and reset, and two inputs events left and right (to indicate the direction of moving and its new position) of type int through the interface eventsInf, and provides a variable x of type int through the interface dataInf. This variable is shared in Ctrl and also to the two machines by requesting dataInf.

The machine CalSTM sets x to 0 in the action (x=0) of its default transition t0 to state cal, and then it is ready for calibration from the input trigger cal?l of the transition t1 if [x==0], or to update x (the action update!x of the transition t2) to MoveSTM through a connection on event update of type int otherwise [x!=0]. We note that it is not mandatory to use this communication mechanism to update x to MoveSTM because x is shared in MoveSTM too. We use this way here to illustrate how nondeterminism is introduced in MoveSTM and how it interleaves with the shared variable.

The machine MoveSTM is responsible for resetting x (x=0) by the self-transition t2 of state move with trigger reset, and the robot movement around the corridor based on the value (of variable x) which passes on update and stored in a local variable l by the other two self-transitions (t1 and t3) from state move. If l is larger than -MAX (guard [l>-MAX] of t3 where MAX is a constant variable), denoting sections S2 and S3, x is updated to l-1, followed by a left!x event. If l is less than MAX (guard [l<MAX] of t1), denoting sections S2 and S1, x is updated to l+1, followed by a right!x event. Consequently, if l is in section S2, the choice between the two transitions is nondeterministic. We note that MAX_INT is not explicitly declared in

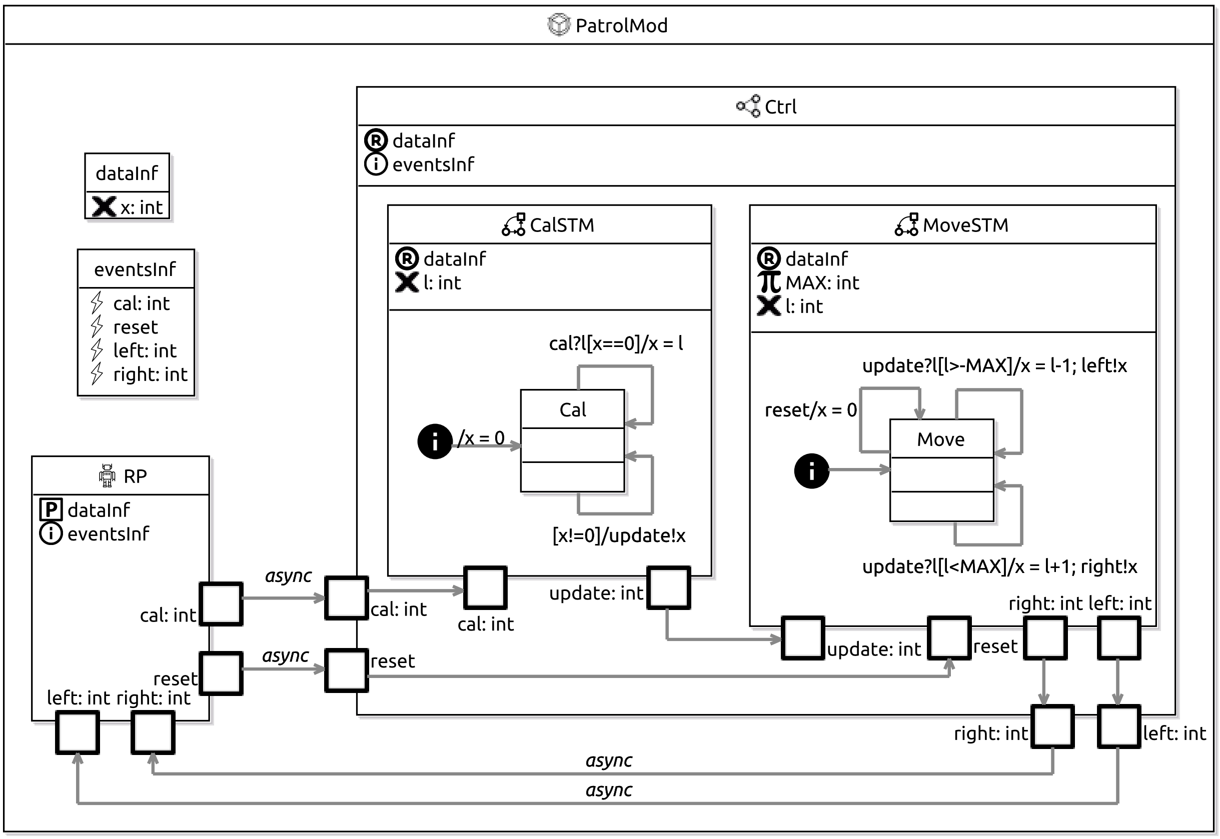

Figure 8: A RoboChart model for a patrol robot with nondeterministic behaviour and interaction using shared variables.

the model and will be given in verification or animation where int is bounded.

Next, we describe the extra ITree-based CSP operators in order for the RoboChart semantics and return to the two models in Sect. 4.

3. Interaction trees

In this section, we briefly introduce interaction trees and extend our existing CSP semantics with additional operators to support the RoboChart semantics. These include three operators (interrupt, exception, and renaming) that are introduced in the previous work [16], one generalised choice operator and two prioritised variants of hiding and renaming operators introduced in this extension paper.

