Analysing RoboChart with Probabilities

M. S. Conserva Filho[1], R. Marinho[1], A. Mota[1], and J. Woodcock[2]

[1] Centro de Informática, Universidade Federal de Pernambuco, Brazil
{mscf,rma6,acm}@cin.ufpe.br
[2] Department of Computer Science, University of York, UK
jim.woodcock@york.ac.uk

Abstract. Robotic systems have applications in many real-life scenarios, ranging from household cleaning to critical operations. RoboChart is a graphical language for describing robotic controllers designed specifically for autonomous and mobile robots, providing architectural constructs to identify the requirements for a robotic platform. It also provides a formal semantics in CSP. RoboChart has a probabilistic operator ($\textcircled{P}$) but no associated probabilistic CSP semantics. When $\textcircled{P}$ is used, currently a non-deterministic choice ($\sqcap$) is used as semantics; this is a conservative semantics but it does not allow the analysis of stochastic properties. In this paper we define the semantics of the operator $\textcircled{P}$ in terms of the probabilistic CSP operator $\boxplus$. We also show how this augmented CSP semantics for RoboChart can be translated into the PRISM probabilistic language to be able to check stochastic properties.

Keywords: Robotic systems, CSP, probabilistic analysis, PRISM

1 Introduction

Robotic systems have been used in many real-life scenarios, ranging from simple domestic assistants [18] (household cleaning) to safety-critical activities, such as driverless cars [4] and pilotless aircraft [27]. Despite their complexity, the current practice for implementing such robots applications is performed in an *ad-hoc* manner. These practices are often based on standard state machines, without formal semantics, to describe the robot controller only.

In [21], a domain-specific modelling language, called RoboChart, based on UML, is proposed. It is a graphical language for describing robotic controllers, specifically designed for autonomous and mobile robots. It provides architectural constructs to identify the requirements for a robotic platform. Features of the RoboChart graphical notation allow, for instance, the behavioural description of timed, continuous, and probabilistic properties.

Concerning formal verification, RoboChart has a formal semantics in CSP [23] that can be automatically calculated by RoboTool[3], a tool that supports the use of RoboChart. CSP is a well established process algebra to model and verify concurrent systems. It defines the behaviours of a system in terms of events and the

[3] www.cs.york.ac.uk/circus/RoboCalc/robotool

interactions of processes. CSP has support for model checking with FDR [25], which provides a high degree of automation for early validation. Using FDR, we can, for instance, establish determinism, and absence of deadlock and divergence.

Besides functional aspects, RoboChart has also a probabilistic operator ($\textcircled{P}$) that can be used to express stochastic behaviours. Currently, however, it has no associated probabilistic CSP semantics. When it is present, for instance, between processes P and Q, a non-deterministic choice ($P \sqcap Q$) is used as semantics.

The work reported in [8] presents a version of FDR supporting the probabilistic operator $\boxplus$. But instead of modifying the FDR algorithm itself to perform probabilistic analysis, the work [8] adds a new algorithm that creates a PRISM specification [14] from a probabilistic CSP specification. This translation was named *WatchDog Transformation*. PRISM is a probabilistic language and model checking tool (both have the same name) which has already been successfully deployed in a wide range of application domains, such as real-time communication protocols, robots applications, and biological signalling pathways.

In this paper we define the semantics of the RoboChart probabilistic operator $\textcircled{P}$ in terms of the CSP probabilistic operator $\boxplus$, preserving all the original CSP semantics of RoboChart. To check for probabilistic properties, we specify a CSP property specification; this is different from the usual way of handling probabilistic model checking, which is based on a temporal logic language to express properties. We then reuse the *WatchDog Transformation* provided by [8], which combines the two CSP processes (property and process under analysis), yielding a PRISM specification. With this specification, we just have to check for a specific probabilistic temporal logic formula using the PRISM tool.

The remaining of this paper is structured as follows. The next section provides an overview of RoboChart, and its CSP semantics. Section 3 briefly presents PRISM. The translation from CSP to PRISM is discussed in Section 4. Section 5 presents our proposed strategy, and case studies are described in Section 6. Finally, we draw our conclusions, and discuss future work in Section 7.

2 RoboChart

RoboChart [21] is a UML-like notation designed for modelling autonomous and mobile robots. It provides constructs for capturing the architectural patterns of typical timed and reactive robotic systems, and probabilistic primitives as well. As opposed to other approaches for describing robotic systems, RoboChart has a formal semantics that can be automatically calculated.

