Proceedings Paper:
Hendry, Holly, Cavalcanti, Ana, McCall, Cade et al. (2025) RoboScene: Notation for Formal Verification of Human-Robot Interaction. In: Boronat, Artur and Fraser, Gordon, (eds.) Fundamental Approaches to Software Engineering - 28th International Conference, FASE 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Proceedings. 28th International Conference on Fundamental Approaches to Software Engineering, FASE 2025, which was held as part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, 03-08 May 2025 Lecture Notes in Computer Science. Springer Science and Business Media Deutschland GmbH, CAN, pp. 166-187.

https://doi.org/10.1007/978-3-031-90900-9_9

RoboScene: Notation for Formal Verification of Human-Robot Interaction

Holly Hendry[1(✉)], Ana Cavalcanti[1], Cade McCall[2], and Mark Chattington[3]

[1] Department of Computer Science, University of York, York, UK
[2] Department of Psychology, University of York, York, UK
{holly.hendry,ana.cavalcanti,cade.mccall,mark.chattington}@york.ac.uk
[3] Thales Research, Technology and Innovation, Reading, RG2 6GF, UK

Abstract. Proving properties about robotic systems with humans-in-the-loop relies on assumptions about human behaviour. Existing technologies require expertise not reasonably expected from psychologists and human-factors engineers, for instance. A user-needs analysis of industrial design techniques for human-robot interaction has identified a lack of standardised approach. We present RoboScene, a notation based on UML sequence diagrams that can be used to capture assumptions derived from human-factors artefacts, through novel constructs enabling consideration of stakeholders with different traits. We describe a tock-CSP semantics for RoboScene, and show how we can connect (mathematically) RoboScene diagrams to platform-independent software models. This is applied in the context of a Human-Centered Engineering process, demonstrated via an industrial case study.

Keywords: RoboStar · Sequence diagrams · CSP · HCE

1 Introduction

Currently, capturing assumptions for mathematical reasoning about human behaviours when interacting with a robotic system requires expert knowledge that many stakeholders do not normally have [25,4,24]. Psychologists, human-factors engineers, and experts in human-robot interaction (HRI) in general are best placed to provide the data for human behaviour and interaction models, but the communication with those capable of generating formal models poses challenges. Cross-discipline communication should be facilitated without loss of precision.

Many recognise the importance of formal reasoning about HRI, and significant progress has been made on modelling and verification. Pioneer works model the human as a black-box deterministic input into the system [10], or obtain by (manual) translation a mathematical model of human-behaviour for verification from an accessible notation [2]. In addition, the pioneering works do not always consider connections with models for the robotic platform and operational scenarios [33,34] that play a key role in system-level reasoning.

A. Boronat and G. Fraser (Eds.): FASE 2025, LNCS 15693, pp.166–187, 2025.
https://doi.org/10.1007/978-3-031-90900-9_9

In this paper, we present RoboScene, a UML-based notation, accessible, with minimal training, to human-behaviour experts, that can be used to capture assumptions about human behaviour. RoboScene diagrams can be connected to models of system software, hardware, and the operational scenario to create a model with a system-level view. RoboScene can be used to generate automatically a mathematical process-algebraic model for verification of the entire system. With RoboScene, we can capture assumptions about humans as nondeterministic timed interactions with the robot or environmental elements.

In designing RoboScene, we have started with a user-needs analysis (UNA) to confirm the need and identify the requirements for a notation to capture human behaviour [16]. RoboScene is the notation we have designed to address these requirements. We present the RoboScene's metamodel and semantics in tock-CSP [32,1], a timed variant of CSP [18]. We demonstrate the use of RoboScene, via a search-and-rescue (SAR) drone example, and work from a Hierarchical Task Analysis (HTA) [37], carried out by Thales, to create RoboScene and RoboChart models. We follow an (automated) engineering process enabled by RoboScene, also described here. In doing this, we identify gaps in the HTA and prove properties about the time taken to complete a search.

We present our work in the context of the RoboStar framework [7], which includes domain-specific notations [5,6,38] and tools for robotics. In particular, we use RoboChart [27] to describe platform-independent models of control software. With RoboScene and RoboChart, we give an accessible and holistic account of a robotic-system design and explore properties and consequences of human behaviours at an early stage of development. RoboScene, however, is an independent novel notation that can be used in connection with models written in other notations. RoboScene is independent of the RoboStar framework, but connects well with it via their common process-algebraic foundations.

RoboScene adopts the structure of UML sequence diagrams, but includes novel constructs. RoboScene diagrams use a notion of capability, and actors to represent human stakeholders, the robotic platform, or other interaction devices. The capabilities of the platform are used to connect the assumptions in the sequence diagram and a software model. Groups of diagrams enable the definition of related assumptions for different human traits, such as "tired" or "untrained". A construct is also available to represent human decisions. Finally, time constructs allow the definition of assumptions about ranges of reaction times.

Our contributions are as follows: **(1)** RoboScene, a notation for modelling expected user behaviour accessible to various stakeholders; **(2)** mathematical semantics for RoboScene; **(3)** a technique to prove properties of a system that depend on human interaction; and **(4)** the description and demonstration of a process to use RoboScene through an industrial case study.

Next, we present related work. Section 3 summarises the UNA used to design RoboScene whose proposed use in an engineering process is defined in Section 4. RoboScene is described in Section 5, and formalised in Section 6. We demonstrate use of RoboScene in Section 7. We conclude and propose future work in Section 8.

2 Related work

In this section, we briefly describe three formal approaches to reason about HRI, and works that formalise UML-like sequence diagrams. In all cases, either there is no support for a notation that facilitates communication, or for verification involving key aspects of human behaviour: time and data exchange.

We single out three state-of-the art approaches to reason about HRI. PVSio-web [25] is an environment for user testing of PVS [30] models. IVY [4] is a textual notation that uses action-logic for modelling of human interaction and behaviour. In both cases, the creation of models requires specialist knowledge. The CIRCUS [24] component HAMSTERS provides a graphical interface for creating an HTA-like user task tree, but has no associated verification approach.

In terms of formal UML-like sequence diagrams, RoboCert [38] is a notation for defining and proving properties of RoboChart models and their components. RoboCert does not cater for human input and some of its constructs (such as parallel and alt) are not in accordance with the UML semantics.

Several works cater for manual [19,39,13,22] or automatic [20,28,36,29,11,35] generation of mathematical models, like we do, but do not cover timed constructs and reasoning, and some consider a restricted or modified set of UML constructs. Both [8] and [23] model these constructs, but do not cater for nondeterministic time specification in their sequence diagrams, and as such do not provide an opportunity to model human decision-making with respect to time.

3 User-Needs Analysis

To understand HRI design and verification techniques used in industry, we have undertaken 14 semi-structured interviews of industry professionals, across a variety of sectors, working in roles at different stages of the robotics-development lifecycle. Interviewees included human factors engineers, UX designers, software engineers, researchers, and technical directors [16].

UML sequence diagrams have been identified as a basis for the study due to their usability and existing use base across the robotics-development lifecycle. Their standardisation and documentation increase usability, and Kutar [21] indicates that these diagrams can be understood by those with little knowledge of the notation. To validate use of these diagrams, the interviews questioned the methods, tools, and standards used during design, how designers think about the human component of HRI, and the use of UML.

Through qualitative thematic and content analysis [3] of the interview transcripts, we have identified the need, and requirements, for a standardised notation to capture human behaviour that can facilitate effective communication. RoboScene is the notation we have designed to address these requirements.

