# Modelling and Verification of Timed Robotic Controllers

Pedro Ribeiro[1], Alvaro Miyazawa[1], Wei Li[2],
Ana Cavalcanti[1], and Jon Timmis[2]

[1] Department of Computer Science, University of York, York, YO10 5GH, UK,
pedro.ribeiro@york.ac.uk
[2] Department of Electronic Engineering, University of York, York, YO10 5DD, UK

**Abstract** Designing robotic systems can be very challenging, yet controllers are often specified using informal notations with development driven primarily by simulations and physical experiments, without relation to abstract models of requirements. The ability to perform formal analysis and replicate results across different robotic platforms is hindered by the lack of well-defined formal notations. In this paper we present a timed state-machine based formal notation for robotics that is informed by current practice. We motivate our work with an example from swarm robotics and define a compositional CSP-based discrete timed semantics suitable for refinement. Our results support verification and, importantly, enable rigorous connection with sound simulations and deployments.

**Keywords:** semantics, refinement, process algebra, CSP, robotics

## 1 Introduction

Robotic systems have applications in many real-life scenarios, ranging from household cleaning to critical search-and-rescue operations. Assessing their expected behaviour is challenging. In spite of that, typically controller software is developed in an ad-hoc manner, driven by simulations and physical experiments, but without a clear relation with models of requirements and design.

Standard state-machine notations, without underlying formal semantics, are often used [1,2] together with natural language annotations to specify more complex behaviours, involving aspects such as time and probabilities. State machines are often neither presented in an abstract way, nor do they contain precise and sufficient information to relate the designs to the simulations and deployments. In this scenario, the ability to faithfully replicate results, even just across different simulators, let alone using different robotic platforms, is significantly hampered.

In this paper we present a timed semantics for RoboChart [3], a state-machine based notation that can be characterised as a UML profile extended with time primitives and with a formal semantics. RoboChart provides constructs for capturing the architectural patterns of typical timed and reactive robotic systems. An abstract characterisation of a robot's operations and events is formalised via the notion of a robotic platform that decouples the software and hardware

platform from controllers. A controller can encapsulate multiple state-machines, and is connected with a particular platform via the notion of a module. This enables an abstract and precise approach to the design of robotic systems, where high-level concepts can be mapped into low-level constructs of typical executable simulations, for example, as we have considered in [3].

Here we propose a compositional semantics for refinement using Timed CSP [4], enriched with deadline constructs from *Circus Time* [5], a discrete-time process algebra that combines constructs of Z [6], CSP [7], and Timed CSP, besides deadline operators. A semantics for the enriched Timed CSP is defined in the Unifying Theories of Programming [5,8].

For RoboChart models that make a modest use of data types, we translate the semantics to CSP using a special event *tock* to mark the time. This version of CSP, called *tock*-CSP [7], is supported by the model checker FDR [9]. We use it to validate the design of RoboChart and our semantics, and check timed properties of RoboChart models. With *tock*-CSP, we can give a discrete-time model for all constructs of Timed CSP and deadlines.

The encoding in *tock*-CSP is mechanised in RoboTool, a graphical editor for RoboChart models. Using RoboTool and the automatically generated semantics, we have tackled a number of examples, and present here four experiments: two chemical detectors [10], an alpha algorithm used in swarm robotics [11], and a transporter that works in a swarm to move an object to a goal position [1].

Our long-term objective is to use our semantics for verification by automated theorem proving using an Isabelle encoding of *Circus Time* [12], and prove that automatically generated simulations are sound, that is, refine the RoboChart models. Translation from Timed CSP with deadlines to *Circus Time* is not challenging, since *Circus Time* is a richer language.

In Section 2 we motivate our work by presenting an example of a typical timed robotic controller, as used in swarm robotics, and giving an insight into related work. In Section 3, we present RoboChart. We discuss in detail the RoboChart timed semantics in Section 4. In Section 5 we present verification results and discuss tool support. Finally, we summarize our contributions and provide pointers for future work in Section 6.

## 2 Modelling Robotic Controllers

We now present an example (Section 2.1) and related works (Section 2.2) to indicate the need for a specialised timed formal language.

### 2.1 Motivating Example

Our goal is not to propose an entirely novel notation, but to define a language that is akin to that currently adopted by roboticists in their informal approach. We present in this section an example, taken from the domain of swarm robotics, whose published model is representative of the current practice.

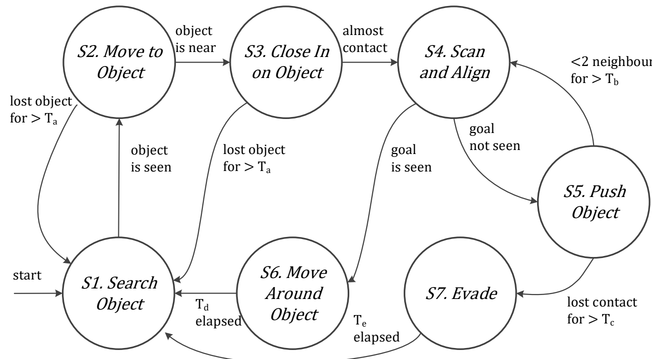


Figure 1: Transport swarm state machine [1].

We consider an individual timed reactive controller used in robots of a swarm for cooperatively transporting tall objects towards a locally perceived goal [1]. The robotic platform has a camera that allows it to distinguish objects and the goal, and proximity sensors that can be used to estimate the distance to an object and to detect other nearby robots.