We give an overview of RoboChart using a toy model, as illustrated in Figure 1. A robotic system is defined in RoboChart by a module. In our example, it is called CFootBot, and specifies a robot that can move around and detect obstacles. A module contains a robotic platform and one or more controllers that run on this platform. The robotic platform FootBot defines the interface of the system with its environment, via variables, operations, and events. In our example, the operation move(lv,av) captures the movement of the robot with linear speed lv and angular speed av. The event obstacle occurs when the robot gets close to any object in its environment; it is an abstraction of a sensor that detects obstacles. There may be one or more controllers, interacting with the platform via asynchronous events, and between them via synchronous or asynchronous events. Our example has just a single controller Movement. The behaviour of a controller is defined by one or more state machines, specifying threads of execution. Here, the behaviour of Movement is defined by the machine SMovement.

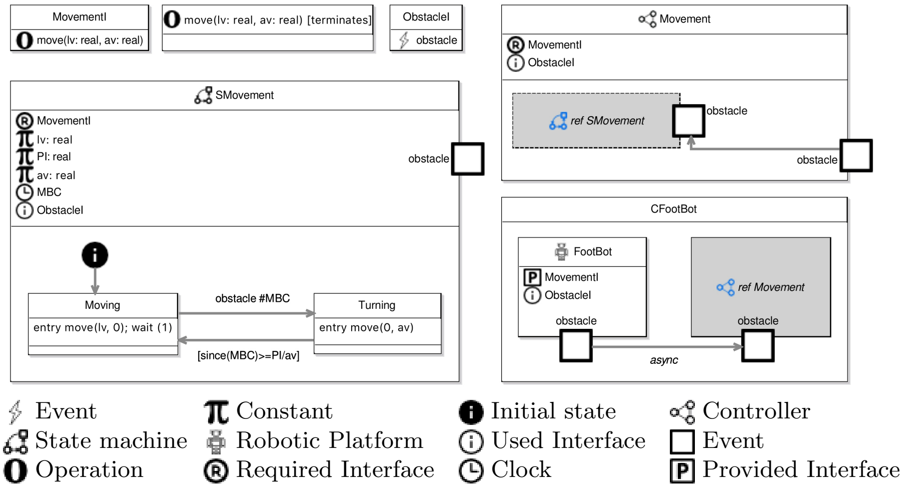


Fig. 1. RoboChart: obstacle detection

Interfaces can group variables, operations, and events. In Figure 1, the interface MovementI has only the operation move(lv,av), provided by the robotic platform, and required by the controller. ObstacleI has just the event obstacle, which is used in the platform, the controller, and the state machine. In general, different events may be connected, as long as they have the same type, or no type. Types are used when an event communicates an input or output value.

A state machine is the main behavioural construct of RoboChart. It is similar to that in UML, except that they have a well defined action language. In our example, the behaviour of SMovement is as follows: upon entry in the state Moving, after calling the operation move(lv,0), the robot waits for one time unit. The operation call move(lv,0) takes no time; it can be, for example, implemented as a simple assignment to the register of an actuator. The machine, however, is blocked by wait(1) for one time unit (which is a budget for the platform to react to this operation) before it completes entry to Moving.

SMovement declares a clock MBC. In Moving, when an obstacle is detected, MBC is reset (#MBC) and the machine moves to the state Turning. There, a call move(0,av) turns the robot. A transition back to Moving is guarded by since(MBC) >= PI/av. As soon as the guard is satisfied, the transition is taken. The guard requires that the value of MBC is greater than or equal to that of

PI/av, to ensure that the robot waits enough time to turn PI degrees, before going back to Moving, and proceeding in a straight line again.

There may be one or more controllers, interacting with the platform via asynchronous events, and between them via synchronous or asynchronous events. Here, we have just a single controller CForaging. The behaviour of a controller is defined by one or more state machines, specifying threads of execution. In the example, the behaviour of CForaging is defined by the machine SForaging.

Interfaces can group variables, operations, and events. In Figure 1, the interface IForaging has the events stop, forage and flip, which are used in the platform, the controller, and the state machine. In general, different events may be connected, as long as they have the same type, or no type. Types are used when an event communicates an input or output value.

Further information regarding RoboChart can be found in [21, 22, 20].

