Article:
Ye, Kangfeng, Woodcock, Jim orcid.org/0000-0001-7955-2702 and Foster, Simon David orcid.org/0000-0002-9889-9514 (2024) Formally verified animation for RoboChart using interaction trees. Journal of Logical and Algebraic Methods in Programming. 100940. ISSN 2352-2216

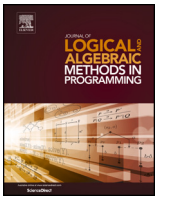
https://doi.org/10.1016/j.jlamp.2023.100940

J. Log. Algebraic Methods Program. 137 (2024) 100940

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp



Formally verified animation for RoboChart using interaction trees ☆

Kangfeng Ye *, Simon Foster, Jim Woodcock

Department of Computer Science, University of York, Deramore Lane, Heslington, YO10 5GH, York, United Kingdom

ARTICLE INFO

Keywords:
Interaction trees
CSP
Operational semantics
Animation of robot software
Theorem proving
Code generation

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, 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 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 and patrol robot models, 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 for the patrol robot when its calibrated position is in other corridor sections. We also verify that the animated scenarios are trace refinements of the CSP denotational semantics of the RoboChart models using FDR, a refinement model checker for CSP. This ensures that our approach to resolve nondeterminism using CSP operators with priority is sound and correct.

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 for 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 to model robotic software, which provides UML-like architectural and state machine modelling notations. RoboChart is distinguished by its formal semantics [3,5,4], which enables automated verification using model checking and theorem proving [6]. In RoboChart, physical robots are abstracted into robotic platforms through variables, events, and opera-

☆ This document presents results from the research project CyPhyAssure (https://www.cs.york.ac.uk/circus/CyPhyAssure/, Grant EP/S001190/1) funded by EPSRC.

* Corresponding author.
E-mail addresses: kangfeng.ye@york.ac.uk (K. Ye), simon.foster@york.ac.uk (S. Foster), jim.woodcock@york.ac.uk (J. Woodcock).

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

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

https://doi.org/10.1016/j.jlamp.2023.100940
Received 28 February 2023; Received in revised form 18 December 2023; Accepted 19 December 2023

Available online 28 December 2023

tions. RoboSim [7] is a domain-specific language for simulation in robotics. It can be seen as a correct implementation or refinement of RoboChart into the simulation level in terms of control software, called d-model. RoboSim is also enriched to specify robotic platforms' physical and dynamic behaviours, provided through p-models. The d-model and p-model of RoboSim are related through a platform mapping, which describes how variables, events, and operations in the d-model are interpreted as continuous variables in the p-model. The hybrid models then can be verified using Differential Hoare logics [8,9], or hardware/software co-verification [10]. Our work presented here targets the animation of RoboChart models for robotic control software, and so users can interact with RoboChart models through provided command line interfaces. From this aspect, users play a role in the environment of the models. They inspect the behaviour of the models by choosing what the model is allowed to do and observing its response.

Previous work [3] gives RoboChart a denotational semantics based on the CSP process algebra [11,12]. This paper defines direct operational semantics for RoboChart using Interaction Trees (ITrees) [13]. ITrees support a coinductive encoding of labelled transition systems that can model infinite behaviours of a reactive system interacting with its environment. Crucially, ITrees provide abstract yet directly executable semantics, which means they can support animation to support model exploration, verification using coinductive proof techniques, and generation of correct-by-construction implementations [14]. An ITree-based semantics for RoboChart thus further empowers developers with techniques for prototyping and analysis of models. ITrees have been mechanised in both Coq [13] and Isabelle/HOL [14]. ITrees also unify trace-based denotational failures-divergences semantics [15,12] for CSP, transition-based operational semantics, and algebraic semantics, as demonstrated in our previous work [14].

The existing implementation of RoboChart's semantics in RoboTool is restricted to machine-readable CSP (or CSP-M) for verification with FDR [16], a CSP refinement checker, and so only a subset of RoboChart's rich types and expressions (which is based on that of the Z notation [17]) can be supported, and quantified predicates cannot be solved. Our contribution here is a richer ITree-based CSP semantics for RoboChart in terms of types and expressions to address these restrictions, thanks to the mechanised Z toolkit[3] [18] in Isabelle/HOL. Our semantics also allow us to characterise systems with an infinite number of states symbolically, avoiding the need to generate an explicit transition system.

We mechanise the ITree-based CSP semantics in Isabelle/HOL, which ensures that all our definitions and proofs are theoretically well-grounded, yet also practically applicable in development tasks. Isabelle/HOL provides us with an array of tools to support software engineering and verification, including a flexible syntax frontend, integration of automated theorem provers, and code generation. Animation of a RoboChart model is realised in two stages. The first stage generates the ITree-based CSP semantics for the model, and the second stage utilises the code generator [19] in Isabelle/HOL to automatically produce Haskell code for animation. We note that the work presented in this paper results from manually generating CSP semantics for RoboChart models in the first stage. Supporting full automation for semantics generation is part of our future work. In the second stage, Isabelle's code generator [19] translates executable definitions in the source HOL logic to target functional languages (such as SML and Haskell), and the translation preserves semantic correctness using higher-order rewrite systems [20]. 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 for 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 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 list or sequence type (that is not defined in the Z toolkit) 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 several 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 [12] as in the animator of ProB [21] and FDR. All definitions and theorems in this paper are mechanised, and accompanying icons () link to corresponding repository artefacts. We assume basic knowledge of Isabelle/HOL from interested readers to understand these definitions and theorems.

The remainder of this paper is organised as follows. Section 2 gives a brief discussion of CSP and then introduces RoboChart through two examples: an autonomous chemical detector model and a patrol robot model. Section 3 briefly describes the mechanisation of ITrees in Isabelle, lists the semantics of previously defined CSP operators, and presents the extra CSP operators in detail. Then we introduce the RoboChart semantics in ITrees in Sect. 4 using general mathematical rules, 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 [22]. 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. It 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. 2.2.2 and illustrated in Sect. 5.2 for its animation. We also add semantic rules and important details in Sect. 4 to give a clear

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

Table 1
A summary of CSP processes and operators.

Symbol	Name	Description
τ	internal event	Invisible event.
Skip	skip	Terminate immediately without change to the state.
Stop	deadlock	Refuse any interaction with the state unchanged.
$c \rightarrow P$	prefix	Synchronise on channel c and then behave like P.
$c?x \rightarrow P(x)$	input	Accept an input of any value (of type T) on channel c if c has a type T, record the value on variable x, and then behave like $P(x)$.
$c?x : S \rightarrow P(x)$	restricted input	Similar to input, but it only accepts the values from set S.
$c!v \rightarrow P$	output	Synchronise on channel c with value v, and then behave like P.
$b \ \& P$	guarded process	Behave like P if b is true, and deadlock otherwise.
$P \Box Q$	external choice	Offer the environment a choice of the first events of P and Q, and then behave accordingly.
$P \sqcap Q$	internal choice	Nondeterministic choice between P and Q without offering the environment a choice.
$P; Q$	sequential composition	Behave like P initially, and behave like Q if P terminates.
$P \triangle Q$	interrupt	Behave like P, but offer the environment a choice of the initial events of Q at any time until P terminates. If one of these events is performed, Q takes over and behaves accordingly.
$P \Theta_E Q$	exception	Behave like P until P performs an event from set E, at that point, then behave like Q.
$P \parallel_E Q$	generalised parallel composition	P and Q run simultaneously, synchronise on the events in E, progress independently on the events not in E, and terminate if both terminate.
$P \mid\mid\mid Q$	interleave	Equal to $P \parallel_{\varnothing} Q$ where P and Q always progress independently on any event.
$P \setminus E$	hiding	Behave like P except that the events from E become internal.
$P[\![c \leftarrow d]\!]$	renaming	Rename the event c in P to d.
$\Box i : I \bullet P(i)$	replicated external choice	Offer a choice of the indexed processes $P(i)$ by i from set I. The similar replicated internal choice, sequential composition, parallel composition, and interleaving are omitted here.

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. We give more details and examples to illustrate the semantics of the existing subsections. 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 with examples.

2. CSP and RoboChart

2.1. CSP

Communicating Sequential Processes or CSP [23,12] is a well-established process algebra to model concurrent systems using communication (message-passing via channels) for the interaction between processes. It has primitives for specifying sequential behaviour and concurrent interaction using events. Table 1 summarises the CSP processes and operators used in our semantics for RoboChart.

CSP has several rich semantic models [12], including traces, stable failures, failures-divergences, and refusal testing models based on its denotational semantics and characterising the observable behaviours of processes. CSP also has consistent denotational and operational semantics [12]: all these semantic models have congruence theorems in the operational semantics. CSP also has been extended to specify time in concurrent systems like *tock*-CSP [12,24] and Timed CSP [12,25], and to specify shared variables, priority, and mobility [12] in concurrent systems. Several languages are also built on CSP, such as Circus [26], CML [27], and RoboChart presented here.

FDR is a widely used model-checking tool for verifying CSP processes through refinement: $P \sqsubseteq_{\mathcal{M}} Q$ represents a specification P is refined by an implementation Q in terms of a semantic model $\mathcal{M}$. FDR supports hierarchical compression methods[4] to tackle the state space explosion problem in model checking. According to [16], the cluster version of FDR3 is able to analyse problems with 10^{12} compressed states (approximately 10^{20} uncompressed states). FDR also has a built-in CSP process animator called ProBE [16], which can be used to manually explore the full transition tree of a CSP process.

This paper applies an ITree-based semantics for CSP, which we developed previously [14]. It is consistent with the failures-divergences semantics [12] but focuses on deterministic CSP processes, which are directly executable. Since our encoding of CSP is symbolic and implicit, we can avoid the state explosion problem.

[4] https://cocotec.io/fdr/manual/cspm/prelude.html#compressions

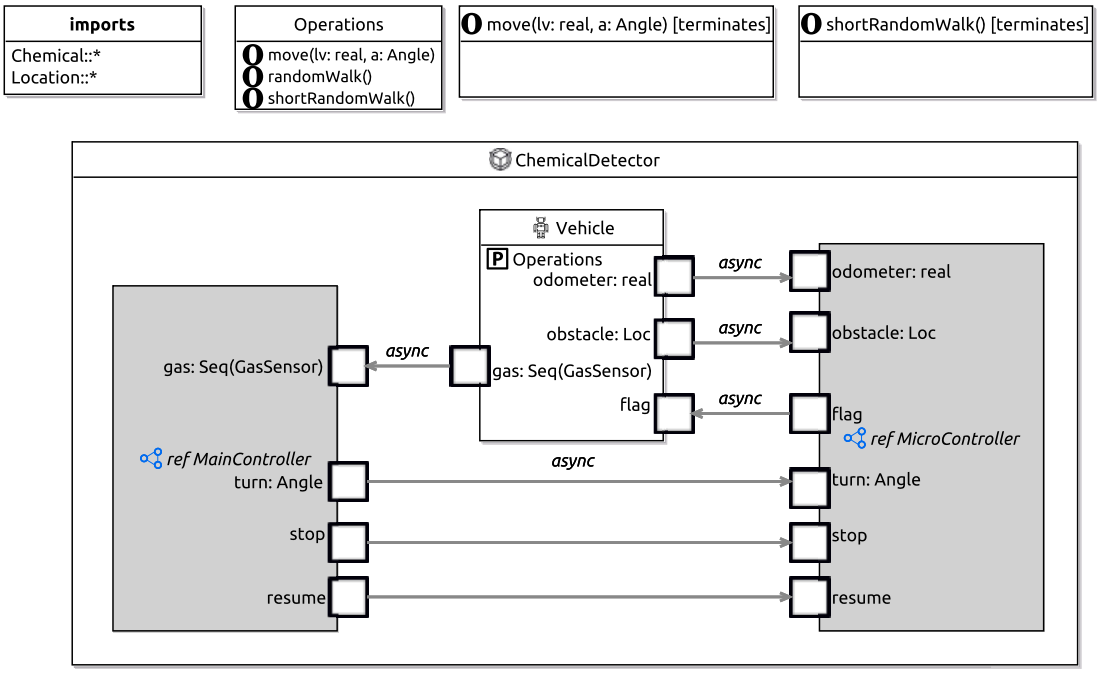


Fig. 1. The module of the autonomous chemical detector model.

2.2. RoboChart

RoboChart is motivated by problems in the current practice of programming robotic applications using standard state machines: (a) no precise syntax and formal semantics, (b) informally discussed time and uncertainty requirements, (c) loosely connected artefacts, (d) no tool support, and (e) no assurance. RoboChart is a state machine-based notation designed for roboticists to model robotic control software and apply modern verification techniques. RoboChart has precise syntax and formal semantics and allows users to specify time and probability features in state machines formally. Modelling using RoboChart and verification is supported by its accompanying tool, RoboTool. Roboticists can use RoboTool to create RoboChart models using consistent diagrammatic and textual modelling. They can either draw diagrams (and RoboTool automatically generates the corresponding textual model) or program models (RoboTool can also automatically render its diagrammatic representation).

RoboChart has an unambiguous mathematical semantics. RoboTool automatically generates semantics for models and applies verification techniques like model checking and theorem proving to analyse the semantics based on the properties specified by users. RoboChart models and properties use languages familiar to roboticists. Other artefacts, like semantics, are automatically linked to the models and properties, making modern software engineering and verification techniques more accessible for roboticists.

RoboChart also has a component model with notions of controller, module, and state machines to foster reuse. In a RoboChart model, physical robots are abstracted into robotic platforms through variables, events, and operations. We describe some of the features of RoboChart for modelling controllers of robots using two examples: an autonomous chemical detector [28,3,29] in Sect. 2.2.1 and a patrol robot in Sect. 2.2.2. We refer to the RoboChart reference manual [29] for a complete account of the notation and its semantics.

2.2.1. 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[5] [3] has been studied and analysed in RoboTool, using FDR.