In Figure 1 we reproduce the transport swarm controller in [1]. In state $S1$ the robot searches for an object and, once it sees one, it transitions to state $S2$. If the object is near, then it transitions to state $S3$. While in states $S2$ and $S3$, if the object is lost for a certain amount of time $T_a$, the robot initiates another search for the object by transitioning to state $S1$. When the robot is close enough to the object, by transitioning from state $S3$ to $S4$, it performs an alignment procedure and checks whether the goal can be seen. The underlying idea is that if the goal is occluded by the object, and the robot is close to the object, then it pushes the object towards the goal. While pushing, in state $S5$, the robot may lose contact with the object, in which case after a time threshold of $T_c$ it evades the vicinity; or it may lose sight of nearby neighbours, in which case it tries to align itself again by transitioning to state $S4$. The transitions between states $S7$ and $S1$, and $S6$ and $S1$, are equally timed according to thresholds $T_e$ and $T_d$.

We observe that the state machine in Figure 1 is specified in natural language and a few aspects are unclear, such as the behaviour and time spent in each state, whether timed transitions take place immediately or need to wait until the behaviour has completed, and thresholds related to the distances to the object. Even when taking into account the implementation details [1], it is ultimately unclear whether the controller, as presented, could be independently and correctly implemented. In our experience, this is not an uncommon scenario in the development of robot applications. We refer, for instance, to [13,14] for examples of other applications modelled with similar state machines.

### 2.2 Related Work

According to a recent survey [15], there is increasing interest in domain-specific and model-driven approaches in robotics. We discuss below those closest to ours in tackling aspects such as architectural design, time, and verification.

$\mathsf{G^{en}{}_{o}M}$ [16] provides a component-based approach for designing middleware-agnostic robotic controllers. Functional aspects are captured by recording the input and output parameters of functions together with their worst-case execution time. Implementations are provided by code fragments, for example, using C code. Verification of schedulability via model-checking is available using Fiacre [16], through the Timed Petri Net model-checker TINA, while deadlocks can be checked using BIP. $\mathsf{G^{en}{}_{o}M}$ is primarily an executable language, whereas RoboChart is a modelling language catering for different levels of abstraction.

Proof techniques, including model-checking, have also been used to identify optimal configurations of adaptive architectures [17]. Related approaches such as CIRCA [18] tackle the problem of meeting real-time constraints given dynamic plan generation. Behavioural properties are not the main focus of these works.

Orccad [19] supports modelling, simulation, and programming, as well as verification of timed behavioural properties via translation into Esterel and Timed Argos. Unlike RoboChart, its support for graphical modelling is limited, while the modelling constructs employed are closest to those of our semantics.

UML has been used for model-based engineering of robotic systems [20]. The profile RobotML [21] supports design modelling and automatic generation of platform-independent code, but verification is not considered. On the other hand, several formal models of UML state machines exist; some of them use CSP [22,23]. However, none of these deal with time modelling.

UML has a simple notion of time. Its profile UML-MARTE [24] supports logical, discrete and continuous time through the notion of clocks. Specification of time budgets and deadlines, however, is focused on particular instances of behaviour via sequence and time diagrams. It is not possible to define timed constraints directly in terms of transitions and states as we require.

UML-RT [25], an extension to UML, includes the notion of capsules, which encapsulate state machines. Communication between capsules is governed by protocols. A timing protocol can raise timeouts, but it is not obvious how timed constraints, such as deadlines, can be specified directly on state machines. In [26] a semantics is given for a subset of UML-RT without considering time. An extension to UML-RT is considered in [27] with semantics given in CSP+T [28], an extension of CSP that records the timing of events.

Timed automata [29] use synchronous continuous-time clocks. Temporal logic properties can be checked using the model checker UPPAAL [30]. It is not directly comparable to RoboChart, which provides modelling abstractions catering for robotic applications and has a semantics for refinement. It is our aim to explore a semantics for RoboChart using UPPAAL for property verification.

## 3 RoboChart: a Formal Notation for Robotics

A system in RoboChart is characterised by a module that contains a robotic platform, associated with one or more controllers. A controller is specified by one or more state-machines. Our focus here is on the state machines, since that is where we define the time properties. The untimed RoboChart semantics defined

| Primitive | Metamodel Element | Description |
|---|---|---|
| #C | ClockReset | Resets clock C. |
| since(C) | ClockExp | Time elapsed since the most recent reset of clock C. |
| sinceEntry(S) | StateClockExp | Time elapsed since state S was entered. |
| A <{d} | TimedStatement | Deadline on action A to terminate within d time units. |
| e <{d} | Transition | Deadline on event e to happen within d time units. |
| Wait(d) | Wait | Explicit time budget of $d$ time units. |

Table 1: Timed primitives of RoboChart.

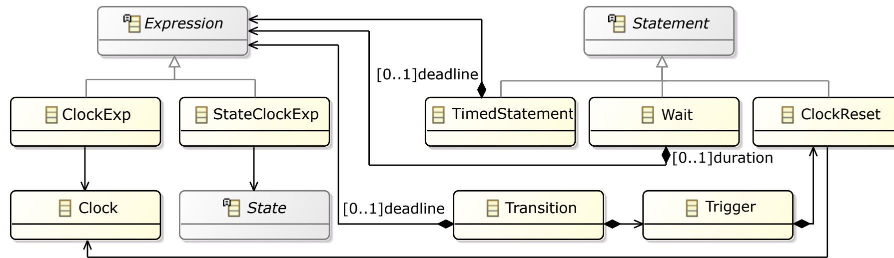


Figure 2: Timed metamodel of RoboChart.

in [31] already describes how CSP models of state machines can be composed to define models for controllers, and how these can be composed to define a complete module and provide a formal model of a robotic system.