2.1 Semantics

The semantics of RoboChart is defined using a dialect of CSP called tock-CSP [23]. It is used to describe concurrent reactive systems that are composed by interacting components, which are independent entities called processes, that can be combined using high level operators to create complex concurrent systems. In tock-CSP, a special event tock marks the discrete passage of time.

Before presenting the semantics for our example, we first introduce the required CSP syntax. The process $SKIP$ represents the terminating process, and $STOP$ represents a deadlock process. The prefixing $a \rightarrow P$ is initially able to perform only the simple event a, and behaves like process P after that. Events may also be compound. For instance, $b.n$ is composed by the channel b and the value n. The process $P \Box Q$ is an external choice between process P and Q. The process $P;\ Q$ combines the processes P and Q in sequence. The process $\mathit{if}\ b\ \mathit{then}\ P\ \mathit{else}\ Q$ behaves as P if b holds and as Q otherwise. Further information regarding CSP can be found in [23].

We present below a CSP process CFootBot that specifies the behaviour of our example in Figure 1. The formal semantics of RoboChart is implemented by a tool (RoboTool) that automatically calculates a process that is equivalent to CFootBot below.

$$CFootBot = EMoving;\ Obstacle;\ ETurning;\ wait(PI/av);\ CFootBot$$

CFootBot composes in sequence processes EMoving, Obstacle, ETurning, and wait(PI/av) followed by a recursive call. EMoving is below; it engages in the event moveCall.lv.0, which represents the operation call move(lv,0) in the entry action of the state Moving. In sequence (prefixing operator →), EMoving engages in the moveRet event that marks the return of that operation, and then behaves like the process wait(1).

$$\begin{array}{l} EMoving = moveCall.lv.0 \rightarrow moveRet \rightarrow wait(1) \\ wait(n) = \mathit{if}\ n == 0\ \mathit{then}\ SKIP\ \mathit{else}\ tock \rightarrow wait(n-1) \end{array}$$

The definition of the parameterised process $wait(n)$ is recursive; it engages in n occurrences of $tock$ to mark the passage of n time units, and after that terminates: $SKIP$. So, $wait(1)$ corresponds directly to the wait(1) primitive of RoboChart. The process $Obstacle$ defined below allows time to pass until an event $obstacle$ occurs, when it then terminates. So, the events $obstacle$ and $tock$ are offered in an external choice ($\Box$).

$$Obstacle = obstacle \rightarrow SKIP \Box tock \rightarrow Obstacle$$

Finally, the process $EntryTurning$ models the entry action of Turning.

$$EntryTurning = moveCall.0.av \rightarrow moveRet \rightarrow SKIP$$

3 PRISM

Probabilistic model checking [1] is a complementary form of model checking aiming at analyzing stochastic systems. The specification describes the behaviour of the system in terms of probabilities (or rates) in which a transition can occur.

Probabilistic model checkers can be used to analyze quantitative properties of (non-deterministic) probabilistic systems by applying rigorous mathematics-based techniques to establish the correctness of such properties. The use of probabilistic model checkers reduces the costs during the construction of a real system by verifying in advance that a specific property does not conform to what is expected about it. This is useful to redesign models.

There are some tools that specialize in probabilistic model checking. The most well-known are: PRISM [14], Storm [3], PEPA [26], and MRMC [11].

This work focuses in the syntax of the language PRISM, which can be analyzed by the PRISM tool, the Storm model checker and other probabilistic model checkers as well. The next section gives an overview of PRISM.

The PRISM Language The PRISM language [14] is a probabilistic specification language designed to model and analyze systems of several application domains, such as multimedia protocols, randomized distributed algorithms, security protocols, and many others.

The PRISM tool uses a specification language also called *PRISM*. It is an ASCII representation of a Markov chain/process, having states, guarded commands and probabilistic temporal logics such as PCTL, CSL, LTL and PCTL*.

PRISM can be used to effectively analyze probabilistic models such as Discrete-Time Markov Chains (DTMCs), Continuous-Time Markov Chains (CTMCs), Markov Decision Processes (MDPs), Probabilistic Automata (PAs), and Probabilistic Timed Automata (PTAs).

To introduce the syntax of the PRISM language, consider the simple probabilistic algorithm due to Knuth and Yao [12] for emulating a 6-sided die with a fair coin (see Figure 2) that can be found in the PRISM website[4]. The PRISM code corresponding to this algorithm can be seen in what follows.