Internal, external, and construct threats to validity have been considered. For instance, we have mitigated selection bias by ensuring no two participants had the same role and affiliation. We have addressed the risk of over representation via participant variety. We have also improved reliability by the interviewing researcher doing the initial encoding of the transcripts.

4 Modelling approach

Communication between robotic-system designers, identified through the analysis of the interviews as a key requirement, is facilitated through the approach described in Fig 1. The process begins with a model of the robotic system as defined by an expert on the human component; in our example this is a Hierarchical Task Analysis (HTA) [37] provided by a human-factors engineer.

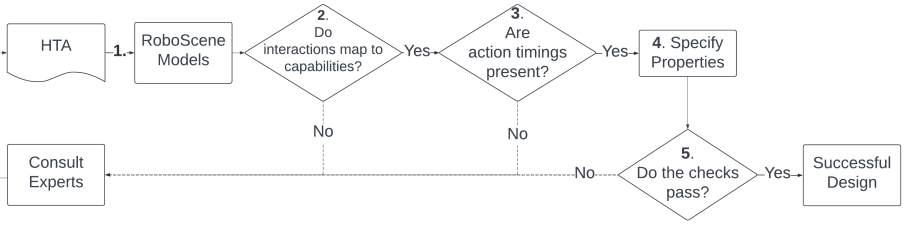


Fig. 1. Modelling flow for SAR example

HTA is a well-known notation and technique [37], with support provided by several tools [26]. In ongoing work, we are defining a metamodel for a version of HTA that is appropriate for use with RoboScene, and are defining a technique to extract RoboScene models from HTA. As we generate models of user scenarios in RoboScene we can handle large and complex HTAs. Here, we consider the HTA just as an informal source of information to write RoboScene models.

The next step of the process is the creation of HRI models using RoboScene. Validating the RoboScene model, in terms of how interactions are carried out using the capabilities of the robot and of information on timings for human actors, requires communication with domain experts in the case of an issue. These experts include software engineers, roboticists, hardware engineers, human-factors engineers, and maybe even prospective users. Issues of time may be addressed by additional computational capacity or improved interfaces, for example. So, our work is complementary to those in HCI concerned with identifying and improving the design of human-robot interfaces.

The process is iterative, and finishes when the design meets the properties. All steps except (5) are manual; our work presented here automates (5).

5 RoboScene

RoboScene addresses the requirements identified in the study described in Section 3. This is achieved by novel constructs to capture HRI scenarios: groups of sequence diagrams per scenario of use, a notion of traits for actors, memory (holding variables) to capture information exchange, time constructs to capture reaction times, and nondeterminism to capture uncertainty or time ranges. Moreover, self-directed messages are used to capture decisions. Here, we describe the metamodel of RoboScene (Section 5.1) and then present examples (Section 5.2).

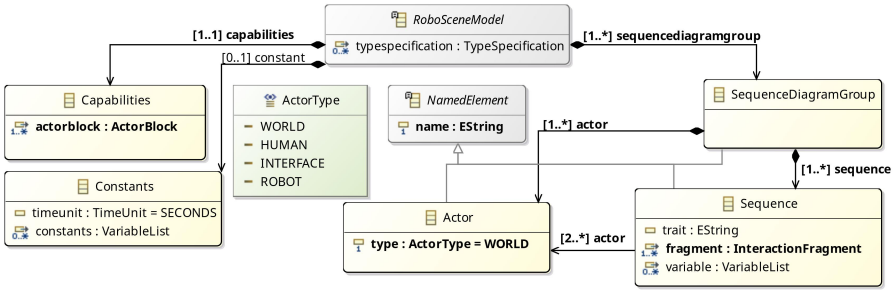


Fig. 2. An excerpt of the RoboScene metamodel

5.1 Metamodel

The RoboScene metamodel, sketched in Figure 2 and fully defined in [17], represents a RoboSceneModel via four attributes. It has exactly one declaration of a capabilities block, one or more sequencediagramgroups, and optional constants and typespecifications. The capabilities can include one or more actorblocks. An actorblock consists of exactly one actor, any number of variables, and any number of ins and outs corresponding to the lifeline events of the actor. An actor has a name and a type, set to WORLD as default. The sequencediagramgroups have a name, one or more actors, and one or more sequence diagrams. A sequence has a name, two or more actors, and one or more fragments, like in UML, but also optionally includes a trait and variables. Moreover, RoboScene has additional forms of fragments for wait and deadline.

Well-formedness conditions identify the valid instances of the RoboScene metamodel for which we can give semantics. These can be found at [17].

5.2 RoboScene overview

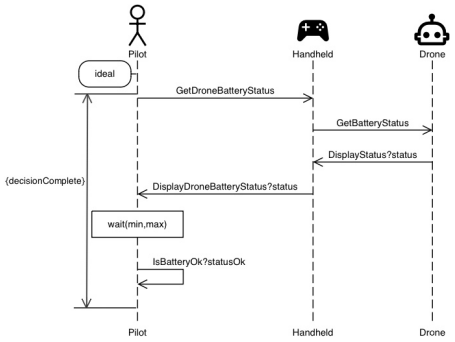
Figure 3 shows a small example inspired by our case study: it includes 7 capabilities, 3 actors, and 7 fragments. The full case study, developed in an industrial setting, has 32 capabilities, 5 actors, and 105 fragments [17]. In Figure 3, we give some Constants, Capabilities, and one diagram. Further examples of RoboScene sequence diagrams can be seen in Figures 4, 5, 6, 7 and 16.

Capabilities are variables, events, or operations, representing the actors possible interactions used in their lifelines. Capabilities declarations are grouped in blocks by actors. Variables (var) record inputs or outputs of messages for later use in any lifeline. The lifeline for the actor that declares the variable has write access to it; the others have only read access. Figure 3 shows three actors. For a Pilot, for instance, we have two variables status and statusOk.

Events and operations are declared to specify incoming (in) or outgoing (out) messages, respectively, for the lifelines of their actors. A declaration defines the source or target of the message. A message is a communication, possibly passing data, resulting in action on the source (events) or target (operations). For

```
Constants
        const min: int = 3
        const max: int = 5
        const decisionComplete: int = 6

Capabilities
    actor Pilot
        var status: Status
        var statusOk: boolean
        in IsBatteryOk <- Pilot
        in DisplayDroneBatteryStatus: Status <- Handheld
        out GetDroneBatteryStatus -> Handheld

    actor Handheld
        out DroneStopSearch -> Drone

    actor Drone
        out DisplayStatus: Status -> Handheld
```

(a) Capabilities Component



(b) Sequence Diagram Component

Fig. 3. RoboScene model of a simple drone system

instance, for the Pilot, DisplayDroneBatteryStatus is an incoming message from the Handheld; GetDroneBatteryStatus is outgoing to the Handheld.

Each group of sequence diagrams can represent a scenario of interaction between humans and robotic devices, such as nominal and off-nominal (failure) paths. Inside a group, diagrams can reflect different traits of the humans. In our full example, we have one diagram capturing the interactions and times for the Pilot when fatigued, and another, for when attentive. For a given scenario, we may have sequence diagrams for all possible combinations of traits. These RoboScene elements facilitate accurate representation of user stories and scenarios.

A scenario has several elements of interest. The first is the robot (which might be specified, for example, by a RoboChart model [27]); this is an actor of type robot. RoboScene can deal with multiple robot actors, although our verification approach presented in Section 7 is for a single-robot.