A state-machine includes states and composite states with entry, during and exit actions, junctions, and transitions, possibly guarded by expressions. The language for actions is well defined to include assignments, operation calls, and a primitive to raise events. In Figure 2 we include part of the RoboChart metamodel showing constructs related to time, whose syntax is summarized in Table 1. The RoboChart Reference Manual [31] gives a complete description.

We have a notion of Clock (see Figure 2) that allows transitions to be guarded by time expressions that define constraints relative to the occurrence of other events via the since(C) (ClockExp in Figure 2) and #C (ClockReset) primitives, and relative to activation of a state via sinceEntry(S) (StateClockExp). We also have primitives to impose a deadline $d$ on action $A$ (A <{d}) (TimedStatement), or transition trigger $e$ (e <{d}) (Trigger), and to specify a budget $d$ (Wait(d)) (Wait) for an operation, where $d$ is an Expression.

Similarly to timed automata, expressions involving clocks are restricted to comparing single timed primitives with constant expressions. We, however, allow conjunctive as well as disjunctive expressions involving more than one clock.

To illustrate the RoboChart notation we consider a robot that moves at constant speed in a square pattern while avoiding obstacles. The state machine is shown in Figure 3, where the annotations T0 to T6 uniquely identifying the transitions are not actually part of RoboChart, but are included to guide the later discussion of the semantics in Section 4.

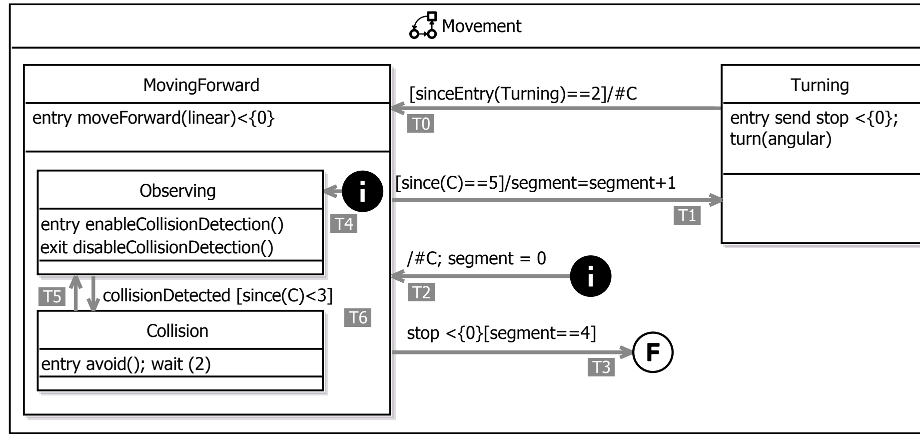


Figure 3: Example of a square trajectory state machine controller.

When the robot is started, it transitions from the initial state, denoted by a black circle, to the state MovingForward, while resetting (#C) a clock C and assigning 0 to the local variable segment. The local declarations are elided in Figure 3, but a RoboChart state machine is self-contained, in that it declares all the variables, events, and operations that it uses. The local variable segment records how many sides of the square have been covered so far; the robot stops when it completes the square (segment == 4). This is achieved by sending an event stop to the platform and transitioning to the final state: a white circle. The event stop is given a deadline 0, indicating that it is expected that the robotic platform is always ready to accept this event immediately.

In the composite state MovingForward, the motion is linear, unless an obstacle is detected. Linear motion is activated by calling the operation moveForward in the entry action with a constant value linear passed as a parameter. This operation is annotated with a deadline of 0, since moveForward can typically be implemented just as an assignment to a variable whose duration is regarded as negligible. Operations may be specified by other state machines or have their implementation provided by the robotic platform.

Before MovingForward is actually entered, its entry action executes, followed by that of its substate Observing, enabling the collision detection capability. Once a collision is detected, the event collisionDetected is raised by the robotic platform: the transition from Observing to the state Collision is then triggered, but only if there is enough time (since(C)<3) before the next turn, executing the exit action of Observing and subsequently the avoid operation that performs the actual collision avoidance. Here we do not specify this operation, but record its budget of 2 time units by sequentially composing it with the timed primitive wait(2). In RoboChart time elapses explicitly via budgets, unless a state has been entered and no transitions are enabled, or, every enabled transition is associated with an external event. Once the collision is resolved, a transition back to Observing is taken. Transitions are triggered once the guard is true and the associated event is raised, or, if there is no event associated, immediately.

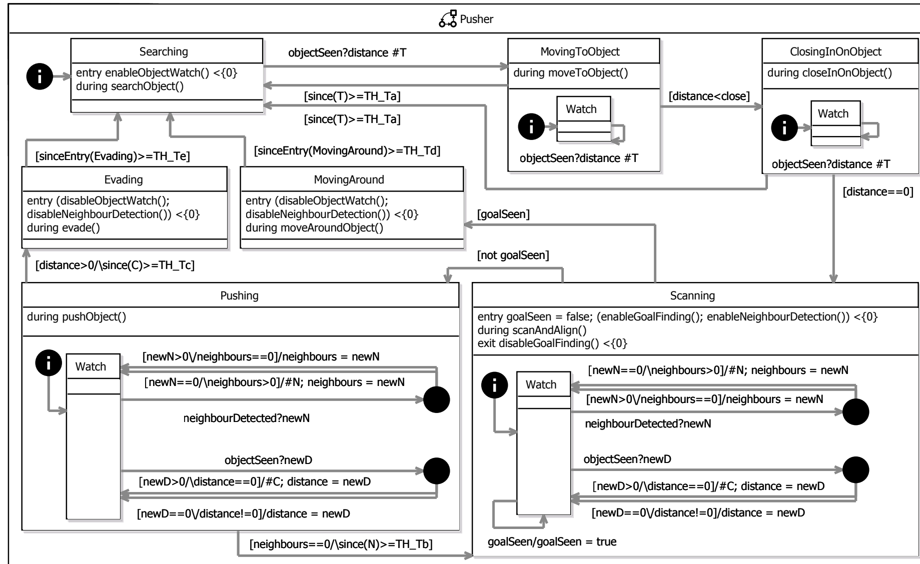


Figure 4: RoboChart model of the transport swarm state machine.