[4] http://www.prismmodelchecker.org/tutorial/die.php

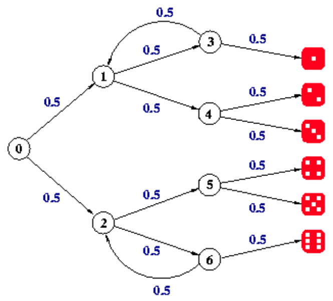


Fig. 2. Graphical illustration of the 6-sided die with a fair coin

```
dtmc
module die
  s : [0..7] init 4;
  d : [0..6] init 0;
  [] s = 0 → 0.5 : (s' = 1) + 0.5 : (s' = 2);
  [] s = 1 → 0.5 : (s' = 3) + 0.5 : (s' = 4);
  [] s = 2 → 0.5 : (s' = 5) + 0.5 : (s' = 6);
  [] s = 3 → 0.5 : (s' = 1) + 0.5 : (s' = 7)&(d' = 1);
  [] s = 4 → 0.5 : (s' = 7) & (d' = 2) + 0.5 : (s' = 7) &(d' = 3);
  [] s = 5 → 0.5 : (s' = 7) & (d' = 4) + 0.5 : (s' = 7) &(d' = 5);
  [] s = 6 → 0.5 : (s' = 2) + 0.5 : (s' = 7) & (d' = 6);
  [] s = 7 → (s' = 7);
endmodule
```

The first thing to note is the reference to the kind of Markov chain being addressed. In this example, a Discrete-Time Markov Chain (DTMC) was used.

This PRISM specification is composed of a single *module*, but if more than one module is presented an implicit parallel composition of them is considered as semantics [14]. This standard parallel composition can be customised to a new semantics by the use of a *system* . . . *endsystem* section.

Inside a module, we can have local variables such as the s and d of this example. Both are natural numbers, ranging from 0..7 and 0..6, respectively. They need an initialisation. In this case, s is initially set to 4 and d to 0.

The rest of the module's body is basically composed of a sequence of probabilistic transitions, each one starting with a choice ([]) operator. A transition has a guard (expression before the → operator), followed by the destination alternatives. The alternatives are identified by the use of + signals. Each alternative has a probability (or rate) before the colon and update rules afterwards. Each update comes inside parenthesis and the apostrophe is used to characterise the value of the variable in the next state of the system. The symbol & is used to describe logic conjunction.

Thus the transition

$$[]\ s = 0 \rightarrow 0.5 : (s' = 1) \ + \ 0.5 : (s' = 2);$$

may be read as follows: if $s = 0$ holds, then this transition is fired. At this state, we have a 50-50% alternative to update the variable s. In the first alternative, the new value of s is set to 1. Otherwise, 2. Figure 3 shows its Markov chain[5].

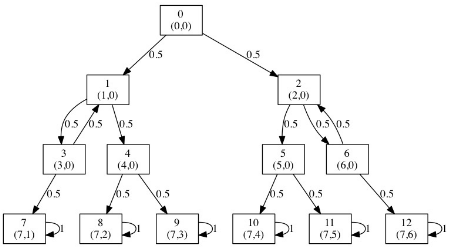


Fig. 3. Markov chain for the 6-sided die

Finally, we can perform probabilistic analysis. For this example, we can calculate the probability of getting one of its 6-sides by writing the following property (standing for "What is the chance of eventually s becomes 7 and d becomes x?")

$$\begin{array}{l} \mathit{const\ int\ x;} \\ P\ =?\ [F\ \ s = 7\ \&\ \ d = x] \end{array}$$

where the constant x refers to a specific face of the die ($x \in \{1, 2, \ldots, 6\}$). In this example, the probability for each value of x equals 16.67%.

4 WatchDog Transformation

The concurrent language CSP was extended to incorporate probabilistic and timed aspects in [16]. The probabilistic operator $\boxplus$ was defined. However, this extension was entirely theoretical; no tool support was available at that time. In the work [8], a version of FDR was extended to handle the operator $\boxplus$.

Essentially, the standard CSP$_M$ notation (the machine-readable version of CSP used by FDR) was augmented by the following new operator ($[\cdot \sim \cdot]$). Let P and Q be CSP processes. Then

$$P\ {}_{\frac{m}{(m+n)}}\boxplus_{\frac{n}{(m+n)}}\ Q\ ==\ P\,[m \sim n]\,Q$$

where m and n are natural numbers.

[5] To create such a graph, we export the Markov model in the PRISM tool and use the graphviz tool (http://www.graphviz.org/) to create Figure 3.