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 the

[5] https://robostar.cs.york.ac.uk/case_studies/autonomous-chemical-detector/autonomous-chemical-detector.html

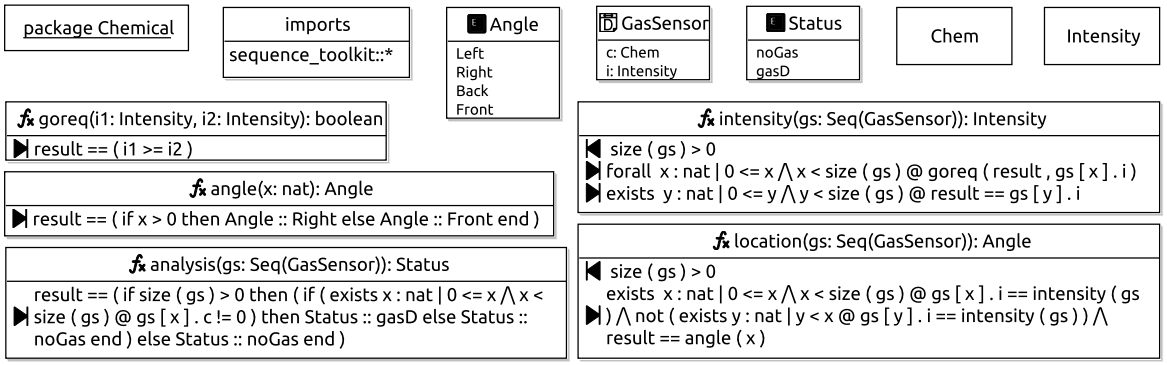


Fig. 2. The Chemical package of the autonomous chemical detector model.

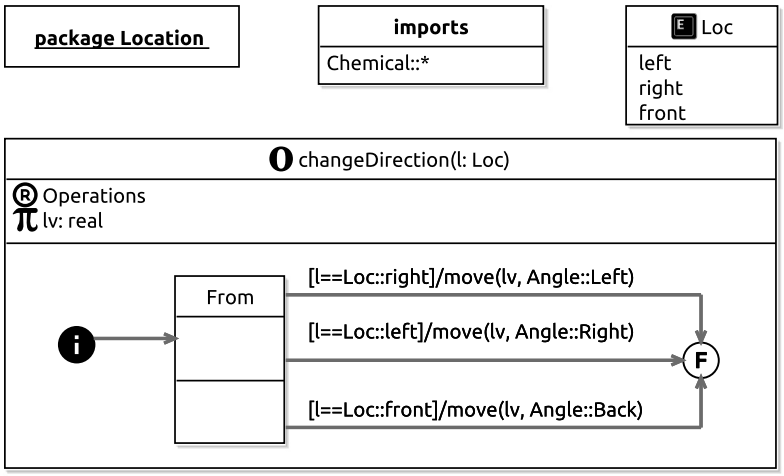


Fig. 3. The Location package of the autonomous chemical detector model.

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

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 Figs. 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 changeDirection using a state machine. The operation has a parameter l of type Loc and a constant lv. 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 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

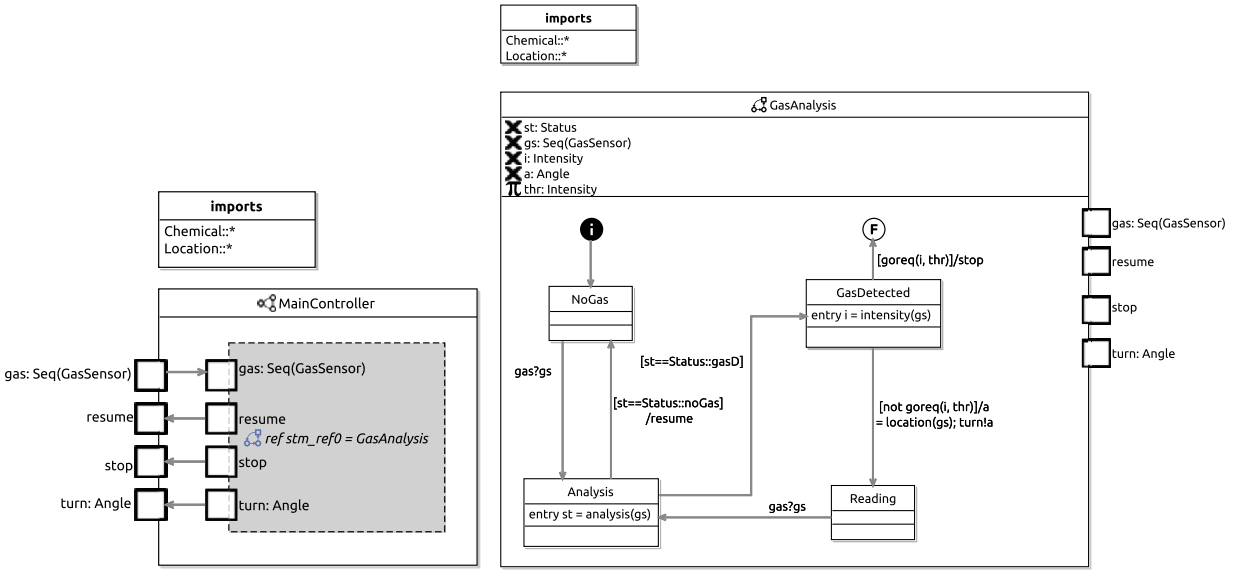


Fig. 4. MainController and GasAnalysis state machine of the autonomous chemical detector model.

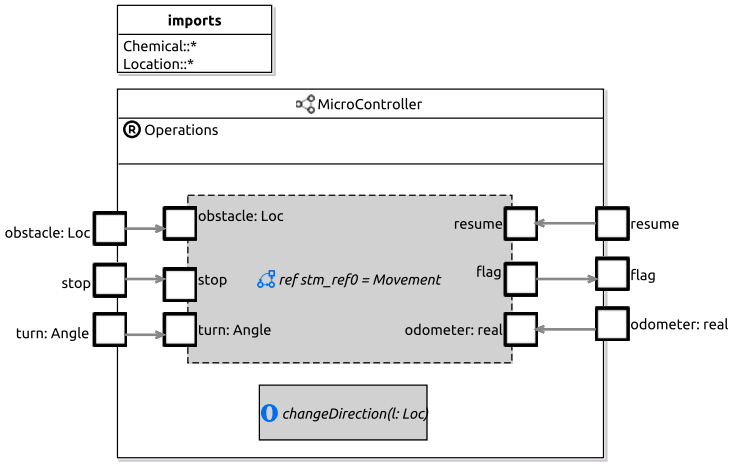


Fig. 5. MicroController of the autonomous chemical detector model.

gas?gs enabling the machine to receive sensor readings from the channel gas and store the value in the variable gas, 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 transition 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 changeDirection. 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,

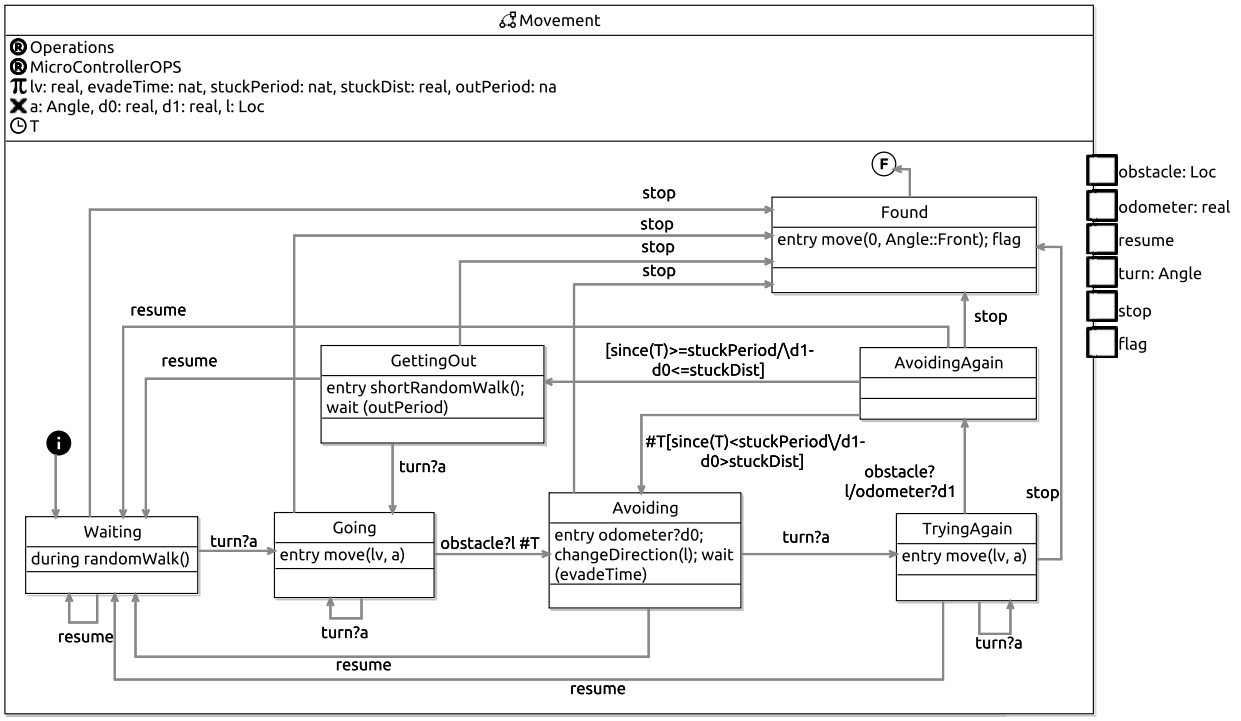


Fig. 6. Movement state machine of the autonomous chemical detector model.

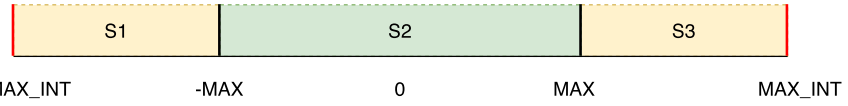


Fig. 7. Sections for the patrol robot: S1 - adjacent to the left boundary; S2 - central area; S3 - adjacent to the right boundary.

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 and means the robot is doing a random walk when waiting for further instructions from MainController based on the gas analysis result. 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.

2.2.2. One-dimensional patrol robot

In addition to the features introduced in the chemical detector example, RoboChart supports abstraction via nondeterministic choice and interaction via shared variables, 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, and the robot will tend to move to S2 if it is in these sections.

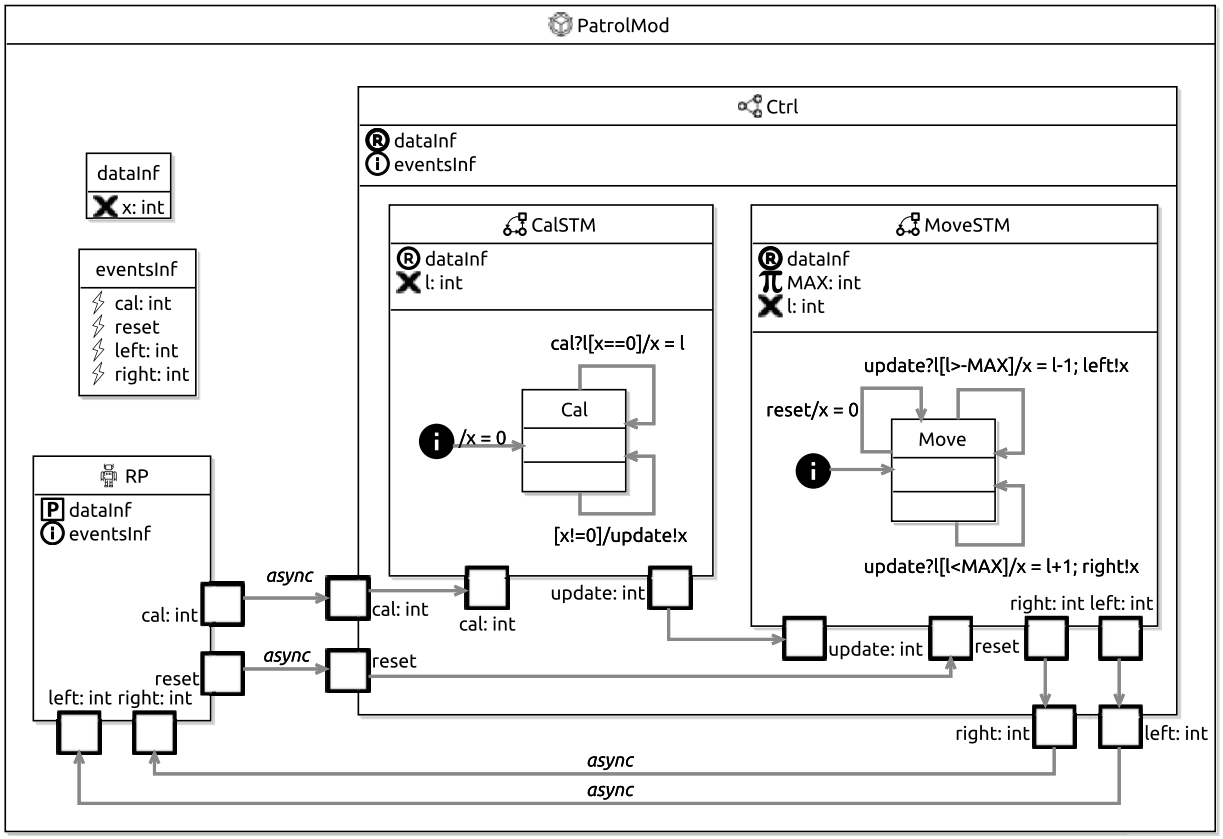


Fig. 8. A RoboChart model for a patrol robot with nondeterministic behaviour and interaction using shared variables.

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.

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 to the two machines by requesting dataInf.

The machine CalSTM sets x to 0 in 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. This illustrates 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. The robot movement around the corridor is based on the value (of variable x), which passes on update and is 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 the model and will be given in verification or animation where int is bounded.

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

3. Interaction trees

This section briefly introduces interaction trees and extends our existing CSP semantics with additional operators to support the RoboChart semantics. These include three operators (interrupt, exception, and renaming) introduced in the previous work [22], one generalised choice operator, and two prioritised hiding and renaming operators presented in this extension paper.

Interaction trees (ITrees) [13] are a data structure for modelling reactive systems interacting with their environment through events. They are potentially infinite and defined as coinductive trees [30] 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 with a value x of type R returned, denoted as $\checkmark_x$[6]; (2) *Sil*P: an internal silent event, denoted as τP for a successor ITree P; or (3) *Vis*F: a choice among several visible events represented by a partial function F of type $E \nrightarrow (E, R)$*itree*. Partial functions are part of the Z toolkit[7] [18], which is also mechanised in Isabelle/HOL. We also use a notation $[]\, e \in E \to P(e)$ for $Vis(\lambda\, e \in E \bullet P(e))$.

