RoboCert: Property Specification in Robotics

Matt Windsor[1[0000−0003−1285−0080]] and Ana Cavalcanti[1[0000−0002−0831−1976]]

University of York, Department of Computer Science, Deramore Lane, Heslington, York, YO10 5GH, UK `{matt.windsor,ana.cavalcanti}@york.ac.uk`

Abstract. RoboStar is a toolkit for model-based development using a domain-specific notation, RoboChart, with enriched UML-like state machines and a custom component model. We present RoboCert: a novel notation, based on UML sequence diagrams, which facilitates the specification of properties over RoboChart components. With RoboCert, we can express properties of a robotic system in a user-friendly, idiomatic manner. RoboCert specifications can be existential or universal, include timing notions such as deadlines and budgets, and both safety and liveness properties. Our work is faithful to UML where it can be, but presents significant extensions to fit the robotics application needs. RoboCert comes with tooling support for modelling and verification by model checking, and formal semantics in *tock*-CSP, the discrete-time variant of CSP.

Keywords: RoboChart · Timed properties · CSP · sequence diagrams

1 Introduction

Mobile and autonomous robots are becoming common among us. While such systems come in many shapes and sizes, from vacuum cleaners to UAVs, each requires a degree of trust from users. A faulty robot risks mission or safety failures, causing reputational damage, financial loss, or injury to humans.

Software and hardware engineering for *trustworthy* robotic systems is, therefore, a key research topic. One approach, embodied by RoboChart [13] and its associated notations, combines model-driven development and formal methods. RoboChart provides practitioners with intuitive, graphical notations that have a well-defined meaning rooted in a formal semantics.

While RoboChart is well-established, with many successful case studies[1], its support for property specification is incomplete. Its assertion language is a thin layer atop the formalisms targeted by the RoboChart semantics—the CSP process algebra and the PCTL probabilistic logic—; users must be experts in those formalisms. We seek high-level notations resembling those already used by practitioners, which they can use with minimal adaptation to their workflows.

We introduce RoboCert, a notation for property specification over RoboChart models. RoboCert exposes a dialect of the sequence, or interaction, diagram notation provided by the Unified Modelling Language (UML) [5]. UML is a

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

widely known and supported, if loosely defined, set of notations for describing software behaviour; as such, it meets our criteria for familiarity and intuition.

Our key contributions are a formal metamodel and semantics for RoboCert. Novelty comes from our treatment of time properties, both as novel constructs in RoboCert diagrams, and in its semantics, based on the *tock*-CSP dialect of CSP [14, 2]. Timing properties are important for robotic systems, and so any useful property language must elegantly capture notions of delay and deadlock.

Section 2 outlines related work. Section 3 is a brief introduction to RoboChart. Section 4 introduces RoboCert through examples. Section 5 explores the well-formedness conditions and formal semantics of RoboCert in *tock*-CSP. Section 6 presents tool support currently available. Finally, section 7 gives conclusions.

2 Related work

Before outlining RoboCert, we discuss existing work in the relevant fields.

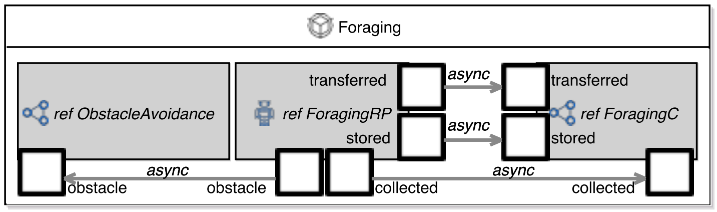
Lindoso et al. [11] have adapted UML activity diagrams as a property notation for RoboChart. Their work has both a semantics in CSP and graphical tooling. That work, however, focus on internal, as opposed to visible like in RoboCert, behaviour of state machines, and does not yet consider time. Sequence and activity diagrams complement each other, and so our work is complementary. Since they are concerned with refinement checking, unsurprisingly, some of their constructs (any and until) have analogues in RoboCert.

The main stream work on sequence diagrams is that of UML, which overlaps with work on the *Message Sequence Charts* standard [16]. Offshoots from these works pursue specific goals. *Live Sequence Charts* [4] (LSC) extend MSC to provide modalities for liveness analysis. *Property Sequence Charts* [1] (PSC) recast parts of UML and MSC, alongside novel constructs for constraining message chaining. In RoboCert, like in LSC, we have facilities to define universal and existential properties. We have a notion similar to the constrained *intraMSG*s of PSC in RoboCert (our 'until' construct). Importantly, in terms of notation, what is novel in RoboCert are the explicit constructs to deal with time, and the specific combination of constructors to support refinement-based compositional reasoning about models of robotic systems.

STAIRS is a sequence-diagram refinement process [8] based on a trace-pair semantics of UML. In contrast, we use refinement to compare a RoboCert diagram to a RoboChart design, not different sequence diagrams. This is, of course, a possibility, given the nature of refinement reasoning. The notion of refinement in STAIRS, however, is different: it allows for addition of behaviours for incremental development of diagrams. RoboCert refinement captures safety, liveness, reduction of nondeterminism, and timing preservation. Like STAIRS, RoboCert defines two forms of choice: internal and external. *Timed STAIRS* [7] extends STAIRS to consider UML timing constraints, which can specify budgets, like in RoboCert. In its semantics, events are time stamped.

Micskei and Waeselynck [12] surveyed the formal semantics for UML sequence diagrams available in 2011. They explored variants of the set of trace-pairs ap-

Fig. 1. The Foraging module.



proach of UML, as well as other denotational and operational semantics. This comparison shows that different use cases and interpretations have given rise to different semantics. Two of them are concerned with refinement: that for STAIRS, and [6], where refinement is inclusion of languages defined by input-enabled automata for a collection of sequence diagrams. None of the works compared provides a reactive semantics for refinement based on a process algebra.

Closest to our work are those in Lima et al. [10, 9]. They provide a process algebraic semantics to SysML and UML sequence diagrams, not including any time constructs. In contrast, RoboCert has constructs to define core properties of a timed system: time budgets, deadlines, and timeouts. RoboCert also adopts a component model (similar to that of RoboChart). There is a notion of robotic platform, system, and controller, as well as state machine. Messages correspond to events, operations, and variables that represent services of a robotic platform or other robotic components. A RoboCert sequence diagram is defined in the context of a specific component of a robotic system: the whole system, its sets of controllers, an individual controller, a set of timed state machines that define the behaviour of a controller, or, finally, a single machine.