The work [8], however, did not change the FDR algorithm to perform probabilistic analysis. Instead, FDR was extended to create a PRISM specification from a refinement assertion. This translation is briefly described as follows.

The *WatchDog Transformation* consists in analysing the probability with which a probabilistic CSP implementation (I) refines a non-probabilistic CSP specification process (S). In this paper it is enough to briefly present the *WatchDog Transformation* concerning the traces semantics

$$S \sqsubseteq_{\mathcal{T}} I$$

It consists in mapping a specification process, say S, to a watchdog process that monitors the traces of an implementation process, say I, to indicate whether or not I refines S according to CSP's traces semantics. Precisely, a watchdog process WDT_S is defined such that it can perform a distinguished *fail_* event when I performs a trace not allowed by S.

$$\begin{aligned} WDT_S(i) = {} & (\Box\, e \in \alpha I \cap A(i) \bullet e \rightarrow WDT_S(\mathit{after}(i, e))) \\ & \Box \\ & (\Box\, e \in \alpha I \setminus A(i) \bullet e \rightarrow \mathit{fail\_} \rightarrow STOP) \end{aligned}$$

where $A(i)$ is calculated by FDR as part of the transformation.

The intention is that $WDT_S(i_0)$ can perform any trace tr of I that S can perform, but it can also perform events from the alphabet of I not allowed by S/tr (after which it can only perform the event *fail_*). Note that this definition of WDT is expressed in terms of the alphabet, αI, of the implementation process I which again must be calculated as part of the transformation. The original refinement check $S \sqsubseteq_T I$ is true precisely when $WDT_S(i_0) \parallel_{\alpha I} I$ can never perform the event *fail_*.

The previous CSP process in its LTS semantic form is translated into a PRISM specification based on a very few set of variables. The boolean variable *trace_error* matches the event *fail_*. With this, the *WatchDog Transformation* is able to calculate a probability using the following formula[6]

$$Pmax =? \, [F \; trace\_error]$$

which mathematically corresponds to the following relation

$$S \sqsubseteq_{\mathcal{T}} I \Leftrightarrow Pmax = 0\% \, [F \; trace\_error]$$

The refinement holds exactly when the maximum probability of *trace_error* becomes *true* is zero (or the event *fail_* never happens). Other interesting probabilities emerge when such a refinement can eventually fail. The interpretation of the above relation is that $Pmax =? \, [F \; trace\_error]$ gives us the degree on which the refinement $S \sqsubseteq_{\mathcal{T}} I$ may fail.

Thus, to explore the traces refinement to get useful probabilistic temporal analysis, one has to think of CSP processes as properties in such a way that the

[6] *Pmin* can be used to calculate the minimum probability as well.

traces refinement holds up to a certain point and then fail. From the temporal formula operator F, the traces refinement must resemble a reachability analysis.

By default, the *WatchDog Transformation* creates an MDP PRISM specification. But, if non-determinism may be ignored (for instance, in the 6-sided die example presented in Section 3), one can simply change the *mdp* directive to a *dtmc* one in the PRISM specification. In such a case, the PRISM tool can calculate a single probability instead of a min/max probabilistic interval.

4.1 CSP to PRISM

To illustrate that the Markov chain generated by this approach is exactly what we need, we use the same example of Section 3. That algorithm written in CSP can be described as follows.

$$
\begin{array}{l}
\mathit{channel}\ \mathit{die} : \{1..6\} \\
\mathit{SixSidedDie} = \\
\quad \mathit{let} \\
\qquad S0 = S1\ [1 \sim 1]\ S2 \\
\qquad S1 = S3\ [1 \sim 1]\ S4 \\
\qquad S2 = S5\ [1 \sim 1]\ S6 \\
\qquad S3 = S1\ [1 \sim 1]\ \mathit{DIE}(1) \\
\qquad S4 = \mathit{DIE}(2)\ [1 \sim 1]\ \mathit{DIE}(3) \\
\qquad S5 = \mathit{DIE}(4)\ [1 \sim 1]\ \mathit{DIE}(5) \\
\qquad S6 = S2\ [1 \sim 1]\ \mathit{DIE}(6) \\
\qquad \mathit{DIE}(x) = \mathit{die}.x \to \mathit{DIE}(x) \\
\quad \mathit{within}\ S0
\end{array}
$$

To obtain the exact Markov chain as depicted in Figure 3, it suffices to apply the *WatchDog Transformation* on the following refinement

$$\mathit{RUN}(\alpha \mathit{SixSidedDie}) \sqsubseteq_{\mathcal{T}} \mathit{SixSidedDie}$$

The reason is simple. The process $\mathit{RUN}(\alpha \mathit{SixSidedDie})$ can be refined by any process in the traces model and thus the PRISM variable *trace_error* is always *false* and the formula $\mathit{Pmax} =?\,[F\ \mathit{trace\_error}]$ will always return 0%.