Interaction trees (ITrees) [9] are a data structure for modelling reactive systems that interact with their environment through events. They are potentially infinite and defined as coinductive trees [19] in Isabelle/HOL.

```
codatatype ('e, 'r) itree =
  Ret 'r | Sil "('e, 'r) itree" | Vis "'e ⇸ ('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 ($\checkmark_x$) with a value x of type R returned; (2) *Sil* P: an internal silent event (τP, for a successor ITree P); or (3) *Vis* F: a choice among several visible events represented by a partial function F of type $E \pto (E, R)\mathit{itree}$. Partial functions are part of the Z toolkit[5] [20] which is also mechanised in Isabelle/HOL.

Deterministic CSP processes can be given executable semantics using ITrees. The determinism is inherent, since we use a partial function to model events and their continuations. Specifically, each unique event must map to at most one continuation. The benefit of this approach is that ITrees are easy to implement and animate, since the behaviour depends solely on provided input events. Therefore, our CSP operators cannot introduce nondeterminism, which must be statically resolved.

Our animator takes a model with an ITree-based semantics in Isabelle/HOL, and generates Haskell code for the underlying ITree. Though this often an infinite structure, Haskell's intrinsic use of lazy evaluation allows to model and partially evaluate infinite objects. We can therefore step through an ITree's behaviour by unfolding the various constructors of the underlying algebraic (co)datatype. When a *Vis* constructor is encountered, the user is presented with a choice for one of the enabled events. When a *Sil* is encountered, it is simply removed and the successor ITree animated. This allows us to compress long sequences of τ events, and so simplified animation. Finally, *Ret* leads to termination of the animation.

Previously, the following CSP processes and operators have been defined: (1) basic processes: *skip*, *stop*, and *div*; (2) input prefixing: *inp* $c\ V$ (communicate any value from A over the channel c);(3) output prefixing $c!v$; (4) *guard* b; (5) external choice $P \mathbin{\Box} Q$; (6) parallel composition $P \parallel_A Q$; (7) hiding $P \setminus A$; (8) sequential composition $P \mathbin{⨾} Q$; (9) *loop* and *iterate*. Though operators like external choice and parallelism can introduce nondeterminism, we restrict this by construction. Nevertheless, different strategies can be employed for statically resolving nondeterminsim, which we explore further in this section.

Here, we give an ITree semantics to extra CSP operators to allow us to give an ITree-based semantics to RoboChart. We (1) generalise external choice; (2) introduce three new operators—interrupt, exception, and renaming operators—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; and (3) add prioritised variants of renaming and hiding to statically resolve nondeterminism based on an order. We restrict ourselves to deterministic operators as it makes the animation of large models more efficient.

3.1. Generalised choice and external choice

Previously [10], we have given a semantics to CSP's external choice operator as a corecursive definition. Our mechanisation of ITrees intrinsically supports external choice through the use of partial functions to model visible events. However, our definition of choice can be generalised to support more flexible choice schemes. For example, priority can be given to one branch of the choice, in order to statically resolve any nondeterminism. We achieve this through a novel generalised choice operator, $P \boxdot_{\mathcal{M}} Q$. For this, we use a merge function $\mathcal{M}$, which merges two choice functions of type $E \pto (E, R)\mathit{itree}$. The operator is defined as a corecursive function using the equations listed below.

Definition 3.1 (Generalised choice).

$$\begin{aligned}
(\mathit{Vis}\ F) \boxdot_{\mathcal{M}} (\mathit{Vis}\ G) &= \mathit{Vis}\ (\mathcal{M}\ F\ \ G) & (\mathit{Ret}\ x) \boxdot_{\mathcal{M}} (\mathit{Ret}\ y) &= (\textbf{if}\ x = y\ \textbf{then}\ \mathit{Ret}\ x\ \textbf{else}\ \mathit{stop}) \\
(\mathit{Sil}\ P') \boxdot_{\mathcal{M}} Q &= \mathit{Sil}\ (P' \boxdot_{\mathcal{M}} Q) & P \boxdot_{\mathcal{M}} (\mathit{Sil}\ Q') &= \mathit{Sil}\ (P \boxdot_{\mathcal{M}} Q') \\
(\mathit{Ret}\ v) \boxdot_{\mathcal{M}} (\mathit{Vis}\ G) &= \mathit{Ret}\ v & (\mathit{Vis}\ F) \boxdot_{\mathcal{M}} (\mathit{Ret}\ v) &= \mathit{Ret}\ v
\end{aligned}$$

When making a choice between two visible event functions, *Vis* F and *Vis* G, the merge function is applied to combine the two. When combining two return value ITrees, *Ret* x and *Ret* y, we require that the two possible values are identical, and otherwise deadlock, to avoid nondeterminism. For silent events (τ) and returns, we also prioritise their occurrence before any visible activity can occur. In particular, any τ events are greedily consumed before any visible event or return can occur.

With generalised choice, we can define external choice using a merge function $F \odot G \mathrel{\widehat=} (\mathrm{dom}(G) \ndres F) \oplus (\mathit{dom}(F) \ndres G)$, and defining $P \mathbin{\Box} Q \mathrel{\widehat=} P \boxdot_{\odot} Q$. This function combines all event maplets from F and

[5] https://github.com/isabelle-utp/Z_Toolkit.

G, whilst ignoring any maplets whose events occur in the intersection $\mathrm{dom}(F) \cap \mathrm{dom}(G)$. For example $\{e_1 \mapsto P_1, e_2 \mapsto P_2\} \odot \{e_3 \mapsto P_3, e_2 \mapsto P_4\} = \{e_1 \mapsto P_1, e_3 \mapsto P_3\}$, since e_2 is ignored. This again avoids nondeterminism, and when the two domains are disjoint the operator can be considered as a union.

Another benefit of the generalised choice operator is that, as with Hoare and He's parallel-by-merge operator [2], properties of the choice operator reduce to properties of the merge function itself. This simplifies proofs of algebraic properties for choice functions. The most basic property of a merge function is well-formedness:

Definition 3.2 (Wellformed merge function). A merge function $\mathcal{M}$ is well-formed provided that for any choice function F, $\mathcal{M}\ \varnothing\ F = \mathcal{M}\ F\ \varnothing = F$.

A wellformed merge function has the empty choice function $\varnothing$ as a left and right identity. For example, it is clear that $\odot$ is wellformed because $\mathrm{dom}(\varnothing) = \varnothing$. From this definition of wellformedness, we obtain the following properties.

Theorem 3.1 (Generalised choice). *If $\mathcal{M}$ is well-formed, then*

$$\begin{aligned} P \boxdot_{\mathcal{M}} \mathit{stop} = \mathit{stop} \boxdot_{\mathcal{M}} P = P \\ P \boxdot_{\mathcal{M}} \mathit{div} = \mathit{div} \boxdot_{\mathcal{M}} P = \mathit{div} \\ P \boxdot_{\mathcal{M}} Q = Q \boxdot_{\mathcal{M}^\sim} P \end{aligned}$$

Generalised choice has *stop* as a unit, since it can add no further behaviour. Moreover, *div* is a zero since τ events always take priority, and so a diverge process prevents choices being made. We can commute a choice by taking the converse of the merge function, where $\mathcal{M}^\sim = (\lambda\, F\ G.\, \mathcal{M}\ G\ F)$. As a result of the final law, if $\mathcal{M}$ is symmetric, that is $\mathcal{M} = \mathcal{M}^\sim$, then choice is commutative: $P \boxdot_{\mathcal{M}} Q = Q \boxdot_{\mathcal{M}} P$. From these properties, we can conclude that external choice is commutative and has *stop* as a unit. The former follows because $f \oplus g = g \oplus f$ whenever $\mathrm{dom}\, f \cap \mathrm{dom}\, g = \varnothing$, a property that is ensured by the construction of $\odot$.

We now consider how we can derive alternative choice schemes. In some circumstances, it may be undesirable that possible events are lost by the external choice. For example, $a \to P \Box a \to Q$ actually ends up as *stop*, since the two processes both have a as an initial event. We can instead resolve any nondeterminism by choosing to prioritise the addition of events from either the left, or the right branch. We introduce a biased choice operator, $P \mathrel{{\triangleleft}\Box} Q \mathrel{\widehat{=}} Q \boxdot_{\oplus} P$, which chooses events from P whenever initial events are present in both P and Q. For example, $a \to P \mathrel{{\triangleleft}\Box} a \to Q = a \to P$, since the event from the left branch is prioritised.

3.2. Interrupt

The second operator we introduce is interrupt [7, 8], $P \bigtriangleup Q$, 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, is 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 3.3 (Interrupt).

$$\begin{array}{ll} (\mathit{Sil}\ P') \bigtriangleup Q = \mathit{Sil}\ (P' \bigtriangleup Q) & P \bigtriangleup (\mathit{Sil}\ Q') = \mathit{Sil}\ (P \bigtriangleup Q') \\ (\mathit{Ret}\ x) \bigtriangleup Q = \mathit{Ret}\ x & P \bigtriangleup (\mathit{Ret}\ x) = \mathit{Ret}\ x \\ (\mathit{Vis}\ F) \bigtriangleup (\mathit{Vis}\ G) = \mathit{Vis} \big(\ \{e \mapsto (P' \bigtriangleup Q) \mid (e \mapsto P') \in (\mathrm{dom}(G) \mathbin{\lhd\!\!\!-} F)\} \oplus G\ \big) & \end{array}$$

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

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, $(\mathrm{dom}(G) \ndres F)$ 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 is a corecursive call $P' \triangle Q$. 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.

3.3. Exception

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

Definition 3.4 (Exception).

$$(\mathsf{Ret}\ x) \llbracket A \rhd Q = \mathsf{Ret}\ x \qquad (\mathsf{Sil}\ P') \llbracket A \rhd Q = \mathsf{Sil}\ (P' \llbracket A \rhd Q)$$

$$(\mathsf{Vis}\ F) \llbracket A \rhd Q = \mathsf{Vis}\begin{pmatrix} \{e \mapsto (P' \llbracket A \rhd Q) \mid (e \mapsto P') \in (A \ndres F)\} \oplus \\ \{e \mapsto Q \mid e \in (A \cap \mathrm{dom}(F))\} \end{pmatrix}$$

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

Similar to Definition 3.3, 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, $e \in \mathrm{dom}(A \ndres F)$), can occur independently and its continuation is a corecursive call $P' \llbracket A \rhd Q$. Following the execution of an initial event e of P that is in A (that is, $e \in (A \cap \mathrm{dom}(F))$, the exception behaves like Q, which is expressed by the second partial function.

3.4. Renaming

The other new operator we define for this work is renaming, $P\llbracket\rho\rrbracket$, 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.

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

This produces the minimal functional relation that is consistent with R. For example,

$$\mathit{mk\_functional}\left(\{e_1 \mapsto e_2, e_1 \mapsto e_3, e_2 \mapsto e_3\}\right) = \{e_2 \mapsto e_3\}$$

This function is used to avoid nondeterminism introduced by renaming multiple events to the same event. We use this function to define the renaming operator.

Definition 3.5 (Renaming).

$$\begin{aligned} (\mathsf{Ret}\ x)\llbracket\rho\rrbracket &= \mathsf{Ret}\ x \\ (\mathsf{Sil}\ P')\llbracket\rho\rrbracket &= \mathsf{Sil}\ (P'\llbracket\rho\rrbracket) \\ (\mathsf{Vis}\ F)\llbracket\rho\rrbracket &= \begin{pmatrix} \textbf{let}\ G = F \circ \mathit{mk\_functional}\left((\mathrm{dom}(F) \lhd \rho)^{\sim}\right) \\ \bullet\ \mathsf{Vis}\left(\ \{e_2 \mapsto (P'\llbracket\rho\rrbracket) \mid (e_2 \mapsto P') \in G\}\ \right) \end{pmatrix} \end{aligned}$$

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

In the *Vis* case, G is a partial function $(E_2 \pfun (E_1, R)\mathit{itree})$ 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 $(\mathrm{dom}(F) \lhd \rho)$ which is the domain restriction $\lhd$ of ρ to the domain $\mathrm{dom}(F)$ of F. Basically, the multiple events of E_1 that are mapped to the same event of E_2 in ρ and also are the initial events of P, or in $\mathrm{dom}(F)$, 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 $P'\llbracket\rho\rrbracket$ where $(e_2 \mapsto P') \in G$.

Because many-to-one mappings in ρ are removed by *mk_functional* in G, the potential nondeterminism is excluded. For example,

$$\begin{aligned} &(e_1 \to P \Box e_2 \to Q \Box e_3 \to R)\llbracket\{e_1 \mapsto e, e_2 \mapsto e, e_3 \mapsto ea, e_4 \mapsto eb\}\rrbracket \\ &= (ea \to R\llbracket\{e_1 \mapsto e, e_2 \mapsto e, e_3 \mapsto ea, e_4 \mapsto eb\}\rrbracket) \qquad \text{(renaming excludes nondeterminism)} \end{aligned}$$

Here, the only available initial event after renaming is ea because the relation $(\mathrm{dom}(F) \lhd \rho)^{\sim}$ is equal to $\{e \mapsto e_1, e \mapsto e_2, ea \mapsto e_3\}$ and *mk_functional* removes the first two pairs (because of duplicate distinct values), and so results in $\{ea \mapsto e_3\}$.

Remark 3.1. Roscoe [8] defines three ways for renaming in CSP. Injective functional renaming will not change the behaviour of a CSP process and is an alternative to parametrised CSP processes on channels. Non-injective functional renaming may change the behaviour of a process by ignoring some level of detail or introducing nondeterminism. Relational renaming, a more general and powerful operator than the other two functional renamings, allows many-to-one (may introduce nondeterminism), or one-to-many (to offer more choice) mappings. What machine-readable CSP (CSPM) supports and FDR implements is relational renaming. We investigated all these approaches and chose to implement here the relational renaming with many-to-one and one-to-many mappings. This is mainly because RoboChart's semantics is defined using CSPM and verified using FDR. For many-to-one mappings, our definition here, however, blocks these many events and the definition of renaming with priority in Sect. 3.6, as follows, chooses one of these many events according to their priority.

3.5. Hiding with priority