Each relevant input device, providing an interface to a robot, can be represented by an actor. In our example, we have the drone, represented by a robot actor, and a handheld device through which most interactions with a human happen. Since the handheld impacts on reaction times of the human and the robot, which transmit information through it, we need to model it as an interface.

Like a robot, an interface can be defined by, and connected to, another behavioural model, or be just an abstraction reflecting the management of the interaction flow. This choice might be influenced by the knowledge of the device software and the properties of interest. When using third-party software, we can create software models to record, for example, just response times.

In our example, we model the handheld in RoboChart as a buffer to store and pass data to and from the robot. Such a simple behaviour could be captured in RoboScene itself, but use of a RoboChart software model indicates that the times are a feature of the handheld, not of a particular scenario.

Every scenario can have at most one world actor, which represents entities that participate in the system but we do not need to represent individually. For

our example, in Figure 16, this includes the weather forecast agency, an external SAR management system, and the missing person call, among others.

The final type of actor is human. We can include multiple human actors to capture different roles of humans: for example, tele-operators, such as the Pilot and the Operator actors in our example, those interacting as a user, for instance, the missing person in our example, or just passers by.

UML constructs are defined as fragments in the metamodel and used to define the system flow. Communication between actors is via the UML message construct, which defines a sender and receiver for all data and events. Typically, interactions with robots are asynchronous, so RoboScene covers only this type of message. In our example, for instance, the human does not require the agreement of the handheld to request the drone battery status (although the request may be ignored) nor does the handheld require agreement of the human to display this status. Where synchronicity is required, we can use a protocol. In our example, we model both call and response messages (see Figure 3b).

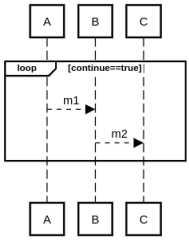


Fig. 4. Loop

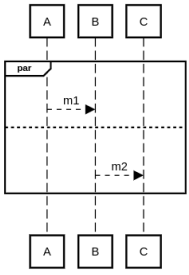


Fig. 5. Parallel

As already mentioned, we use self-directed messages of human actors to capture decision making. This message can be used to represent an input from the human, which can take time, and be recorded for later use in guards and messages. In Figure 3b, the Pilot's final message is IsBatteryOK?statusOk representing that a decision about the battery status is made by the Pilot and recorded in the variable statusOk.

To model nondeterminism through the system flow, RoboScene includes the UML constructs loop (see Figure 4 for an example), alt and opt. Each fragment can be guarded by an expression, perhaps using a human decision-making variable, or remain unguarded and entirely nondeterministic.

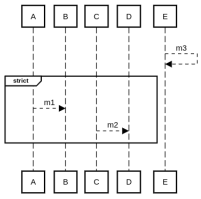


Fig. 6. Strict

The par construct enables parallel execution of lifelines fragments divided in threads. The sequential flow is maintained within a thread, but across threads it is relaxed. For example, without the par, in Figure 5 m1 has to come before m2, but with the par this restriction is lifted. This adds further nondeterminism and some freedom of action for actors.

In contrast to par, a strict fragment sequentially orders all its contained fragments, even across disconnected lifelines. For the sequence diagram in Figure 6, for instance, it ensures that m1 happens before m2; without the strict, that order is unconstrained. The order of messages outside the strict is unaffected; m3 in Figure 6, for example, can take place before, after, or between m1 and m2.

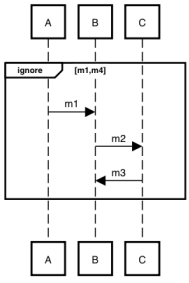


Fig. 7. Ignore

The ignore and consider fragments define which interactions are included in system traces. Messages are ignored or considered dependent on the fragment parameter list, regardless of whether those messages are present in the fragment block. For instance, for the diagram in Figure 7, traces do not include messages m1 or m4, the latter not actually occurring in the body of the ignore anyway.

To capture passage of time, RoboScene includes a wait construct defining a specific value, or minimum and maximum values for a nondeterministic waiting period. An example in Figure 3b is on the Pilot lifeline. Nondeterminism more accurately represents a human's interaction with a system, as humans are inherently variable, and even those trained will react within a margin of precision.

RoboScene also includes a deadline construct to restrict the time of a sequence of interactions: it ensures that every actor involved in the constrained interactions responds quickly enough. In our example, a deadline wraps (with a vertical bidirectional arrow) all interactions within the diagram, defining that they can take at most decisionComplete time units.

Next, we describe the formal semantics of RoboScene.

6 Formalisation

We now give a brief introduction to CSP and tock-CSP (Section 6.1), and an overview of the tock-CSP semantics of RoboScene (Section 6.2). The semantics are presented as rules that are the basis for automatic translation of RoboScene models to tock-CSP; an example is provided in Section 6.3.

6.1 CSP and *tock*-CSP

CSP is a process algebraic notation for modelling concurrent systems. Processes and channels are the key constructs of CSP, used to model systems, their components, and their interactions. Processes define possible interactions and passage of time as a sequence of events. Channels are used to define events representing interactions with possible communication of values.

As said, we use tock-CSP [32,1], a timed variant of CSP that provides the special event *tock* for the representation of the passage of a time unit. Table 1 summarises the tock-CSP operators used in this paper.

6.2 Overview of Semantics

The overall structure of the RoboScene semantics is outlined, using an informal notation of blocks for processes and || for parallel composition, in Figure 8. A RoboScene model, (1) in Figure 8, is given semantics as a collection of tock-CSP processes, one for each sequence diagram, defined over channels representing messages and variables corresponding to the Capabilities used in the diagram. For instance, for our drone, the process defines the allowed interactions over channels representing statusOk and DisplayBatteryStatus, among others.

Operator	Meaning	Operator	Meaning
tock	Passage of one time unit	$P \fatsemi Q$	Sequential composition
Skip	Termination	$g \& P$	Guarding
$e \rightarrow P$	Timed event prefixing	$P \mid\mid\mid Q$	Interleaving parallel
wait t	Delay of t time units	$P[\![c \leftarrow d]\!]$	Renaming
$P \blacktriangleright t$	Deadline; at most t *tocks* can pass within P	$P \ [\![X]\!] \ Q$	Time-synchronising parallel composition
$P \sqcap Q$	Internal choice	$\mu P \bullet f(P)$	Recursive function
$P \ \Box \ Q$	External choice	$P \setminus X$	Hiding

Table 1. *Tock*-CSP rules used in the RoboScene semantics

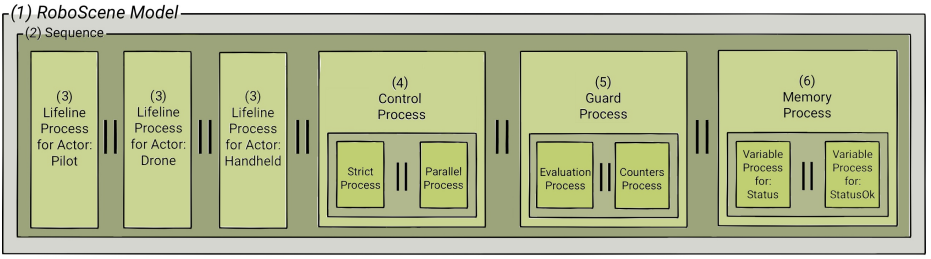


Fig. 8. Pictoral informal representation of the semantic structure of RoboScene

In Figure 8, we show the structure of a process for a sequence diagram, (2). For our example, that process is in Figure 9. It is, at the core, a parallel composition of processes that capture the behaviour of each of the lifelines. In tock-CSP, $P[\![cs]\!]Q$ is the parallel composition of processes P and Q synchronising on the events in the set cs and interleaving on the others.