Though ITrees are elementary structures, they can be used to encode, animate, and symbolically reason about the complex behaviour of systems of interacting concurrent processes. ITrees is a model that gives semantics to various languages and potentially unifies them. In particular, deterministic CSP processes can be given executable semantics using ITrees. Determinism is inherent since we use a partial function to model events and their continuations. Precisely, 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 ITree-based semantics in Isabelle/HOL and generates Haskell code for the underlying ITree. Though this is often an infinite structure, Haskell's intrinsic use of lazy evaluation allows us 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 removed, and the successor ITree is animated. This allows us to compress long sequences of τ events and simplified animation. Finally, *Ret* leads to the termination of the animation.

Previously [14], 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 V over the channel c); (3) output prefixing $c!v$ (send a value v over the channel c); (4) *guard*b (guarded based on a Boolean value b); (5) external choice $P \,\Box\, Q$ between two processes P and Q; (6) parallel composition $P \parallel_A Q$; (7) hiding $P \setminus A$; (8) sequential composition $P \fatsemi Q$; (9) *loop* and *iterate*. We summarise their definitions in Table 2 and show some definitions omitted in the table (due to the large space they will take) as follows.

External choice $P \,\Box\, Q$ is defined corecursively. A corecursive definition can have several equations ordered by priority.

$$(Vis\,F) \,\Box\, (Vis\,G) = Vis(F \odot G)$$

$$(Sil\,P') \,\Box\, Q = Sil(P' \,\Box\, Q)$$

$$P \,\Box\, (Sil\,Q') = Sil(P \,\Box\, Q')$$

$$(Ret\,x) \,\Box\, (Vis\,G) = Ret\,x$$

$$(Vis\,F) \,\Box\, (Ret\,y) = Ret\,y$$

$$(Ret\,x) \,\Box\, (Ret\,y) = (if\, x = y\, then\, (Ret\,x)\, else\, stop)$$

The merge function $F \odot G \mathrel{\widehat{=}} (\mathrm{dom}(G) \lhd F) \oplus (\mathrm{dom}(F) \lhd G)$ is used to define the *Vis* case. The $\lhd$ is called the domain anti-restriction, and $A \lhd R$ denotes the domain restriction of relation R to the complement of set A. The $\oplus$ is a relational overriding operator. For example, $A \oplus B$ agrees with the relation B and with the relation in A outside the domain of B. The relation in A inside the domain of B is overridden by B. This function $F \odot G$ combines all event maplets from F and G, 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 avoids nondeterminism; when the two domains are disjointed, the operator can be considered a union.

Sequential composition $(P \fatsemi Q)$ is defined for Kleisli trees [13,31] P (of type $S_1 \Rightarrow (E, S_2)$*itree*) and Q (of type $S_2 \Rightarrow (E, S_3)$*itree*). A Kleisli tree is a Kleisli lifting of an ITree. The Kleisli composition is supported through a monadic bind operator ($\gg\!=$):

$$P \fatsemi Q \mathrel{\widehat{=}} (\lambda\, x.P(x) \gg\!= Q)$$

[6] We note that $\checkmark_x$ is just a shorthand for *Ret*x and $\checkmark$ cannot be used alone. In some CSP literature, such as [12], $\checkmark$ means a special event representing termination. We do not refer to the single $\checkmark$ in this paper.

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

Table 2
A summary of the existing basic CSP operators defined in the previous work [14]. The definitions for $\Box$, $\|_A$, $|||$, $\setminus$, and $;$ are omitted in the table (because their definitions take a large space) and are presented and discussed in the overview of Sect. 3.

Operator	Definition	Description
skip	$Ret()$	Terminate immediately, and return a unit type (), a degenerate form of *Ret*.
stop	$Vis\{\mapsto\}$	Deadlock, or no event is possible.
div	**primcorec** $div = \tau\, div$	ITree of infinite depth, a divergent ITree that does not terminate and only performs internal activity.
$runE$	$[]\, e \in E \to runE$	Repeatedly perform any event from E. We give an alternative definition instead of the original definition for simplicity.
$inp\, c\, V$	$Vis\left(\begin{array}{l}\lambda e \in \left(\begin{array}{l}\mathrm{dom}(match_c)\\ \cap build_c(\!| A |\!)\end{array}\right)\\ \bullet Ret(match_c\ e)\end{array}\right)$	Accept a constrained input (only the values from V) over channel c and return the value.
$outp\, c\, v$	$Vis\{build_c\ v \mapsto Ret()\}$	Send a value v over channel c, or denoted as $c!v$.
$guard\ b$ $b\ \&\ P$	**if** b **then** *skip* **else** *stop* **do**$\{guard\ b;\ P\}$	Behave as *skip* if b is true and otherwise *stop*. Guarded process.
$P\, ; Q$		Sequential composition of P and Q.
$P \Box Q$		External choice, redefined in Sect. 3.1.
$P \|\|_A Q$		Parallel composition of P and Q over A.
$P \|\|\| Q$		Interleave of P and Q.
$P \setminus A$		Hide the events in A from P.
$iterate\, b\, P\, s$	**if** $(b\, s)$ **then** $Sil(P\, s \gg\!= iterate\, b\, P)$ **else** $\checkmark_s$	Continue to execute P while the condition $b\, s$ holds and otherwise terminates and returns the current state s. Also called *while* in [14].
$loop\, P$	$iterate\, (\lambda s.True)\, P$	Infinite loop.

where $P(x)$ defines an ITree given an input value x. The composition, defined on the right of the definition, executes P first, and then passes the value of x to its continuation Q upon the termination of P. The $P \gg\!= Q$ is defined corecursively, including several equations ordered by priority.

$$\checkmark_r \gg\!= Q = Q(r)$$

$$\tau P' \gg\!= Q = \tau(P' \gg\!= Q)$$

$$Vis F \gg\!= Q = Vis\,(\lambda e \in \mathrm{dom}(F) \bullet F(e) \gg\!= Q)$$

With the monadic **do** notation, we can write a sequential composition like the one shown below.

$$\textbf{do}\{x \leftarrow inp\, c\, V;\ outp\, d\, (x+1);\ \checkmark_x\}$$

This is equivalent to the composition:

$$inp\, c\, V \gg\!= \left(\lambda x.outp\, d\, (x+1) \gg\!= (\lambda y.\checkmark_x)\right)$$

This process accepts input from set V on channel c, records the value in x, passes x to its continuation, which sends $x+1$ on channel d, and afterwards terminates and returns the value of x.

Parallel composition $P \|_E Q$ over a set E of events is defined corecursively below.

$$(Sil P') \|_E Q = Sil(P' \|_E Q)$$

$$P \|_E (Sil Q') = Sil(P \|_E Q')$$

$$(Vis F) \|_E (Vis G) = Vis\left(\begin{array}{l}\{e \mapsto (P' \|_E (Vis G)) \mid (e \mapsto Left(P')) \in merge_E(F,G)\}\\ \oplus \{e \mapsto ((Vis F) \|_E Q') \mid (e \mapsto Right(Q')) \in merge_E(F,G)\}\\ \oplus \{e \mapsto (P' \|_E Q') \mid (e \mapsto Both(P',Q')) \in merge_E(F,G)\}\end{array}\right)$$

$$(Ret x) \|_E (Ret y) = Ret(x,y)$$

$$(Ret x) \|_E (Vis G) = Vis\{e \mapsto \left(Ret x \|_E Q'\right) \mid (e \mapsto Q') \in G\}$$

$$(Vis F) \|_E (Ret y) = Vis\{e \mapsto \left(P' \|_E Ret y\right) \mid (e \mapsto P') \in F\}$$

The definition of the *Vis* case uses an operator $merge_E(F, G)$ to merge two event functions. We omit its definition here for simplicity and refer to [14] for more details. For the sake of presentation, we present partial functions as sets and use set comprehensions for construction. A set comprehension $\{e \mid P\}$ is a shorthand for $\{e \mid xy.P\}$ if x and y are free variables of e and occur in P. For example, $\{e \mapsto (Ret x \parallel_E Q') \mid (e \mapsto Q') \in G\}$ in the above definition means $\{e \mapsto (Ret x \parallel_E Q') \mid e\, Q'.(e \mapsto Q') \in G\}$, that is, any maplet $e \mapsto Q'$ (a visible event e and its continuation Q') in the partial function G becomes $e \mapsto (Ret x \parallel_E Q')$.

Interleave $P \mid\mid\mid Q$ is simply the parallel composition over the empty set: $P \parallel_\varnothing Q$.

Hiding $P \setminus A$ is defined corecursively below.

$$Vis F = \begin{cases} Sil\,(F(e) \setminus A) & \textbf{if } A \cap \mathrm{dom}(F) = \{e\} \\ Vis\,\{(e, P \setminus A \mid (e, P) \in F\} & \textbf{if } A \cap \mathrm{dom}(F) = \varnothing \\ stop & \text{Otherwise} \end{cases}$$

$$Sil(P) \setminus A = Sil(P \setminus A)$$

$$Ret x \setminus A = Ret x$$

The hiding operator is restricted to at most one event once to avoid nondeterminism introduced by hiding multiple events. However, hiding more than one event can be achieved through hiding with priority described in Sect. 3.5.

Though operators like external choice and parallelism can introduce nondeterminism, we restrict this by construction. Nevertheless, different strategies such as biased operators, which give priority to the left-hand side or right-hand side process of the operators, and priority based on an order of events for hiding or renaming, can be employed for statically resolving nondeterminism, which we explore further in this section. We use these strategies to define alternative operators for different purposes.

Next, 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 resolve nondeterminism based on an order statically. We restrict ourselves to deterministic operators as it makes the animation of large models more efficient.

3.1. Generalised choice and external choice

Previously [14], we have given 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 to resolve any nondeterminism statically. 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 $E \nrightarrow (E, R)itree$. The operator is defined as a corecursive function using the equations listed below.

Definition 3.1 *(Generalised choice).*

$$(Vis F) \boxdot_{\mathcal{M}} (Vis G) = Vis\,(\mathcal{M}\, F\, G) \qquad (Ret x) \boxdot_{\mathcal{M}} (Ret y) = (\textbf{if } x = y \textbf{ then } Ret x \textbf{ else } stop)$$

$$(Sil P') \boxdot_{\mathcal{M}} Q = Sil\,(P' \boxdot_{\mathcal{M}} Q) \qquad P \boxdot_{\mathcal{M}} (Sil Q') = Sil\,(P \boxdot_{\mathcal{M}} Q')$$

$$(Ret v) \boxdot_{\mathcal{M}} (Vis G) = Ret v \qquad (Vis F) \boxdot_{\mathcal{M}} (Ret v) = Ret v$$

When choosing 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 the two possible values to be identical and otherwise deadlocked 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 redefine external choice $P \,\Box\, Q \mathrel{\widehat{=}} P \boxdot_{\odot} Q$.

Another benefit of the generalised choice operator is that, as with Hoare and He's parallel-by-merge operator [2], its properties reduce to the merge function itself. This simplifies proof 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 well-formed because $\mathrm{dom}(\varnothing) = \varnothing$. From this definition of well-formedness, we obtain the following properties.

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

$$P \boxdot_{\mathcal{M}} stop = stop \boxdot_{\mathcal{M}} P = P$$

$$P \circledcirc_{\mathcal{M}} \mathit{div} = \mathit{div} \circledcirc_{\mathcal{M}} P = \mathit{div}$$

$$P \circledcirc_{\mathcal{M}} Q = Q \circledcirc_{\mathcal{M}^\sim} P$$

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, so a divergent process prevents choices. 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: $P \circledcirc_{\mathcal{M}} Q = Q \circledcirc_{\mathcal{M}} P$. These properties show 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 external choice. For example, $a \to P \,\Box\, a \to Q$ ends as *stop* since both processes have a as an initial event. Instead, We can resolve any nondeterminism by prioritising the addition of events from either the left or right branches. We introduce a biased choice operator, $P \mathbin{\triangleleft\Box} Q \mathrel{\widehat{=}} Q \circledcirc_{\oplus} P$, which chooses events from P whenever initial events are present in both P and Q. For example, $a \to P \mathbin{\triangleleft\Box} a \to Q = a \to P$, since the event from the left branch is prioritised.

3.2. Interrupt

The second operator we introduce is interrupt [11,12], $P \bigtriangleup Q$, which behaves like P except that if at any time Q performs one of its initial events, it takes over. 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).*

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

The *Sil* cases allow τ events to happen independently with priority and without resolving $\bigtriangleup$. The *Ret* cases terminate with x returned from $\bigtriangleup$'s left or right side.

The *Vis* case also results in a *Vis* process constructed from an overriding $\oplus$ of the further two sets, representing two partial functions. In the partial function, $(\mathrm{dom}(G) \lhd F)$ 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, not the initial event of Q, can occur independently (without resolving the interrupt), and its continuation is a corecursive call $P' \bigtriangleup Q$. The second function is just G, which denotes that the initial events of Q can happen whether they are in F or not. 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).*

$$\begin{array}{ll} (\mathit{Ret}\, x) \llbracket A \rhd Q = \mathit{Ret}\, x & (\mathit{Sil}\, P') \llbracket A \rhd Q = \mathit{Sil}\,(P' \llbracket A \rhd Q) \\ (\mathit{Vis}\, F) \llbracket A \rhd Q = \mathit{Vis}\left(\begin{array}{l}\{e \mapsto (P' \llbracket A \rhd Q) \mid (e \mapsto P') \in (A \lhd F)\} \oplus \\ \{e \mapsto Q \mid e \in (A \cap \mathrm{dom}(F))\}\end{array}\right) & \end{array}$$

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

Similar to Definition 3.3, the *Vis* case is also represented by the overriding of two partial functions. The first partial function means that an initial event e of P that is not in A (that is, $e \in \mathrm{dom}(A \lhd F)$) can occur independently. 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 $\rho : E_1 \leftrightarrow E_2$, which is equivalent to $\mathbb{P}(E_1 \times E_2)$. This relation is possibly heterogeneous, 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 with duplicate distinct values. This is the case when the renaming relation is functional, restricted to the initial events of P.