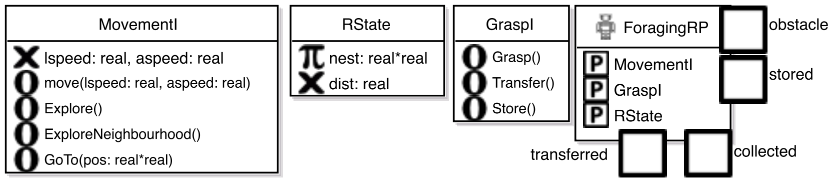
3 RoboChart

RoboChart [13] is a notation for the design of robotic control software. It provides domain-specific constructs, such as the notion of service-based *robotic platforms* that abstract over the robotic hardware. Another key feature is its discrete time constructs for specification of deadlines, delays, and time-outs.

A RoboChart model consists of several diagrams arranged in a hierarchy proceeding from the top-level *module* to individual *controllers*, each in turn containing state machines. Another diagram captures the robotic platform. Communications with the platform are asynchronous, reflecting their typical reification as sensors in the robot hardware. Communications between state machines are always synchronous; communications between controllers may be of either sort.

As a running example, we use the foraging robot of Buchanan et al. [3], which searches for items to store in a ‘nest’ location; it can transfer items to nearby robots and avoid obstacles. Figure 1 shows Foraging, the top-level module for our example. The platform block, ForagingRP, exposes four events to the software

Fig. 2. The robotic platform and the interfaces it provides.



(depicted by boxes and arrows). One (obstacle) signals to the ObstacleAvoidance controller that an obstacle is in the way. The others (collected, stored, and transferred) inform the ForagingC controller of the status of the items.

Figure 2 shows the robotic platform, ForagingRP. ForagingRP exposes the aforementioned events as well as three *provided interfaces* (**P**). Each (MovementI, GraspI, and RState) exposes operations (**O**), constants (π), and shared variables (**X**) for use by the software. For instance, the platform provides a move operation for setting linear and angular speed set-points. It then exposes the current speeds as variables lspeed and aspeed. A constant nest provides the fixed nest location. These elements abstract over actuators, sensors, and aspects of the environment.

Figure 3 depicts the controllers, and how they can both *require* (**R**) and locally define (**i**) interfaces. Both controllers contain state machines: ForagingC does so by reference to other diagrams, omitted here for space reasons, while ObstacleAvoidance directly contains Avoid. The complete example is available[2], as well as many other case studies and RoboChart's reference manual and tutorial.

The behaviour of Avoid, after transitioning from the initial state i, is a cycle of Waiting for an event obstacle, then Turning, then stopping the turn once complete. Both behaviours are described by calls to move on the platform.

Avoid makes use of RoboChart time primitives. The marker #T on the obstacle transition resets a clock T. Subsequently, since(T) gets the number of discrete time units observed since the reset; this information forms part of a condition (in square brackets) which guards the transition back to Waiting. These primitives form a time-based check for the completion of the turn.

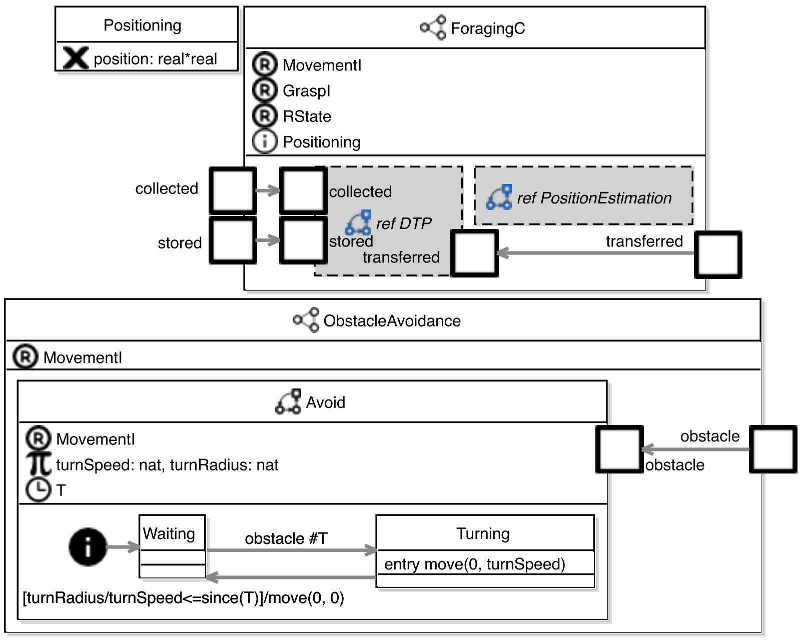
In RoboChart, state machines within a controller operate in parallel with one another, as do controllers within a module. This permits the modelling of multiple separate computational resources, threads, and other such constructs. Sequence diagrams, as a notation for capturing communications between parallel entities, capture these rich scenarios well. They are presented next.

4 RoboCert sequence diagrams

In this section, we present the main novel features of RoboCert sequence diagrams (or interactions, in UML terminology) via examples of properties of the foraging robot. A metamodel can be found in its draft report [17], and formal

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

Fig. 3. The ForagingC and ObstacleAvoidance controllers.



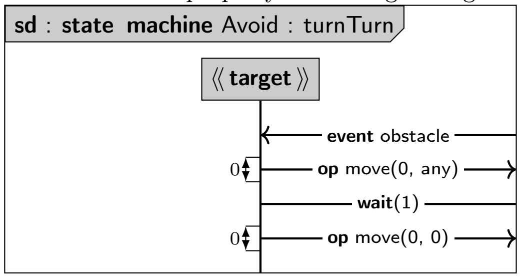
semantics is discussed in Section 5. RoboCert also includes the controlled English, presented in [13, 15] and omitted here, that can be used to define core properties (such as deadlock and livelock freedom) of RoboChart models.