Lifeline processes, (3), specify the interactions of a lifeline via a sequential composition of processes defining the interactions of fragments. In our example, the lifeline processes are $lifeline(Pilot)$, $lifeline(Handheld)$, and $lifeline(Drone)$. In Figure 10, we present $lifeline(Pilot)$ for the Pilot lifeline, defined by a single deadline fragment, and so a single process $deadline(Pilot)$ with a deadline given by the constant *decisionComplete*. The fragment process $deadline(Pilot)$ is in turn a sequential composition of fragment processes: one for each message and for the wait. Sequential composition is defined through $\fatsemi$ and timed event prefixing $\rightarrow$. A final control event *terminate* just signals the end of the lifeline.

Lifeline processes synchronise on the intersections of what we define as their alphabets: the sets of channels representing the capabilities of their actors. In Figure 9, we have alphabets $alpha(Pilot)$, $alpha(Handheld)$, and $alpha(Drone)$. In $alpha(Pilot)$, we have, for instance, the channel *GetDroneBatteryStatus*.

The semantics of a message fragment is a prefixing on the channel for the message; for the fragment with GetDroneBatteryStatus, see the process in Figure 10. If a value is communicated, then the corresponding variable is also set.

$$
\begin{array}{l}
BatteryCheck = \\
\left(\left(\left(\left(\left(\left(\begin{array}{l} lifeline(Pilot) \\ \quad [\![\ alpha(Pilot) \cap alpha(Handheld)\ ]\!] \\ lifeline(Handheld) \end{array}\right) \right. \right. \right. \right. \right. \\
\left. \left. \left. \left. \left. \quad [\![\ ((alpha(Pilot) \cup alpha(Handheld)) \cap alpha(Drone)\ ]\!] \right) \right. \right. \right. \right. \\
\left. \left. \left. \left. \quad lifeline(Drone) \right) \right. \right. \right. \\
\left. \left. \left. \quad [\![\ \{terminate, par, str\}\ ]\!] \right) \right. \right. \\
\left. \left. \quad Control \right) \right. \\
\left. \quad [\![\ \{loop, alt, guard, terminate\}\ ]\!] \right) \\
\quad Guard \\
\left. \quad [\![\ sharedVars \cup \{terminate\}\ ]\!] \right) \\
\quad Memory \\
\setminus \{loop, alt, par, str, guard\} \cup sharedVars
\end{array}
$$

Fig. 9. This tock-CSP process, called *BatteryCheck*, results from the application of a rule to translate the `Sequence` in Figure 3. That rule is defined using rules to translate `Actor`, `VariableList` and `InteractionFragment` constructs.

$$lifeline(Pilot) = deadline(Pilot) \blacktriangleright decisionComplete$$

$$deadline(Pilot) = GetDroneBatteryStatus \rightarrow DisplayDroneBatteryStatus?status \rightarrow setStatus!status \rightarrow (\sqcap x : \{min\,..\,max\} \bullet wait(x)) \fatsemi IsBatteryOk?statusOk \rightarrow setStatusOk!statusOk \rightarrow terminate \rightarrow SKIP$$

Fig. 10. Process for the Pilot lifeline in Figure 3, generated by the rule for `Sequence`, defined using rules for `Actor`, `VariableList`, and `InteractionFragment`.

For instance, for DisplayDroneBatteryStatus?status, the fragment process in Figure 10 uses *setStatus*!*status* to record the input value in the memory process.

The wait fragment is defined using the tock-CSP wait construct, and nondeterminism, if applicable. In our example, the fragment wait(min,max) is a nondeterministic choice (operator $\sqcap$) of values x between *min* and *max* to define a $wait(x)$. Just like for deadlines, there is direct rendering in tock-CSP.

The control flow of a lifeline, however, can be affected by control fragments. So, Control and Guard processes, (4) and (5) in Figure 8, synchronise with the lifeline processes through additional control channels to define the valid sequences of interactions. As our example has no Control nor Guarded elements, their corresponding processes are ready to engage in the terminate event immediately.

In general, however, the Control process gives semantics to strict and par fragments. *Control* composes in parallel Strict and Parallel processes, which themselves compose processes for each strict and par fragment in the diagram. The lifeline processes communicate with *Control*, via events *str.ID_STR* and *par.ID_PAR*, to pass across the control of its flow while inside a strict or par fragment. Each fragment is numbered and the event communicates an identifier, for instance, x in *str.ID_STR.x*, for each control fragment.

The *Parallel* process for the par in Figure 5 is shown in Figure 11. We also show the definition of the lifeline process for A, which immediately uses *par.ID_PAR*.1 to pass control to *Parallel* before terminating, since *A* has no other fragment. *Parallel* accepts events *par*?*ID_PAR.x* for all values of x; in our example, only 1 is used. For each identifier there is a guarded (&) choice ($\Box$) of an interleave (|||) of processes for the parallel threads. In our example, there are similar thread processes *parallel_1_1* and *parallel_1_2*. The definition of a

$$A = par.ID\_PAR.1 \rightarrow terminate \rightarrow Skip$$

$$Parallel = par?ID\_PAR.x \rightarrow ((x == 1)\&(parallel\_1\_1 \;|||\; parallel\_1\_2)) \fatsemi Parallel \\ \Box\ terminate \rightarrow Skip$$

Fig. 11. Process for a Par fragment is generated by our rules.

$$A = loop\_A\_1 \fatsemi terminate \rightarrow Skip$$

$$loop\_A\_1 = loop!ID\_LOOP.1 \rightarrow guard.ID\_LOOP.1.1?x \rightarrow \begin{pmatrix} (x)\&(m1 \rightarrow loop\_A\_1) \\ \Box \\ (not(x))\&(SKIP) \end{pmatrix}$$

$$Evaluation = reset\_counters \fatsemi guards\_response$$

$$guards\_response = loop?ID\_LOOP.id \rightarrow \\ \left(\begin{array}{l} (id == 1)\& \\ \left(\begin{array}{l} getcontinue?continue \rightarrow getCount.ID\_LOOP.1?x \rightarrow \\ \left(\begin{array}{l} not(continue == true)\& \\ \quad (setCount.ID\_LOOP.1!0 \rightarrow Skip \fatsemi guard.ID\_LOOP.1.1!false \rightarrow Skip) \end{array} \right) \\ \Box \ldots \end{array} \right) \\ \Box\ terminate \rightarrow Skip \end{array} \right) \fatsemi guards\_response$$

Fig. 12. Process for a Loop fragment generated by our rules.

thread process is similar to that for a sequence diagram matching the fragments in the thread, but without a memory or a guard process. To ensure that *Parallel* terminates when the lifeline processes do, *terminate* is offered in choice.

The definition of *Strict* is very similar to that of *Parallel* shown in Figure 11.

The loop, alt, and opt fragments determine the control flow based on the evaluation of guards (entry conditions). The processes for these fragments are captured in the lifeline processes, using the Guard process to evaluate their entry conditions and synchronising those evaluations as needed. Figure 12 shows the semantics for the diagram in Figure 4. The semantics for alt and opt are similar.