The generated PRISM code in what follows is naturally different from the one presented in Section 3. But its Markov chain is the same of Figure 3.

```
dtmc
moduleWATCHDOG
    pc : [0..1] init 0;
    trace_error : bool init false;
    [e2] pc! = 0 → 1 : (trace_error' = true); //die.1
    [e2] pc = 0 → 1 : (pc' = pc = 0?0 : 1); //die.1
    [e3] pc! = 0 → 1 : (trace_error' = true); //die.3
    [e3] pc = 0 → 1 : (pc' = pc = 0?0 : 1); //die.3
    [e4] pc! = 0 → 1 : (trace_error' = true); //die.2
    [e4] pc = 0 → 1 : (pc' = pc = 0?0 : 1); //die.2
    [e5] pc! = 0 → 1 : (trace_error' = true); //die.4
    [e5] pc = 0 → 1 : (pc' = pc = 0?0 : 1); //die.4
    [e6] pc! = 0 → 1 : (trace_error' = true); //die.6
    [e6] pc = 0 → 1 : (pc' = pc = 0?0 : 1); //die.6
    [e7] pc! = 0 → 1 : (trace_error' = true); //die.5
    [e7] pc = 0 → 1 : (pc' = pc = 0?0 : 1); //die.5
endmodule
moduleP0
    pc0 : [0..13] init 0;
    [] pc0 = 0 → 0.5 : (pc0' = 1) + 0.5 : (pc0' = 2); //_prob.0
    [] pc0 = 1 → 0.5 : (pc0' = 3) + 0.5 : (pc0' = 4); //_prob.1
    [] pc0 = 2 → 0.5 : (pc0' = 5) + 0.5 : (pc0' = 6); //_prob.2
    [] pc0 = 3 → 0.5 : (pc0' = 1) + 0.5 : (pc0' = 7); //_prob.3
    [] pc0 = 4 → 0.5 : (pc0' = 8) + 0.5 : (pc0' = 9); //_prob.4
    [] pc0 = 5 → 0.5 : (pc0' = 10) + 0.5 : (pc0' = 11); //_prob.5
    [] pc0 = 6 → 0.5 : (pc0' = 2) + 0.5 : (pc0' = 12); //_prob.6
    [e2] pc0 = 7 → 1 : (pc0' = pc0 = 7?7 : 13); //die.1
    [e3] pc0 = 9 → 1 : (pc0' = pc0 = 9?9 : 13); //die.3
    [e4] pc0 = 8 → 1 : (pc0' = pc0 = 8?8 : 13); //die.2
    [e5] pc0 = 10 → 1 : (pc0' = pc0 = 10?10 : 13); //die.4
    [e6] pc0 = 12 → 1 : (pc0' = pc0 = 12?12 : 13); //die.6
    [e7] pc0 = 11 → 1 : (pc0' = pc0 = 11?11 : 13); //die.5
endmodule
system
    WATCHDOG || P0
endsystem
```

Instead of the variables s and d, we have integer variables whose prefix start with pc (resembling the program counter of an assembly code and match the LTS state) and the *trace_error* boolean variable. Therefore, if one is interested to analyse this generated PRISM code directly, instead of using the formula $P =? [F\, s = 7 \,\&\, d = x]$, it is necessary to use $P =? [F\, pc_0 = y]$ where y must assume one of the values 7, 8, 9, 10, 11, or 12, corresponding to the events *die*.1,..., *die*.6, which can be detected by simply reading the comments.