A key characteristic of RoboCert compared to UML is that it adheres to the RoboChart component model. A RoboCert sequence diagram has a *target*: the RoboChart model element being constrained by the diagram. Each diagram also has a *world*: the environment of components residing above the target in the component hierarchy. Each component is represented in a RoboCert diagram by a vertical *lifeline*, with the vertical axis loosely representing the passage of time.

RoboCert sequence diagrams show traces of messages among components and the world, and associated time restrictions. Communications are represented by arrows between lifelines, with block-based control flow (parallelism, choice, and so on) depicted by boxes surrounding lifelines and relevant messages.

RoboCert sequence diagrams can capture *existential* (stating *some* expected behaviours of the system) or *universal* (capturing *all* expected behaviours) properties [4]. We can check whether a target conforms to a sequence diagram according to two semantic models. In the 'traces' model, we only check trace-set inclusion, treating the passage of time as a trace event and ignoring liveness. In the 'timed' model, we require conformance in terms of liveness, and that timing properties in both specification and model are fully compliant.

Fig. 4. An existential property containing timing features.



Component targets. There are two types of target: *component* and *collection* targets. Sequence diagrams for components capture black-box specifications of the behaviour of a component with respect to its world: they capture only the side of the communications visible to the target. Figure 4 shows a sequence diagram called turnTurn for an existential property whose target is the **state machine** Avoid. Every diagram is given in a box; on its top left-hand corner, the label sd includes the name of the target and the name of the diagram. In a component diagram, as in this example, there is a single lifeline, labelled ⟨⟨ **target** ⟩⟩. The world is represented by the box enclosing the diagram. Figure 4 captures a trace starting with the (input) event obstacle, from the world to the target.

Diagrams with module targets show messages between the control software represented by the module and its robotic platform. Diagrams with controller targets depict controller interactions with the platform and other controllers. Finally, the world for state machine and operation diagrams includes other machines in the same controller, the other controllers, and the platform.

Time. As well as the UML constructs for loops, parallel composition, optionality, and alternative choice, RoboCert has constructs for capturing timing properties: deadlines, time budgets, and timeouts. The **deadline** fragment constrains the amount of time units that can pass on a lifeline. The **wait** occurrence pauses a lifeline for a given amount of time units, to define time budgets.

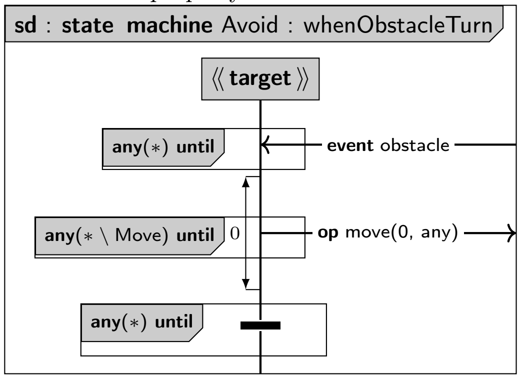
The diagram in Figure 4 depicts traces that start with the event obstacle followed by an immediate (taking 0 time units) call to move(0, **any**), with arguments 0 and **any** other value, effecting a turn. Afterwards, we have a time budget (**wait**) of one time unit, and an immediate move call stopping the turn.

To specify a timeout, we use **wait** in conjunction with **until**, presented next.

Until. Another RoboCert extension to UML is **until** fragments, useful to define partial specifications, which characterise sets of traces, rather than single traces, of universal properties. These fragments permit the passage of time and exchange of *any* messages in a given set *until* some occurrence inside the body takes effect. (They are related to UML the concepts 'strict' and 'ignore'.)

Figure 5 shows a diagram targeting the Avoid state machine. Its first **until** fragment states that any messages can be exchanged ($*$) **until** the event obstacle

Fig. 5. A universal property of the obstacle avoidance logic.



occurs. Afterwards, a series of messages can be exchanged immediately (with **deadline** 0) **until** a turn: a call to move(0, turnSpeed). Other calls to move, in the set Move, whose definition we omit, are excluded in the **until** via $*\backslash$Move. We define named sets such as Move outside of the diagrams, to allow reuse.

The last **until** fragment wraps a **deadlock** occurrence (━). This construct specifies that a lifeline has stopped making progress. Wrapping it in an **until** has the effect of allowing messages from the **until** set to occur indefinitely; this is the RoboCert idiom for encoding a universal diagram over partial traces.

A third use of **until** is to wrap a **wait** occurrence to define a timeout. This permits messages in the set until the time specified by the **wait** has passed.

Temperature. An extension we make to UML is the **hot** and **cold** modality for messages. On reaching a **hot** message, a model that satisfies the specified property must be willing to accept the message at any time. By default, messages are **cold**; the model is able to refuse these for any length of time.

Mandatory choice. As well as the UML **alt** fragment, RoboCert uses the **xalt** fragment. Informally, **alt** is a *provisional* choice where the model can decide which alternatives are available; **xalt** is a *mandatory* choice where the environment has full say and the model must accept all such choices.

The constructs **hot** and **xalt** add support to specify liveness aspects of a property. We can verify such a property by using the **timed** model.

Collections. Collection targets are defined by a component (module or controller), and reveal the internal representation of that component as a collection of subcomponents. In this case, the world remains the same as if the diagram had the corresponding component target, but each lifeline now maps to a distinct subcomponent; the target itself is not tangible here. We can observe both messages between the subcomponents and the world *as well as* messages from one subcomponent to another. Like in UML, the **component** lifelines produced by such subcomponents act in parallel except where connected by such messages.

Fig. 6. Existential property of obstacle avoidance in parallel with the main logic.

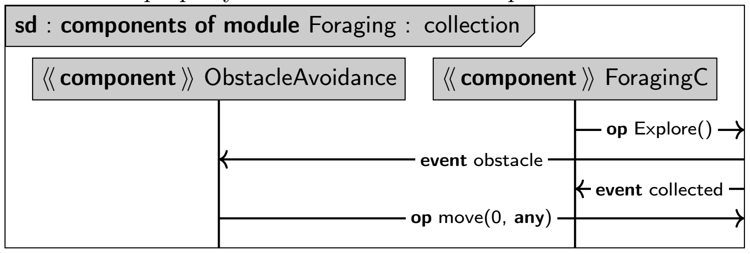


Figure 6 gives an example over Foraging, describing traces that interleave activities of ObstacleAvoidance with those of ForagingC. First ForagingC calls the operation Explore() of the world (platform). Inputs are then provided to the controllers by the world via events obstacle and collected The parallelism of the diagram means that Figure 6 admits traces where collected occurs before obstacle. Finally, ObstacleAvoidance calls move with arguments 0 and **any**.