The current semantics [10] of hiding $P \setminus A$ is deadlock if more than one initial events of P are in A. This is to avoid nondeterminism caused by hiding of two possible events. This restriction can be relaxed by placing events to be hidden in an order. For example, $(a \to P \Box b \to Q) \setminus \{a, b\} = \mathit{stop}$, but $((a \to P \Box b \to Q) \setminus \{a\}) \setminus \{b\} = \tau\,((P \setminus \{a\}) \setminus \{b\})$, and $((a \to P \Box b \to Q) \setminus \{b\}) \setminus \{a\} = \tau\,((Q \setminus \{b\}) \setminus \{a\})$. Hiding events in a different order, therefore, resolves the external choice differently without deadlock. This difference is due to the maximal progress assumption of hiding: $(a \to P \Box b \to Q) \setminus \{a\}$ is actually equal to $\tau\,(P \setminus \{a\})$.

We define hiding with priority, $P \setminus_p \mathit{el}$ (), to put these events to be hidden in an order based on their order in a list el.

$$P \setminus_p \mathit{el} = \mathit{foldl}\,((\lambda\, Q\ e.\ Q \setminus \{e\})\,, P, \mathit{el})$$

The foldl builds a return value by applying the function $(\lambda\, Q\ e.\ Q \setminus \{e\})$ (say f) to the combined result (initially P) and elements in el based on their orders. For example, $\mathit{foldl}\,(f, P, [a, b])$ will be expanded to $f(f(P, a), b)$, which is just $(P \setminus \{a\}) \setminus \{b\}$. The above examples now can be expressed as $(a \to P \Box b \to Q) \setminus_p [a, b]$ and $(a \to P \Box b \to Q) \setminus_p [b, a]$.

3.6. Renaming with priority

Because the relation ρ of type $E_1 \leftrightarrow E_2$ in the renaming definition 3.5 is possibly heterogeneous (so E_1 and E_2 are different types), renaming cannot be like hiding with priority to place the events to be renamed in an order to rename events one by one. This is because renaming a process changes its event type from E_1 to E_2, and so we cannot rename other events of type E_1 anymore. For this reason, we define renaming with priority, $P\llbracket\varrho\rrbracket_p$, which renames events of P according to a finite sequence ϱ of type $\mathrm{seq}\,(E_1 \times E_2)$. We use the finite sequence type here to describe the mathematical definition of this operator and its representation in Isabelle is actually lists. A finite sequence of type $\mathrm{seq}\,X$ is a finite partial function $\mathbb{N} \mathrel{+\!\!\!\!\twoheadrightarrow} X$ from natural numbers $\mathbb{N}$ to X. With ϱ, a priority is given based on the indices of pairs in the sequence in order to resolve potential nondeterminism in a particular way. For those pairs that have the same second element (in other words, many-to-one mappings), the pair with the smallest index has the highest priority. This renaming with a priority operator will only rename the event with the highest priority and block other events with lower priority.

Prior to the definition of $P[\![\varrho]\!]_p$, we need to define another two functions. The first function is the domain restriction $\lhd_l$ of ϱ to a set A.

$$A \lhd_l \varrho = \mathit{squash}\ \{s : \mathrm{seq}\,(E_1 \times E_2) \mid s \in \varrho \bullet (s.2).1 \in A\}$$

This produces a new sequence, compacted from a function, or a set of ordered pairs, in which each member s is in ϱ and the first element $(s.2).1$ (of type E_1) of the second element $s.2$ (of type $E_1 \times E_2$) of s is in A, by the *squash* function [20]. When a type is obvious, we use a short form $x \in A \bullet P(x)$ for $x : T \mid x \in A \bullet P(x)$. This could be used in set comprehension, quantification, etc.

We give an example below to illustrate how $\lhd_l$ works.

$$\{e_1, e_2, e_4\} \lhd_l \langle (e_1, e), (e_2, e), (e_3, ea), (e_4, eb)\rangle = \langle (e_1, e), (e_2, e), (e_4, eb)\rangle$$

For the second function $\mathit{drop\_dup}(\varrho)$, we need to drop the pairs with lower priority (and so bigger indices). We define an auxiliary *least* function first.

$$\mathit{least}(\varrho, p) \mathrel{\widehat{=}} (\nexists q \in \varrho \bullet q.1 < p.1 \wedge (q.2).2 = (p.2).2) \qquad (\textit{least}\ \text{definition})$$

This function characterises if p (of type $\mathbb{N} \times (E_1 \times E_2)$) has the least index number $p.1$ in the members q of ϱ that have the same target event $(q.2).2$ (of type E_2) as that $(p.2).2$ of p. In other words, p has the least index number in a set of renaming pairs that have multiple events renamed to the same event as in p. Now $\mathit{drop\_dup}(\varrho)$ is defined below.

$$(\forall\, p \in \mathit{drop\_dup}(\varrho) \bullet \mathit{least}(\rho, p)) \wedge \qquad \text{(maximal)}$$
$$(\forall\, p \in \varrho \bullet \mathit{least}(\varrho, p) \Rightarrow p \in \mathit{drop\_dup}(\varrho)) \qquad \text{(minimal)}$$

The first predicate (maximal) in the conjunction states that every element in the resultant $\mathit{drop\_dup}(\varrho)$ is *least* in ϱ, and the second predicate (minimal) states that every least element in ϱ must be in the resultant $\mathit{drop\_dup}(\varrho)$. For example,

$$\mathit{drop\_dup}\,(\langle (e_1, e), (e_2, e), (e_4, eb)\rangle) = \langle (e_1, e), (e_4, eb)\rangle$$

Here, the pair (e_2, e) is dropped because it does not have the highest priority in terms of the target event e (due to the fact that e appears early in (e_1, e)). With the two functions given above, we can define $P[\![\varrho]\!]_p$ corecursively.

Definition 3.6 (Renaming with priority).

$$(\mathsf{Ret}\ x)\,[\![\varrho]\!]_p = \mathsf{Ret}\ x$$
$$(\mathsf{Sil}\ P')\,[\![\varrho]\!]_p = \mathsf{Sil}\ \left(P'[\![\varrho]\!]_p\right)$$
$$(\mathsf{Vis}\ F)\,[\![\varrho]\!]_p = \left(\begin{array}{l} \textbf{let}\ G = F \circ \mathit{mk\_functional}\left((\mathrm{ran}\,(\mathit{drop\_dup}\,(\mathrm{dom}(F) \lhd_l \varrho)))^{\sim}\right) \\ \bullet\ \mathsf{Vis}\left(\ \left\{e_2 \mapsto \left(P'[\![\varrho]\!]_p\right) \mid (e_2 \mapsto P') \in G\right\}\ \right) \end{array}\right)$$

This definition is similar to Definition 3.5 except that ϱ here is a sequence of renaming pairs and the relation in the inverse now has all its domain elements mapped to distinct values by *drop_dup*. Because *drop_dup* defines a sequence of type $\mathbb{N} \mathrel{+\!\!\!\!\twoheadrightarrow} (E_1 \times E_2)$, we get the range of the sequence by the ran function, which is a relation.

The difference between renaming and renaming with priority is exemplified below.

$$(e_1 \to P \mathrel{\Box} e_2 \to Q \mathrel{\Box} e_3 \to R)\,[\![\langle (e_1, e), (e_2, e), (e_3, ea), (e_4, eb)\rangle]\!]_p$$
$$= \left(\begin{array}{l} e \to P[\![\langle (e_1, e), (e_2, e), (e_3, ea), (e_4, eb)\rangle]\!]_p \mathrel{\Box} \\ ea \to R[\![\langle (e_1, e), (e_2, e), (e_3, ea), (e_4, eb)\rangle]\!]_p \end{array}\right) \qquad \text{(renaming resolves nondeterminism)}$$

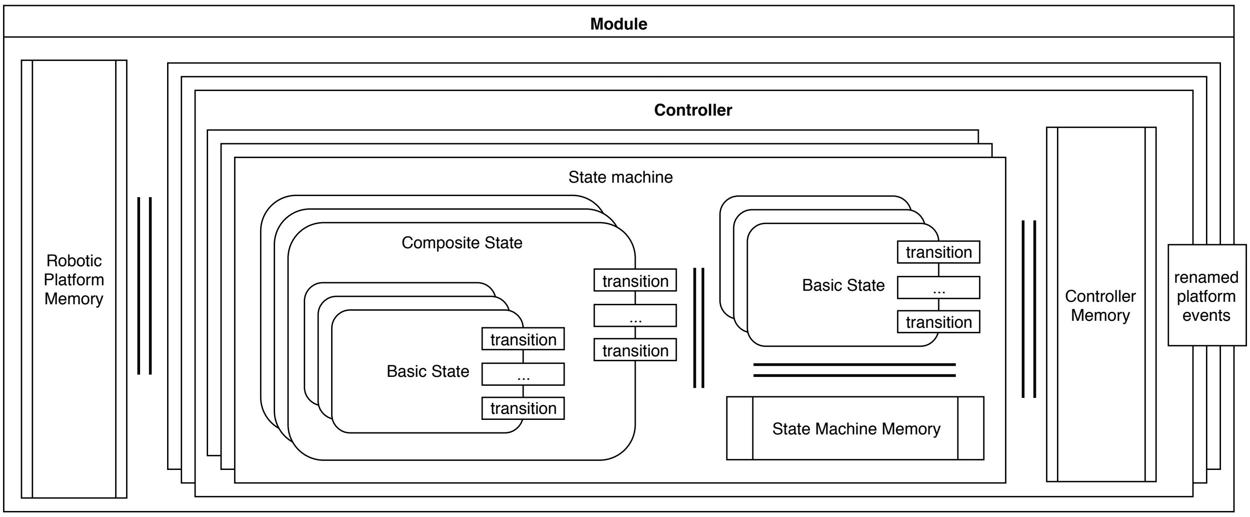

Figure 9: From [3, Fig. 11]. Structure of the RoboChart semantics: stacked components and parallel lines indicate parallel composition; bordered boxes indicate points of interaction; the semantics of a container is composed of the semantics of its contained components.

Compared to (renaming excludes nondeterminism), the potential nondeterminism introduced by renaming both e_1 and e_2 to e is resolved by giving priority to the renaming map (e_1, e). If (e_1, e) and (e_2, e) are swapped, then the renaming will give priority to e_2 and its continuation is Q (instead of P).

4. RoboChart semantics in interaction trees

In this section, we describe how we give 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.

In particular, nondeterministic choice between transitions, for example in Fig. 8, is resolved by the use of the prioritised renaming operator defined earlier through ordered transition event mappings. This is not covered in the previous work [16] and is the new contribution of this paper.

4.1. Overview of RoboChart semantics

The RoboChart CSP semantics is sketched in Fig. 9. The semantics for modules, controllers, state machines, and (either composite or basic) states are CSP processes. The semantics for a robotic platform, however, is different from controllers and state machines in that it does not have a particular behaviour and so its semantics is not a CSP process because the platform is an abstraction of a physical robot through variables, events, and operations.

Semantically, RoboChart has a hierarchical memory model with a memory for the robotic platform at the top, memories for controllers in the middle, and memories for state machines inside their container controllers. A memory process records the reads and writes of variables for components (the platform, controllers, and state machines) in scope. The memory for a state machine caches the (both local and shared) variables it requires, and so the semantics of the machine is independent of the location where these variables are declared. The memory for a controller or the platform, however, is different from that of a state machine in that it not only accepts updates to the variables in the memory but also propagates the updates down the hierarchy to the memories of state machines that require the updated variables.

The RoboChart semantics of the autonomous chemical detector model in Sect. 2.1 is shown in Fig. 10. The semantics of the module is a parallel composition of the two controllers with the robotic platform memory

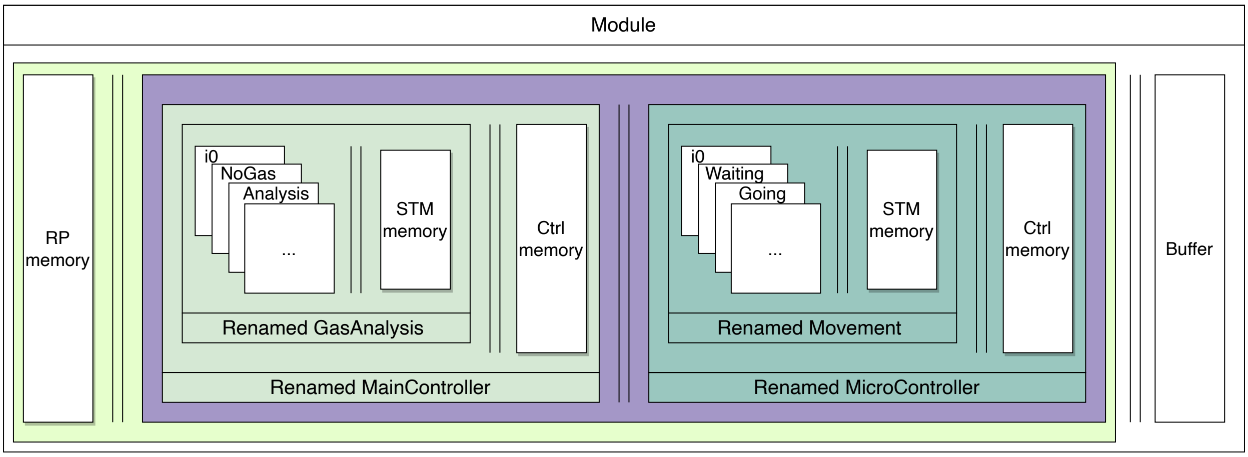

Figure 10: RoboChart semantics of the autonomous chemical detector.

(RP memory) and a buffer process (Buffer). The buffer process models the asynchronous connection from MainController to MicroController on event turn in Fig. 1. This semantics for asynchronous connections is omitted in Fig. 9 for simplicity.

The platform memory and the controller memories for this chemical detector model are simply the CSP process *skip* because the platform and the controllers do not provide any variables for sharing. The platform memory and the controller memory for the one-dimensional patrol robot in Sect. 2.2, nevertheless, record the access and update of the shared variable x and propagate its update to the controller, and finally to the state machines in the controller.

The semantics of MainController (or MicroController) is the composition of the CSP process for the semantics of its contained state machine GasAnalysis (or Movement) with the controller memory. We note the CSP processes for the controllers are renamed, such as Renamed MainController. This is because RoboChart uses directed connections for communication, but CSP's parallel composition requires composites to have the same channel names for communication. We, therefore, need to rename the exported channels of the controller CSP processes to ensure the channel names for the two controllers in a connection are the same, and so they can communicate with each other.

Similarly, the CSP processes for the semantics of the state machines (GasAnalysis and Movement) are also renamed for the same reason to ensure the state machines in a controller can communicate on directed connections. The renaming is not compulsory for the chemical detector model because each controller contains only one state machine, but it is mandatory for the patrol robot because the controller contains two state machines connected on event update. So in general, we always rename them.

The semantics of the state machine GasAnalysis is a parallel composition of the node (including the junction i0 and the states NoGas, etc.) processes with the machine memory recording the access and update of the local variables of the machine. The semantics of the machine Movement has a similar structure. The memories of the state machines CalSTM and MoveSTM in Fig. 8, nonetheless, record not only their local variables l but also the shared variable x.

Practical consideration. Our ITree-based semantics for RoboChart in this paper aims for consistency with the restricted CSP semantics generated in RoboTool. Here the restriction particularly refers to bounded types for basic RoboChart types and corresponding closed arithmetic operators. In the future, we could use unbounded basic types with usual operators, then enforce enumerations just in the animator, not in the semantics like this work. One advantage of using this restricted semantics is the possibility to reuse the current CSP semantics generator in RoboTool to generate corresponding ITrees-based theories for Isabelle to automatically generate Haskell code.

RoboTool automatically generates multiple versions of CSP semantics for a RoboChart model, including a standard version (**S**), an optimised version (**D**) that reduces internal interaction to some extent, an optimised and visible version (**VS**) that further exposes internal interaction, and an optimised and compressed version (**O**) with compressions using strong bisimulation and diamond elimination [8]. All these versions use some compressions. Our ITree-based semantics for RoboChart in this paper is based on the **D** version because RoboTool uses this version for verification. Our semantics here does not support compressions in the **D** version. Instead, we implement a form of compression for internal events in the animator.

4.2. Types

RoboChart has its type system based on that of the Z notation [21]. 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 improves the 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 `'a Chem = (ChemT, 'a) PrimType` (). An example of 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`). In order to construct an element of type Chem, we define a dedicated constructor `ChemC` which is simply a type cast of `PrimTypeC`. Finally, we use `(ChemC 0::2)` and `(ChemC 1::2)` to construct such two elements of type `(2 Chem)`.

For enumerations and records, we use **`datatype`** and **`record`** in Isabelle. The `Status` and `GasSensor` below are such examples for the corresponding types in the Chemical package shown in Fig.2