For the lifeline A, the process uses a loop process, *loop_A_1*, to capture its participation in a loop fragment. The definition of a loop process starts with an event $loop!ID\_LOOP.x$, where x is the identifier of the loop; 1 in our example. That event signals to the guard process the requirement to evaluate the entry condition. The processes for all lifelines involved in the loop synchronise on that event, so evaluation happens only when all lifelines are ready to enter the loop. After that, all lifelines receive the result of the evaluation via a *guard* event. With that result, the loop process decides ($\Box$) whether to iterate or finish (*SKIP*). The semantics in the case of an iteration is that of the fragment of the lifeline inside the loop (just $m1$ in the example), followed by a recursive call to the *loop* process.

Guard is a parallel composition of Evaluation and Counters processes. *Counters* handles counter variables to keep track of the number of times a loop is executed. To enable model checking, we limit the number of times that every loop can iterate. The *Evaluation* process resets the counters (process *reset_counters*) and proceeds as defined by *guards_response*, which composes processes for each of the guards of the loop, alt and opt fragments.

The *guards_response* process gets from the memory process the values for the variables used in the guard conditions, preventing potential inconsistencies

$$A = ignore\_A\_1 \,⨟\, terminate \rightarrow Skip$$

$$ignore\_A\_1 = (m1) \setminus \{m1, m4\}$$

Fig. 13. Process for an Ignore fragment generated by our rules.

$$[\![\mathsf{rsm} : \texttt{RoboSceneModel}]\!]_M \mathrel{\widehat{=}} [\![\mathsf{rsm.constant}]\!]_{CO}\ [\![\mathsf{rsm.caps}]\!]_C\ [\![\mathsf{rsm.seqdgrmgrps}]\!]_{SDG}\ [\![\mathsf{rsm.types}]\!]_{TY}$$

$$\begin{aligned}[\![\mathsf{s} : \texttt{Sequence}]\!]_S \mathrel{\widehat{=}} ((((&|\!|\ \mathsf{a} : \mathsf{actors}(\mathrm{a}_1, ..., \mathrm{a}_n) \bullet \mathrm{alpha(a)} \circ \mathrm{lifeline(a)})\\ &|[\ \overline{\{str, par, terminate\}}\ ]|\ \underline{Control(\mathrm{parFrags}, \mathrm{strFrags})})\\ &|[\ \{alt, opt, loop, guard, terminate\}\ ]|\ \underline{Guard(\mathrm{altFrags}, \mathrm{loopFrags}, \mathrm{optFrags})})\\ &|[\ sharedVars\ \cup\ \{terminate\}]|\ Memory) \setminus \overline{\{alt, opt, loop, guard, str, par\}}\end{aligned}$$

$$\cdots$$

$$\underline{\text{where}}\ \mathsf{actors} = \mathsf{s.actor}\ \underline{\text{and}}\ \mathsf{parFrags} = \underline{\mathrm{par}}(\mathsf{s.fragment})\ \cdots$$

Fig. 14. RoboScene CSP rules for a `Sequence`

arising if each lifeline evaluates the condition separately. In the example, for guard 1 ($id == 1$), *getcontinue* and *getCount.ID_LOOP*.1 are used. Afterwards, a choice ($\Box$), based on the result of the evaluation of the guard, defines the value communicated via the *guard* event. In Figure 4, A uses guard condition continue==true, so in *guards_response*, if $not(continue == true)$ holds, *guard* communicates *false*. Other choices are omitted in Figure 12.

The semantics for the ignore fragment in Figure 7 is in Figure 13. *A* uses the process *ignore_A_1* to capture *A*'s interactions within the ignore fragment, but with the channels for the fragment parameters $\{m1, m4\}$ hidden ($\setminus$). For consider fragments, the same process is used but the hidden set is the set of all possible interactions in the diagram minus the parameter set.

To capture the semantics of a diagram, we also have a memory process, (6), composing in parallel processes that record the values of the variables. This is a standard way of modelling state in CSP. Each variable in the Capabilities ActorBlocks is defined by a pair of *get* and *set* channels and a process that uses these channels to provide and update the value of the variable.

The Memory process (see Figure 9) synchronises with all processes on the variable *get* and *set* channels, which is included in the set *sharedVars* and hidden. Synchronisation occurs in all variable processes on *terminate*.

Next, we explain how the semantics of RoboScene models is formalised.

6.3 Formalisation

We have defined 41 semantic rules that map RoboScene models to tock-CSP; they can be found at [17]. They are the basis for automatic translation from RoboScene to tock-CSP. The rules use multiple typefaces to differentiate between references to the `metamodel`, a metanotation, variable names, and *CSP terms*.

The top rule defines the function $[\![\_]\!]_M$, which maps a `RoboSceneModel` to a collection of channel and type declarations, and process definitions. It is shown in Figure 14. The channel and type declarations are defined by rules for

`Capabilities`, `Constants`, and `TypeSpecifications`, whose simple definitions are omitted. The channels declared are those mentioned in the previous section. The processes are defined by the rule for a function $[\![\_]\!]_{SDG}$ for groups of sequence diagrams. This rule uses the function $[\![\_]\!]_S$ shown in Figure 14.

With $[\![\mathsf{s} : \texttt{Sequence}]\!]_S$, we get a tock-CSP process for a sequence diagram s. The first part of this process is the parallelium of lifeline processes, lifeline(a), for each actor, a, in s.actor over the alphabets, alpha(a), of these actors.

This in turn is put in parallel with the *Control*() process which, as previously explained, manages all parallel and strict fragments through channels *par* and *str*, respectively, and these channels are used to synchronise between the lifeline(a) processes and *Control*(). Similarly, the alt, opt, and loop fragments are managed by the *Guard*() process which uses *alt*, *opt*, *loop* and *guard* channels to synchronise with the lifeline(a) processes, using the *guard* channel to evaluate all fragment guards. The parallel composition of lifeline(a)s, *Control*() and *Guard*() is then synchronised with the *Memory* process over the alphabet of all accessible variables, *sharedVars*. All processes are synchronised on the special channel *terminate* to ensure that they all successfully conclude when the sequence ends. The channels *alt*, *opt*, *loop*, *guard*, *str*, *par* are considered internal events and as such are hidden, $\setminus \{\}$, within the sequence diagram process. Within the where statement, par is a syntactic filtering function that defines the set of fragments in s.fragment that are of the particular type `ParFragment` representing parallel fragments in the metamodel. Similar comments apply to strFrags, altFrags, loopFrags and optFrags, for the other forms of fragments.

Omitted in Figure 14, as $\cdots$, but found at [17], are the definitions of: processes for lifeline(a), *Control* and *Guard*; and the channels and datatypes defined for UML constructs; and, actor alphabets, $alpha(a)$.

Next, we describe how we have used the calculated semantics.

7 Verification

Here, we use a search-and-rescue UAV system, used in the field in the UK [15], to demonstrate the application of RoboScene. We present the needed software models for the UAV system (Section 7.1). We then present the RoboScene model obtained from an HTA (Section 7.2), specify and check properties of interest using RoboCert (Section 7.3), and report our results (Section 7.4).

7.1 Software Models

As RoboScene is intended for providing scenario models and guiding training, in a way usable by human-behaviour experts, it does not model the control software. For system-level verification we need comprehensive software and hardware models. We can connect these models to RoboScene models using capabilities.