Other features. Features not shown here for brevity and pragmatism include: a subset of the RoboChart expression language to define arguments and bounds; variables within interactions, with the possibility of capturing event and operation arguments to variables and using them within expressions; **loop**; and **par**.

5 Well-formedness and semantics

We now present well-formedness conditions and a semantics for RoboCert.

5.1 Well-formedness

To have a well-defined meaning, a sequence diagram must satisfy multiple well-formedness conditions. As expected, it must be well-typed and well-scoped according to standard rules, not discussed here. In addition, the referenced elements of RoboChart must comply with RoboChart well-formedness conditions.

Fragments are the main elements of sequence diagrams. They consist of *occurrences* (points on the vertical time axis, on which elements such as messages, deadlocks, and waits are located), and *combined fragments* (which implement control flow over other fragments), such as **deadline**, **until**, **opt**, for example.

Compound fragments contain one or more blocks of nested fragments; we call these their *operands*. Operands may have a guard, which must be true for the operand to be considered. By default, an operand's guard is **always**: a tautology. There are two other guard types: a Boolean expression, or **otherwise** (the negated disjunction of the guards of all operands in the fragment).

The following well-formedness conditions apply to fragments.

F1 *The operand of an* ***opt*** *or* ***loop*** *must not have an* ***otherwise*** *guard.* Since this guard is the negated disjunction of all other guards, and these fragments only have one operand, the **otherwise** guard would always be false.

F2 *At most one operand of* ***alt****,* ***xalt****, and* ***par*** *can have an* ***otherwise*** *guard.* Since this guard is the negated disjunction of all other guards, the two such guards would induce an ambiguous recursive definition.

F3 *A* ***alt****,* ***xalt****, or* ***par*** *must not have an operand with an* ***always*** *guard and one with* ***otherwise****.* The former would make the latter implicitly false.

F4 *A* ***deadline****,* ***deadlock****, or* ***wait*** *must not be* ***on*** *a* ***world*** *actor.* The **world** does not form a lifeline, so the fragment would not constrain anything.

F5 *An* ***until*** *must not contain another* ***until****.* Since the operand of an **until** defines the interrupting condition for the arbitrary occurrences of the allowed messages, it makes no sense for that condition to be an **until** itself.

Messages correspond to events, operation calls, and assignments to variables of the associated RoboChart model. In general terms, their use must be compatible with the RoboChart model. For example, the operation calls must have the right number of arguments of the right type. More interestingly, the source and target of an event message must be a component that defines that event, and the there must be a connection between them associating these events. The specific conditions in this category are listed in the report.

The semantics of diagrams defined next is for well-formed diagrams.

5.2 Semantics

RoboCert supports verification by refinement. Given the nature of robotic applications, namely concurrent reactive timed systems, we use *tock*-CSP, a discrete-time version of CSP to define the RoboCert formal semantics. Like in CSP, systems, their components, and properties are defined in *tock*-CSP as processes. These define patterns of interaction via atomic and instantaneous events, with the special event *tock* used to mark the passage of time. Table 1 enumerates the operators we use, which behave as per Baxter et al. [2], where a denotational semantics for *tock*-CSP, and an encoding for model checking are provided.

Diagrams. The diagram semantics is a parallel composition of lifeline processes, as well as a process for handling **until** fragments. This reflects the UML semantics of weak sequencing on occurrences across lifelines; the separate **until** process lets us step out of this parallelism when inside an **until** fragment. Figure 7 presents the *tock*-CSP process *turnTurn* defined by the diagram rule applied to Figure 4.

A diagram process, such as *turnTurn*, is defined by an iterated parallelism ($\|$) over its set of actors: just **target** in the example, abbreviated to t. It composes processes $lifeline(a)$, for each actor a, synchronising on the intersection of their sets of events $alpha(a)$. In our example, we have just one process $lifeline(t)$.

A $lifeline(a)$ process is defined as a sequence ($\fatsemi$) of processes representing the fragments of the lifeline, followed by a process $term \rightarrow Skip$, which uses the event *term* to flag termination. This event is internal to RoboCert, and so we

Operator	Meaning	Operator	Meaning
$tock$	event representing discrete passage of time	$Skip$	termination
$Stop$	timed deadlock; refuses all events except $tock$	$TChaos(s)$	timed chaos; permits any event in s, $tock$, or deadlock
$Stop_U$	timelock; refuses all events including $tock$	$P \blacktriangleright n$	deadline; permit at most n $tock$s to pass within P
$wait\ n$	wait for n $tock$s to pass	$e \rightarrow P$	timed event prefixing
$g \,\&\, P$	guard (if g then P else $Stop$)	$\mu P \bullet f(P)$	recursive process definition
$P \bigtriangleup Q$	interrupt	$P \sqcap Q$	internal choice
$P \Box Q$	external choice	$P \fatsemi Q$	sequential composition
$P \interleave Q$	interleaving parallel	$P \lbrack\!\lbrack a \rbrack\!\rbrack Q$	generalised parallel
$\parallel x : a \bullet \alpha \circ P$	indexed alphabetised parallel	$P \sqsubseteq_m Q$	refined-by (under model m)