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

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

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

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

Definition 3.5 *(Renaming).*

$$\begin{aligned}
&(Ret x)\,[\![\rho]\!] = Ret x\\
&\left(Sil P'\right)[\![\rho]\!] = Sil\left(P'[\![\rho]\!]\right)\\
&(Vis F)\,[\![\rho]\!] = \left(\begin{array}{l}\textbf{let } G = F \circ mk_functional\left((\mathrm{dom}(F) \lhd \rho)^{\sim}\right)\\ \bullet Vis\left(\left\{e_2 \mapsto \left(P'[\![\rho]\!]\right) \mid \left(e_2 \mapsto P'\right) \in G\right\}\right)\end{array}\right)
\end{aligned}$$

The *Ret* case behaves like P, and the renaming does not affect 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 $(E_2 \nrightarrow (E_1, R)itree)$ that is the backward partial function composition $\circ$ of F and a partial function made using *mk_functional* from the inverse $^{\sim}$ of the relation $(\mathrm{dom}(F) \lhd \rho)$ which is the domain restriction $\lhd$ of ρ to the domain $\mathrm{dom}(F)$ of F. 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 $P'[\![\rho]\!]$ where $\left(e_2 \mapsto P'\right) \in G$.

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

$$\begin{aligned}
&\left(e_1 \to P \,\Box\, e_2 \to Q \,\Box\, e_3 \to R\right)[\![\{e_1 \mapsto e, e_2 \mapsto e, e_3 \mapsto ea, e_4 \mapsto eb\}]\!]\\
&= \left(ea \to R[\![\{e_1 \mapsto e, e_2 \mapsto e, e_3 \mapsto ea, e_4 \mapsto eb\}]\!]\right) \qquad \text{(renaming example 1)}
\end{aligned}$$

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 [12] 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 (CSP-M) supports and FDR implements is relational renaming. We investigated all these approaches and chose to implement the relational renaming with many-to-one and one-to-many mappings. This is mainly because RoboChart's semantics is defined using CSP-M 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 [14] of hiding $P \setminus A$ is deadlock if more than one initial event of P is in A. This is to avoid nondeterminism caused by the hiding of two possible events. This restriction can be relaxed by hiding events in an order. For example, $(a \to P \,\Box\, b \to Q) \setminus \{a, b\} = stop$, 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 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 equal to $\tau\,(P \setminus \{a\})$.

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

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

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, $foldl(f, P, [a, b])$ 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, so we can no longer rename other events of type E_1. For this reason, we define renaming with priority, $P[\![\varrho]\!]_p$, which renames events of P according to a finite sequence ϱ of type $\mathrm{seq}\left(E_1 \times E_2\right)$. We use the finite sequence type here to describe the mathematical definition of this operator, and its representation in Isabelle is a list. A

finite sequence of type, $\mathrm{seq}X$, is a finite partial function $\mathbb{N} \pfun X$ from natural numbers $\mathbb{N}$ to X. With ϱ, a priority is given based on the indices of pairs in the sequence to resolve potential nondeterminism in a particular way. For pairs with 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.

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

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

This function 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 [18]. Here, we use the selection operator $(s.i)$ to select the ith element in a tuple s. 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 $drop\_dup(\varrho)$, we need to drop the pairs with lower priority (and bigger indices). We define an auxiliary *least* function first.

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

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 with multiple events renamed to the same event as in p. Now $drop\_dup(\varrho)$ is defined below.

$$(\forall p \in drop\_dup(\varrho) \bullet least(\rho, p)) \land \qquad \text{(maximal)}$$

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

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

$$drop\_dup\left(\langle (e_1, e), (e_2, e), (e_4, eb)\rangle\right) = \langle (e_1, e), (e_4, eb)\rangle$$

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

Definition 3.6 *(Renaming with priority).*

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

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 $\mathbb{N} \pfun (E_1 \times E_2)$, 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.

$$\begin{aligned}
&\left(e_1 \to P \,\Box\, e_2 \to Q \,\Box\, e_3 \to R\right)[\![\langle (e_1, e), (e_2, e), (e_3, ea), (e_4, eb)\rangle]\!]_p \\
&= \left(\begin{array}{l} e \to P[\![\langle (e_1, e), (e_2, e), (e_3, ea), (e_4, eb)\rangle]\!]_p \,\Box \\ ea \to R[\![\langle (e_1, e), (e_2, e), (e_3, ea), (e_4, eb)\rangle]\!]_p \end{array}\right) \qquad \text{(renaming example 2)}
\end{aligned}$$

Compared to the (renaming example 1) where 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 regarding ITrees in Isabelle/HOL. These include types, instantiations, functions, state machines, controllers, and modules. In implementing RoboChart's semantics, we also consider the practical details of the CSP semantics generation in RoboTool, such as naming and bounded primitive types.

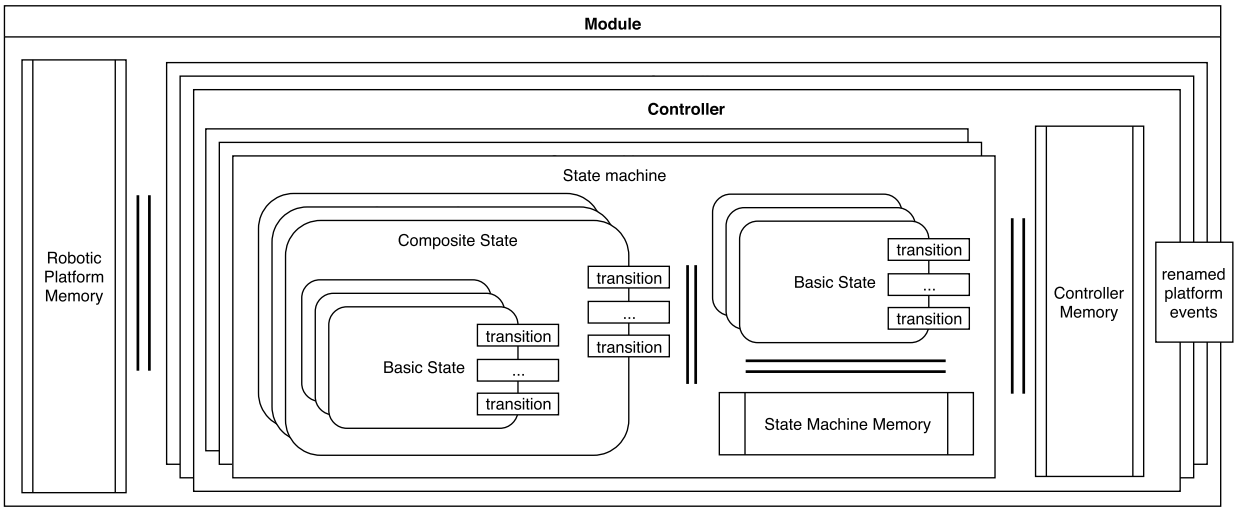


Fig. 9. From [3, Fig. 11]. Structure of the RoboChart semantics: stacked components and parallel lines indicate parallel composition; bordered boxes indicate points of interaction; the semantics of a container is composed of the semantics of its contained components.

In particular, the nondeterministic choice between transitions, for example, in Fig. 8, is resolved using the prioritised renaming operator defined earlier through ordered transition event mappings. This is not covered in the previous work [22] 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. However, the semantics for a robotic platform is different from controllers and state machines in that it does not have a particular behaviour. So, its semantics is not a CSP process because the platform is an abstraction of a physical robot through variables, events, and operations.

RoboChart has a hierarchical memory model with 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 in scope records the reads and writes of variables for components (the platform, controllers, and state machines). The memory for a state machine caches the (both local and shared) variables it requires. So, the semantics of the machine is independent of the location where these variables are declared. However, the memory for a controller or the platform differs 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. 2.2.1 is shown in Fig. 10. The module's semantics is a parallel composition of the two controllers with the robotic platform memory (RP memory) and a buffer process (Buffer). The buffer process models the asynchronous connection from MainController to MicroController on event turn in Fig. 1. For simplicity, this semantics for asynchronous connections is omitted in Fig. 9.

The platform and 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. 2.2.2, 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 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. Still, it is mandatory for the patrol robot because the controller includes 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.

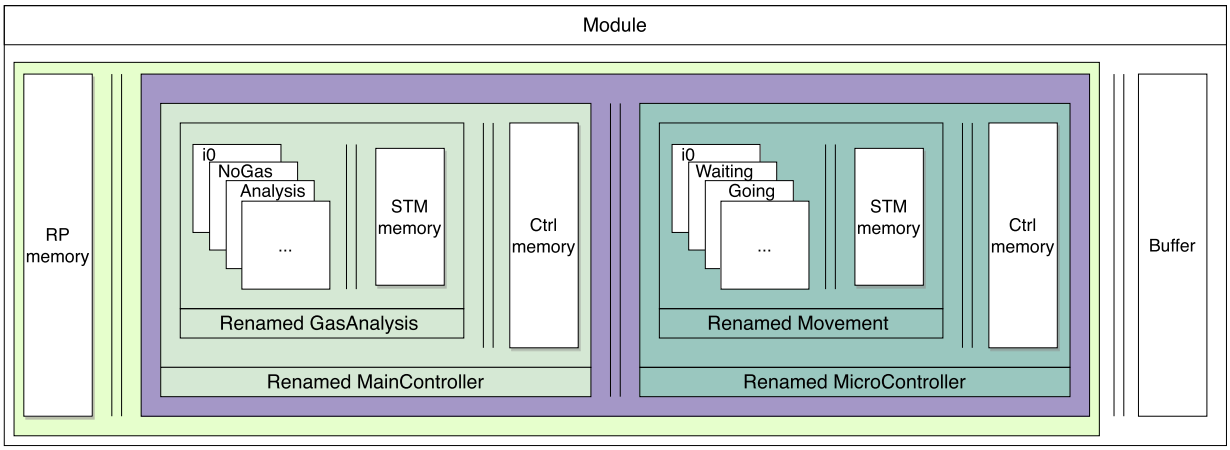


Fig. 10. RoboChart semantics of the autonomous chemical detector.

Practical consideration. In this paper, our ITree-based semantics for RoboChart aims for consistency with the restricted CSP semantics generated in RoboTool. Here, the restriction mainly 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 of reusing the current CSP semantics generator in RoboTool to generate corresponding ITrees-based theories for Isabelle to generate Haskell code automatically.

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 [12]. 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 the Z notation [17]. It supports basic types: PrimitiveType, Enumeration, records (or schema types), and other additional types from the mathematical toolkits of Z.

The core package of RoboTool provides five primitive types: Booleans, natural numbers, integer, real numbers, and strings. 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. We define the mappings below where we use the notation $[\![\mathsf{rc}]\!]_c = \mathtt{ic}$ to denote a mapping from a RoboChart entity rc into an ITree-based CSP counterpart `ic` in the context c. In other words, the semantics of rc in the context c is `ic` in our ITree-based CSP.

$$[\![\mathsf{bool}]\!]_t = \mathtt{bool} \quad [\![\mathsf{nat}]\!]_t = \mathtt{natural} \quad [\![\mathsf{int}]\!]_t = \mathtt{integer}$$

Here, the context t means the type.

For a record type T (datatype T) such as GasSensor in Fig. 2 in RoboChart, we use **`record`** in Isabelle. Its semantics is defined in the rule below.

$$[\![\mathsf{datatype\ T}]\!]_t = \quad \mathtt{record}\ \underline{\mathsf{T}}\ =\ \underline{\{\mathsf{f} : \mathsf{T.fields} \bullet \mathsf{f.name}\ \mathtt{::"}\ [\![\mathsf{f.type}]\!]_t\ \mathtt{"}\ ,\ \}}$$

We also use the notation $\underline{\mathsf{a}}$ to denote elements to be expanded, where we use the constructs from Z as a meta-notation and use `a` to denote the resultant construct in our ITree-based CSP in Isabelle. The rule results in the definition of a record type $\underline{\mathsf{T}}$ in Isabelle, which contains a set of fields corresponding to the fields ($\underline{\mathsf{T.fields}}$) in T. Here we use set comprehension $\underline{\{\mathsf{x} : \mathsf{T}_\mathsf{x} \bullet \mathsf{expr(x)}\}}$ from Z. For the sake of presentation, we introduce a comma , after the expression expr(x) to represent a character to separate each expression after the set is converted to concrete syntax in Isabelle. In the rule above, the character is empty, so there is a blank space to separate each expression. In the above rule, for each field f, we use its name $\underline{\mathsf{f.name}}$. The semantics of its type ($\underline{[\![\mathsf{f.type}]\!]_t}$) to construct each field in Isabelle by using Isabelle's syntax (`name::"type"`).

For an enumeration type such as Angle and Status in Fig. 2, we use **`datatype`** in Isabelle to define it.

$$[\![\mathsf{enumeration\ T}]\!]_t = \quad \mathtt{datatype}\ \underline{\mathsf{T}}\ =\ \underline{\{\mathsf{l} : \mathsf{T.literals} \bullet \mathsf{l.name},\ |\ \}}$$

Similar to the rule for the record type, we use set comprehension to get a set of constructors: one for each literal ⌊ in the enumeration type T. In Isabelle, only the name ⌊.name of the literal matters and constructors are separated by |.

RoboChart models can also have abstract primitive types (type T) 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 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
```

We show the rule to generate the semantics for primitive types below.

$$[\![\text{type T}]\!]_t = \begin{cases} \texttt{typedef } \underline{\texttt{T}}\texttt{T} = \texttt{"\{()\}" by auto} \\ \texttt{type\_synonym ('a)}\underline{\texttt{T}} \texttt{ = "(}\underline{\texttt{T}}\texttt{T, 'a) PrimType"} \end{cases}$$

This rule results in two statements in Isabelle: first to declare a new type ⌊T using **typedef**, and then the corresponding type ⌊ is just a synonym for `PrimType` instantiated to ⌊T. We note that ⌊T can be any type, and here we choose the type { () } containing only the unit `()`.

RoboChart additionally supports a large collection of data types from Z. We show the rules for set, product, and sequence types below and refer to the counterparts in the Z toolkit for other types.

$$[\![\text{Set(T)}]\!]_t = [\![\text{T}]\!]_t \texttt{ set} \qquad [\![\text{T}_1 * \text{T}_2]\!]_t = [\![\text{T}_1]\!]_t \times [\![\text{T}_2]\!]_t \qquad [\![\text{Seq(T)}]\!]_t = [\![\text{T}]\!]_t \texttt{ blist['n]}$$

The rules for set and product types are straightforward. Seq(T) in RoboChart is a type representing an infinite set of all finite (any length) sequences of elements of type *T*. We need to bind the size of the set and also the length of sequences for code generation. For this reason, 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. We introduce a notation `'a blist['n]` for this type. So the rule above for Seq(T) results in a type with bounded sequences. We describe the definition of bounded sequences in Isabelle below.

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

`CARD` here retrieves the cardinality of `'n`. If `'a` is a finite type, then `'a blist['n]` also defines a finite type. We 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. These functions are lifted from the corresponding functions for lists in Isabelle. Additionally, we instantiate the type to be comparable and enumerable, and so the equality of two bounded lists of the same type `'a blist['n]` can be established, and all elements in such a bounded type can be enumerated. Equality and enumerability are essential for code generation in Isabelle.

Example 1 *(types in the autonomous chemical detector).* Using the rule $[\![\_]\!]_t$, we get the corresponding definitions in Isabelle for the RoboChart types in Fig. 2.