RoboChart [27] is a notation to describe platform-independent behaviour of robotics software. In contrast to RoboScene, RoboChart defines complete behavioural models, not scenarios, with human-interaction abstracted via inputs

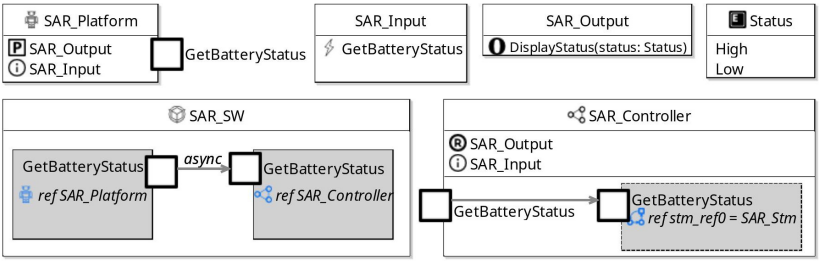


Fig. 15. RoboChart model of a simple drone

or outputs. RoboChart can be used to define design models for code generation, test generation, and verification of software properties, such as deadlock freedom.

RoboChart A brief overview of RoboChart is given here via a simplified version of our example. We describe just the RoboChart features needed to explain its joint use with RoboScene. The models for the full case study are available [17].

RoboChart models are modules characterised by an abstraction for the robotic platform defining services required by the software events and operations. The behaviour of a module is defined by one or more parallel controllers. Each controller is defined by one or more state machines also in parallel.

The RoboChart model for a simple drone software can be seen in part in Figure 15. The module is defined by the block labelled SAR_SW; it connects the robotic-platform block SAR_Platform with a single controller block, SAR_Controller. The blocks inside SAR_SW are references (keyword *ref*) to definitions outside. The simplified drone has a single event GetBatteryStatus as input: an abstraction declared in an interface called SAR_Input for a mechanism to receive commands. A single operation DisplayStatus declared in an interface SAR_Output is an abstraction for a light providing information about the battery. A call to DisplayStatus takes an argument of type Status: either High or Low. The definition of SAR_Platform declares the interfaces SAR_Input and SAR_Output.

SAR_Controller uses GetBatteryStatus and DisplayStatus (and so declares the interfaces). SAR_SW passes on the input from SAR_Platform to SAR_Controller, asynchronously. SAR_Controller in turn passes it on to a simple state machine SAR_Stm, omitted here. It just defines that DisplayStatus is called in response to an input GetBatteryStatus, after a certain amount of time.

The capabilities of the robotic platform provide potential links to a RoboScene model. These capabilities define the points of interaction available in the robot, so it is via them that a human can observe, affect, or be affected by the robot. For our example, the messages to and from the Drone in Figure 3 match the services of SAR_Platform in Figure 15.

Connecting RoboScene and RoboChart models To capture the behaviour of the entire system in the scenario defined by a RoboScene diagram, we can compose the processes for the RoboScene and the RoboChart models in parallel. Optionally, to capture the behaviour of any additional devices, such as a handheld,

we may define RoboChart models for them as well. For instance, for our SAR example, we have been advised to define a RoboChart model for the handheld to capture the time delay for communication between with the handheld and the actual drone. In this case, two RoboChart processes are composed. The parallel composition can be generated automatically based on the definition of the correspondence between capabilities of the diagrammatic models.

Synchronisation between the RoboScene and RoboChart models must reflect the way in which their capabilities are connected. For example, in Figure 3a we see that Handheld has a capability GetBatteryStatus, which is an output to the Drone. GetBatteryStatus is in the platform for the Drone RoboChart model, Figure 15, as an event and is an operation in the Handheld RoboChart model (available at [17]). The scenario captured in the RoboScene Model in Figure 3b controls the flow between the Handheld and the Drone using the GetBatteryStatus capability. In tock-CSP, renaming of the channels representing RoboChart platform capabilities in the RoboChart semantics, to their corresponding channels in our RoboScene semantics, establishes the required connection in the parallelism. For example, *GetBatteryStatusCall*, representing Handheld operation calls, is renamed to *GetBatteryStatus*, representing the Drone event.

7.2 Search-and-Rescue Drone Human-Interaction Model

Our example is from a research study that observed the Brecon Beacons search-and-rescue team utilising a UAV [15]. All information has been captured in a HTA [37], provided as a document defining tasks identified during the analysis. These tasks are associated with data recorded by the analyst: times, humans, and devices, or systems. The HTA, recorded in a spreadsheet, contains 121 tasks, each with fields for device, hardware, software, user, external communication source, time estimates, time windows, and more. We show here how we can use RoboScene to add value to this work with the process in Figure 1.

Step 1 in our workflow can be achieved by traversing the paths of the HTA, and ensuring they are covered within the RoboScene model. This scenario traversal is best achieved through a DFS, to cover all the subtasks of a task in the HTA.

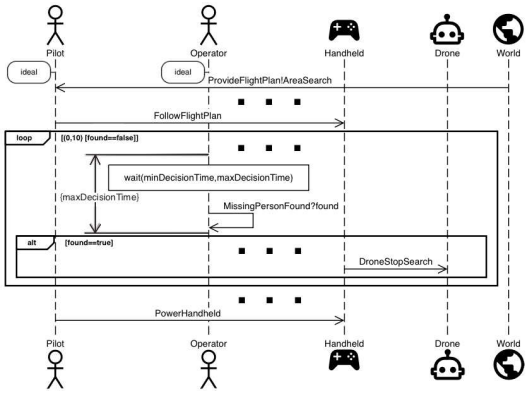
A RoboScene group of sequence diagrams captures a single scenario, rather than the entire HTA. In our example, available in [17], we cover the path for the Area Search. This has led to a group of three sequence diagrams representing the "ideal" users (Figure 16), an operator under pressure, and a fatigued operator. The sequence diagrams for the latter two traits are available at [17].

To create the RoboScene model, we first define actors, for the humans and devices, and capabilities to represent the tasks assigned to these actors. Figure 16 sketches the very large model of our example, showing some capabilities, and part of the end flow of one of the diagrams. The complete model has 32 capabilities, one group, with three diagrams; the one sketched has 35 fragments. The generated tock-CSP semantics (53 processes) is at [17].

We have five RoboScene actors: a UAV (of type robot), a handheld controller (interface), a Pilot (human), an Operator (human), and all further ex-

```
Constants
        const minDecisionTime: int = 1
        const maxDecisionTime: int = 5
        ...
        const communicationTime: int = 5

Capabilities
    actor Pilot
        var flight plan: SearchType
        out PowerHandheld -> Handheld
        out FollowFlightPlan -> Handheld
        ...

    actor Handheld
        out DroneStopSearch -> Drone
        ...

    ...

    actor Operator
        var found : boolean
        in MissingPersonFound : boolean -> Operator
        ...

    actor World
        out ProvideFlightPlan: SearchType -> Pilot
```

(a) Capabilities Component



(b) Sequence Diagram Component

Fig. 16. RoboScene model of a search-and-rescue UAV system

ternal organisations (world). The diagram in Figure 16b is for the "ideal" Pilot and Operator. A tired Operator might make mistakes, but with experience and training the fatigue of the operator may only be present as slower reaction and decision times [31]. This is captured by an increase on the minDecisionTime and maxDecisionTime for the operator when determining if a missing person has been found, and the time taken to communicate the found status. For an Operator under pressure, reaction times are quickened and the likelihood of erroneous behaviour is increased due to the skipping of tasks [12]. Erroneous behaviour may also come from the incorrect identification of the missing person. All this can be captured through (non)deterministic choice using alt or opt fragments.

Step 2 can highlight issues in the HTA, whose development is entirely informal. The first possible issue is that some of the tasks may not map to capabilities of the software. This reveals an inconsistency between the system analysis and the robot design that needs to be resolved. The second potential issue is that some necessary interactions between system actors may be missing. This indicates that the analysis has failed to require proper use of the robot. To address these issues, the human-factor engineers and roboticists need to give input.