Table 1. The *tock*-CSP operators used in our semantics

$$
\begin{aligned}
turnTurn &= \Big(\parallel a : \{t\} \bullet alpha(a) \circ lifeline(a) \Big) \setminus \{term\} \\
alpha(t) &= ctl \ \cup \ \{Avoid :: obstacle.in\} \ \cup \ \{x \in \mathbb{R} \bullet Avoid :: moveCall.0.x\} \\
lifeline(t) &= (\mu\, x \bullet (Avoid :: obstacle.in \rightarrow Skip) \sqcap tock \rightarrow x) \\
&\fatsemi ((\mu\, x \bullet (Avoid :: moveCall.0?x \rightarrow Skip) \sqcap tock \rightarrow x) \blacktriangleright 0) \\
&\fatsemi wait\ 1 \\
&\fatsemi ((\mu\, x \bullet (Avoid :: moveCall.0.0 \rightarrow Skip) \sqcap tock \rightarrow x) \blacktriangleright 0) \\
&\fatsemi term \rightarrow Skip
\end{aligned}
$$

Fig. 7. Semantics of turnTurn in *tock*-CSP – let t stand for $\langle\!\langle$ **target** $\rangle\!\rangle$

hide it at the top level using $\setminus$ $\{term\}$. In turnTurn, the only lifeline has four fragments: an occurrence fragment including obstacle, a **deadline** fragment (with the call move(0,**any**)), another occurrence fragment with the **wait**, and a final **deadline** fragment. Accordingly, we have four fragment processes in $lifeline(t)$.

For fragments with a cold message, we have a process that makes repeated internal choices as to whether engage in the event representing the message or allow time to pass ($tock$). For example, the obstacle fragment is defined by a recursive process ($\mu\ x$) whose body makes an internal choice ($\sqcap$). The first option is to engage in the event $Avoid :: obstacle.in$ representing the *in*put obstacle of the machine Avoid and terminate ($Skip$). The alternative choice engages in $tock$ and recurses (x). For a fragment with a hot message, the process engages in the event representing the message and terminates. There is no choice or recursion.

A **deadline** fragment has an operand: a list of fragments under a guard. Its semantics is a process defined by applying the *tock*-CSP deadline operator ($\blacktriangleright$) to the process for the operand. The semantics of an operand is a guarded process

$$
\begin{aligned}
&\mathit{whenObstacleTurn} = \Big(\Big(\parallel a : \{t\} \bullet \mathit{alpha}(a) \circ \mathit{lifeline}(a)\Big) \mathbin{[\![} \mathit{ctl} \mathbin{]\!]} \mathit{until}\Big) \setminus \mathit{ctl} \\
&\mathit{lifeline}(t) = (\mathit{sync}.0.\mathit{in} \to \mathit{sync}.0.\mathit{out} \to \mathit{Skip}) \mathbin{;} (\mathit{sync}.1.\mathit{in} \to \mathit{sync}.1.\mathit{out} \to \mathit{Skip}) \mathbin{;} \\
&\qquad\qquad (\mathit{sync}.2.\mathit{in} \to \mathit{sync}.2.\mathit{out} \to \mathit{Skip}) \mathbin{;} \mathit{term} \to \mathit{Skip} \\
&\mathit{until} = \mu\, x \bullet (\mathit{term} \to \mathit{Skip}) \\
&\qquad \Box\ (\mathit{sync}.0.\mathit{in} \to (\mathit{TChaos}(\mathit{Events}) \bigtriangleup \mathit{Avoid} :: \mathit{obstacle.in}) \to \mathit{sync}.0.\mathit{out} \to x) \\
&\qquad \Box\ \ldots
\end{aligned}
$$

Fig. 8. Semantics of whenObstacleTurn in *tock*-CSP

defined by a sequence of fragment processes. In the **deadline** fragments of our example, the operands are occurrence fragments. The semantics of a **wait** fragment is direct, since **wait** is also a primitive of *tock*-CSP to model time budget.

For diagrams with **until** fragments, the lifelines must coordinate so that each is aware of when the fragments start and end. We consider a diagram with lifelines a, b, and c, a fragment **any** (m) **until**: $a \twoheadrightarrow b$: **event** e, and, below it, a fragment $c \twoheadrightarrow w$: **op** o(). While messages in m are being exchanged, c cannot progress past the **until** fragment to the **op** o() call. At the same time, that call becomes available as soon as **event** e occurs, even though c is not involved in that event. So, an **until** fragment affects all lifelines of a diagram.

To capture this semantics, we add a process to handle the bodies of **until** fragments, placing it in parallel with the lifeline processes. We synchronise lifelines with this process to effect the **until** fragments.

To illustrate, Figure 8 sketches the *tock*-CSP translation of Figure 5. The sequence process *whenObstacleTurn* composes the parallelism of the *lifeline* processes in parallel with a new process *until* synchronising on the events in the hidden channel set *ctl*. We define $\mathit{ctl} = \{\mathit{term}\} \cup \mathit{sync}$, where *sync* is a set of events representing synchronisation induced by **until**: there is a pair of events for each **until** fragment. Each event is of the form $\mathit{sync}.i.d$, where i is the index of an **until**, and d is an *in* or *out* direction with respect to the **until** block. For **any** (m) **until**: $a \twoheadrightarrow b$: **event** e in the example above, there is a pair $\mathit{sync}.i.\mathit{in}, \mathit{sync}.i.\mathit{out}$ where i is the index of the **until** fragment in the diagram. In Figure 8, we have $\mathit{sync}.0.\mathit{in}$ and $\mathit{sync}.0.\mathit{out}$ capturing entering and exiting the fragment **any** (∗) **until**: $w \twoheadrightarrow t$: **event** obstacle, and similar pairs ($\mathit{sync}.1.\mathit{in}$, $\mathit{sync}.2.\mathit{in}$) and ($\mathit{sync}.2.\mathit{out}$, $\mathit{sync}.2.\mathit{out}$) for the next two fragments.