```
datatype Status = Status_noGas | Status_gasD
record 'a GasSensor = gs_c :: "'a Chem"   gs_i :: "'a Intensity"
```

An instantiation type `(2 GasSensor)` is a record containing two fields of finite types `(2 Chem)` and `(2 Intensity)`, both of which have two elements. We now can construct an element `⦇gs_c = Chem (1::2), gs_i = IntensityC (0::2)⦈` of this type: the chemical is q and the intensity is 0.

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.

```
typedef ('a,'n::finite) blist = {xs::'a list. length xs ≤ CARD('n)}
```

`CARD` here retrieves the cardinality of `'n`. We introduce a notation `'a blist['n]` for this type and also define several functions: (a) `blength` to get the length of a bounded sequence; (b) `bnth` to get the nth element of a bounded sequence; (c) `bappend` ($@_s$) to concatenate two bounded sequence; and (d) `bmake` to construct a bounded sequence from a finite list.

For the type Seq(GasSensor) in RoboChart, its bounded type in Isabelle is `(2 GasSensor) blist[2]` which denotes the length of sequences bounded to 2 and elements (in the sequences) of type `(2 GasSensor)`. We now can use `bmake TYPE(2) [⦇gs_c = Chem (0::2), gs_i = IntensityC (0::2)⦈, ⦇gs_c = Chem (1::2), gs_i = IntensityC (1::2)⦈]` to construct a sequence containing two sensor readings.

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

4.3. 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 three for the arithmetic operators, 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
Minus(e1, e2, T) = if member(e1 - e2, T) then (e1 - e2) else e1
Neg(e, T) = if member(-e, T) then (-e) else e
```

In the definition of `Plus`, if `e1+e2` is within `T` then this is the result, otherwise, it is `e1`. Other definitions `Minus` and `Neg` are similar. We use `locale` [22] 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.

```
locale robochart_confs =
  fixes min_int::"integer" and max_int::"integer" and max_nat::"natural" and
        min_real::"integer" and max_real::"integer"
begin
  abbreviation "core_int_list ≡ [min_int .. max_int]"
  abbreviation "core_int_set ≡ set core_int_list"
  definition Plus where "Plus e1 e2 T = (if (e1+e2) ∈ T then (e1+e2) else e1)"
  definition Minus where "Minus e1 e2 T = (if (e1-e2) ∈ T then (e1-e2) else e1)"
  definition Neg where "Neg e T = (if -e ∈ T then -e else e)"
  ...
end
```

We define bounded core types as sets by ranges given by a lower limit and an upper limit, such as parameters `min_int` and `max_int`. Additionally, in code generation, we use lists to implement sets, and so we define a list of ranges first in `core_int_list` and then convert it into a set `core_int_set`.

We note a named type in CSP can be used as a type as well as a set expression, and so a bounded core type is a type and also an expression. Isabelle, however, has different concepts for types and expressions, and they cannot be used interchangeably. For this reason, we have core types in Sect. 4.2 and bounded sets for these types here. The definition `Plus` illustrated above corresponds to that in the `instantiation.csp` file and is an example to define a closure operator.

In the theory of Isabelle for a RoboChart model, we instantiate this locale using `interpretation` (), which allows us to assign concrete values for the parameters. An example is shown below which instantiates the parameters (limits) of the locale to `-2, 2, ...` etc.

```
interpretation rc: robochart_confs "-2" "2" "2" "0" "1".
```

Then we can use `rc.core_int_set` and `rc.Plus` to access the instantiated definitions.

4.4. 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.

As mentioned in Sect. 2.1, three (goreq, angle, and analysis) among the five functions defined in the Chemical package of the autonomous chemical detector model in Fig. 2 are unspecified, and two (analysis and intensity) among them are specified in the original model. Our model presented in Fig. 2 specify all five functions. With the capability of solving preconditions and postconditions of functions introduced in our work, we detect two problems in the definitions of the two specified functions in the original model, which are corrected in our model. Next, we present the semantics of the two functions (intensity and location) in our implementation.

The intensity function defined in Fig. 2 has a precondition (◀) that the length (size) of the parameter gs is larger than 0, and two postconditions (▶) involving universal and existential quantifications where @ separates constraint and predicate parts, and goreq is a $\geq$ relation on intensities. The result of the function is the largest intensity in gs. 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 nth 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.

The location function defined in Fig. 2 has one precondition and one postcondition and their corresponding definitions in Isabelle are shown below. The result of this function is the location (on the right or in the front of the robot, according to the definition of the angle function) where the largest intensity in gs is detected.

```
definition "pre_Chemical_location gs = (blength gs > 0)"
definition "Chemical_location gs = (THE result. (∃x::nat<blength gs.
    (Chemical_intensity(gs) = gs_i (bnth gs x)) ∧
    (¬ (∃y::nat < x. (Chemical_intensity(gs) = gs_i (bnth gs y)))) ∧
    (∀x::nat < blength gs. Chemical_goreq(result, gs_i (bnth gs x)))
    (result = Chemical_angle(x))))"
```

Inside the definite description is an existential quantification over an index x of gs such that (a conjunction of three conjuncts) (1) the intensity gs_i (bnth gs x) of the sensor reading at index x is the largest intensity (Chemical_intensity(gs)) in gs; (2) there does not exist an index y such that y is less than x and the intensity at y is also the largest intensity in gs (in other word, x is the smallest index whose intensity is the largest); and (3) the result of the function is just the angle Chemical_angle(x) of x.

We also found another error in the postcondition of location in the original model: the postcondition is not strong enough to identify a unique result of the function for the same input (and so the result is a relation and not a function). Our definition of the function fixes this problem by specifying that x is the smallest index.

While there are problems with the definitions of intensity and location in the original model, their implementations in CSP for verification with FDR previously, however, are correct. This leads to an inconsistency between models and their semantic implementation for verification. Our approach presented here translates the specification of functions from models to their semantic implementation in Isabelle directly, and so the consistency is preserved. This is one benefit of our approach.

4.5. Channels and alphabet transformation

A CSP process in our semantics is an interaction tree of type (E, R)*itree*, parametrised over a type E of events representing the event alphabet (Σ) of the CSP process and a type R of return values. E is declared through the **chantype** command and expressed as a finite set of channels declared in the command. Each channel c is modelled by a prism [23]: $V \Longrightarrow_{\triangle} E$ where V is the value type of c. Each prism contains two functions [10, Definition 5]: a destructor *match*: $E \nrightarrow V$ and a constructor *build*: $V \Rightarrow E$. We show an example to illustrate the creation of such an event alphabet `chan`.

```
chantype chan = Input::integer      Output::integer     State::"integer list"
```

This `chan` declares three channels: `Input` and `Output` of type `integer`, and `State` of type `integer list`. Each channel is modelled by a prism. For example, the channel `Input`, carrying a value of `integer`, has a type `integer` $\Longrightarrow_{\triangle}$ `chan`. We use $\mathit{build}_{\mathit{Input}}\ 1$ (a notation `Input_C 1` introduced for it) to construct an event (`Input 1`) of type `chan`.

External choice $P \;\Box\; Q$ requires that P and Q have the same type $(E, R)\mathit{itree}$ and so the same event type E. Parallel composition $P \parallel_A Q$ additionally requires that the type R of return values is empty (`()` or `unit` in Isabelle) because CSP processes in parallel usually do not return data, and so both P and Q should be the type of $(E, ())\mathit{itree}$. A here is a set of events and of type $\mathbb{P}\, E$.

For the CSP processes that have different types, they need to be transformed into the same type prior to composition. The alphabet transformation of a process P of type $(E_1, R)\mathit{itree}$ is through renaming (or renaming with priority) according to a renaming relation ρ of type $E_1 \leftrightarrow E_2$ (or a finite sequence ϱ of type $\operatorname{seq}(E_1 \times E_2)$), and the resultant process $P[\![\rho]\!]$ (or $P[\![\varrho]\!]_p$) is of type $(E_2, R)\mathit{itree}$.

We also note that the renaming relation or the finite sequence for the renaming operators is "total" — only events of type E_1 in the relation or the sequence are renamed and others are blocked — no matter whether this is a homogeneous (E_1 and E_2 are the same) or heterogeneous (E_1 and E_2 are different) renaming. This is different from the relational renaming operator in CSPM where the relation is partial and so the events that are not in the renaming relation are not affected by the renaming. This difference is due to the fact that ITrees or Isabelle's terms are strongly typed.

In our ITree-based semantics for RoboChart, illustrated in Fig. 10 for the autonomous chemical detector model, each state machine, controller, or module has a different event alphabet by declaring an individual channel type. For example, the state machine GasAnalysis, the controller MainController, and the module ChemicalDetector declare channel types `Chan_GasAnalysis`, `Chan_MainController`, and `Chan_ChemicalDetector`. The renamed GasAnalysis transforms the event alphabet of GasAnalysis from `Chan_GasAnalysis` to `Chan_MainController`, and so it can be composed in parallel with the controller memory (whose event type is `Chan_MainController`). Similarly, the two controllers are renamed to the event alphabet `Chan_ChemicalDetector` of the module, and so they are able to be composed in parallel with the robotic platform memory and buffer processes.