The square motion pattern is achieved by limiting the linear motion to 5 time units before switching to angular motion for 2 time units, and then switching again to linear motion. Accordingly, we guard the transition from MovingForward to the state Turning with the expression since(C) == 5. Upon such a transition, the value of segment is incremented. Similarly, the angular motion is limited by guarding the transition from Turning to MovingForward using the timed primitive sinceEntry(Turning). Upon this transition, the clock is reset.

In Figure 4, we also show the RoboChart model for the transport swarm controller described in Section 1. We assume that the robotic platform can raise events: objectSeen, with a distance value passed as a parameter in response to seeing an object at an estimated distance; goalSeen in response to detecting the goal; and neighbourDetected, with a number of neighbours passed as a parameter. We also assume that the controller needs to enable the platform to receive those events, by calling appropriate operations, such as enableObjectWatch.

Operations likely to be implemented as assignments to variables have been annotated with zero deadlines. Overall we have the same structure as the original specification [1], with the same number of states, but with additional substates. This stems from interactions that are not clear in the original model, such as the need to keep counting neighbours while in states Pushing and Scanning, and the need to keep track of the object across multiple states.

The existing semantics of RoboChart deals with the structure (modules, controllers, and parallel state machines) of models. That semantics defines the visible behaviour of a module: the order and availability of the events of the platform. That semantics, however, ignores all time constructs of a model: clocks, and associated statements, waits, and deadlines. We address them in the next section.

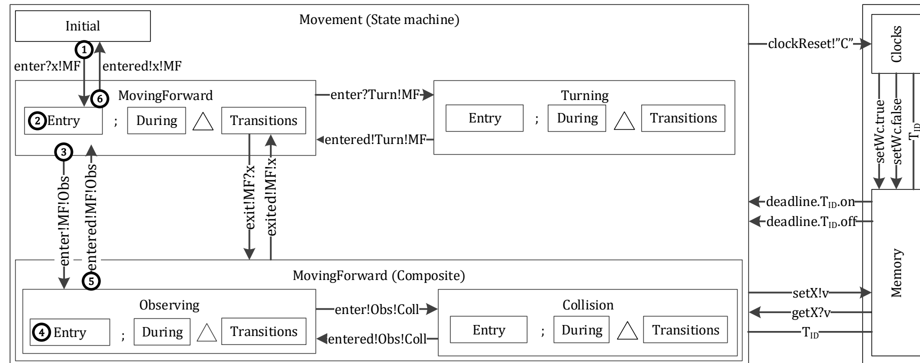


Figure 5: Semantics Architecture based on Example of Figure 3.

# 4 Semantics

Here, we describe the semantics of RoboChart state machines (Section 4.1) [31]. We then focus on the semantics of each timed RoboChart construct, namely budgets and deadlines (Section 4.2) and clocks (Section 4.3).

Before defining the semantics, we first introduce the required CSP syntax. A communication on event $e$ (also known as a channel), optionally parametrised by $x$, is defined as $e.x \to P$, with $e?x$ being syntactic sugar for allowing $x$ to range over the type of $e$ and introducing $x$ in the scope of $P$, and with $e!v$ being used for a specific value $v$. Processes can be composed in parallel ($P \lbrack\!\lbrack s \rbrack\!\rbrack Q$), where $s$ is the set of events on which $P$ and $Q$ require agreement, and if $s$ is empty this is an interleaving ($P \mid\mid\mid Q$). An external choice $P \sqcap Q$ offers an initial choice between behaving as $P$ or $Q$, while $P \bigtriangleup Q$ behaves as $P$ but can be interrupted by $Q$ at any time, with the timed version $P \bigtriangleup_d Q$ in addition also interrupting $P$ exactly at $d$ time units. $P \mathit{\Theta}_A Q$ initially behaves as $P$ but can be interrupted by an event in $A$ to behave as $Q$. Sequential composition of $P$ and $Q$ is $P \,;\, Q$, with $\mathit{SKIP}$ being the unit. Hiding ($P \setminus h$) makes the events in set $h$ internal to $P$. Finally, the events in a process $P[f]$ can be renamed according to function $f$.

## 4.1 State Machines

A state machine is given a CSP semantics as the parallel composition of a process *States*, itself the parallel composition of processes that model a state, with a process *Initial*, that models the transition from the initial state. In Figure 5 we illustrate the architecture of the CSP semantics of the example from Figure 3. A state is modelled by a process *Entry*, modelling its entry action, sequentially composed with *During*, a model for its during action, that can be interrupted by a process *Transitions* that models the possible outgoing transitions.

A state machine defines a sequential and hierarchical control flow. To model this flow, there are *enter*, *entered*, *exit*, and *exited* events that model state activation and deactivation, with the associated entry and exit actions. Each event takes two parameters: the state that requested the activation or deactivation to start, and the target state of the request.