The *until* process is a recursion that offers either to acknowledge the termination of the lifeline processes and then terminates itself ($\mathit{term} \to \mathit{Skip}$) or to handle any of the three **until** fragments in Figure 5 (we show one of the choices in Figure 8). Each fragment choice is over an event on *sync* representing entering the fragment, followed by timed chaos (*TChaos*) on the events named in the fragment. (*TChaos*) captures the ability to perform any of the given events, consuming any amount of time or deadlocking. This can be interrupted ($\bigtriangleup$) by the

Rule	Definition
$[\![-]\!]^{\mathsf{S}}$	For a diagram named s, with body $b = \langle b_1, \ldots, b_n \rangle$: $\underline{[\![\textbf{sequence}\ s\ \textbf{actors}\ a_1, \ldots, a_n\ b]\!]^{\mathsf{S}}} \triangleq \left(\left(\parallel a : \underline{lines(a_1, \ldots, a_n)} \bullet alpha(a) \circ lifeline(a)\right) [\![ctl]\!] \underline{[\![b]\!]^{\mathsf{U}}}\right) \setminus ctl$ where $alpha(a) = \underline{\alpha(a, b)}$; $lifeline(a) = \underline{[\![b_1]\!]^{\mathsf{F}}_{\{a\}}} \fatsemi \underline{\ldots} \fatsemi \underline{[\![b_n]\!]^{\mathsf{F}}_{\{a\}}}$; and $until = \underline{[\![untils(b)]\!]^{\mathsf{U}}}$.
$[\![-]\!]^{\mathsf{U}}$	For a list $\langle f_1, \ldots f_n \rangle$ of **until** fragments: $\underline{[\![\langle f_1, \ldots f_n \rangle]\!]^{\mathsf{U}}} \triangleq \mu\, x \bullet (term \rightarrow Skip) \Box \underline{[\![f_1]\!]^{\mathsf{u}}_x} \Box \underline{\ldots} \Box \underline{[\![f_n]\!]^{\mathsf{u}}_x}$
$[\![-]\!]^{\mathsf{u}}$	For a single **until** fragment inside the *until* process: $\underline{[\![\textbf{any in}\ x\ \textbf{until}\ p]\!]^{\mathsf{u}}_x} \triangleq sync.\underline{isync(u)}.in \rightarrow (TChaos(\underline{[\![x]\!]^{\mathsf{MS}}}) \triangle \underline{[\![p]\!]^{\mathsf{P}}_{\mathsf{U}}}) \rightarrow sync.\underline{isync(u)}.out \rightarrow x$
$[\![-]\!]^{\mathsf{F}}$	By case analysis on types of fragment: $\underline{[\![\textbf{alt}\ x_1\ \textbf{else}\ x_2\ \textbf{else}\ \ldots x_n\ \textbf{end}]\!]^{\mathsf{F}}_a} \triangleq \underline{[\![x_1]\!]^{\mathsf{P}}} \sqcap \underline{[\![x_2]\!]^{\mathsf{P}}} \sqcap \underline{\ldots} \sqcap \underline{[\![x_n]\!]^{\mathsf{P}}}$ $\underline{[\![\textbf{xalt}\ x_1\ \textbf{else}\ x_2\ \textbf{else}\ \ldots x_n\ \textbf{end}]\!]^{\mathsf{F}}_a} \triangleq \underline{[\![x_1]\!]^{\mathsf{P}}} \Box \underline{[\![x_2]\!]^{\mathsf{P}}} \Box \underline{\ldots} \Box \underline{[\![x_n]\!]^{\mathsf{P}}}$ $\underline{[\![\textbf{opt}\ x\ \textbf{end}]\!]^{\mathsf{F}}_a} \triangleq \underline{[\![\textbf{alt}\ x\ \textbf{else}\ [\textbf{always}]\ \textbf{nothing end}]\!]^{\mathsf{F}}_a}$ $\underline{[\![\textbf{deadline}\ (x)\ \textbf{on}\ o]\!]^{\mathsf{F}}_a} \triangleq \underline{\text{if } o \in a \text{ then } ([\![o]\!]^{\mathsf{O}} \blacktriangleright [\![x]\!]^{\mathsf{E}}) \text{ else } [\![o]\!]^{\mathsf{O}}}$ $\underline{[\![o]\!]^{\mathsf{F}}_a} \triangleq \underline{\text{if } a \cap A(o) = \emptyset \text{ then } Skip \text{ else } [\![o]\!]^{\mathsf{O}}}$ (occurrence fragments) $\underline{[\![u]\!]^{\mathsf{F}}_a} \triangleq sync.\underline{isync(u)}.in \rightarrow sync.\underline{isync(u)}.out \rightarrow Skip$ (until fragments)
$[\![-]\!]^{\mathsf{O}}$	Messages are omitted here for brevity. $\underline{[\![\textbf{deadlock on}\ a]\!]^{\mathsf{O}}} \triangleq Stop$ $\underline{[\![\textbf{wait}(x)\ \textbf{on}\ a]\!]^{\mathsf{O}}} \triangleq wait\ \underline{[\![x]\!]^{\mathsf{E}}}$

Table 2. Selected semantic rules for RoboCert

trigger given in the fragment. Upon completion of a fragment, the process then engages in another *sync* event representing exiting the fragment, and recursing.

The *sync* communications synchronise with the *lifeline* processes. In our example, we only have **until** fragments; therefore, each fragment is modelled by a process that engages in a pair of *sync* events corresponding to the appropriate fragment in *until*. These pairs effect the handing-over of control from lifelines to the **until** fragment, then the hand-back once it is finished. All *lifeline* processes synchronise on all *sync* events, so that they all handover control to *until*.

Table 2 presents selected rewrite rules, from RoboCert to *tock*-CSP, defining the semantics of diagrams as we have just illustrated. Grey font denotes metanotation; standard mathematical (italics) font denotes *tock*-CSP target notation.

Rule $[\![-]\!]^{\mathsf{S}}$ expands a diagram s with actors a_1 to a_n. With $\mathit{lines}(a_1, \ldots, a_n)$ we get the set of all actors except the world. The set $\alpha(a, b)$ is the *alphabet* of an actor a within the fragment list b. For Figure 6, if o stands for ObstacleAvoidance, f for ForagingC, w for **world**, and b for the body of collection , then

$$\alpha(o, b) = \mathit{ctl} \cup \{w \twoheadrightarrow o\colon \textbf{event}\ \mathsf{obstacle}\} \cup \{x \in \mathbb{R} \bullet o \twoheadrightarrow w\colon \textbf{op}\ \mathsf{move}(0,\ x)\}$$

The definition of $\mathit{lifeline}(a)$ expands, per actor, to the sequential composition of the fragment rule $[\![-]\!]^{\mathsf{F}}$ for each fragment in b. That rule takes an actor set; some sequence elements only appear on the lifeline process if they relate to one of the actors in the set. Usually, the set contains only the actor of the lifeline being defined; the exception is when we expand fragments inside an **until**.