4.6. State machines

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 entering 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. We show below an example of these data types for the Movement machine in Fig. 6.

```
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[6] 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.

4.6.1. Memory for variables and transitions

The memory process Memory_x for a shared variable x, such as x in the state machine CalSTM in Fig. 8, is shown below.

```
loop (λv. get_x!v → ✓_v □ set_x?x → ✓_x  □ set_EXT_x?x → ✓_x)
```

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). For a local variable of the state machine, its memory process does not provide an external update.

The memory process for transitions of the state machine Movement in Fig. 6 is partially (3 in 24 transitions) illustrated below.

```
Movement_MemoryTrans = loop (λid.
  internal!TID_t1 → ✓_id □
  resume_!(TID_t0, din) → ✓_id □
  turn_?(TID_t3, din, a∈Chemical_Angle) → ✓_id □
  ...  □
  get_d1?d1 → get_d0?d0 →
    ((rc.Minus d1 d0 rc.core_int) > 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 an output resume (din because the resume event acts as an input in the state machine according to the connection in Fig. 5); and (3) the third for the transition from Waiting to state Going which has an input trigger turn?a where we use a notation turn_?(TID_t3, din, a∈Chemical_Angle) to denote *inp* turn_ {(TID_t3, din, a) | a∈Chemical_Angle} (input values from a set of triples by set comprehension

[6] In the Isabelle code, we include suffixes to ensure that names do not collide, but omit them here

over a channel turn_); and (4) the fourth for the transition from AvoidingAgain to Avoiding whose guard is (d1-d0>stuckDist) (time semantics is ignored, - becomes a closed operator rc.Minus under core_int) and evaluated in this memory transition process. We note that these choices will not lead to nondeterminism because their initial input or output events are parametrised over a distinct TID (for each choice).

For the transition with an input trigger and a guard together, such as the self transition of the state move in Fig. 8 with a trigger update?l and a guard [l>=-MAX], its semantics in CSP is a constrained input prefix: update?l:(l≥-MAX) where (l≥-MAX) is a constraint on l. So the memory process for transitions of the state machine MoveSTM in Fig. 8 has choices illustrated below.

```
MoveSTM_MemoryTrans = loop (λid.
  internal!(TID_t0, din) → ✓_id □
  reset_!(TID_t2, din) → ✓_id □
  inp update_ {(TID_t1, din, l) | l ∈ rc.core_int. l ≥ (rc.Neg MAX rc.core_int)} □
  inp update_ {(TID_t3, din, l) | l ∈ rc.core_int. l ≤ MAX})
```

In Isabelle, we encode constraints in the predicate part of set comprehension.

4.6.2. Nodes

The semantics of each node in a state machine is a CSP process. In this paper, we consider three kinds of nodes: initial junctions, basic states, and final states, which are used in the two RoboChart models. Semantics for other nodes including normal junctions and composite states are part of our future work.

Initial junctions. We show the semantics for the initial junction i0 in the state machine CalSTM in Fig. 8, which has an outgoing transition t0 with an action x=0 to set the variable x to 0.

```
I_i0 = internal!(TID_t0, din) → set_x!0 →
  enter!(SID_CalSTM, SID_CalSTM_cal) → entered!(SID_CalSTM, SID_CalSTM_cal) → ✓
```

The process outputs (1) a pair (TID_t0, din) on the channel internal, denoting the empty trigger of t0 from i0 to state cal; (2) 0 on the channel set_x, denoting the initialisation of x to 0 (by parallel composition with the memory for x presented later), which corresponds to the action of t0; (3) a pair (SID_CalSTM, SID_CalSTM_cal) on the channel enter, denoting the start of entering the state cal; and (4) a pair (SID_CalSTM, SID_CalSTM_cal) on the channel entered, denoting the completion of entering cal.

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

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

S and S_exec are defined by mutual recursion. Initially, S accepts *enter*ing 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 the 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 the 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* (identified 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 is denoted by *EvtChs*. If this event *e* of a transition *t* is signalled ($e?(TID\_t, \_)$), the during-action is interrupted and S accepts an *exit* from one of the 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 in Fig. 6 is shown below.

```
1 State_Waiting = loop (λid::integer.
2   sd ← inp enter {(s, SID_Waiting) | s. ¬ s ∈ SIDS_Movement_no_Waiting} ;
3   ret ← Ret (True, id, fst sd);
4   (iterate (λr. fst r) (λr.
5     entered!(snd (snd r), SID_Waiting); (
6     (CALL__randomWalk(id); stop) △ (
7       (resume_!(TID_t0, din) → exit!(SID_Waiting, SID_Waiting) →
8        exited!(SID_Waiting, SID_Waiting) → enter!(SID_Waiting, SID_Waiting);
9        Ret (True, fst (snd r), SID_Waiting)) □
10      (turn_?(TID_t2, din, a∈Chemical_Angle) → set_a!a →
11       exit!(SID_Waiting, SID_Waiting) → exited!(SID_Waiting, SID_Waiting) →
12       enter!(SID_Waiting, SID_Going) → entered!(SID_Waiting, SID_Going) ;
13       Ret (False, fst (snd r), SID_Waiting)) □
14       ...
15  ))) (ret)); Ret (id))
```

Basically, `State_Waiting` is an infinite loop, an ITree, whose state (a state of an ITree is its carried data or its return value) is an integer `id` (corresponding to the parameter `id`), with a nested conditional iteration by `iterate` (corresponding to `S_exec`). The state of the iteration is a triple such as `(True, id, fst sd)`, whose first element is a boolean value to indicate if this iteration terminates or not, whose second element is `id`, and whose third element is the RoboChart state (for example `fst sd`, see lines 2 and 3) that initiates entering of the state Waiting of `S_exec`. Initially, the state of the iteration is passed from its preceding `Ret` construct and recorded in a variable `ret` (also the third argument of `iterate`, see line 15).

In the iteration body, `snd (snd r)` on line 5 is the third element of the tuple `r` and is a RoboChart state. `CALL__randomWalk(id)` on line 6 corresponds to the call to an operation randomWalk (provided by the robotic platform) in the during-action of Waiting. Lines 7 to 9 correspond to the self transition `t0` of

Waiting, whose trigger is resume, and lines 10 to 13 correspond to the transition t2 (from Waiting to Going) whose trigger is turn?a. We note that there is no entered event for t0 and the first element in the return value *Ret*(True, fst (snd r), SID_Waiting) is True. This is because t0 is a self transition and it does not need to wait for entering shown on line 2. The first element True makes iterate continue its inner iteration, and the third element SID_Waiting means this is entering from the state Waiting itself. The transition t2 is different from t0 in that it (1) has an input trigger to get the angle and record it in the local variable a, as shown on line 10; and (2) will exit Waiting (on line 11) and enter Going (on line 12). The execution of this transition will leave the inner iteration by setting the first element of the return value to False (on line 13).

The restricted behaviour of Waiting is shown below.

$$\texttt{State\_Waiting\_R = }\lambda\texttt{id. State\_Waiting(id)} \parallel_{\texttt{internal\_events - trigger\_events}} \mathit{skip}$$

State_Waiting_R is a parallel composition of State_Waiting and a *skip* process over a set of events for synchronisation which is the set difference of all internal events internal_event (including e_ events and internal_ events) and all trigger events trigger_events of the machine Movement. The parallel composition with *skip* blocks all events in the synchronisation set and leaves all events in trigger_events to proceed independently. We note that for a basic state such as Waiting, its restricted process will not change its behaviour State_Waiting. The reason why restricted behaviour is needed is because of composite states. We will deal with it in the future.

4.6.3. Composition of STM and memory processes

As sketched in Figs 9 and 10, the semantics of a state machine is the parallel composition of the composed processes for its nodes and its memory. We show below the composed processes for the nodes of the state machine CalSTM in Fig. 8.

```
STM_CalSTM = λid. I_i0(id) |[ flow_events_from_other_states ]| State_cal_R
```

The composed process STM_CalSTM (called the node process) is a parallel composition of the process I_i0 for the initial junction and the restricted process State_cal_R for the state cal over all the flow events flow_events_from_other_states that enter cal from other states.

Then the node process of a state machine is composed in parallel with the memory of the machine with proper event hiding. The definitions of such composition (called the node memory process) are omitted here for simplicity and can be found online (MemorySTM_Movement for the machine Movement and MemorySTM_CalSTM for the machine CalSTM). We note that in such a process, the internal events used for synchronisation between the node process and the memory, the get events of all variables, and the set events of all local variables are hidden because these events are used to synchronise between the node process and the memory and are internal. The set events of shared variables are not hidden because the update to shared variables by the set events needs to be propagated to other state machines.

As discussed previously, each event e of a state machine has two corresponding event channels e_ (for triggers) and e (for actions) in the semantics of the machines, such as obstacle_ and obstacle for the event obstacle of the machine Movement. Trigger event channels e_ in the composition of the nodes and the memory of a machine are renamed to e by forgetting the first element (a transition identifier) of the value carried on e_. For example, MemorySTM_Movement is renamed according to a renaming relation event_map.

```
Renamed_MemorySTM_Movement = MemorySTM_Movement ⟦event_map⟧
```

The event_map contains the mappings not only for the trigger events like (obstacle_.(TID_t6, din, Loc_left), obstacle.(din, Loc_left)) but also for other event channels (events for actions, terminate, shared variable channels set and set_EXT, and operation call) that will not be renamed like (obstacle.(din, Loc_left), obstacle.(din, Loc_left)) and (terminate.(), terminate.()) because the renaming relation is total.

Renaming, however, may result in nondeterminism in the RoboChart semantics, which is excluded in our renaming operator. For example, the renaming of MemorySTM_MoveSTM for the state machine MoveSTM in Fig. 8 blocks both self transitions of the state move that have an input trigger update?l.

$$\texttt{Renamed\_MemorySTM\_MoveSTM = MemorySTM\_MoveSTM } [\![\texttt{event\_map}]\!]$$

The `event_map` contains the mappings like `(update_.(TID_t1, din, v), update.(din, v))` and `(update_.(TID_t3, din, v), update.(din, v))` where `v` is a literal integer number in `core_int`. For `v` that is between `-MAX` and `MAX`, the two events `update_.(TID_t1, din, v)` and `update_.(TID_t3, din, v)` are both enabled for communication (see the definition of `MoveSTM_MemoryTrans` where both predicates in the two set comprehensions are satisfied) and they are mapped to a same event `update.(din, v)` in this relation. According to the definition of renaming in Sect. 3.4, both events are blocked and so the state move is not able to take the two self transitions. This problem can be solved by using the renaming with priority defined in Sect. 3.6.

$$\texttt{Renamedp\_MemorySTM\_MoveSTM = MemorySTM\_MoveSTM } [\![\texttt{event\_map\_list}]\!]_p$$

The `event_map_list` has the same mappings as `event_map`, but it is a list (and so the mappings are ordered) instead of a relation. This renaming gives priority to the event at the front of the list. If `(update_.(TID_t3, din, v), update.(din, v))` is after `(update_.(TID_t1, din, v), update.(din, v))`, then `t1` has a priority and so the nondeterminism is resolved. This corresponds to moving towards only one direction (to the left) when the robot is in Section S2 in Fig. 7, instead of the nondeterministic choice of two directions. If we change the order of the two mappings, then `t3` will have a priority and the direction of the movement to the right will be chosen.