A state is modelled in a compositional way, capturing information only about itself, irrespective of whether it is inside a state machine or another state. In Figure 5, the execution sequence is numbered. For example, the process modelling MovingForward offers events $enter?x!\mathsf{MF}$ for any other state $x$, including the initial state to request it to enter, followed by the process that models its own entry action, a request on $enter!\mathsf{MF}!\mathsf{Obs}$ for the child Observing to enter, the entry action of Observing, and the acknowledgements $entered!\mathsf{MF}!\mathsf{Obs}$ and $entered!x!\mathsf{MF}$. The process then offers an external choice of events that trigger its transitions.

Following a transition event, the *exit* and *exited* events to request and acknowledge deactivation are offered. For instance, in our example, following a transition triggered from state MovingForward, the process offers to synchronize on events $exit!\mathsf{MF}.S$, where $S$ ranges over all state identifiers except MF itself, as a way of requiring deactivation of either Observing or Collision.

Each state transition $T$ is modelled by a process that synchronizes on $T_{ID}$, an event that uniquely identifies the transition in the state machine. If an event trigger $e$ is associated with the transition, then at the outer level we rename the complete state machine process by mapping $T_{ID}$ to $e$.

Variables declared in a state machine are modelled using a process *Memory* that exposes events *get* and *set* for each variable. In our example, *Memory* is parametrised by $s$, which holds the value of the variable segment, and offers the events *getSegment* and *setSegment* in an external choice followed by a recursion.

$$Memory(s) = \begin{pmatrix} getSegment!s \rightarrow Memory(s) \;\Box\; setSegment?y \rightarrow Memory(y) \\ \Box\; s == 4 \;\&\; T3 \rightarrow Memory(s) \end{pmatrix}$$

Moreover, it also models transition guards by constraining synchronization on transition events ($T_{ID}$). In our example, the transition from MovingForward to the final state is guarded, so *Memory* captures this guard by only offering the event $T3$ that uniquely identifies the transition (Figure 3) when *segment* is 4.

### 4.2 Budgets and Deadlines

As mentioned before, RoboChart budgets can be specified as part of actions. using the wait(d) construct. Its semantics is given by **Wait** $t$, a Timed CSP process that terminates exactly after $t$ units of time elapse. Deadlines specified on actions are defined using the deadline operator $A \blacktriangleright t$ of *Circus Time*, where the process $A$ modelling action A must terminate within $t$ time units.

When a deadline is imposed on a transition trigger, however, it must be enforced only when the transition is enabled, that is, the transition's guard is true and the source state has been entered. In our model, we define a pair of events $deadline.T_{ID}.on$ and $deadline.T_{ID}.off$ for each transition T whose trigger has a deadline. Whenever T's guard is true, the *Memory* process offers the event $deadline.T_{ID}.on$, and when the guard is false, it offers $deadline.T_{ID}.off$. The *Memory* process of our example is defined as follows.

$$Memory(s) = \begin{pmatrix} ... \;\Box\; s == 4 \;\&\; deadline.T3.on \rightarrow Memory(s) \\ \Box\; \neg(s == 4) \;\&\; deadline.T3.off \rightarrow Memory(s) \end{pmatrix}$$

In addition to the *get* and *set* events for setting and getting the value of variable *segment*, and the guarded synchronization on $T3$, the event $deadline.T3.on$ is guarded by the expression corresponding to the guard on the transition identified by T3, and the negation of this expression guards the event $deadline.T3.off$.

For each process that models a state where an outgoing transition has a trigger with a deadline, we then compose in interleaving with the process modelling its during action, a $Dline_i$ process for each deadline $d_i$ as defined below.

$$Dline_i = deadline.T_{ID}.on \to ((deadline.T_{ID}.off \to SKIP) \blacktriangleright d_i) \ ; \ Dline_i$$

$Dline_i$ initially synchronizes on $deadline.T_{ID}.on$, and thereafter must synchronize on $deadline.T_{ID}.off$ within $d_i$ time units, followed by a recursion. The deadline is imposed on $deadline.T_{ID}.off$ rather than the transition identifier $T_{ID}$. The deadline can be satisfied either as a result of the transition's guard no longer being true, in which case the process synchronizes on $deadline.T_{ID}.off$, or as a result of the process being interrupted due to some transition out of the source state of T, modelled by a process *Transitions*, being triggered, possibly T itself. Effectively an enabled deadline on a transition becomes a deadline on the external choice between all enabled transitions out of the same state.

As an example, we show the process $M$ for the state MovingForward.

$$M = enter?S!\mathsf{MF} \to \begin{pmatrix} moveForward \ ; \ enter!\mathsf{MF}!\mathsf{Obs} \to \\ entered!\mathsf{MF}!\mathsf{Obs} \to entered!S!\mathsf{MF} \to SKIP \ ; \\ ((SKIP \ ||| \ Dline_{MF}) \bigtriangleup Transitions_{MF}) \end{pmatrix} \ ; \ M$$

Initially it offers events $enter?S!\mathsf{MF}$, so that any other state identified by $S$ may request it to be entered. It then behaves as *moveForward*, the process that models the operation moveForward, and then requests the substate Observation to *enter* by synchronising on $enter!\mathsf{MF}!\mathsf{Obs}$, subsequently waiting for an acknowledgement via $entered!\mathsf{MF}!\mathsf{Obs}$ and then acknowledging its own entry through $entered!S!\mathsf{MF}$. $M$ then behaves as an interleaving ($|||$) between the process modelling its during action, in this case *SKIP* as there is none, and the process $Dline_{MF}$ that models the deadlines on triggers of every outgoing transition of state MovingForward, while offering for any event in $Transitions_{MF}$, the process that models every outgoing transition from this state, to interrupt the interleaving.

### 4.3 Clocks

As previously mentioned, RoboChart clocks allow conditions to be set relative to the time elapsed since a particular clock reset. To model a reset #C on clock C we introduce an event $clockReset.C$, where $C$ is the name of the clock.