Fortunately, we do not need to know anything about the *pc* variables, nor the above automatically generated PRISM specification. Instead, we just have to formulate the appropriate traces refinement directly in CSP terms and check for $Pmax =? [F\, trace\_error]$. For this example, the CSP refinement could be

$$\texttt{Prop} \sqsubseteq_{\mathcal{T}} SixSidedDie$$

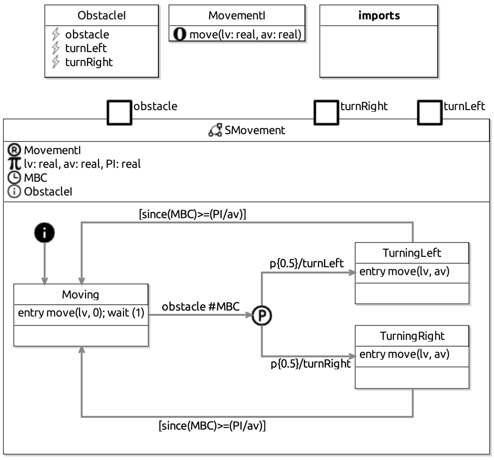
where $\texttt{Prop} = die.x \rightarrow STOP$ and $x \in \{1..6\}$ to check the probability of each face of the die. The property $Pmax =? [F\, trace\_error]$ yields the probability of 83.33%. This means that the refinement $Prop \sqsubseteq_{\mathcal{T}} SixSidedDie$ has a 16.67% complementary probability of holding. This corresponds exactly to the calculation in the PRISM website and also shown here in Section 3.

5 Using probabilities in RoboChart

In this section we present our strategy that allows probabilistic analysis of RoboChart models. We first introduce the use of the probabilistic RoboChart operator and then the extended RoboChart probabilistic semantics.

5.1 The RoboChart probabilistic operator

To create RoboChart models with probabilistic properties, we have to use a specific operator: Probabilistic Junction (Ⓟ). To illustrate its usage, we now consider an extension of the state machine of the RoboChart model presented in Section 2. We consider that the robot is equally likely to turn to the right and to the left. This new model is shown in Figure 4.



Ⓟ Probabilistic Junction

Fig. 4. RoboChart: obstacle detection with probabilities.

In this new version, when an obstacle is detected, the control of the model proceeds to a probabilistic junction between two equally likely alternatives. One alternative moves into the TurningLeft state, in which the robot turns to the left. The other alternative moves into the TurningRight state, turning the robot to the right. Afterwards, in both cases, the control goes back to the Moving state.

In the current state of RoboChart, probabilistic properties cannot be automatically analysed, since there is no direct translation from RoboChart to a probabilistic language. Recall from Section 2.1 that RoboChart models are automatically translated into CSP. This translation loses probabilistic aspects by using internal choices as semantics. This does not represent the correct meaning of a probabilistic specification; stochastic analyses cannot be done.

5.2 Dealing with probabilities

To extend the semantics of RoboChart models to deal with probability aspects, we have to consider the CSP probabilistic operator $\boxplus$ to define the semantics of $\textcircled{P}$, instead of internal choices. Let $\langle\!| \cdot |\!\rangle$ be this extended semantics.

To be able to analyse probabilistic RoboChart models, it suffices to apply the strategy presented in Section 4. That is, take a RoboChart model R, formulate the desired property about R as a CSP specification S, apply the *WatchDog Transformation* on the refinement

$$S \sqsubseteq_{\mathcal{T}} \langle\!| R |\!\rangle$$

and use the PRISM model checker using the single LTL formula

$$Pmax =? \, [F \; trace\_error]$$

Finally, interpret the result of the above formula as it is related to $S \not\sqsubseteq_{\mathcal{T}} \langle\!| R |\!\rangle$.

6 Case Study

In this section, we present a RoboChart model with probabilistic primitives. Furthermore, we also discuss some probabilistic analysis of the RoboChart model that the robotic engineers are now able to do.

6.1 Obstacle Detection

In this section we just illustrate the kind of analysis we can perform on our running example from Section 5.1.

We can use as property the following CSP specification.

$$\begin{aligned}&Prop = moveCall \rightarrow moveRet \rightarrow SMovement\_obstacle \rightarrow\\&\qquad SMovement\_turnRight \rightarrow STOP\end{aligned}$$

After performing the refinement

$$Prop \sqsubseteq_{\mathcal{T}} Obstable\_Detection$$

and checking for the probability of $Pmax =? \, [F \; trace\_error]$ we get 100%, indicating that (by the probabilistic complement) such a refinement does not hold.