Let $\mathit{untils}(b)$ extract from fragment list b all **until**s nested in the list. For example, *untils* over the body of the diagram in fig. 5 yields:

$$\begin{aligned}\langle\, &\textbf{any in}\ (*)\ \textbf{until}\colon\ w \twoheadrightarrow t\colon\ \textbf{event}\ \mathsf{obstacle},\\ &\textbf{any in}\ (* \setminus \mathsf{Moves})\ \textbf{until}\colon\ t \twoheadrightarrow w\colon\ \textbf{op}\ \mathsf{move}(0,\ \textbf{any}),\\ &\textbf{any in}\ (*)\ \textbf{until}\colon\ \textbf{deadlock on}\ t\,\rangle\end{aligned}$$

Rule $[\![-]\!]^{\mathsf{U}}$ builds the process composing **until** fragment bodies, as extracted by *untils*. Inside a recursive process $\mu\, x$, we produce first the termination acknowledgement $\mathit{term} \to \mathit{Skip}$, then add, in parallel composition, one application of the sub-rule $[\![f]\!]^{\mathsf{u}}_{x}$ for every fragment f in the list. This rule, in turn, produces the timed chaos over the message set of f, interrupted by the expansion of the trigger of f. This uses a rule $[\![-]\!]^{\mathsf{P}}$ for fragment operands, which we omit for brevity. Each fragment expansion then ends in a recursive call to x.

Rule $[\![-]\!]^{\mathsf{F}}$ gives the semantics of fragments. As mentioned, this rule takes as an extra argument the set of actors for which we are expanding the fragment semantics. For **alt**, we combine the semantics of the branches using internal choice. The semantics for **xalt** is identical, but uses external choice. As in UML, **opt** equates to an **alt** where one branch is the body and another is empty.

For **deadline** fragments, we lift the operand into the *tock*-CSP deadline operator if, and only if, its bound actor is one of the ones we are considering. Otherwise, we pass through the operand semantics unchanged.

For **until** fragments, we synchronise with the *until* process. To do so, we find the correct $\mathit{sync.i}$ channel using *isync*, then emitting $\mathit{sync.i.in}$ followed by $\mathit{sync.i.out}$. While the fragment body is not used in this rule, it will execute sequentially on the *until* process once all lifelines synchronise on $\mathit{sync.i.in}$.

We elide productions for **loop** and **par** in $[\![-]\!]^{\mathsf{F}}$. The semantics for **loop** resembles the standard UML semantics; that is, no synchronisation between iterations. Furthermore, the semantics for **par** is similar to that of Lima et al.

Occurrences at fragment position whose *relevant* actors include any of the actors given to $[\![-]\!]^{\mathsf{F}}$ map to their occurrence semantics. Otherwise, they become *Skip*. In the rule, $A(e)$ refers to the relevant actors for e: the actor bound in any **on** clause on e; the endpoints if e is a message; or $\mathbf{U}$ for any other e.

Finally, $\llbracket - \rrbracket^{\mathsf{O}}$ is the semantic rule for occurrences. Both **deadlock** and **wait** map to their *tock*-CSP equivalents.[3] We omit the production for messages here, as it just delegates to the rule $\llbracket - \rrbracket^{\mathsf{M}}$ for messages with one caveat: it applies the previously mentioned recursive-process transformation when messages are **cold**.

While the full RoboCert semantics is available in [17], we elide some semantic rules for brevity, as well as the productions mentioned previously. We omit $\llbracket - \rrbracket^{\mathsf{E}}$ (expressions) as it is largely similar to its equivalent in the RoboChart semantics; the RoboChart semantics is fully defined in [13]. We also omit $\llbracket - \rrbracket^{\mathsf{MS}}$ (message sets), as it directly maps to set operations.

We omit the rule for messages ($\llbracket - \rrbracket^{\mathsf{M}}$), as it follows that of the analogous RoboChart constructs. The semantics of **op** messages is that of RoboChart calls; the semantics of **event** messages is that of RoboChart communications. Instances of **any** become CSP inputs. For example, we translate the messages in fig. 6 as:

$$\begin{aligned}
\underline{\llbracket \mathsf{f} \twoheadrightarrow \mathsf{w}\colon \textbf{op}\ \mathsf{Explore()} \rrbracket^{\mathsf{M}}} &\triangleq \mathit{ForagingC} :: \mathit{ExploreCall} \\
\underline{\llbracket \mathsf{w} \twoheadrightarrow \mathsf{o}\colon \textbf{event}\ \mathsf{obstacle} \rrbracket^{\mathsf{M}}} &\triangleq \mathit{ObstacleAvoidance} :: \mathit{obstacle.in} \\
\underline{\llbracket \mathsf{w} \twoheadrightarrow \mathsf{f}\colon \textbf{event}\ \mathsf{collected} \rrbracket^{\mathsf{M}}} &\triangleq \mathit{ForagingC} :: \mathit{collected.in} \\
\underline{\llbracket \mathsf{o} \twoheadrightarrow \mathsf{w}\colon \textbf{op}\ \mathsf{move(0,\ \textbf{any})} \rrbracket^{\mathsf{M}}} &\triangleq \mathit{ObstacleAvoidance} :: \mathit{moveCall}.0?x
\end{aligned}$$

6 Tool support

RoboCert has tool support in the form of RoboTool Eclipse plug-ins[4]. It adopts a textual encoding of RoboCert diagrams loosely based on the *Mermaid*[5] markup language. The tool reifies the well formedness rules and semantics in section 5.

Listing 1.1 shows the encoding of the diagram in Figure 4. In RoboTool, sequence diagrams are included in a *specification group* that has a name, and defines a target, possibly instantiating some of its constants. This is optional, but for successful verification by model checking, all constants need to have a value. In our example, the group is SAvoid. The **target** stanza defines the group's target as Foraging. Each **set to** line fixes constants of Foraging.

A specification group also allows the definitions of (short)names for the **actors** such as **target**s, **world**s, and **component**s up-front for later use in the diagrams. In our example, we define names T and W for the target and the world.