In our example, the HTA had a task "set return to home", which did not map to any capability of the Drone. The experts indicated that we needed to model the Handheld with a InputHomeCoordinate capability, matching "set return to home". The Handheld records that information and uses it to control the Drone.

The Handheld model primarily defines a buffer but has a notion of time passing, including specific communication times that occur between the receiving and subsequent sending of data. It can be found in full at [17].

We also had in the HTA a task "initiate matrix search programme", which did not match any capability of the Drone. In that case, the human-factors team decided to expand the HTA. Many such issues have been identified.

Step 3 is concerned with lack of timing information for the HTA actions and human decisions. Obtaining this information may require studies, for example, identifying expected response times by providing users with simulations. For our example, the HTA did not record the decision-making times. To resolve that, we have worked with a human-factors engineer who, reflecting the unpredictability of human behaviour, provided nondeterministic timings to be modelled.

All of these problems have been due to a combination of missing data and ambiguity in the textual descriptions of the HTA. This does raise the question of whether a HTA is an appropriate source for a RoboScene diagram, but provided the analysis is complete, and has times recorded, then it is well-positioned.

7.3 Properties To Prove

RoboScene models support verification of safety and time properties using tock-CSP traces refinement. With that, we can, for instance, determine whether a task is, or is not, started or completed within a predicted time frame.

Trace refinement verifies whether specific execution paths exist by traversing through the model. Before specifying properties and checking them, however, we can provide an extensive collection of trace examples to the experts. These provide a basis for a detailed discussion of the common understanding of the model among the human-factors team, engineers, and verifiers. This can be done for each group of sequence diagrams, allowing in addition a preliminary analysis of the impact of the human traits on the system.

Step 4 The RoboCert [38] notation provides a sequence-diagram based approach for property definition, enabling those without expertise in formal methods to specify properties. Like RoboScene, it has a semantics defined in tock-CSP, and so it is convenient for verification using a CSP model checker [14].

As an example, we define a property via the diagram AreaSearchTimedProperty, partially seen in Figure 17, to verify that, once the flight plan is initiated, the area search completes within 170 time units. In AreaSearchTimedProperty, we require that first a sequence of events (omitted but indicated via ... in Figure 17) occurs, starting with ProvideFlightPlan and ending in FollowFlightPlan, then any event can occur, specified by any (*) until, but the event DroneStopSearch must be encountered within 170 time units. This is defined by the the vertical line enclosing the stated interactions with a deadline range of {0..170}. The lifeline continues with a number of events before concluding with PowerHandheld.

A RoboCert assertion using AreaSearchTimedProperty requires traces refinement, $\sqsubseteq_T$, by the SAR system (composing the RoboScene and RoboChart models). This assertion checks that the set of possible traces within the SAR system is a subset of those defined by AreaSearchTimedProperty.

Step 5 On checking AreaSearchTimedProperty, it fails and the counterexample demonstrates that at least 170+1 time steps could occur after FollowFlightPlan and before DroneStopSearch occurs. This has been identified as being due to missing constraints on the passage of time throughout the RoboChart model. As

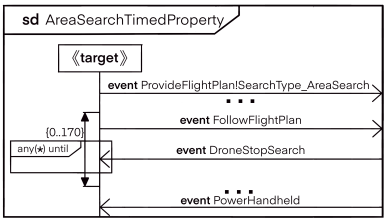


Fig. 17. RoboCert property for the Search-and-Rescue Drone

such, we have consulted domain experts and updated our RoboChart model to record the deadlines for the software response. This has resulted in the successful check of AreaSearchTimedProperty for all diagrams.

If a property of interest is concerned with a specific trace, we can also use a RoboCert assertion to check that it is possible for the system. This type of property can also be used to ensure specific traces are not possible, such as a failure trace. Finally, we can include an indication of expected time.

A trace can define specific amounts of time between events. For traces where these times correlate to the lower bound expectations from the "ideal" Pilot and Operator, the verification succeeds for the diagram whose trait is "ideal". The diagram for the Operator under pressure also passes these checks, since the upper bound for reaction time of the Operator in this case is the lower bound of the "ideal" case. Finally, the reaction time for the "fatigued" Operator has a minimum bound exceeding that of the "ideal"; so the check fails for that diagram.

7.4 Discussion

The process in Figure 1 is a systematic approach to creation and analysis of RoboScene models from a HTA source. It can be applied to other artefacts (such as a Circus [24] model), provided they specify the human actions and timings. The essential component of this flow is the experts consultation. To enable accurate model creation, through communication between human-behaviour experts and software engineers, use of RoboScene has proved key.

Verification failure suggests one of the following: inaccurate software modelling, missing or malformed RoboScene source data, incorrect mapping between the software and RoboScene models, or expectation of human interaction that does not align with that needed to satisfy the requirement. All these kinds of issues have been encountered and addressed as part of our industrial case study. These issues are symptoms of the communication difficulties between human-factors and software experts. Including them in the RoboScene model-creation process decreases the likelihood of these issues lingering during development.

Wrong expectations regarding human interaction with the system can indicate that a redesign of the robot is needed. Alternatively, the account of human interactions in RoboScene may be unrealistic and require further user testing, through simulation, prototyping, or analysis of users of an existing system. The timings defined for human reaction need checking separately for validity.

Not featured in this case study are hardware and operational scenario models. Including these robotic system models can identify further potential failure cases unique to HRI, such as unexpected weather events during a UAV flight.

A verified design that satisfies the properties of interest has successful property checks for all diagrams of a RoboScene model. These proofs are evidence that can potentially be used in a safety case. The expectations on human interaction can also inform training mechanisms for future users.

8 Conclusion

Through an industrial case study, we have presented RoboScene, a notation to capture assumptions for mathematical timed reasoning about human-robot interaction. RoboScene formalises UML 2.0 sequence diagram constructs and, unlike other approaches [19,20,22,13,9,8,23], adds the ability to capture (non-deterministic) time properties within the modelled user scenarios. Use of RoboScene requires no knowledge of a mathematical notation, making it accessible to various stakeholders. RoboScene enables cross-discipline communication between human-factors and software experts without the loss of precision.

The UML constructs, Assert and Neg are not included in RoboScene as they are concerned with trace validation, which is handled by RoboCert [38]. The constructs break and critical are also not included since we can use, respectively, an alt construct and loop guard, or alt and strict constructs, to express the same behaviour. Similarly, synchronous messages can be achieved through defining an asynchronous message as a response. Finally, as the standard flow within all sequence diagrams corresponds to weak sequencing, the omitted seq construct can be replicated through exiting and reentering the containing strict constructs.

Future work Evaluation through an additional industrial case study is the next step in our work. Quantitively comparing the impact of RoboScene relative to traditional methods and usability testing, with human-factors and software engineers, are planned to take place during the case study. To overcome the scalability concerns with the use of model checking, the RoboStar team is investing in support of theorem proving using Isabelle. Additionally, RoboScene diagrams can be used as a source to define runtime verifiers that can flag when assumptions about human behaviours are violated.