Although clocks could be explicitly modelled in the semantics, for example, by adding variables in the *Memory* process for each clock, this would make the model intractable for model-checking as the variables would have an unbounded domain. Since we assume clocks can only be compared with constant expressions, we adopt a model where a timed expression involving a comparison between a constant and constructs since(C) or sinceEntry(S) is encoded by a boolean

variable together with an auxiliary CSP process synchronizing with the *Memory* process. For example, a transition with unique identifier T1 guarded by the expression $\mathsf{x} = 1 \vee \mathsf{since(C)}>= \mathsf{d}$ is encoded in the *Memory* process as follows.

$$Memory(..., x, wc_{T1}) = \begin{pmatrix} ... \ \Box \ setWc_{T1}?wc \rightarrow Memory(..., x, wc) \\ \Box \ (x = 1 \vee wc_{T1}) \ \& \ T1 \rightarrow Memory(..., x, wc_{T1}) \end{pmatrix}$$

A boolean variable $wc_{T1}$ encodes the timed condition since(C)>=d, with channel $setWc_{T1}$ used to set it *true* or *false*. Synchronizing in parallel with the *Memory* process we introduce a *WaitingCondition* process $WC_T1$ defined below.

$$\begin{aligned} &WC_T1 = Do(T1) \ \triangle \ WC_T1_reset \\ &WC_T1_reset = clockReset.\mathsf{C} \rightarrow setWc_{T1}!false \rightarrow WC_T1_body \\ &WC_T1_body = (Do(T1) \ \triangle_d \ setWc_{T1}!true \rightarrow Do(T1)) \ \triangle \ WC_T1_reset \end{aligned}$$

This process ensures that while $wc_{T1}$ is being updated the event $T1$ is not offered. Initially it is ready to synchronise on $T1$ indefinitely (as defined using the process $Do(e) = e \rightarrow Do(e)$), but can be interrupted by the event *clockReset*.C offered in the process $WC_T1_reset$. Whether $T1$ is actually enabled or not is controlled by *Memory* and not $WC_T1$. So, the availability of $T1$ in $WC_T1$ indicates only that $wc_{T1}$ is not being updated. If there is a clock reset, $WC_T1_reset$ sets the value of the *Memory* process variable $wc_{T1}$ to *false* via the synchronization $setWc_{T1}!false$ and behaves as $WC_T1_body$. This ensures that, when the clock is reset, the transition cannot take place, even if the value of the condition is not yet updated. Initially this process continuously offers the event $T1$ until exactly $d$ units elapse ($\triangle_d$), after which it sets $wc_{T1}$ to *true* via the synchronization $setWc_{T1}!true$ and then continuously offers the event $T1$. At any point the process may be interrupted by $WC_T1_reset$ due to a *clockReset*.C.

The complete semantics of a timed state machine is given by the parallel composition of the process modelling the state machine, $STM$, the *Memory* process and a *Clocks* process whose definition is the parallel composition of all *WaitingCondition* processes as defined for each timed condition.

$$(((STM \ [\![ \ g \cup dc \ ]\!] \ ((Memory \ [\![ \ w \cup t \ ]\!] \ Clocks) \setminus w)) \setminus I)[f]) \ \Theta_{\{term\}} \ SKIP$$

*Memory* and *Clocks* synchronise on the events in the sets $w$, containing all *setWc* events, which are then subsequently hidden (\). They also synchronise on the events of the set $t$ of identifiers for transitions whose conditions are timed. This parallel process synchronises with $STM$ on the events from $g$, containing the *get* and *set* events for reading and writing the value of state variables and the transition identifiers, and from $dc$, containing the *deadline* and *clockReset* events. This is illustrated by the lines on the top right corner of Figure 5. The set of identifiers for internal transitions ($I$) are hidden (\). Also, as explained, we use a function $f$ to rename transition identifiers to external events of the platform. Finally if the state machine has a final state, the process $STM$ can signal termination via the event *term*, which interrupts the process to behave as $SKIP$.

Our RoboTool presented next automatically calculates the timed semantics of a RoboChart model just described. Instead of Timed CSP, it uses *tock*-CSP for direct use of FDR. The time constructs are encoded as described in [4].

## 5 Tool Support and Model-Checking

To provide support for designing robotic systems using RoboChart, we have developed RoboTool[1], an Eclipse plugin that allows specifications to be input using both graphical and textual editors, implemented using the Sirius and Xtext[2] frameworks. RoboTool automatically generates the semantics of RoboChart models in $\texttt{CSP}_\texttt{M}$, the machine readable version of CSP used by FDR [9].

FDR includes facilities to translate untimed processes into *tock*-CSP. For example, the prefixing $a \to P$ is translated into an external choice offering *tock*, the event that marks the passage of time, in addition to $a$: $X = a \to P \,\Box\, tock \to X$. Other operators are similarly accommodated, while more intricate concepts need to be manually specified using *tock*-CSP. For example, deadlines are encoded by timelocking once a deadline expires, that is, by refusing *tock*.

Using the timed semantics of RoboChart we can perform a number of core checks using FDR, namely, determinism and divergence freedom. In addition, for a given *tock*-CSP process $STM_T$ modelling a state machine, and whose set of externally observable events is $E$, we can establish that there are no timelocks provided the following refinement is satisfied [7]. Since in our model unmet deadlines lead to timelocks this is a useful check to identify infeasible deadlines.

$$RUN(\{tock\}) \mathbin{|||} CHAOS(E) \sqsubseteq_F STM_T \upharpoonright (E \cup \{tock\})$$

With the above we require that $STM_T$, with every event other than those in $E$ and *tock* hidden (using the projection operator $\upharpoonright$), is a refinement ($\sqsubseteq_F$) in the failures model of the process $RUN(\{tock\})$, that is always offering *tock*, in interleaving ($|||$) with the process $CHAOS(E)$ that can perform any event in the set $E$ nondeterministically. Zeno freedom, that is, the absence of a behaviour where an infinite sequence of events is performed in finite time, can be ascertained by checking that $STM_T \upharpoonright (E \cup \{tock\})$ is divergence free. Assertions to establish all these core properties are also automatically generated by RoboTool.