```
typedef ChemT = "()"
type_synonym ('a) Chem = "(ChemT, 'a) PrimType"
abbreviation ChemC::"('a::finite ⇒ 'a Chem)" where "ChemC ≡ PrimTypeC"
datatype Status = Status_noGas | Status_gasD
record 'a GasSensor = gs_c :: "'a Chem"   gs_i :: "'a Intensity"
```

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`). 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)`.

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 use record brackets (⦇…⦈) in Isabelle to construct an element ⦇`gs_c = Chem (1::2), gs_i = IntensityC (0::2)`⦈ of this type: the chemical is 1, and the intensity is 0. □

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. □

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 such as core_int and arithmetic operators under which these bounded types are

closed. We use **locale** [32] 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 the 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 ...  end
```

Here, we omit the definitions for simplicity. 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.

Example 2 *(instantiation).* An example is shown below that 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 in the locale. □

4.4. Functions

Functions in RoboChart benefit from the rich expressions and Z toolkit in Isabelle. The expressions that are not supported in CSP-M, such as logical quantification, are naturally present in Isabelle. Using the code generator, the preconditions and postconditions of a function definition can be solved effectively (thanks to Isabelle's data refinement in the code generation), which is impossible in CSP-M and FDR.

We define the semantics of a function definition f (of type Function — a class in RoboChart's metamodel) in RoboChart below.

$$[\![f : \mathrm{Function}]\!]_{\mathcal{F}} = \begin{cases} \textbf{definition}\ "\mathtt{pre\_}\underline{f.name}\ \langle p : f.parameters \bullet p.name, \rangle\ =\ \{ c : f.preconditions \bullet [\![c]\!]_e, \wedge \}" \\ \textbf{definition}\ "\underline{f.name}\ \langle p : f.parameters \bullet p.name, \rangle\ =\ (\mathtt{THE\ result.}\ \{ c : f.postconditions \bullet [\![c]\!]_e, \wedge \})" \end{cases}$$

Here, the context $\mathcal{F}$ means functions. One function f corresponds to two definitions in Isabelle: one for its preconditions 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. The name of the first definition has the name f.name of f with a prefix `pre_`. Then, in the definition, the name is followed by a sequence of parameter names, constructed from sequence comprehension $\langle p : f.parameters \bullet p.name, \rangle$, which is similar to set comprehension except that the result is a sequence instead of a set. We use the sequence here because the order of parameters matters. The body of the definition is a set of the semantics for preconditions, given by $[\![c]\!]_e$ in the context of expressions e, combined using a conjunction operator $\wedge$. The second definition for the postconditions is similar except that a definite description (`THE result`, denoting the unique `result` such that the predicate holds) is used to return the `result` of the function.

Example 3 *(functions defined in the autonomous chemical detector).* As mentioned in Sect. 2.2.1, 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 in Fig. 2 specifies 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) by $[\![\_]\!]_{\mathcal{F}}$ in our implementation.

The intensity function defined in Fig. 2 has a precondition ( ) that the length (size) of the parameter gs is more significant 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 through the `instantiation.csp` file or the assertion language for RoboChart [29]. However, our definition of this function in Isabelle is directly generated 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.

We note that there is an error in the definition of intensity in the original model where ≤ (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; 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 words, `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 function definition 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,[8] their previous implementations[9] (defined in CSP-M and shown below) in CSP for verification with FDR, however, are correct.

```
intensity(gs) = let
  aux(<>,max) = max
  aux(<g>^gs,max) =
    if GasSensor_i(g) > max then aux(gs,GasSensor_i(g)) else aux(gs,max)
within
  aux(gs,0)

location(gs) = let
  aux(<>,max,n) = Front
  aux(<g>^gs,max,n) = if GasSensor_i(g) == max then angle(n) else aux(gs,max,n+1)
within
  aux(gs,intensity(gs),0)
```

Both definitions make use of pattern matching (an empty sequence `<>` or a non-empty sequence `<g>^gs`) to decompose values, initially starting (after `within`) from the function application of `aux` to `gs`. The pattern matching, therefore, starts from the beginning of the sequence `gs`. The definition of `intensity` does not refer to the length of `gs` and so avoids the comparison error introduced by using ≤ in its specification in the model. The definition of `location` starts the comparison with the maximum intensity `intensity(gs)` in `gs` from the beginning of the sequence, and so the result is the `angle` of the smallest index `n` whose corresponding intensity is maximal. This is deterministic, and so avoids the problem in the specification of `intensity` in the original model. □

The manual implementations, however, lead 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 directly in Isabelle, 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 a **`chantype`** command , created in our ITree-based CSP, and expressed as a finite set of channels declared in the command. We show an example to illustrate the creation of such an event alphabet `chan`.

[8] https://robostar.cs.york.ac.uk/case_studies/autonomous-chemical-detector/autonomous-chemical-detector.html#version4

[9] The implementations can be found in the `instantiation.csp` file in the original model available at https://robostar.cs.york.ac.uk/case_studies/autonomous-chemical-detector/autonomouschemicaldetector_v4.zip.

```
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`.

External choice $P \Box Q$ requires that P and Q have the same type (E, R)*itree* and 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. So both P and Q should be the type of $(E, ())$*itree*. A is a set of events and of type $\mathbb{P}E$, the power set of E.

The CSP processes with different types must be transformed into the same type before composition. The alphabet transformation of a process P of type (E_1, R)*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 $\mathrm{seq}\left(E_1 \times E_2\right)$), and the resultant process $P[\![\rho]\!]$ (or $P[\![\varrho]\!]_p$) is of type (E_2, R)*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. 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 differs from the relational renaming operator in CSP-M, where the relation is partial. So, the renaming does not affect the events not in the relation. This difference is because 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 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 entering 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`.

4.6.1. General definitions

The state machine semantics uses a general type `InOut` for the direction of an event in a connection.

```
datatype InOut = din | dout
```

Every state machine also has two data types for state and transition identifiers (`SIDS` and `TIDS`), and an event alphabet (E) for the process of this state machine. They are generated by three functions defined in the rules below.

```
sidsOfSTM(stm : StateMachineDef) =
  datatype SIDS = SID_ | {s : stm.nodes | s ∈ State • SID_s.name, | }
tidsOfSTM(stm : StateMachineDef) =
  datatype TIDS = {t : stm.transitions • TID_t.name, | }
channelsOfSTM(stm : StateMachineDef) =
  chantype Chan = internal::TIDS terminate::unit
   enter::SIDS×SIDS entered::SIDS×SIDS exit::SIDS×SIDS exited::SIDS×SIDS
   ⋃ {v : allLocalVariables(stm) • { get_v.name::[[v.type]]t, set_v.name::[[v.type]]t } , }
   ⋃ { v : requiredVariables(stm) • { get_v.name::[[v.type]]t,
                                      set_v.name::[[v.type]]t,
                                      set_EXT_v.name::[[v.type]]t } , }
   ⋃ { e : allEvents(stm) • { e.name_::TIDS×InOut×[[e.type]]t,
                              e.name::InOut×[[e.type]]t } , }
```

$$\{op : requiredOperations(stm) \bullet op.nameCall :: \langle p : op.parameters \bullet [\![p.type]\!]_t \times \rangle, \}$$

The function sidsOfSTM declares an enumeration type SIDS[10] containing state identifiers representing the machine itself SID_ and other states of the machine, that is, the nodes stm.nodes of the machine that are states s ∈ State. The function tidsOfSTM declares TIDS contains transition identifiers. Channels in the machine are declared by the statement generated by the function channelsOfSTM. The channel type Chan includes four kinds of channels. Firstly, flow control channels include (a) internal 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 local variable (v of all local variables allLocalVariables(stm) of the machine) with an additional set_EXT for each shared variable (v of the required variables requiredVariables(stm) of the machine) to accept an external update. We use generalised union ⋃ to combine all sets of channels for these variables into a large set. Thirdly, event channels include two channels for each event (e of all events allEvents(stm)) of the machine: one channel named e.name and another named e.name_. The distinction of two event channels (ech and ech_) for each event (e) 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 (ech_) with a transition id of type TIDS. We note, however, that events ech_.tid of this new channel are eventually renamed to the event channel ech in the process for the machine. Fourthly, operation call channels include a channel named op.nameCall for each required operation (op of all required operations requiredOperations(stm)) of the machine. The type of the channel is the product × of the corresponding types ($[\![p.type]\!]_t$ for each parameter p) of the operation parameters op.parameters in a sequence constructed by comprehension and so the order of the parameters is preserved.

Example 4 *(general types for Movement).* Below is an example of these data types for the Movement machine in Fig. 6.

```
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_MovementXSIDS_Movement" ...
  get_l_Movement    :: "Location_Loc"
  set_l_Movement    :: "Location_Loc"
  obstacle__Movement:: "TIDS_MovementXInOutXLocation_Loc"
  obstacle_Movement :: "InOutXLocation_Loc" ...
  moveCall_Movement :: "core_realXChemical_Angle" ...
```

4.6.2. Memory for variables and transitions

The memory for a state machine is composed of the memory for its local variables, for its shared variables, and that for its transitions. Its semantics is defined below.

$$memSharedVar(s : Variable) =$$

$$loop\ (\lambda v.\, get\_s.name!v \rightarrow \checkmark_v \Box\ set\_s.name?x \rightarrow \checkmark_x \Box\ set\_EXT\_s.name?x \rightarrow \checkmark_x)$$

$$memLocalVar(l : Variable) = loop\ (\lambda v.\, get\_l.name!v \rightarrow \checkmark_v \Box\ set\_l.name?x \rightarrow \checkmark_x)$$

$$guardExpr(e : Expression) = (\textbf{if}\ e \neq null\ \textbf{then}\ guard\ [\![e]\!]_e\ \textbf{else}\ skip)$$

$$memTransition(t : Transition) =$$

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

$$
\begin{array}{l}
\left(\begin{array}{l}
\textbf{if}\ \text{t.trigger} = \text{null}\ \textbf{then} \\
\quad \texttt{internal!TID\_}\underline{\text{t.name}} \rightarrow \mathit{skip} \\
\textbf{else if}\ \text{t.trigger.type} = \text{CommunicationType.SIMPLE}\ \textbf{then} \\
\quad \underline{\text{t.tigger.event.name}}\texttt{\_!(TID\_}\underline{\text{t.name}}\texttt{,din)} \rightarrow \mathit{skip} \\
\textbf{else if}\ \text{t.trigger.type} = \text{CommunicationType.INPUT}\ \textbf{then} \\
\quad \textbf{let}\ e == \text{t.trigger.event};\ p == \text{t.trigger.parameter} \bullet \\
\quad \underline{\text{e.name}}\texttt{\_?}\underline{\text{p.name}} : \{\text{p.name} : \text{p.type} \mid \text{t.condition} \bullet \texttt{(TID\_}\underline{\text{t.name}}\texttt{,din,}[\![\text{p.name}]\!]_e\texttt{)}\ \}\rightarrow \mathit{skip} \\
\textbf{else if}\ \text{t.trigger.type} = \text{CommunicationType.OUTPUT}\ \textbf{then} \\
\quad \underline{\text{guardExpr(t.condition)}}\texttt{;}\underline{\text{t.tigger.event.name}}\texttt{\_!(TID\_}\underline{\text{t.name}}\texttt{,dout,}[\![\text{t.trigger.value}]\!]_e\texttt{)} \rightarrow \mathit{skip} \\
\textbf{else} \\
\quad \underline{\text{guardExpr(t.condition)}}\texttt{;}\underline{\text{t.tigger.event.name}}\texttt{\_!(TID\_}\underline{\text{t.name}}\texttt{,din,}[\![\text{t.trigger.value}]\!]_e\texttt{)} \rightarrow \mathit{skip}
\end{array}\right) \\
\underline{\text{memTransitions(stm : StateMachineDef)}} = \\
\quad \mathit{loop}\ \ (\lambda\, id \bullet\ (\Box \underline{\text{t : stm.transitions}} \bullet \underline{\text{memTransition(t)}})) \\
\underline{[\![\text{stm : StateMachineDef}]\!]_{\mathcal{M}em}} = \\
\left(\begin{array}{l}
\left(\Box \underline{\text{v : allLocalVariables(stm)}} \bullet \underline{\text{memLocalVar(v)}}\right) \Box \\
\left(\Box \underline{\text{v : requiredVariables(stm)}} \bullet \underline{\text{memSharedVar(v)}}\right) \Box \\
\underline{\text{memTransitions(stm)}}
\end{array}\right)
\end{array}
$$

The memory memSharedVar(s) for a shared variable s is an infinite *loop*. It provides three choices: output the value v on `get_s` without updating the variable and accept a local (or external) update of the variable through `set_s` (or `set_EXT_s`). For a local variable l, its memory process memLocalVar(l) does not provide an external update.

The memory memTransition(tr) for a transition tr depends on the trigger tr.trigger of tr. Suppose tr has no trigger (such as the transition from the initial junction to Waiting in the state machine Movement in Fig. 6), tr.trigger = null. In that case, its memory is an output of the corresponding transition identifier `TID_`tr.name on channel `internal`. Otherwise, if the trigger type tr.trigger.type of tr is SIMPLE (with an event name but no value such as resume), its memory is an output of a tuple composed of the transition identifier and `din` on the corresponding channel tr.tigger.event.name_. Suppose the trigger type is INPUT (such as turn?a). In that case, we use local definitions **let** to introduce e for the trigger event and p for the trigger input variable. The memory for the transition is an input of tuples, composed of the transition identifier, `din`, and the value $[\![\text{p.name}]\!]_e$ (restricted to the type p.type of the trigger event and satisfy the guard tr.condition of the transition), from the corresponding channel. Suppose the trigger type is OUTPUT (such as turn!v). In that case, the memory for the transition evaluates its guard first by guardExpr(tr.condition) which uses the `guard` operator to test the expression if it is not null, or is just `skip` otherwise, followed by an output of the corresponding value tr.trigger.value on the corresponding channel. If the trigger type is not from one of the presented types, it must be of type SYNC (such as *turn.v*). Then the memory for tr (the **else** branch) is similar to the output trigger except that the event direction is `din` now (because of RoboChart's semantics).

The memory memTransitions(stm) for all transitions in a state machine stm is a loop infinitely offering the memory memTransition(tr) for each transition tr from the transitions stm.transitions of stm. Here, we use replicated external choice $\Box$ to construct external choice from a set of processes.

Then the memory $[\![\text{stm}]\!]_{\mathcal{M}em}$ of a state machine offers an external choice for the memories of all the local variables allLocalVariables(stm), all the shared variables requiredVariables(stm), and all the transitions.

Example 5 *(memory of a shared variable).* The memory process `Memory_x` for the shared variable x in the state machine CalSTM in Fig. 8, is shown below.

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

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

$$
\begin{array}{l}
\texttt{Movement\_MemoryTrans} = \mathit{loop}\ \ (\lambda id. \\
\quad \texttt{internal!TID\_t1} \rightarrow \checkmark_{id}\ \Box \\
\quad \texttt{resume\_!(TID\_t0, din)} \rightarrow \checkmark_{id}\ \Box \\
\quad \texttt{turn\_?(TID\_t3, din, a}\in\texttt{Chemical\_Angle)} \rightarrow \checkmark_{id}\ \Box \\
\quad \ldots\ \ \Box \\
\quad \texttt{get\_d1?d1} \rightarrow \texttt{get\_d0?d0} \rightarrow \\
\qquad (\texttt{(rc.Minus d1 d0 rc.core\_int)} > \texttt{stuckDist})\ \&\ (\texttt{internal!TID\_t12} \rightarrow \checkmark_{id})\ \Box
\end{array}
$$