Acknowledgments. We would like to acknowledge the participants of the UNA. The work is funded by the UK EPSRC Grants EP/R025479/1, and EP/V026801/2, by the Royal Academy of Engineering Grants No CiET1719/45 and IF2122\183, and by a Thales EPSRC iCASE Grant EP/W522296/1, Award No 2605294.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Baxter, J., Ribeiro, P., Cavalcanti, A.L.C.: Sound reasoning in tock-CSP. Acta Informatica **59**, 125–162 (2022). https://doi.org/10.1007/s00236-020-00394-3
2. Bolton, M., Bass, E.: A Method for the Formal Verification of Human-interactive Systems. Proc. of the HFES Annual Meeting **52**, 764–768 (11 2009)
3. Braun, V., Clarke, V.: Using thematic analysis in psychology. Qualitative Research in Psychology **3**, 77–101 (01 2006)
4. Campos, J.: IVY Workbench, http://ivy.di.uminho.pt/
5. Cavalcanti, A., Baxter, J., Hierons, R., Lefticaru, R.: Testing robots using CSP (September 2019), https://eprints.whiterose.ac.uk/150135/
6. Cavalcanti, A.: Modelling and Verification of Robotic Platforms for Simulation Using RoboStar Technology, pp. 3–5. Springer International Publishing (05 2020)
7. Cavalcanti, A., Barnett, W., Baxter, J., Carvalho, G., Filho, M.C., Miyazawa, A., Ribeiro, P., Sampaio, A.: RoboStar Technology: A Roboticist's Toolbox for Combined Proof, Simulation, and Testing. Software Engineering for Robotics (2020)
8. Chen, X., Mallet, F., Liu, X.: Formally Verifying Sequence Diagrams for Safety Critical Systems. In: TASE 2020. pp. 217–224 (2020)
9. Chen, Z., Zhenhua, D.: Specification and Verification of UML2.0 Sequence Diagrams Using Event Deterministic Finite Automata. In: SSIRI-C 2011. pp. 41–46 (2011)
10. Choi, B., Park, J., Park, C.H.: Formal Verification for Human-Robot Interaction in Medical Environments. In: HRI '21 Companion. pp. 181–185 (03 2021)
11. Cunha, E., Custodio, M., Rocha, H., Barreto, R.: Formal Verification of UML Sequence Diagrams in the Embedded Systems Context. In: SBESC 2011. pp. 39–45 (2011)
12. Donkin, C., Little, D.R., Houpt, J.W.: Assessing the speed–accuracy trade-off effect on the capacity of information processing. J Exp Psychol Hum Percept Perform **40**(3), 1183–1202 (Mar 2014)
13. Ejnioui, A., Otero, C.E., Qureshi, A.A.: Formal semantics of interactions in sequence diagrams for embedded software. In: 2013 ICOS. pp. 106–111 (2013)
14. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3 - A Modern Refinement Checker for CSP. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 187–201 (2014)
15. Hart, S., Steane, V., Bullock, S., Noyes, J.M.: Understanding human decision-making when controlling UAVs in a search and rescue application. In: IHIET 2022: Artificial Intelligence & Future Applications. AHFE International (2022)
16. Hendry, H., Cavalcanti, A., McCall, C., Chattington, M.: Modelling of Human Behaviour in Robotic Systems, https://robostar.cs.york.ac.uk/publications/reports/Human_Behaviour_in_RS.pdf, Working Paper/Draft
17. Hendry, H., Cavalcanti, A., McCall, C., Chattington, M.: RoboScene Materials, https://github.com/UoY-RoboStar/RoboScene
18. Hoare, C.: Communicating Sequential Process. CACM **21**, 666–677 (08 1978)
19. Jacobs, J., Simpson, A.: On a Process Algebraic Representation of Sequence Diagrams. In: Canal, C., Idani, A. (eds.) Software Engineering and Formal Methods. pp. 71–85. Springer International Publishing (2015)
20. Kaizu, T., Isobe, Y., Suzuki, M.: Refinement and Verification of Sequence Diagrams Using the Process Algebra CSP. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **96-A**, 495–504 (2013)

21. Kutar, M., Britton, C., Barker, T.: A Comparison of Empirical Study and Cognitive Dimensions Analysis in the Evaluation of UML Diagrams. In: Proc. of the 14th Workshop of the PPIG (01 2002)
22. Li, X., Liu, Z., Jifeng, H.: A formal semantics of UML sequence diagram. In: 2004 ASWEC Proc. pp. 168–177 (2004)
23. Lima, V., Talhi, C., Mouheb, D., Debbabi, M., Wang, L., Pourzandi, M.: Formal Verification and Validation of UML 2.0 Sequence Diagrams using Source and Destination of Messages. Electr. Notes Theor. Comput. Sci. **254**, 143–160 (10 2009)
24. Martinie, C., Navarre, D., Palanque, P., Barboni, E., Pottier, G., Winckler, M.: Circus Tool Suite, https://www.irit.fr/recherches/ICS/softwares/circus/
25. Masci, P., Oladimeji, P.: PVSio-web, http://www.pvsioweb.org
26. Microsoft Corporation: Microsoft visio (2024), https://products.office.com/en/visio/flowchart-software
27. 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**, 1–53 (10 2019)
28. Muram, F.U., Tran, H., Zdun, U.: A model checking based approach for containment checking of uml sequence diagrams. In: 2016 23rd Asia-Pacific Software Engineering Conference (APSEC). pp. 73–80 (2016). https://doi.org/10.1109/APSEC.2016.021
29. do Nascimento, F.A.M., da Silva Oliveira, M.F., Wagner, F.R.: Using MDE for the Formal Verification of Embedded Systems Modeled by UML Sequence Diagrams. In: SBCCI'09. SBCCI '09, ACM, New York, NY, USA (2009)
30. Owre, S., Rushby, J., Shankar, N.: PVS: A Prototype Verification System. In: Kapur, D. (ed.) Automated Deduction—CADE-11. vol. 607, pp. 748–752. Springer, Berlin, Heidelberg (02 2001)
31. Román, C.A.F., DeLuca, J., Yao, B., Genova, H.M., Wylie, G.R.: Signal Detection Theory as a Novel Tool to Understand Cognitive Fatigue in Individuals With Multiple Sclerosis. Front Behav Neurosci **16**, 828566 (Mar 2022)
32. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall Series in Computer Science, Prentice-Hall (1998)
33. Sadigh, D., Driggs-Campbell, K., Puggelli, A., Li, W., Shia, V., Bajcsy, R., Vincentelli, A., Sastry, S., Seshia, S.: Data-Driven Probabilistic Modeling and Verification of Human Driver Behavior. In: AAAI Spring Symposium. pp. 55–61 (03 2014)
34. Sadigh, D., Sastry, S., Seshia, S., Dragan, A.: Planning for Autonomous Cars that Leverage Effects on Human Actions. In: Robotics: Science and Systems (06 2016)
35. Saputra, A.B., Basuki, T.A., Tirtawangsa, J.: Transformation of UML 2.0 sequence diagram into Coloured Petri Nets. In: 2014 ICAICTA. pp. 243–248 (2014)
36. Shen, H., Robinson, M., Niu, J.: A Logical Framework for Sequence Diagram with Combined Fragments (2011), https://api.semanticscholar.org/CorpusID:8677948
37. Shepherd, A.: HTA as a framework for task analysis. Ergonomics **41**, 1537–52 (12 1998)
38. Windsor, M., Cavalcanti, A.: RoboCert: Property Specification in Robotics. In: Riesco, A., Zhang, M. (eds.) Formal Methods and Software Engineering. pp. 386–403. Springer (2022)
39. Zafar, N.: Formal Specification and Verification of Few Combined Fragments of UML Sequence Diagram. Arabian Journal for Science and Engineering **41** (02 2016)