The semantics of a state machine also needs to take its termination (for example the final state is reached) into consideration. This is shown in the process for MoveSTM below.

$$\texttt{D\_MoveSTM = (Renamedp\_MemorySTM\_MoveSTM } [\![\{\texttt{terminate.()}\} \rhd \textit{skip}) \setminus \texttt{internal\_events}$$

Based on the definition of the exception operator, if `Renamedp_MemorySTM_MoveSTM` ever performs a `terminate` event, then *skip* will take over and so the process terminates. The process also hides all `internal` events and flow control events, defined in `internal_events`.

4.7. Operations

Operations in RoboChart can be provided by robotic platforms such as move and randomWalk in Fig. 1, or defined by state machines such as changeDirection in Fig. 3.

The semantics of a call (an action) to an operation that is provided by a robotic platform is an event to record the call with appropriate parameters. For example, the semantics of move(lv,a), the entry action of the state Going in Fig. 6, is `get_a?a` → `moveCall!(const_lv, a)` → ✓ where the message on the channel `moveCall` contains the value `const_lv` of the constant variable lv and the value of the local variable a, retrieved from the memory through `get_a`.

The semantics of a state machine-defined operation is different from that of a state machine because an operation is not an independent execution element like a state machine. Its behaviour is within the scope of the state machine that calls the operation. For this reason, the semantics of the operation does not include a separate channel type (or an event alphabet) and does not have a separate memory. Instead, all channels required for the operation are declared along with the channels for the caller state machine in a channel-type declaration. For example, the channel type `Chan_Movement` of Movement has the following additional channels for the operation changeDirection.

```
chantype Chan_Movement =
  ...
  internal_changeDirection :: TIDS_changeDirection
  enter_changeDirection    :: "SIDS_changeDirection×SIDS_changeDirection"
  entered_changeDirection  :: "SIDS_changeDirection×SIDS_changeDirection"
  exit_changeDirection     :: "SIDS_changeDirection×SIDS_changeDirection"
  exited_changeDirection   :: "SIDS_changeDirection×SIDS_changeDirection"
  terminate_changeDirection:: unit
  get_l_changeDirection    :: "Location_Loc"
  set_l_changeDirection    :: "Location_Loc"
```

The get_l and set_l channels are for the parameter l of the operation and not for a local variable l (indeed, there is no such local variable). Different from local variables, l is only set once (set_l) by the caller of the operation for passing its value, not inside the operation like local variables. The call changeDirection(l) to the operation in the entry action of the state Avoiding has its semantics in CSP as follows.

```
CALL__changeDirection_Movement =
  get_l_Movement?l → set_l_changeDirection!l → ✓; D_changeDirection
```

The first input event gets the value of the local variable l of Movement and the value is recorded in l. The second event updates the value of the parameter l (in the memory) of changeDirection to l. Finally, the process for this call behaves like the process D_changeDirection for changeDirection.

We note that the process D_changeDirection is not similar to the process D_MoveSTM to have a composition of the node process and the memory process of the machine. D_changeDirection basically is a node process with appropriate event hiding and termination. The memory of changeDirection is part of the memory of Movement and so it is composed in parallel with the node process of Movement like the memory of Movement.

4.8. Controllers

The event alphabet of the process for a controller contains a termination channel, shared variable channels, event channels, and operation call channels. The event channels include not only the events of the controller but also those in connections between its state machines. The channel type of the controller Ctrl in Fig. 8 is such one example.

```
chantype Chan_Ctrl =
  terminate_Ctrl          :: unit
  set_x_Ctrl              :: core_int
  get_x_Ctrl              :: core_int
  set_EXT_x_Ctrl          :: core_int
  set_EXT_x_Ctrl_CalSTM   :: core_int
  set_EXT_x_Ctrl_MoveSTM  :: core_int
  rec_Ctrl                :: "InOut×core_int"
  reset_Ctrl              :: "InOut"
  update_Ctrl             :: "InOut×core_int"
```

This channel type includes a terminate channel, two get and set channels for the required shared variable x, three set_EXT channels for accepting the update of x (through set_EXT_x_Ctrl) and then propagating the update to the state machines of the controller that require x: CalSTM and MoveSTM (through set_EXT_x_Ctrl_CalSTM and set_EXT_x_Ctrl_MoveSTM), and three event channels. The event channels are not only for the events rec and reset of Ctrl but also for the event update that is used for communication between the two state machines.

The memory of a controller deals with the update of shared variables. The memory of Ctrl is such an example.

```
Memory_Ctrl = loop (λid.
    set_EXT_x_Ctrl?x → set_EXT_x_Ctrl_CalSTM!x → set_EXT_x_Ctrl_MoveSTM!x → ✓_id)
```

This process accepts an update to x through an input event and propagates it to CalSTM and MoveSTM through two output events.

Parallel composition of the heterogeneous state machine processes (D_CalSTM and D_MoveSTM, for example, have different event alphabets) for a controller requires they all share a common event type E, and so we rename them. The events of the state machines are renamed to the corresponding events in the controller alphabet, according to the connections between the controller and its state machines. For example, D_CalSTM is renamed based on a renaming relation that maps the events of D_CalSTM to the events of D_Ctrl, such as (set_x_CalSTM.v, set_x_Ctrl.v) and (update_CalSTM.(dout, v), update_Ctrl.(dout, v)). D_MoveSTM is similarly renamed but its renaming relation contains a mapping (update_MoveSTM.(din, v), update_Ctrl.(dout, v)) with opposite directions din and dout. Finally, both update_CalSTM.(dout, v) and update_MoveSTM.(din, v) are renamed to the same event update_Ctrl.(dout, v), which allows the two state machines to communicate on the channel update through parallel composition according to the directions (CalSTM uses it for output and MoveSTM uses it for input) of the event update in the connection between the state machines.

The semantics of a controller is a parallel composition of the composed state machine processes and its memory with appropriate event hiding and termination. The process for Ctrl is given as follows.

```
1 D_Ctrl = ((((D_CalSTM ⟦CalSTM_events_map⟧
2                  ‖stms_events
3                  D_MoveSTM ⟦MoveSTM_events_map⟧
4                ) \ (stms_events - {terminate_Ctrl.()})
5              ) ‖mem_events Memory_Ctrl
6            ) \ mem_events
7          )  ⟦{terminate_Ctrl.()} ▷ skip
```

The parallel composition of the two state machines of Ctrl is defined on lines 1 to 3 where the events of the two machine processes (D_CalSTM and D_MoveSTM) are renamed to those of Ctrl and smts_events are a set of events including the termination event terminate_Ctrl and the events update.(dout,v) used for communication between the machines. The events used for communication then are hidden on line 4. The process after hiding is then composed in parallel with the memory of Ctrl on line 5 where mem_event is a set of variable events used for propagating the update of shared variables to the state machines, including set_EXT_x_Ctrl_CalSTM.v and set_EXT_x_Ctrl_MoveSTM.v. These events are also hidden on line 6. Finally, the exception on line 7 deals with the termination of Ctrl.

4.9. Modules

Similar to the event alphabet of the process for a controller, that of the process for a module also contains a termination channel, shared variable channels, 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. The semantics for this connection is a one-place buffer.

```
buffer = loop (λla.
    (((length la ≥ 0)∧(length la ≤ 1)) &turn?(dout, a∈Chemical_Angle) → ✓[a]) □
    ((length la > 0) &turn!(din, hd la) →  ✓[])
  )
```

The state of a buffer process is of type V list where V is the value type (Chemical_Angle) of the event channel (turn) corresponding to the event (turn) of the connection (c) which this buffer process is for. The buffer process is an infinite loop whose body is an external choice of two processes: (1) if the length of its state (a list) is either 0 or 1, it accepts an input (a) on the channel turn whose event direction is dout (and so the output from MainController) and then terminates with the updated state (a new list [a] with the only element a); and (2) if its state is not an empty list, it outputs the head element (hd la) of the list on the channel turn whose event direction is din (and so the input to MicroController) and then terminates with the update state (an empty list []).

We also note that when renaming the processes for controllers, it is not necessary to swap the direction of the event channels for the events of the controllers that are connected asynchronous. This is because of the usage of a buffer having connected the output of the event on one controller and the input of the event

```
 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: ()
```

Figure 11: Animation of the example when dangerous chemical detected.

on another controller through shared buffer elements. This is different from renaming the processes for state machines where one side uses an opposite direction.

The memory of a module deals with the update of shared variables and also the propagation of the update to its controllers, which is similar to that of a controller. The definition of the process for a module is omitted for simplicity and can be found online (D_ChemicalDetector for the autonomous chemical detector and D_PatrolMod for the patrol robot).

5. Code generation, animation, and case studies

As discussed previously in [10, Sect. 5], the animation of ITrees is achieved through code generation [13] 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. Using the same approach for animation, we are able to animate the two RoboChart models shown in Sect. 2.

5.1. Autonomous chemical detector

We illustrate two scenarios of the animation of the autonomous chemical detector in Figs. 11 and 12. 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 represent 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. 12, we omit the lines for enabled events and append the chosen event to the chosen number for simplicity.

Figure 11 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. 6; (2) then a sequence of gas sensor readings is received through the gas event, and we choose number 9 (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. 12, 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 (online #2); (3) the controllers call the move operation (the entry action of the state Going in Fig. 6) 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

```
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]: ...
```

Figure 12: Animation of the example when chemical detected with low intensity.

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 (online #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 the 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. In our implementation in ITrees, the buffer process defined previously for the connection reflects this behaviour: overwriting the buffer is always allowed.

5.2. The patrol robot

In this example, we instantiate MAX_INT to 3 and MAX to 2 and illustrate three scenarios of the animation of the patrol robot corresponding to the three sections (S1, S2, and S3) of the corridor in Fig. 7.

We show the first scenario in Fig. 13, which is related to the calibrated position in S1. The model behaves as follows: (1) initially the controller provides eight events on lines #2-4 for users to choose: one to reset the position and another seven to set the calibrated position to an integer value between -3 and 3; (2) the second event (2) is chosen on line #5, denoting the calibrated position is -3 and so the robot is in S1; (3) the only available event on lines #6-10 is Right_PatrolMod[7] which corresponds to the right event in the RoboChart model, denoting the movement of the robot towards the right side of the corridor at the new positions (-2, -1, and 0 respectively); (4) since the new position on line #10 is 0 now, the controller could accept a right event and calibration events on lines #11-13; (5) the right event is chosen on line 14, and after that (6) the controller returns to its initial state: having the same available events on lines #15-17 as initially available events on lines 2-4. When x is equal to -3 and -2 (in section S1), the robot moves towards the right side and x is increased by 1, as illustrated on line #6-9. This behaviour is consistent with the semantics of the model. After x becomes -1 (in section S2), the model nondeterministically chooses to move towards the left side or the right side. Our animation on lines #6-10, and #14, however, shows only the right side is chosen. This

[7] The change of the name from right_PatrolMod to Right_PatrolMod is due to the code generation in Isabelle to generate Haskell. In Haskell, it is conventional to use capitalised names for data types.

```
1  Starting ITree Simulation...
2  Events: (1) Reset_PatrolMod Din; (2) Cal_PatrolMod (Din,-3); (3) Cal_PatrolMod (Din,-2);
3          (4) Cal_PatrolMod (Din,-1); (5) Cal_PatrolMod (Din,0); (6) Cal_PatrolMod (Din,1);
4          (7) Cal_PatrolMod (Din,2); (8) Cal_PatrolMod (Din,3);
5  [Choose: 1-8]: 2 Cal_PatrolMod (Din,-3)
6  [Choose: 1-1]: Right_PatrolMod (Dout,-2)
7  [Choose: 1-1]: Right_PatrolMod (Dout,-2)
8  [Choose: 1-1]: Right_PatrolMod (Dout,-1)
9  [Choose: 1-1]: Right_PatrolMod (Dout,-1)
10 [Choose: 1-1]: Right_PatrolMod (Dout,0)
11 Events: (1) Right_PatrolMod (Dout,0); (2) Cal_PatrolMod (Din,-3); (3) Cal_PatrolMod (Din,-2);
12         (4) Cal_PatrolMod (Din,-1); (5) Cal_PatrolMod (Din,0); (6) Cal_PatrolMod (Din,1);
13         (7) Cal_PatrolMod (Din,2); (8) Cal_PatrolMod (Din,3);
14 [Choose: 1-8]: 1 Right_PatrolMod (Dout,0)
15 Events: (1) Reset_PatrolMod Din; (2) Cal_PatrolMod (Din,-3); (3) Cal_PatrolMod (Din,-2);
16         (4) Cal_PatrolMod (Din,-1); (5) Cal_PatrolMod (Din,0); (6) Cal_PatrolMod (Din,1);
17         (7) Cal_PatrolMod (Din,2); (8) Cal_PatrolMod (Din,3);
18 [Choose: 1-8]:
```

Figure 13: Animation of the patrol robot when the calibrated position is in S1.

is because of the use of renaming with priority in Renamedp_MemorySTM_MoveSTM and the higher priority of the update event on t1 than t3 to resolve the nondeterminism (and so the priority is given to the movement towards the right side).