```
...
```

Example 7 *(memory of transitions with an input trigger and a guard)*. The memory process for the transitions of the state machine MoveSTM in Fig. 8 has the choices 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})
```

4.6.3. Transitions

This paper considers three kinds of nodes, initial junctions, basic states, and final states, used in the two RoboChart models. Semantics for other nodes, including normal junctions and composite states, are part of our future work. In the semantics for a transition defined below, we consider two kinds of source nodes because a final state is also a State.

$$
\begin{array}{l}
[\![(\mathsf{t} : \mathsf{Transition}, \mathsf{stm} : \mathsf{StateMachineDef})]\!]_{\mathcal{T}} = \\
\left(\begin{array}{l}
\textbf{if } \mathsf{t.src} \in \mathsf{Initial} \textbf{ then} \\
\quad \mathtt{internal!TID\_}\mathsf{t.name} \rightarrow [\![\mathsf{t.action}]\!]_{\mathcal{A}} ; \ \mathtt{enter!(SID\_}\mathsf{stm.name}\mathtt{,SID\_}\mathsf{t.target.name}\mathtt{)} \rightarrow \\
\quad \mathtt{entered!(SID\_}\mathsf{stm.name}\mathtt{,SID\_}\mathsf{t.target.name}\mathtt{)} \rightarrow \mathit{skip} \\
\textbf{else if } \mathsf{t.src} \in \mathsf{State} \textbf{ then} \\
\quad [\![\mathsf{t.trigger}]\!]_{\mathcal{TR}} ; \ \mathtt{exit!(SID\_}\mathsf{t.src.name}\mathtt{,SID\_}\mathsf{t.src.name}\mathtt{)} \rightarrow [\![\mathsf{t.src.exit}]\!]_{\mathcal{A}} ; \\
\quad \mathtt{exited!(SID\_}\mathsf{t.src.name}\mathtt{,SID\_}\mathsf{t.src.name}\mathtt{)} \rightarrow [\![\mathsf{t.action}]\!]_{\mathcal{A}} ; \\
\quad \mathtt{enter!(SID\_}\mathsf{t.src.name}\mathtt{,SID\_}\mathsf{t.target.name}\mathtt{)} \rightarrow \\
\quad \mathtt{entered!(SID\_}\mathsf{t.src.name}\mathtt{,SID\_}\mathsf{t.target.name}\mathtt{)} \rightarrow \mathit{skip}
\end{array}\right)
\end{array}
$$

From the semantic point of view, a transition starts with the synchronising of its trigger, then initiates an exit from its source node, executes the exit action of the source node, makes its source node exited, executes its transitions action, and finally enters the target node of the transition. If the source node t.src of a transition t is an initial junction, t.src ∈ Initial, then t has no trigger and so channel `internal` with `TID` for the transition is synchronised. Because the source node is an initial junction, t does not need to exit the junction explicitly. The next step after the trigger, therefore, is the execution of the transition action $[\![\mathsf{t.action}]\!]_{\mathcal{A}}$ (whose definition in the context of actions $\mathcal{A}$ is omitted here). After that, t `enter`s its target node, and the node is `entered`.

If the source node t.src is a state, the transition trigger $[\![\mathsf{t.trigger}]\!]_{\mathcal{TR}}$, defined below, is synchronised.

$$
\begin{array}{l}
[\![\mathsf{tr} : \mathsf{Communication}, \mathsf{t} : \mathsf{Transition}]\!]_{\mathcal{TR}} = \\
\left(\begin{array}{l}
\textbf{if } \mathsf{tr} = \mathsf{null} \textbf{ then} \\
\quad \mathtt{internal!TID\_}\mathsf{t.name} \rightarrow \mathit{skip} \\
\textbf{else if } \mathsf{tr.event.type} = \mathsf{null} \textbf{ then} \\
\quad \mathsf{tr.event.name}\mathtt{\_!(TID\_}\mathsf{t.name}\mathtt{,din)} \rightarrow \mathit{skip} \\
\textbf{else} \\
\quad \textbf{let } \mathsf{e} == \mathsf{tr.event}; \ \mathsf{p} == \mathsf{tr.parameter} \bullet \\
\quad \mathsf{e.name}\mathtt{\_?}\mathsf{p.name} : \{\mathsf{p.name} : \mathsf{p.type} \bullet \mathtt{(TID\_}\mathsf{t.name}\mathtt{,din,}[\![\mathsf{p.name}]\!]_{\mathsf{e}}\mathtt{)}\ \} \\
\qquad \rightarrow \ \mathtt{set\_}\mathsf{p.name}\mathtt{!}\mathsf{p.name} \rightarrow \mathit{skip}
\end{array}\right)
\end{array}
$$

A trigger is of type Communication. Depending on whether the trigger exists and the trigger type, the semantics of the trigger might synchronise on channel `internal`, the channel corresponding to the trigger event tr.event.name, or accept input from the channel and then set the received value on p.name to the corresponding variable p.name in the memory using channel `set_`p.name.

After the trigger, the transition t starts to `exit` from its source node, executes the exit action t.src.exit of the node, and makes the node `exited`. The subsequent behaviour is similar to the Initial case.

4.6.4. Nodes

The semantics of each node in a state machine is a CSP process.

Initial junctions. One state machine has only one initial junction and one outgoing transition. Its semantics is defined below.

$$
\begin{array}{l}
[\![(\mathsf{stm} : \mathsf{StateMachineDef})]\!]_{\mathcal{NI}} = \\
\quad \textbf{let } \mathsf{i} == (\mu\, \mathsf{n} : \mathsf{stm.nodes} \mid \mathsf{n} \in \mathsf{Initial}) ; \ \mathsf{t} == (\mu\, \mathsf{t} : \mathsf{stm.transitions} \mid \mathsf{t.source} = \mathsf{i}) \bullet [\![\mathsf{t}, \mathsf{stm}]\!]_{\mathcal{T}}
\end{array}
$$

The semantics of the only initial junction i (retrieved through the definite description operator μ) in a state machine stm is just the semantics $[\![\mathsf{t}, \mathsf{stm}]\!]_{\mathcal{T}}$ of the only outgoing transition t (retrieved through μ too) from i.

Example 8 *(initial junction).* 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 → set_x!0 → enter!(SID_CalSTM, SID_CalSTM_cal) →
  entered!(SID_CalSTM, SID_CalSTM_cal) → ✓
```

States. The semantics for a state node is given below.

$$[\![(s : \mathrm{State}, \mathrm{stm} : \mathrm{StateMachineDef})]\!]_{\mathcal{NS}} = \mathit{loop}$$

$$\left(\begin{array}{l}
\lambda\, \mathit{id}.\ \texttt{enter?sd:}\{\texttt{sid} : \texttt{SIDS} \mid \texttt{sid} \neq \texttt{SID_s.name}\bullet(\texttt{sid}, \texttt{SID_s.name})\} \rightarrow \mathit{Ret}\,(\texttt{True, fst sd});\\
\left(\begin{array}{l}
\mathit{iterate}\ (\lambda \texttt{r.}\ \texttt{fst r})\\
\left(\begin{array}{l}
\lambda \texttt{r.}\\
\left(\begin{array}{l}
[\![\mathrm{s.entry}]\!]_{\mathcal{A}}\,;\texttt{entered!(fst r, SID_s.name)} \rightarrow\ ([\![\mathrm{s.during}]\!]_{\mathcal{A}};\ \texttt{stop})\ \triangle\\
\left(\begin{array}{l}
(\Box\ \mathrm{t} : \mathrm{selfTransFromNode(s)}\bullet([\![\mathrm{t, stm}]\!]_{\mathcal{T}};\ \mathit{Ret}\,(\texttt{True, SID_s.name})))\ \Box\\
(\Box\ \mathrm{t} : \mathrm{nonSelfTransFromNode(s)}\bullet([\![\mathrm{t, stm}]\!]_{\mathcal{T}};\ \mathit{Ret}\,(\texttt{False, SID_s.name})))\ \Box\\
\left(\begin{array}{l}
\Box\ \mathrm{e} : \mathrm{EvtChns}\bullet\\
\mathrm{e.name_?x}:\{\mathrm{t} : \texttt{TIDS} \mid \mathrm{t} \in \mathrm{ITIDS(stm)} \wedge \mathrm{t} \notin \mathrm{ITIDS_s(s)}\bullet(\mathrm{t}, _, _)\} \rightarrow\\
\texttt{exit?sd:}\{\texttt{sid} : \texttt{SIDS} \mid \texttt{sid} \neq \texttt{SID_s.name}\bullet(\texttt{sid}, \texttt{SID_s.name})\} \rightarrow [\![\mathrm{s.exit}]\!]_{\mathcal{A}};\\
\texttt{exited!(fst sd, SID_s.name)} \rightarrow\ \mathit{Ret}\,(\texttt{False, SID_s.name})
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)$$

Basically, the semantics 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` (the parameter of the semantics process), with a nested conditional iteration by `iterate`. The state of the iteration is a pair such as `(True, fst sd)`, whose first element is a Boolean value to indicate if this iteration terminates or not, and whose second element is the RoboChart state (for example `fst r`) that initiates entering of the state s. Initially, the state of the iteration is passed from its preceding `Ret` construct.

Initially, the state s is waiting for the `enter`ing from other states (sid ≠ SID_s.name). After that, the source node, the first `fst` element of the pair `sd`, is passed to the subsequent iteration. The first argument `(λr.fst r)` of the iteration is its condition, a boolean function from the state of the iteration (a pair and `fst r` is the termination condition), and the second argument is its body, an ITree process. The process starts with the execution of the entry action $[\![\mathrm{s.entry}]\!]_{\mathcal{A}}$ of s and then signals that s is `entered` to the node, `fst sd`, that initiates the entering. Afterwards, the during action $[\![\mathrm{s.during}]\!]_{\mathcal{A}}$ of s is being executed while offering the possibility of interruption $\triangle$ by a transition process on the right of $\triangle$. The during action is composed sequentially with `stop`. So, the interrupt cannot be terminated here and can only be terminated by the transition process offered in an external choice of three groups: (1) self-transitions selfTransFromNode(s) of s; (2) other transitions nonSelfTransFromNode(s) from s that are not self-transitions; and (3) transitions that can interrupt s.

The process for each self-transition t is terminated (*Ret*), after t is taken ($[\![\mathrm{t.stm}]\!]_{\mathcal{A}}$), and returned with a state `(True, SID_)` s.name. The iteration, therefore, is not terminated (because the condition is `True`), and so the self-transition does not need to `enter` s again.

The process for each other transition t is terminated and returned with a state `(False, SID_)` s.name. The iteration, therefore, is terminated (because the condition is `False`), and so the semantics of s needs to re-`enter` it from the body of the loop again.

The third group is for the transitions ITIDS(stm) of stm that are from any state (and so they could interrupt other states) of stm, but do not include those transitions $\mathrm{ITIDS_s(s)}$ that are from or contained in s. This group applies to composite states in RoboChart, which further contain nodes and transitions. No composite state is used for the two RoboChart examples in this paper, so the parent of s is just stm. Therefore, no such interruptible transitions are in the examples. In general, for each event e of all trigger events EvtChns in stm, its corresponding event channel is e.name_. Each trigger event can interrupt the during action if the corresponding transition is in this group. After the trigger event, the semantics of s accepts `exit` from any other state sid (sid ≠ SID_s.name), executes its exit action, signals that s is `exited`, and then terminates.

Example 9 *(state).* 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, 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, 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, SID_Waiting)) □
14        ...
15  ))) (ret)); Ret (id))  □
```

The semantics of a state s also captures the transitions from its sibling states, which have the same parent as s. These transitions shall be blocked because they are part of the behaviours of its sibling states. This is implemented as the restricted behaviour of s shown below.

$$\text{restrictedState}(s : \text{State}, stm : \text{StateMachineDef}) =$$

$$\lambda id.\ [\![(s : \text{State}, stm : \text{StateMachineDef})]\!]_{\mathcal{NS}} \parallel_{\text{eventsAllTransFromSiblings}(s,stm)} \mathit{skip}$$

The synchronisation of the semantics of s with *skip* over a set of events eventsAllTransFromSiblings(s, stm) on the transitions that are from the sibling states of s. These events are blocked in the restricted semantics because of the synchronisation with *skip*.