Using our semantics we have considered several case studies. We have verified core properties and also defined requirements directly in CSP and *tock*-CSP. A complete account of the experiments can be found in [32].

Table 2 summarises the results of checking for divergence freedom, a particularly expensive check in FDR, including state-space complexity ($S/T$) in terms of number of states ($S$) and transitions ($T$) visited, compilation time ($C_T$) and verification time ($V_T$). We also include the experimental results obtained with the untimed models, defined without using *tock*, for comparison. Results were obtained using FDR version 4.2.0 on a computer with 16GiB of RAM and an Intel i5-5287U CPU. Times correspond to an average of 5 runs. For the purpose of verification, in examples $E2$, $E3$ and $E4$ the types for reals and integers are instantiated in $\texttt{CSP}_\texttt{M}$ as ranging from 0 to 1, whereas in $E1$ reals are instantiated within the range from $-90$ to 180 due to the specification using such values.

[1] https://www.cs.york.ac.uk/circus/RoboCalc
[2] `www.eclipse.org/sirius` and `www.eclipse.org/xtext`

| Examples | Untimed | | | Timed | | |
|---|---|---|---|---|---|---|
| | S/T | $C_T$ | $V_T$ | S/T | $C_T$ | $V_T$ |
| E1. Chemical Detector | 80/265 | 0.23s | 2.3s | 240/861 | 0.15s | 4.58s |
| E2. Autonomous Chemical Detector | 5/112 | 2.03s | 0.65s | 6/72 | 1.82s | 1.99s |
| E3. Alpha Algorithm | 52/184 | 0.26s | 1.28s | 12045/30918 | 0.66s | 1.30s |
| E4. Transport Swarm | 8/28 | 1.12s | 0.56s | 436/1085 | 2.49s | 0.17s |

Table 2: Verification results of checking divergence freedom with FDR.

Our results show that assertions in the failures-divergences model can typically be checked within a few seconds. Diligent application of compression functions significantly reduces the time required to compile and verify the assertions. We use *diamond*, which removes silent transitions from the LTS, and *wbisim*, that reduces the LTS by computing the maximal weak bisimulation.

To cope with additional variables in the *Memory* process, typically as the result of modelling timed conditions, we have optimized this process. Each variable is captured in separate, but parallel, "cell" processes, that synchronize with an auxiliary non-parametrised process, modelling the transitions' conditions, such that whenever a variable is changed it introduces in scope the current value of all variables. This yields a reduction in the number of possible states. The efficiency gain is particularly noticeable when a state machine has several variables, or timed conditions, which we have also optimized by generating equivalent timed expressions only once as a Waiting Condition CSP process.

As expected, the usage of *tock* increases the state-space complexity of examples compared to their untimed counterparts. The exception here is $E2$, likely due to *wbisim* that can yield better compression than *diamond* in some cases. We observe that *diamond* is not permitted by FDR within timed processes.

## 6 Conclusion

RoboChart can be viewed as a UML profile extended with timed primitives and a formal semantics. We have used constructs from *Circus Time* to capture budgets and deadlines in a timed semantics for refinement and model checking. Support for refinement is essential to our future plans to prove soundness of automatically generated simulation and deployment code.

To optimise model checking, clocks are modelled implicitly, with timed conditions modelled explicitly. Our use of clocks makes a translation into UPPAL feasible, and of interest for further analysis. For example, we have considered UPPAAL models of the transport swarm, including a model based on the architecture of our semantics and a simplified version. Both require additional states and transitions when compared to RoboChart to achieve a faithful model.

A semantic model generator has been implemented in RoboTool via translation into *tock*-CSP [7]. We have tackled several examples and verified whether the generated models satisfy expected system requirements, in addition to core properties like divergence freedom and zeno freedom. Results suggest an increase in complexity, but not necessarily in verification time, when compared to the verification of untimed models. The verifications are tractable given modest

data types and diligent use of FDR's compression functions. For realistic data types we do not expect scalability, instead we will consider theorem proving.

We have a precise account of the timed semantics of RoboChart embedded in RoboTool. We will capture this semantics via translation functions that generate *Circus Time* models suitable for use in Isabelle/UTP [12], which supports reasoning about the *Circus* family of languages via theorem proving. Furthermore, to account for the environment and probabilistic behaviour we will ultimately consider richer semantics models in the context of the UTP.