In the RoboChart model, we expect each left or right event corresponds to decrease or increase x by 1. The animation, however, shows that the new positions (-2, -1, and 0) on the right event on lines #7, #9, and #14 stay the same as their previous positions on lines #5, #8, and #10. This is actually due to the semantics of shared variables in RoboChart, specifically the mechanism used to update shared variables and propagate the updates, which is subtle. We illustrate the implemented mechanism in our semantics in Fig. 14 where the exchange of the value of x in the module, the controller, and the two machines is through communication (labelled with an identifier, a channel, and a message over the channel).

The communications 1 to 4 show that the change of x to -2 in CalSTM is updated to the module PatrolMod, and then this update is propagated down the memory hierarchy to the controller Ctrl, subsequently to the state machines CalSTM and MoveSTM. The update and propagation, however, are not atomic. Between these communications, the memories can be accessed and evaluated using the outdated value. This is further demonstrated by communications 5 to 11. Consider a new update of x to -1 (by communication 5) in the memory of MoveSTM, the update and propagation are similar to the previous update (to -2). We here, however, consider the value of x in CalSTM is accessed (by the event get_x on communication 8) and evaluated in the guard [x!=0] of the transition in the machine before the new value -1 is propagated to the machine (on communication 9). This means the action of the transition still outputs -2 on the event update!-2, not the -1 because the new value has not been seen in this machine. As a consequence, the input trigger update?l in the machine MoveSTM will receive a value -2 and so x is set to -1 (x=l+1) again, as indicated on communication 11. The updates of x to -1 on communications 5 and 11 correspond to the action x=l+1 (rc.Plus l 1 rc.core_int_set in our semantics) in the self transition of MoveSTM, which is followed by an output event right!x. The animation, therefore, shows two Right_PatrolMod events on lines #8 and #9.

It is worth mentioning that the RoboChart semantics in this model with the shared variable x has a high degree of nondeterminism because of the interleaving of events between the module, the controller, and the state machines, and eventually nondeterminism due to the hiding of these interleaving events. Our implementation of the semantics reduces nondeterminism in a particular way: the maximal progress assumption (internal events τ have a higher priority) [10]. We also note that the animated behaviour of

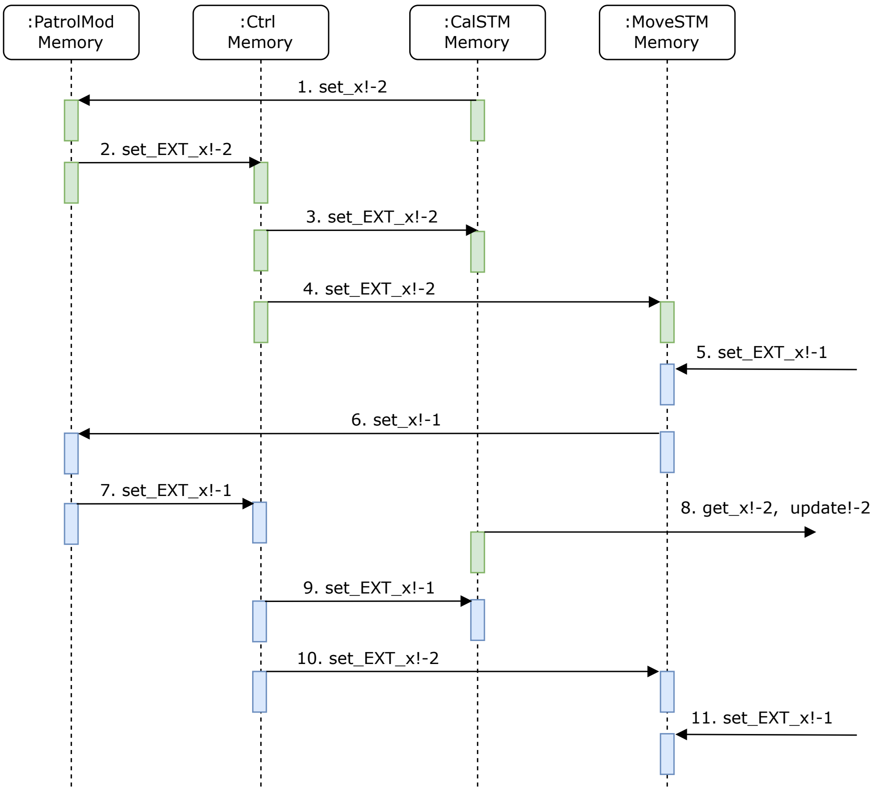

Figure 14: The update of the shared variable x and its propagation in the patrol robot model.

two Right_PatrolMod events for a position is one behaviour of the RoboChart's standard semantics. This has been verified using FDR that this scenario is a trace refinement of the standard semantics (generated in RoboTool). In order for the verification, we encode the scenario in Fig. 13 in a CSP process Scenario1 below and then use FDR to check the assertion satisfied.

```
Scenario1 = PatrolMod::cal.in.-3 -> PatrolMod::right.out.-2 -> PatrolMod::right.out.-2 ->
            PatrolMod::right.out.-1 -> PatrolMod::right.out.-1 -> PatrolMod::right.out.0 ->
             PatrolMod::right.out.0 -> Scenario1
assert PatrolMod [T= Scenario1
```

In the assertion, PatrolMod is the CSP process for the module PatrolMod in the generated CSP semantics in RoboTool.

Though semantically allowed, the model does not reflect the optimal way to use shared variables. We could, for example, design models to allow only one state machine to update a shared variable and other state machines to access its value or add additional events (such as start_update and end_update) to enforce a synchronisation of updates to shared variables. Our patrol robot model here is presented to reveal the subtle semantics of using shared variables.

In Fig. 15, we consider the second scenario where the calibrated position is 1 (in S2). The model behaves as follows: (1) the sixth event (6) is chosen on line #5, denoting the calibrated position is 1 and so the robot is in S2; (2) then the robot moves towards the right side at position 2 (lines #6 and #7); (3) subsequently the robot moves towards the left direction to position 1 (lines #8 and #9); and (4) finally, the robot repeats steps (2) and (3) to patrol between position 1 and position 2. We also verified this scenario is a trace refinement of the standard semantics, shown below.

```
Repeat = PatrolMod::right.out.2 -> PatrolMod::right.out.2 ->
```

```
1  Starting ITree Simulation...
2  Events: (1) Reset_PatrolMod Din; (2) Cal_PatrolMod (Din,-3); (3) Cal_PatrolMod (Din,-2);
3          (4) Cal_PatrolMod (Din,-1); (5) Cal_PatrolMod (Din,0); (6) Cal_PatrolMod (Din,1);
4          (7) Cal_PatrolMod (Din,2); (8) Cal_PatrolMod (Din,3);
5  [Choose: 1-8]: 6 Cal_PatrolMod (Din,1)
6  [Choose: 1-1]: 1 Right_PatrolMod (Dout,2)
7  [Choose: 1-1]: 1 Right_PatrolMod (Dout,2)
8  [Choose: 1-1]: 1 Left_PatrolMod (Dout,1)
9  [Choose: 1-1]: 1 Left_PatrolMod (Dout,1)
10 [Choose: 1-1]: 1 Right_PatrolMod (Dout,2)
11 [Choose: 1-1]: 1 Right_PatrolMod (Dout,2)
12 [Choose: 1-1]: 1 Left_PatrolMod (Dout,1)
13 [Choose: 1-1]: 1 Left_PatrolMod (Dout,1)
14 [Choose: 1-1]: ...
```

Figure 15: Animation of the patrol robot when the calibrated position is in S2.

```
1  Starting ITree Simulation...
2  Events: (1) Reset_PatrolMod Din; (2) Cal_PatrolMod (Din,-3); (3) Cal_PatrolMod (Din,-2);
3          (4) Cal_PatrolMod (Din,-1); (5) Cal_PatrolMod (Din,0); (6) Cal_PatrolMod (Din,1);
4          (7) Cal_PatrolMod (Din,2); (8) Cal_PatrolMod (Din,3);
5  [Choose: 1-8]: 8 Cal_PatrolMod (Din,3)
6  [Choose: 1-1]: 1 Left_PatrolMod (Dout,2)
7  [Choose: 1-1]: 1 Left_PatrolMod (Dout,2)
8  [Choose: 1-1]: 1 Left_PatrolMod (Dout,1)
9  [Choose: 1-1]: 1 Left_PatrolMod (Dout,1)
10 [Choose: 1-1]: 1 Right_PatrolMod (Dout,2)
11 [Choose: 1-1]: 1 Right_PatrolMod (Dout,2)
12 [Choose: 1-1]: ...
```

Figure 16: Animation of the patrol robot when the calibrated position is in S3.

```
        PatrolMod::left.out.1 -> PatrolMod::left.out.1 -> Repeat
Scenario2 = PatrolMod::cal.in.1 -> Repeat
assert PatrolMod [T= Scenario2
```

The third scenario, we consider, is shown in Fig. 16 where initially the calibrated position is 3 (in S3) on line #5. The robot starts to move towards the left side to position 2 (on lines #6 and #7) and position 1 (on lines #8 and #9), and then towards the right side back to position 2 (on lines #10 and #11). After that, the behaviour is the same as that in the second scenario in Fig. 15. Similarly, we verified this scenario is a trace refinement of the standard semantics.

```
Scenario2 = PatrolMod::cal.in.3 -> PatrolMod::left.out.2 -> PatrolMod::left.out.2 ->
            PatrolMod::left.out.1 -> PatrolMod::left.out.1 -> Repeat
assert PatrolMod [T= Scenario3
```

From the three scenarios, we have seen that the event Reset_PatrolMod (reset in the model) is only enabled when the current position (the value of x on the event left or right) is 0. The events enabled on lines #2 and #15 in Fig. 16 and on line #2 in Figs. 16 and 16 are such examples. The standard CSP semantics, however, allows reset when the current position is other than 0. The following analysis using FDR illustrates it clearly.

```
Reset = PatrolMod::cal.in.-2 -> PatrolMod::left.out.-1 ->
```

```
2         PatrolMod::reset.in -> PatrolMod::right.out.0 ->
3         PatrolMod::right.out.1 -> PatrolMod::reset.in -> Reset
4 assert PatrolMod [T= Reset
```

In this example, reset is enabled when x is -1 (on line #2) and 1 (on line #3). This difference is due to the maximal progress assumption in the definition of external choice and hiding in our approach: internal events have priority over external events. In this patrol robot model, the event update is internal and reset is external, so update has priority over reset. When x is not 0, the guard [x!=0] in the self transition of Cal in the machine CalSTM is true, which enables CalSTM to communicate with the machine MoveSTM on update, and then the transition with the trigger reset cannot be taken due to its lower priority than update. The reset, therefore, is only enabled when x is 0.