The behaviour of multiple states is composed of their restricted behaviours.

$$\text{composeStates}(ss : \text{seq State}, stm : \text{StateMachineDef}) =$$

$$\left(\begin{array}{l}\textbf{if } \#ss = 1 \textbf{ then} \\ \quad \text{restrictedState}(\text{head } ss, stm) \\ \textbf{else} \\ \quad \text{restrictedState}(\text{head } ss, stm) \parallel_{\text{flowEvents}(ss,stm)} \text{composeStates}(\text{tail } ss, stm)\end{array}\right)$$

Here, ss is a sequence of states. If there is only one state in the sequence (that is, the length #ss of ss is 1), the semantics is just that of the only state (in the head of ss). Otherwise, the semantics is the parallel composition of the behaviour of the head state with the composed behaviour composeStates(tail ss, stm) of the tail states (tail ss, ss after its head is removed), synchronised on a set of events flowEvents(ss, stm) that are flow channel (`enter`, `entered`, `exit`, and `exited`) events used in the restricted behaviours of all states in tail ss to enter the head state or exit from the head state.

The behaviour of nodes in a state machine is then the composition of the initial junction and that of states.

$$[\![stm : \text{StateMachineDef}]\!]_{\mathcal{N}} =$$

$$[\![stm : \text{StateMachineDef}]\!]_{\mathcal{NI}} \parallel_{\text{initFlowEvents}(stm)} \text{composeStates}(\langle n : stm.\text{nodes} \mid n \in \text{State}\rangle, stm)$$

Here, we use sequence comprehension to construct all states in stm in a sequence. The composition synchronises on a set of events initFlowEvents(stm) that are flow channel events used to enter the states of stm from the initial junction and not from other states.

Example 10 *(nodes).* The semantics of nodes in GasAnalysis in Fig. 4 can be found online  .

4.6.5. Composition of STM and memory processes

The semantics of nodes in stm is composed with the memory of stm.

$$\text{MemorySTM}(stm : \text{StateMachineDef}) =$$

$$\left(\left([\![stm]\!]_{\mathcal{N}} \setminus_p \text{allFlowEvents}(stm)\right) \parallel_{(\text{varChannelEvents}(stm) \cup \text{trigEvents}(stm))} [\![stm]\!]_{\mathcal{M}em}\right) \setminus_p \text{localVarEvents}(stm)$$

In the semantics, all flow channel events allFlowEvents(stm) are hidden from the behaviour of nodes. Then the nodes synchronise with the memory on the union of two sets of events: varChannelEvents(stm) for all `get_`, `set_`, and `set_EXT_` variable channel events, and trigEvents(stm) for all trigger events. Subsequently, all local variable channel events localVarEvents(stm) are hidden.

As discussed previously in Sect. 4.6.1, 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 MemorySTM(stm) are renamed to `e` by forgetting the first element (a transition identifier) of the value carried on `e_`.

$$\text{MemorySTM\_renamedp}(stm : \text{StateMachineDef}) = \text{MemorySTM}(stm)[\![\text{triggerMap}(stm)]\!]_p$$

Here, we use the renaming operator with priority to avoid nondeterminism. The renaming mapping list triggerMap(stm) includes the mappings for renaming and also the mappings whose names are not changed.

Example 11 *(rename)*. We show the renaming of the state machine MoveSTM in Fig. 8.

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

The `event_map_list` contains the mappings like `(update_.(TID_t1, din, v), update.(din, v))` and `(update_.(TID_t3, din, v), update.(din, v))` where `v` is an integer number. 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. In our example, `event_map_list` is arranged based on the order in which the transitions are created. □

The semantics of a state machine also needs to take its termination (for example, the final state is reached) into consideration.

$$[\![\text{stm} : \text{StateMachineDef}]\!]_{\mathcal{STM}} = \left(\text{MemorySTM\_renamedp(stm)}\ [\![\{\texttt{terminate.()}\} \triangleright \textit{skip}\right) \setminus_p \text{internalEvents(stm)}$$

Based on the definition of the exception operator, if MemorySTM_renamedp(stm) 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 internalEvents(stm).

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 provided by a robotic platform is an event to record the call with appropriate arguments.

$$[\![\text{s} : \text{Call}]\!]_{\mathcal{A}} = \text{s.operation.name}\texttt{Call!}(\langle \text{a} : \text{s.args} \bullet [\![\text{a}]\!]_{\text{e}}, \rangle) \rightarrow \textit{skip}$$

Here, s.operation.name`Call` is the corresponding channel name to the operation s.operation.name which *s* calls. The call arguments s.args become a tuple composed of corresponding arguments $[\![\text{a}]\!]_{\text{e}}$ for each argument a.

The semantics of a state machine-defined operation differs 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. However, the semantics of an operation's nodes, transitions, and memory are the same as those in a state machine. We omit the semantics of operations for simplicity. Instead, we illustrate it with an example below.

Example 12 *(operation)*. 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"
```

tThe `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 as the process `D_changeDirection` (for changeDirection). □

4.8. Controllers

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

channelsOfCTRL(c : ControllerDef) =

chantype Chan = terminate::unit

$$\bigcup \left\{ \begin{array}{l} \text{v : allLocalVariables(c)} \bullet \\ \bigcup \left\{ \begin{array}{l} \{ \texttt{get\_}\text{v.name} :: [\![\text{v.type}]\!]_t, \texttt{set\_}\text{v.name} :: [\![\text{v.type}]\!]_t \ \}, \\ \{ \text{s : c.machines} \mid \text{v} \in \text{requiredVariables(s)} \bullet \texttt{set\_EXT\_}\text{s.name}\texttt{\_}\text{v.name} :: [\![\text{v.type}]\!]_t, \} \end{array} \right\}, \end{array} \right\}$$

$$\bigcup \left\{ \begin{array}{l} \text{v : requiredVariables(c)} \bullet \\ \bigcup \left\{ \begin{array}{l} \{ \texttt{get\_}\text{v.name} :: [\![\text{v.type}]\!]_t, \texttt{set\_}\text{v.name} :: [\![\text{v.type}]\!]_t, \texttt{set\_EXT\_}\text{v.name} :: [\![\text{v.type}]\!]_t \ \}, \\ \{ \text{s : c.machines} \mid \text{v} \in \text{requiredVariables(s)} \bullet \texttt{set\_EXT\_}\text{s.name}\texttt{\_}\text{v.name} :: [\![\text{v.type}]\!]_t, \} \end{array} \right\}, \end{array} \right\}$$

$$\{ \text{e : allEvents(c)} \bullet \text{e.name} :: \texttt{InOut} \times [\![\text{e.type}]\!]_t, \}$$

$$\{ \text{e : internalEvents(c)} \bullet \text{e.name} :: \texttt{InOut} \times [\![\text{e.type}]\!]_t, \}$$

$$\{ \text{op : requiredOperations(c)} \bullet \text{op.name}\texttt{Call} :: \langle \text{p : op.parameters} \bullet [\![\text{p.type}]\!]_t, \times \rangle, \}$$

The channel type for a controller includes channels for termination, local variables, shared variables, events, and operations. Different from those for a state machine, for each local or shared variable v a controller c, additionally, declares a `set_EXT_`s.name`_`v.name channel for each state machine s of c that requires v. These channels propagate the update of v to all state machines that require v.

In addition to event channels for all events, allEventsc of c, an event channel for each event e of all internal events (that is, the events that are used in connections between state machines) is also declared. These channels are used for communication between state machines.

Example 13 *(controller channel type).* The channel type of the controller Ctrl in Fig. 8 is defined below.

```
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"
```

□

The memory of a controller deals with the update of its local and shared variables, but not transitions like that of a state machine.

memSharedVarCtrl(s : Variable, c : ControllerDef) =

$$loop \left(\begin{array}{l} \lambda v \bullet \texttt{set\_EXT\_}\text{s.name}\texttt{?x} \rightarrow \\ \left(\fatsemi \ \text{m : } \{ \text{m : c.machines} \mid \text{s} \in \text{requiredVariables(m)} \} \bullet \texttt{set\_EXT\_}\text{m.name}\texttt{\_}\text{s.name}\texttt{!x} \rightarrow skip \right) \end{array} \right)$$

memLocalVarCtrl(l : Variable, c : ControllerDef) = *loop*

$$\left(\begin{array}{l} \lambda v \bullet \\ \left(\begin{array}{l} (\texttt{set\_}\text{l.name}\texttt{?x} \rightarrow \\ \left(\fatsemi \ \text{m : } \{ \text{m : c.machines} \mid \text{l} \in \text{requiredVariables(m)} \} \bullet \texttt{set\_EXT\_}\text{m.name}\texttt{\_}\text{l.name}\texttt{!x} \rightarrow skip \right) ; \checkmark_x) \end{array} \right) \\ \Box \ \texttt{get\_}\text{l.name}\texttt{!v} \rightarrow \checkmark_v \end{array} \right)$$

$[\![\text{c : ControllerDef}]\!]_{\mathcal{M}em} =$

$$\left(\begin{array}{l}\left(\Box v : \mathit{allLocalVariables}(c) \bullet \mathit{memLocalVarCtrl}(v)\right) \Box \\ \left(\Box v : \mathit{requiredVariables}(c) \bullet \mathit{memSharedVarCtrl}(v, c)\right)\end{array}\right)$$

The memory $\mathit{memSharedVarCtrl}(s, c)$ of a shared variable s in a controller c is an infinite loop. It accepts an update of s on channel `set_EXT_`$s.name$ and then propagates the updated value x to the state machines that require s through the corresponding channels, using a replicated sequential composition $;$. The memory $\mathit{memLocalVarCtrl}(l, c)$ of a local variable l also offers an update of l but on a different channel `set_`$l.name$, then propagates the update. Additionally, it offers access to the value of l through channel `get_`$l.name$. We also note that the loop state v of $\mathit{memSharedVarCtrl}(s, c)$ is dummy (that is, not used and updated) because c does not store the value of a shared variable, but the loop state v of $\mathit{memLocalVarCtrl}(l, c)$ is usual (for update and access).

The memory $[\![c]\!]_{\mathcal{M}em}$ of a controller c offers an external choice for the memories of all the local variables $\mathit{allLocalVariables}(c)$, and all the shared variables $\mathit{requiredVariables}(c)$.

Example 14 *(controller memory).* The memory of Ctrl in Fig. 8 is shown below.

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

□

Each state machine in a controller has a different event alphabet or channel type. To compose state machines, their event alphabets must be transformed into the same channel type for the controller. We use renaming defined in Sect. 3 to achieve it.

$$\mathit{STM\_renamedp}(\mathit{stm} : \mathit{StateMachineDef}, c : \mathit{ControllerDef}) = \left([\![\mathit{stm}]\!]_{\mathcal{STM}}\right) [\![\mathit{stm2CtrlMap}(\mathit{stm}, c)]\!]_p$$

We use renaming with priority to resolve potential nondeterminism introduced from the renaming mapping list $\mathit{stm2CtrlMap}(\mathit{stm}, c)$. This list contains event mappings between stm and c for the `terminate` channel, (local and shared) variable channels, event channels, and operation call channels. Additionally, it includes event channels that are used for internal communication between state machines, such as the update event between CalSTM and MoveSTM in Fig. 8.

Example 15 *(state machine renaming).* The renaming list for CalSTM in Fig. 8 can be found online . □

Renamed state machines are composed together using parallel composition.

$$\mathit{composeMachines}(\mathit{ms} : \mathit{seq}\, \mathit{StateMachineDef}, c : \mathit{Controllers}) =$$

$$\left(\begin{array}{l}\textbf{if } \#\mathit{ss} = 1 \textbf{ then} \\ \quad \mathit{STM\_renamedp}(\mathit{head}\, \mathit{ms}, c) \\ \textbf{else} \\ \quad \mathit{STM\_renamedp}(\mathit{head}\, \mathit{ms}, c) \;\|_{\mathit{connEvents}(\mathit{ms},c)}\; \mathit{composeMachines}(\mathit{tail}\, \mathit{ms}, c)\end{array}\right)$$

Here, ms is a sequence of state machines. If there is only one machine in the sequence, the semantics is just that of the only machine (in the head of ms). Otherwise, the semantics is the parallel composition of the behaviour of the head machine with the composed behaviour $\mathit{composeMachines}(\mathit{tail}\, \mathit{ms}, c)$ of the tail machines, synchronised on a set of events $\mathit{connEvents}(\mathit{ms}, c)$ which includes the `terminate` channel and event channels used in all machines in $\mathit{tail}\, \mathit{ms}$ to communicate with the head machine.

The events for internal communication between state machines are hidden.

$$\mathit{hiddenMachines}(c) = \mathit{composeMachines}(\langle m : c.\mathit{machine} \bullet m \rangle, c) \setminus_p \mathit{intConnEvents}(c)$$

Here, $\mathit{intConnEvents}(c)$ defines a set of events used for internal communication between state machines in c.

The semantics of a controller is the parallel composition of that of its composed machines with its memory and also deals with its termination.

$$[\![c : \mathit{ControllerDef}]\!]_{\mathcal{C}} =$$

$$\left(\left(\mathit{hiddenMachines}(c) \;\|_{\mathit{ctrlMemEvents}(c)}\; [\![c]\!]_{\mathcal{M}em}\right) \setminus_p \mathit{ctrlMemEvents}(c)\right) [\![\{\texttt{terminate.()}\} \triangleright \mathit{skip}$$

The $\mathit{ctrlMemEvents}(c)$ is a set of variable channel events in c used to access, update, and propagate variables.

Example 16 *(controller).* The semantics of Ctrl in Fig. 8 can be found online . □

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

Fig. 11. SCE-ACD-1: animation of the example when dangerous chemical detected.

4.9. Modules

Similar to the event alphabet of the process for a controller, the process for a module also contains a termination channel, shared variable channels, event channels, and operation call channels. The event channels include the events of its platform and 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 an asynchronous connection is a one-place buffer.

$$\text{singleBuffer}(\text{efrom} : \text{Event}, \text{eto} : \text{Event}) =$$
$$\mathit{loop}\left(\lambda\, lv \,.\, \left(\begin{array}{l}\left(\begin{array}{l}\texttt{guard (length lv} \geq \texttt{0} \wedge \texttt{length lv} \leq \texttt{1);}\\ \text{efrom.name}?x:\{v : \text{efrom.type} \bullet (\texttt{dout}, v)\} \rightarrow \checkmark_{[\mathit{snd}\, x]}\end{array}\right) \Box \\ \left(\texttt{guard (length lv > 0);}\ \text{eto.name}!(\texttt{din, hd lv}) \rightarrow \checkmark_{[]}\right)\end{array}\right)\right)$$

The connection is from a efrom event to a eto event. The semantics is an infinite loop whose state is a list `lv` in Isabelle. It offers two choices: either accepting a write operation if the list is empty or contains one element, using channel efrom.name and then updating the state to a list containing the write value `snd x` in the second part of `x`, or accepting a read operation, if the list is not empty, using channel eto.name with current element `hd lv` and then updating the list to be empty `[]`.

The memory of a module deals with the update of shared variables and 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. It 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 [14, Sect. 5], the animation of ITrees is achieved through code generation [19] in Isabelle. Infinite corecursive definitions over ITrees are implemented using lazy evaluation in Haskell. Associative lists are used to implement partial functions in ITrees, and a simple animator in Haskell is presented. Using the same approach for animation, we can animate the two RoboChart models shown in Sect. 2.

5.1. Autonomous chemical detector

We illustrate two scenarios **SCE-ACD-1** and **SCE-ACD-2** 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 describe all enabled events, and the lines beginning 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 selected number for simplicity.

SCE-ACD-1. Fig. 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

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

Fig. 12. SCE-ACD-2: animation of the example when chemical detected with low intensity.

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).

SCE-ACD-2. 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 (on line #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 reading is 0 (on line #5); (6) the controllers call move (the action of a transition in the defined operation changeDirection) to request the robot to move towards its opposite direction (left here) to the obstacle at speed 1 (on line #6); (7) another reading of the gas sensor shows there is still a chemical detected with low intensity (on line #7); (8) the controllers call move (the entry action of state TryingAgain in machine Movement) to request the robot to move towards its front at speed 1 (on line #8); (9) an obstacle on its left is encountered (on line #9); (10) the odometer reading (the action of the transition from state TryingAgain to state AvoidingAgain) is 1 (on line #10); (11) there is another odometer reading (0) on line #11, which corresponds to the entry action of state Avoiding (the entering of this state has resulted from the transition taken 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 **SCE-PR-1**, **SCE-PR-2**, and **SCE-PR-3** of the animation of the patrol robot corresponding to the three sections (S1, S2, and S3) of the corridor in Fig. 7.

SCE-PR-1. We show the first scenario in Fig. 13, 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[11] 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 or right side. However, our animation on lines #6-10, and #14 shows only the right side is chosen. This 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 to correspond 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

[11] 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.

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

Fig. 13. SCE-PR-1: animation of the patrol robot when the calibrated position is in S1.

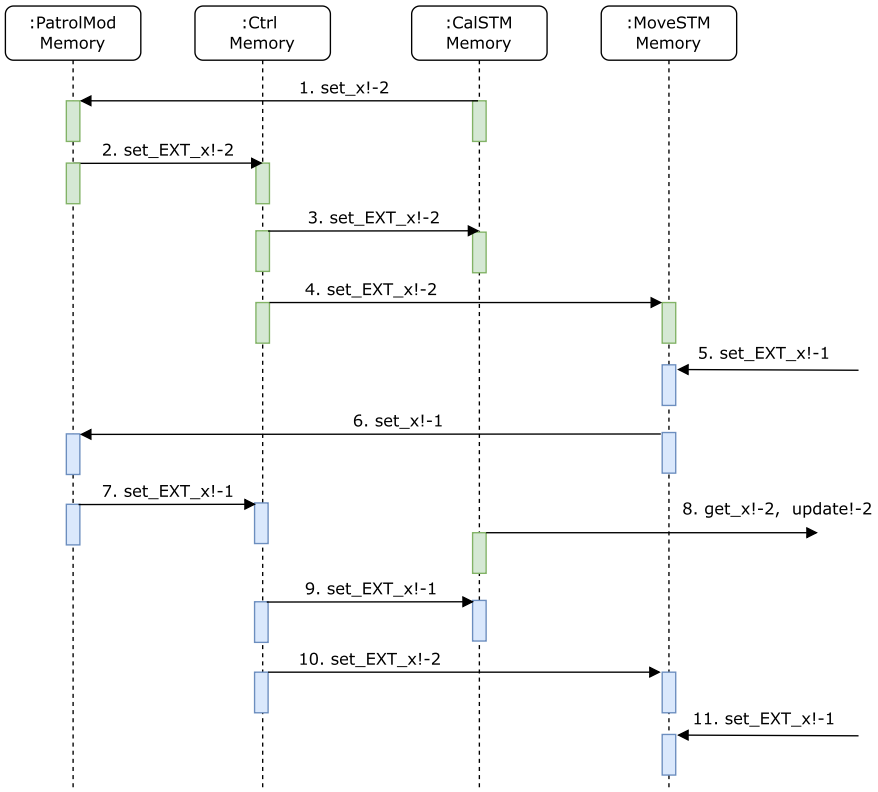


Fig. 14. The update of the shared variable x and its propagation in the patrol robot model.

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. The memories can be accessed and evaluated between these communications using the outdated value. This is further demonstrated by communications 5 to 11. Consider a new update of x to -1 (by communication 5)

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

Fig. 15. SCE-PR-2: animation of the patrol robot when the calibrated position is in S2.

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

Fig. 16. SCE-PR-3: animation of the patrol robot when the calibrated position is in S3.

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) [14]. We also note that the animated behaviour of 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). 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 in terms of the unnecessary interleaving behaviour by updating the shared variables from multiple state machines. 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.

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

SCE-PR-3. The third scenario we consider is shown in Fig. 16 where the calibrated position is initially 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 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
```