**Acknowledgments** This work is funded by EPSRC grant EP/M025756/1. No new primary data was created during this study.

## References

1. Chen, J., Gauci, M., Gross, R.: A strategy for transporting tall objects with a swarm of miniature mobile robots. In: Robotics and Automation (ICRA), 2013 IEEE International Conference on. (May 2013) 863–869
2. Liu, W., Winfield, A.F.T.: Modeling and Optimization of Adaptive Foraging in Swarm Robotic Systems. The International Journal of Robotics Research **29**(14) (2010) 1743–1760
3. Li, W., Miyazawa, A., Ribeiro, P., Cavalcanti, A.L.C., Woodcock, J.C.P., Timmis, J.: From formalised state machines to implementation of robotic controllers. In: DARS 2016, Springer-Verlag (2016)
4. Schneider, S.: Concurrent and real-time systems: the CSP approach. Worldwide series in computer science. John Wiley (2000)
5. Sherif, A., Cavalcanti, A.L.C., He, J., Sampaio, A.C.A.: A process algebraic framework for specification and validation of real-time systems. Formal Aspects of Computing **22**(2) (2010) 153–191
6. Woodcock, J.C.P., Davies, J.: Using Z—Specification, Refinement, and Proof. Prentice-Hall (1996)
7. Roscoe, A.W.: Understanding Concurrent Systems. Texts in Computer Science. Springer (2011)
8. Woodcock, J.C.P.: The miracle of reactive programming. In: Unifying Theories of Programming. LNCS, Springer-Verlag (2009) 202–217
9. Thomas Gibson-Robinson, Philip Armstrong, A.B.A.R.: FDR3 — A Modern Refinement Checker for CSP. In Ábrahám, E., Havelund, K., eds.: Tools and Algorithms for the Construction and Analysis of Systems. Volume 8413 of Lecture Notes in Computer Science. (2014) 187–201
10. Hilder, J.A., Owens, N.D.L., Neal, M.J., Hickey, P.J., Cairns, S.N., Kilgour, D.P.A., Timmis, J., Tyrrell, A.M.: Chemical Detection Using the Receptor Density Algorithm. IEEE Trans. Syst. Man Cybern. C, Appl. Rev. **42**(6) (2012) 1730–1741
11. Dixon, C., Winfield, A.F.T., Fisher, M., Zeng, C.: Towards temporal verification of swarm robotic systems. Robot. Auton. Syst. **60**(11) (2012) 1429–1441
12. Foster, S., Zeyda, F., Woodcock, J.: Isabelle/UTP: A Mechanised Theory Engineering Framework. In: UTP 2015. Volume 8963 of LNCS. Springer (2015) 21–41
13. Nouyan, S., Gross, R., Bonani, M., Mondada, F., Dorigo, M.: Teamwork in Self-Organized Robot Colonies. IEEE Transactions on Evolutionary Computation **13**(4) (Aug 2009) 695–711

14. Pini, G., Brutschy, A., Scheidler, A., Dorigo, M., Birattari, M.: Task partitioning in a robot swarm: Object retrieval as a sequence of subtasks with direct object transfer. Artificial life **20**(3) (2014) 291–317
15. Nordmann, A., Hochgeschwender, N., Wigand, D., Wrede, S.: A survey on domain-specific modeling and languages in Robotics. J Softw Eng Robot **7**(1) (2016) 75–99
16. Foughali, M., Berthomieu, B., Zilio, S.D., Ingrand, F., Mallet, A.: Model Checking Real-Time Properties on the Functional Layer of Autonomous Robots. In: Formal Methods and Soft. Eng., Springer (2016) 383–399
17. Fleurey, F., Solberg, A.: A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In: Int. Conf. on Model Driven Eng. Languages and Systems, Springer-Verlag (2009) 606–621
18. Musliner, D.J., Durfee, E.H., Shin, K.G.: CIRCA: A cooperative intelligent real-time control architecture. IEEE Transactions on Systems, Man, and Cybernetics **23**(6) (1993) 1561–1574
19. Espiau, B., Kapellos, K., Jourdan, M.: Formal verification in robotics: Why and how? In: Robotics Research, Springer London (1996) 225–236
20. Schlegel, C., Hassler, T., Lotz, A., Steck, A.: Robotic software systems: From code-driven to model-driven designs. In: Advanced Robotics, 2009. ICAR 2009. International Conference on. (June 2009) 1–8
21. Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., Ziane, M. In: RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications. Springer Berlin Heidelberg, Berlin, Heidelberg (2012) 149–160
22. Rasch, H., Wehrheim, H.: Checking consistency in UML diagrams: Classes and state machines. In: Formal Methods for Open Object-Based Distributed Systems. Volume 2884 of LNCS. Springer (2003) 229–243
23. Davies, J., Crichton, C.: Concurrency and Refinement in the Unified Modeling Language. Formal Aspects of Computing **15**(2-3) (2003) 118–145
24. Selic, B., Grard, S.: Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems. Morgan Kaufmann Publishers Inc. (2013)
25. Selic, B.: Using UML for modeling complex real-time systems. In Mueller, F., Bestavros, A., eds.: Languages, Compilers, and Tools for Embedded Systems. Volume 1474 of LNCS., Springer (1998) 250–260
26. Ramos, R., Sampaio, A.C.A., Mota, A.C.: A Semantics for UML-RT Active Classes via Mapping into *Circus*. In: Formal Methods for Open Object-based Distributed Systems. Volume 3535 of LNCS. (2005) 99–114
27. Akhlaki, K.B., Tunon, M.I.C., Terriza, J.A.H., Morales, L.E.M.: A methodological approach to the formal specification of real-time systems by transformation of UML-RT design models. Science of Computer Programming **65**(1) (2007) 41–56
28. Zic, J.J.: Time-constrained Buffer Specifications in CSP + T and Timed CSP. ACM Trans. on Programming Languages and Systems **16**(6) (1994) 1661–1674
29. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2) (1994) 183–235
30. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In: Proc. of Workshop on Verification and Control of Hybrid Systems III. Number 1066 in LNCS, Springer (October 1995) 232–243
31. Miyazawa, A., et al.: RoboChart Reference Manual. Technical report, University of York (2017) `http://bit.ly/2plUry4`.
32. RoboCalc Project: RoboChart Case Studies (2017) `www.cs.york.ac.uk/circus/RoboCalc/case-studies/`.