6. Related work

Animation is a lightweight formal method. Kazmierczak et al. [24] 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. [25] 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 the animation of formal specifications. The ProB animator and model checker provides a different framework [15]. 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 [26] 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.

Rosu et al. [27] develop K,[8] a rewriting-based executable semantic framework. The operational semantics of programming languages such as C [28] and Java [29] are proposed based on K. Our ITree-based approach is also an executable semantic framework enabling the definition of operational semantics, but for both abstract specification languages and concrete refinements. And so program development by refinement is supported in our framework. Higher-order logic and nondeterminism are some features of interaction trees, but not for K.

Stateflow is a graphical language integrated in the Simulink of Matlab to model and simulate decision logic using state machines and flow charts. During simulation, transitions in state machines are evaluated, by default, based on the order in which they are created [30]. This is the same as our approach to resolve nondeterministic choice between transitions using the prioritised renaming operator.

7. Conclusions

This work gives RoboChart an ITree-based operational semantics and enables the animation of RoboChart using code generation in Isabelle/HOL. To provide animation support, we extend ITree-based CSP

[8] https://kframework.org/

with extra 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 model and a patrol robot model. 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 chemical detector example and three concrete scenarios of the patrol robot model using animation. The analysis using FDR shows these scenarios are trace refinements of the standard RoboChart CSP semantics, and so our approach gives and animates a refinement of the original models.

This work targets deterministic RoboChart models and also nondeterministic RoboChart models (but nondeterminism is resolved in a priority way in the semantics and so deterministic semantics eventually). Our work covers a large part of RoboChart features (but not all). Our immediate future work is to investigate the support of nondeterminism in the semantics and give semantics to more features such as hierarchical state machines and timed semantics.

In this paper, we manually translate the RoboChart semantics to Isabelle. Particularly, we take RoboChart's CSP semantics generated in RoboTool into account to define consistent and also restricted semantics based on an optimised version of the CSP semantics. This practical consideration could entitle us to largely reuse the current CSP semantics generator in RoboTool to automatically generate ITree-based CSP semantics. Then the whole workflow from RoboChart models to Haskell code can be fully automated and our work here brings insights into it. This is part of our future work.

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 logic as a property language for the 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) [31, 32], which will allow us to give an 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. We also could use this approach to automatically generate a sound runtime monitor from RoboChart models to observe the behaviour of systems that are derived from the models.

We use a basic textual animation in this work. This will be improved to directly allow the visualisation of RoboChart models in RoboTool for animation. Eventually, users of RoboTool are able to animate a state machine, or a controller, or a whole model by clicking transitions or events.

Acknowledgements

This work is funded by the EPSRC projects CyPhyAssure[9] (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, and are licensed under CC 3.0 BY.

References

[1] A. Cavalcanti, W. Barnett, J. Baxter, G. Carvalho, M. C. Filho, A. Miyazawa, P. Ribeiro, A. Sampaio, RoboStar Technology: A Roboticist's Toolbox for Combined Proof, Simulation, and Testing, Springer International Publishing, Cham, 2021, pp. 249–293. doi:10.1007/978-3-030-66494-7_9.
URL https://doi.org/10.1007/978-3-030-66494-7_9

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

[3] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, J. Woodcock, RoboChart: modelling and verification of the functional behaviour of robotic applications, Softw. Syst. Model. 18 (5) (2019) 3097–3149. doi:10.1007/s10270-018-00710-z.

[4] K. Ye, A. Cavalcanti, S. Foster, A. Miyazawa, J. Woodcock, Probabilistic modelling and verification using RoboChart and PRISM, Software and Systems Modeling (Oct 2021). doi:10.1007/s10270-021-00916-8.
URL https://doi.org/10.1007/s10270-021-00916-8

[9] CyPhyAssure Project: www.cs.york.ac.uk/circus/CyPhyAssure/.

[5] J. Woodcock, A. Cavalcanti, S. Foster, A. Mota, K. Ye, Probabilistic Semantics for RoboChart, in: P. Ribeiro, A. Sampaio (Eds.), Unifying Theories of Programming, Springer International Publishing, Cham, 2019, pp. 80–105.

[6] K. Ye, S. Foster, J. Woodcock, Automated Reasoning for Probabilistic Sequential Programs with Theorem Proving, in: U. Fahrenberg, M. Gehrke, L. Santocanale, M. Winter (Eds.), Relational and Algebraic Methods in Computer Science, Springer International Publishing, Cham, 2021, pp. 465–482.

[7] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall Int., 1985.

[8] A. W. Roscoe, Understanding Concurrent Systems, Texts in Computer Science, Springer, 2011.

[9] L.-y. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, S. Zdancewic, Interaction Trees: Representing Recursive and Impure Programs in Coq, Proc. ACM Program. Lang. 4 (POPL) (Dec. 2019). doi:10.1145/3371119.
URL https://doi.org/10.1145/3371119

[10] S. Foster, C.-K. Hur, J. Woodcock, Formally Verified Simulations of State-Rich Processes Using Interaction Trees in Isabelle/HOL, in: S. Haddad, D. Varacca (Eds.), 32nd International Conference on Concurrency Theory (CONCUR 2021), Vol. 203 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2021, pp. 20:1–20:18. doi:10.4230/LIPIcs.CONCUR.2021.20.
URL https://drops.dagstuhl.de/opus/volltexte/2021/14397

[11] S. D. Brookes, C. A. R. Hoare, A. W. Roscoe, A Theory of Communicating Sequential Processes 560–599doi:10.1145/828.833.
URL https://doi.org/10.1145/828.833

[12] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, A. W. Roscoe, FDR3 - A Modern Refinement Checker for CSP, in: Tools and Algorithms for the Construction and Analysis of Systems, 2014, pp. 187–201.

[13] F. Haftmann, T. Nipkow, Code Generation via Higher-Order Rewrite Systems, in: M. Blume, N. Kobayashi, G. Vidal (Eds.), Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings, Vol. 6009 of Lecture Notes in Computer Science, Springer, 2010, pp. 103–117. doi:10.1007/978-3-642-12251-4_9.
URL https://doi.org/10.1007/978-3-642-12251-4_9

[14] R. Mayr, T. Nipkow, Higher-order rewrite systems and their confluence, Theoretical computer science 192 (1) (1998) 3–29.

[15] M. Leuschel, M. J. Butler, ProB: A Model Checker for B, in: K. Araki, S. Gnesi, D. Mandrioli (Eds.), FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings, Vol. 2805 of Lecture Notes in Computer Science, Springer, 2003, pp. 855–874. doi:10.1007/978-3-540-45236-2_46.
URL https://doi.org/10.1007/978-3-540-45236-2_46

[16] K. Ye, S. Foster, J. Woodcock, Formally Verified Animation for RoboChart Using Interaction Trees, in: A. Riesco, M. Zhang (Eds.), Formal Methods and Software Engineering, Springer International Publishing, Cham, 2022, pp. 404–420.

[17] J. A. Hilder, N. D. L. Owens, M. J. Neal, P. J. Hickey, S. N. Cairns, D. P. A. Kilgour, J. Timmis, A. M. Tyrrell, Chemical Detection Using the Receptor Density Algorithm, IEEE Trans. Syst. Man Cybern. Part C 42 (6) (2012) 1730–1741. doi:10.1109/TSMCC.2012.2218236.

[18] A. Miyazawa, A. Cavalcanti, P. Ribeiro, K. Ye, W. Li, J. Woodcock, J. Timmis, RoboChart Reference Manual, Tech. rep., University of York, www.cs.york.ac.uk/circus/publications/techreports/reports/robochart-reference.pdf (2020).

[19] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, D. Traytel, Truly Modular (Co)datatypes for Isabelle/HOL, in: G. Klein, R. Gamboa (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, Vol. 8558 of Lecture Notes in Computer Science, Springer, 2014, pp. 93–110. doi:10.1007/978-3-319-08970-6_7.
URL https://doi.org/10.1007/978-3-319-08970-6_7

[20] J. M. Spivey, The Z Notation: A Reference Manual, 2nd, Prentice-Hall, 1992.

[21] I. Toyn (Ed.), Information Technology — Z Formal Specification Notation — Syntax, Type System and Semantics, ISO, 2002, iSO/IEC 13568:2002(E).

[22] C. Ballarin, Locales and Locale Expressions in Isabelle/Isar, in: S. Berardi, M. Coppo, F. Damiani (Eds.), Types for Proofs and Programs, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 34–50.

[23] M. Pickering, J. Gibbons, N. Wu, Profunctor Optics: Modular Data Accessors, Art Sci. Eng. Program. 1 (2) (2017) 7. doi:10.22152/programming-journal.org/2017/1/7.

[24] E. Kazmierczak, M. Winikoff, P. W. Dart, 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, pp. 254–261. doi:10.1109/APSEC.1998.733727.

[25] Y. Boichut, T. Genet, Y. Glouche, O. Heen, Using Animation to Improve Formal Specifications of Security Protocols, in: 2nd Conference on Security in Network Architectures and Information Systems (SARSSI 2007, 2007, pp. 169–182.

[26] D. H. O. de Souza, Joker: An Animator for Formal Languages, Ph.D. thesis, Departamento de Informática e Matemática Aplicada, Universidade Federal do Rio Grande do Norte (2011).

[27] G. Rosu, T. Serbanuta, An overview of the K semantic framework, J. Log. Algebraic Methods Program. 79 (6) (2010) 397–434. doi:10.1016/j.jlap.2010.03.012.

[28] C. Ellison, G. Rosu, An executable formal semantics of C with applications, in: J. Field, M. Hicks (Eds.), Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012, ACM, 2012, pp. 533–544. doi:10.1145/2103656.2103719.

[29] D. Bogdanas, G. Rosu, K-java: A complete semantics of java, in: S. K. Rajamani, D. Walker (Eds.), Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015, ACM, 2015, pp. 445–456. doi:10.1145/2676726.2676982.

[30] MathWorks, Stateflow: User's Guide (R2022b), The MathWorks Inc., 2022.
URL www.mathworks.com/help/pdf_doc/stateflow/stateflow_ug.pdf

[31] J. G. Kemeny, J. L. Snell, A. W. Knapp, Denumerable Markov Chains, 1976. doi:10.1007/978-1-4684-9455-6.

[32] J. G. Kemeny, J. L. Snell, Finite Markov Chains: With a New Appendix "Generalization of a Fundamental Matrix" (Undergraduate Texts in Mathematics), Springer, 1983.