In the sequence turnTurn, each occurrence inhabits its own line, with message occurrences denoting the source and target actors using the $\twoheadrightarrow$ syntax. Occurrences and fragments over particular actors have an '**on** X' construct. A full syntax of RoboCert, while omitted here, is available in the report.

Listing 1.1 has two assertions: a *core property* (deadlock freedom) and a *sequence property*. In the latter, **is not observed** marks the property as existential

[3] Nondeterministic waits, taking range expressions, are planned for future revisions.
[4] `https://github.com/UoY-RoboStar/robocert-evaluation`
[5] `https://mermaid-js.github.io`

Listing 1.1. A RoboCert script textually encoding the sequence in fig. 4

```
specification group SAvoid
  target = state machine ObstacleAvoidance::Avoid with
    turnRadius, turnSpeed set to 2
  actors = { target as T, world as W }
  sequence turnTurn
    actors T and W
    W->>T: event obstacle
    deadline(0) on T: T->>W: op move(0, any)
    wait(1) on T
    deadline(0) on T: T->>W: op move(0, 0)

assertion Df : target of SAvoid is deadlock-free
assertion Tt : SAvoid::turnTurn is not observed in the traces model
```

(and negated, as we expect this property not to hold). To check universal specifications, we can use **does not hold** or **holds**. Also, **traces** states that we are interested in traces refinement; for the liveness proofs, we can use **timed**.

RoboTool compiles scripts such as those above into CSP-M scripts using the semantic rules in Section 5. Before compilation may proceed, RoboTool performs syntax checking, scope analysis, and the verification of healthiness conditions.

The RoboTool implementation validates our rules. Each rule is implemented separately, by its own method, and, apart from the fact that the rules are functional, and the implementation uses an imperative language, there is a one-to-one correspondence between the rule definitions and code.

The output of the RoboCert plug-in uses the RoboTool output for the targets, either directly (for component targets) or with composition to reflect the top-level structure of the component (for collection targets). We can therefore use the RoboTool verification facilities to model check properties against the automatically generated semantics of the actual artefacts.

7 Conclusions

This paper has introduced a domain-specific property language for robotics based on UML sequence diagrams, but significantly enriched to deal with compositional reasoning and time properties. This language enriches the RoboStar model-based framework, and, in particular, supports the specification and verification by refinement of properties of design models written in RoboChart. We have shown that sequences capture flows of events and operations in RoboChart models in a natural way. We have also shown how the diagrams map onto *tock*-CSP.

Our work complements that of Lindoso et al.'s [11] on activity diagrams for RoboChart models. We will expand support in RoboTool to integrate descriptions of sequence and activity diagrams. In addition, we note that our graphical notation is consistent and follows from the metamodel, but the diagrams here were manually produced. We need a tool for automatic diagram production.

A goal of RoboCert is to be a unified toolbox for property specification of RoboStar models. As such, we want to expand its notation set beyond core properties, sequence diagrams, and CSP. The RoboStar notations allow for parallel modelling of both robotic software and hardware, with the goal of co-verification. Extending RoboCert to express hybrid properties over both (discrete, logical) software and (continuous, physical) hardware behaviour is in our plans.

References

1. Autili, M., Inverardi, P., Pelliccione, P.: Graphical scenarios for specifying temporal properties: An automated approach. Autom. Softw. Eng. **14**, 293–340 (09 2007).
2. Baxter, J., Ribeiro, P., Cavalcanti, A.L.C.: Sound reasoning in tock-CSP. Acta Informatica (2021). https://doi.org/10.1007/s00236-020-00394-3, online April 2021
3. Buchanan, E., Pomfret, A., Timmis, J.: Dynamic Task Partitioning for Foraging Robot Swarms. In Swarm Intelligence. pp. 113–124. Springer (2016)
4. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Formal Methods Syst. Des. **19**(1), 45–80 (2001).
5. OMG Unified Modeling Language. Standard, Object Management Group (Dec 2017), `https://www.omg.org/spec/UML/2.5.1/PDF`
6. Grosu, R., Smolka, S.A.: Safety-liveness semantics for UML 2.0 sequence diagrams. In: 5th ACSD. pp. 6–14 (2005)
7. Haugen, Ø., Husa, K.E., Runde, R.K., Stølen, K.: Why timed sequence diagrams require three-event semantics. In Scenarios: Models, Transformations and Tools. pp. 1–25. Springer (2005)
8. Haugen, Ø., Stølen, K.: STAIRS – Steps To Analyze Interactions with Refinement Semantics. In «UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications. pp. 388–402. Springer (2003)
9. Jacobs, J., Simpson, A.C.: On a process algebraic representation of sequence diagrams. In: SEFM. LNCS, vol. 8938, pp. 71–85. Springer (2014)
10. Lima, L., Iyoda, J., Sampaio, A.: A Formal Semantics for Sequence Diagrams and a Strategy for System Analysis. In: MODELSWARD pp. 317–324. SciTePress (2014).
11. Lindoso, W., Nogueira, S.C., Domingues, R., Lima, L.: Visual Specification of Properties for Robotic Designs. In Formal Methods: Foundations and Applications. pp. 34–52. Springer (2021)
12. Micskei, Z., Waeselynck, H.: The many meanings of uml 2 sequence diagrams: A survey. Softw. Syst. Model. **10**(4), 489–514 (Oct 2011).
13. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J., Woodcock, J.: RoboChart: modelling and verification of the functional behaviour of robotic applications. Software & Systems Modeling **18**(5), 3097–3149 (2019).
14. Roscoe, A.W.: Understanding Concurrent Systems. Texts in Computer Science, Springer (2011)
15. Ye, K., Cavalcanti, A.L.C., Foster, S., Miyazawa, A., Woodcock, J.C.P.: Probabilistic modelling and verification using RoboChart and PRISM. Software and Systems Modeling (2021).
16. Message Sequence Chart (MSC). Standard, ITU-T (Feb 2011), `https://www.itu.int/rec/T-REC-Z.120-201102-I/en`
17. RoboCert Reference Manual. Report, RoboStar (May 2022), `https://robostar.cs.york.ac.uk/publications/reports/robocert.pdf`