Summary. 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. 13 and on line `#2` in Figs. 15 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 ->
       PatrolMod::reset.in -> PatrolMod::right.out.0 ->
       PatrolMod::right.out.1 -> PatrolMod::reset.in -> Reset
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. Therefore, the reset is only enabled when x is 0.

6. Related work

Animation is a lightweight formal method. Kazmierczak et al. [33] describe the advantages of using animation to test 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. However, its biggest drawback is that it cannot verify consistency, correctness, or completeness.

Animation can be tailored to specific application domains. For example, Boichut et al. [34] 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 provide a different framework [21]. ProB contains model and constraint-based checkers that can detect errors in B specifications. It implements a back-end in a framework for various specification languages, including the B language, Event-B, CSP-M, TLA+, and Z.

De Souza [35] 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. [36] develop K,[12] a rewriting-based executable semantic framework. The operational semantics of programming languages such as C [37] and Java [38] 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. Thus, program development by refinement is supported in our framework. Higher-order logic and nondeterminism are some features of interaction trees, but not of K.

[12] https://kframework.org/

Stateflow is a graphical language integrated into Matlab's Simulink 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 [39]. This is the same as our approach to resolving nondeterministic choices between transitions using the prioritised renaming operator.

The automatically generated CSP semantics of a RoboChart model in RoboTool targets at verification with FDR4 and so uses CSP-M with modules. This naturally makes Probe in FDR the first choice for the animation of RoboChart. The benefits of Probe include (1) the consistency of the standard and timed semantics of RoboChart between verification and animation because they use the same generated CSP code; and (2) various compression methods in FDR are also used for animation, which could potentially reduce the size of state space and improve the efficiency of animation. Probe presents all internal behaviours to users to let them choose each nondeterministic choice, making the animation challenging even for experts. Our animation, presented in this paper, simplifies the process by eliminating internal behaviours so only observable events are visible to users, which makes our animation accessible to normal users.

ProB [40] also supports CSP-M for model checking and animation without modules. It also has some limitations[13] in terms of supported CSP constructors. ProB cannot directly model check and animate the generated CSP code (in the new CSP-M version) in RoboTool because ProB only supports early versions of CSP-M. To use the user-friendly (GUI-based) animator in ProB, we need to encode the CSP semantics of RoboChart in the CSP-M supported by ProB. It is possible to encode the CSP semantics of RoboChart in the CSP-M supported by ProB, but we have not investigated it yet. The GUI-based animation in ProB is straightforward and easily used even by normal users. This is an advantage of ProB in terms of animation.

7. Conclusions and future work

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

This work targets deterministic RoboChart and nondeterministic RoboChart models (but nondeterminism is resolved in a priority way in the semantics and so deterministic semantics eventually). Our work covers many RoboChart features (but not all). Our immediate future work is to investigate the support of nondeterminism in 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. Mainly, we take RoboChart's CSP semantics generated in RoboTool into account to define consistent and restricted semantics based on an optimised version of the CSP semantics. This practical consideration could entitle us to reuse the current CSP semantics generator in RoboTool to generate ITree-based CSP semantics automatically. Then, the workflow from RoboChart models to Haskell code can be fully automated, and our work 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 and animation in this paper. We will investigate using temporal logic as a property language for verifying 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) [41,42], 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 could also use this approach to automatically generate a sound runtime monitor from RoboChart models to observe the behaviour of systems derived from the models.

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

CRediT authorship contribution statement

Kangfeng Ye: Formal analysis, Investigation, Methodology, Software, Validation. **Simon Foster:** Formal analysis, Investigation, Methodology, Software, Validation. **Jim Woodcock:** Formal analysis, Investigation, Methodology, Software, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

[13] https://prob.hhu.de/w/index.php?title=CSP-M

Acknowledgements

This work is funded by the EPSRC projects CyPhyAssure[14] (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 https://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.
[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, https://doi.org/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, Softw. Syst. Model. (Oct 2021), https://doi.org/10.1007/s10270-021-00916-8.
[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] A.L.C. Cavalcanti, A.C.A. Sampaio, A. Miyazawa, P. Ribeiro, M.S.C. Filho, A. Didier, W. Li, J. Timmis, Verified simulation for robotics, Sci. Comput. Program. 174 (2019) 1–37, https://doi.org/10.1016/j.scico.2019.01.004, https://www-users.cs.york.ac.uk/%7Ealcc/publications/papers/CSMRCD19.pdf.
[8] S. Foster, J.J. Huerta y Munive, G. Struth, Differential hoare logics and refinement calculi for hybrid systems with isabelle/hol, in: U. Fahrenberg, P. Jipsen, M. Winter (Eds.), Relational and Algebraic Methods in Computer Science, Springer International Publishing, Cham, 2020, pp. 169–186.
[9] S. Foster, J.J. Huerta y Munive, M. Gleirscher, G. Struth, Hybrid systems verification with Isabelle/hol: simpler syntax, better models, faster proofs, in: M. Huisman, C. Păsăreanu, N. Zhan (Eds.), Formal Methods, Springer International Publishing, Cham, 2021, pp. 367–386.
[10] Y. Murray, M. Sirevåg, P. Ribeiro, D.A. Anisi, M. Mossige, Safety assurance of an industrial robotic control system using hardware/software co-verification, Sci. Comput. Program. 216 (2022) 102766, https://doi.org/10.1016/j.scico.2021.102766, https://www.sciencedirect.com/science/article/pii/S0167642321001593.
[11] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall Int., 1985.
[12] A.W. Roscoe, Understanding Concurrent Systems, Texts in Computer Science, Springer, 2011.
[13] 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), https://doi.org/10.1145/3371119.
[14] 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), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 203, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2021, pp. 20:1–20:18, https://drops.dagstuhl.de/opus/volltexte/2021/14397.
[15] S.D. Brookes, C.A.R. Hoare, A.W. Roscoe, A theory of communicating sequential processes 560–599, https://doi.org/10.1145/828.833.
[16] 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.
[17] I. Toyn (Ed.), Information Technology — Z Formal Specification Notation — Syntax, Type System and Semantics, ISO, 2002, iSO/IEC 13568:2002(E).
[18] J.M. Spivey, The Z Notation: A Reference Manual, 2nd edition, Prentice-Hall, 1992.
[19] 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, in: Lecture Notes in Computer Science, vol. 6009, Springer, 2010, pp. 103–117.
[20] R. Mayr, T. Nipkow, Higher-order rewrite systems and their confluence, Theor. Comput. Sci. 192 (1) (1998) 3–29.
[21] 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, in: Lecture Notes in Computer Science, vol. 2805, Springer, 2003, pp. 855–874.
[22] 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.
[23] C.A.R. Hoare, J. He, The weakest prespecification, Inf. Process. Lett. 24 (2) (1987) 127–132.
[24] J. Baxter, P. Ribeiro, A. Cavalcanti, Sound reasoning in tock-CSP, Acta Inform. 04 (2021), https://doi.org/10.1007/s00236-020-00394-3.
[25] S. Schneider, Concurrent and Real Time Systems: The CSP Approach, 1st edition, John Wiley & Sons, Inc., USA, 1999.
[26] J. Woodcock, A. Cavalcanti, The semantics of circus, in: D. Bert, J.P. Bowen, M.C. Henson, K. Robinson (Eds.), ZB 2002:Formal Specification and Development in Z and B, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 184–203.
[27] J.C.P. Woodcock, A.L.C. Cavalcanti, J. Fitzgerald, P.G. Larsen, A. Miyazawa, S. Perry, Features of cml: a formal modelling language for systems of systems, in: 7th International Conference on Systems of Systems Engineering, IEEE Syst. J. 6 (2012), IEEE.
[28] 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, https://doi.org/10.1109/TSMCC.2012.2218236.
[29] A. Miyazawa, A. Cavalcanti, P. Ribeiro, K. Ye, W. Li, J. Woodcock, J. Timmis, RoboChart Reference Manual, Tech. Rep., University of York, 2020, https://www.cs.york.ac.uk/circus/publications/techreports/reports/robochart-reference.pdf.
[30] 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, in: Lecture Notes in Computer Science, vol. 8558, Springer, 2014, pp. 93–110.
[31] S. Foster, J. Baxter, A. Cavalcanti, J. Woodcock, F. Zeyda, Unifying semantic foundations for automated verification tools in Isabelle/UTP, Sci. Comput. Program. 197 (2020) 102510, https://doi.org/10.1016/j.scico.2020.102510.
[32] 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.
[33] 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, 1998, pp. 254–261.
[34] 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.

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

[35] 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.
[36] G. Rosu, T. Serbanuta, An overview of the K semantic framework, J. Log. Algebraic Methods Program. 79 (6) (2010) 397–434, https://doi.org/10.1016/j.jlap.2010.03.012.
[37] 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.
[38] 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.
[39] MathWorks, Stateflow: User's Guide (R2022b), The MathWorks Inc., 2022, https://www.mathworks.com/help/pdf_doc/stateflow/stateflow_ug.pdf.
[40] The ProB Animator and Model Checker, https://prob.hhu.de/w/index.php?title=Main_Page. (Accessed 25 March 2021).
[41] J.G. Kemeny, J.L. Snell, A.W. Knapp, Denumerable Markov chains, https://doi.org/10.1007/978-1-4684-9455-6, 1976.
[42] J.G. Kemeny, J.L. Snell, Finite Markov Chains: With a New Appendix "Generalization of a Fundamental Matrix", Undergraduate Texts in Mathematics, Springer, 1983.