6.2 Foraging Robot

This is a more complex example than the previous one. It is a simple foraging robot. It is equipped with an idealised randomising device with two activities that are equally likely to occur; the device generates an outcome in every time step. The robot uses the device to decide whether to terminate or to continue a particular activity (here, foraging). For reasons of its own, the robot may choose

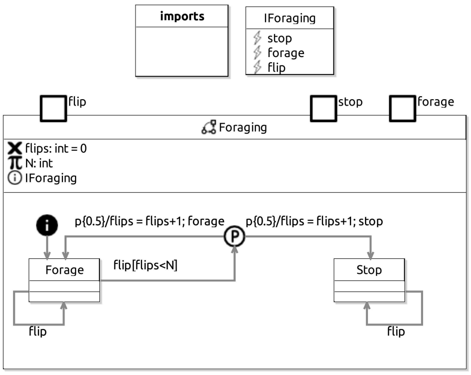


Fig. 5. RoboChart model

to ignore the outcome of the device. Finally, the robot considers only a limited number of times whether to continue foraging (see Figure 5).

The initial transition in Foraging leads to state Forage, in which a number of transitions can be taken. If a flip is allowed (represented by the occurrence of the event flip), the robot may ignore the randomising device and remain in the Forage state. Another possible transition from Forage happens when the event flip occurs and the number of choices has not been exhausted, given by (flips $<$ N). The constant N represents the maximum number of choices. In this case, the control proceeds to a probabilistic junction between two equally likely alternatives. One alternative is to move into the Stop state, which it signals with the stop event. The other is to return to the Forage state, signalling this with the forage event. In both cases, the value of flips is incremented (flips $=$ flips$+1$). This is used by the machine to keep track of the number of choices made.

There is only one transition available in the Stop state: the flip event keeps the controller in Stop; this transition is included to model the fact that flip occurs in every time step, even when the controller has terminated.

The CSP refinement property to check this model can be written as:

$$\begin{aligned} Prop = {} & (\Box\, e : \{flip, forage\} \bullet e \rightarrow Prop) \\ & \Box\, stop \rightarrow STOP \end{aligned}$$

By varying the N from 1 to 20 we get the graph depicted in Figure 6.

7 Conclusions

In this paper we defined the semantics of the RoboChart probabilistic operator Ⓟ as the CSP probabilistic operator $\boxplus$, preserving all the original CSP semantics of RoboChart as provided by [21]. We then reused the *WatchDog Transformation* provided by [8], obtaining a PRISM specification corresponding to the analysis

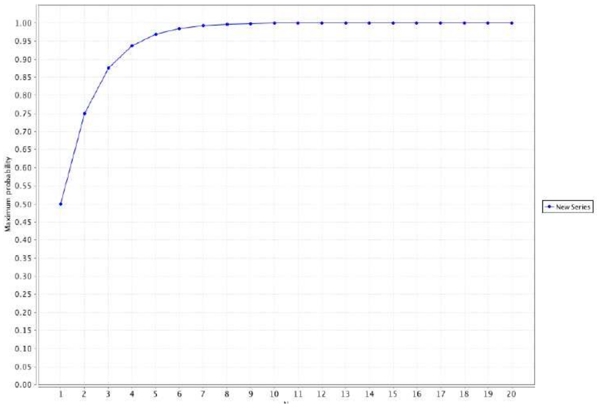


Fig. 6. Graph for Foraging Robot model plotting N from 1 to 20

of a refinement such as $P \sqsubseteq_{\mathcal{T}} Q$, where Q is the probabilistic CSP specification automatically generated by RoboTool and P is some property of interest.

The strategy reported in this paper has two main advantages over other attempts found in literature. The first is that it is based on CSP refinement and not in LTL model checking. This allows a more closer analysis style as already present in RoboChart. The second is that all data structures and functional language already available in CSP is inherited by the automatically generated PRISM specification. This is very interesting because it is hard to find rich data structures as well as a readable PRISM specification in literature.

One drawback of our strategy is that we never get a parameterised PRISM specification. But we can generate several models, each one corresponding to the values of the parameters being analysed. This is what the PRISM tool does directly from a parameterised PRISM specification.

Some works have been proposed for analyzing stochastic properties of robotic systems. In [13], probabilistic analysis are performed focusing on the robotic control software, ignoring the environment. It manually captures probabilistic state machines (using the PRISM language) of swarm systems from [15] in order to check specific properties in PCTL. No formal semantics is reported in this work. Probabilistic properties of swarm robotic models are also verified in [17]. It uses the process algebra Bio-PEPA [2] for modelling such models, which can be mapping to PRISM models by the Bio-PEPA suite of software tools.

We are working on another route to analysis of RoboChart models. This route goes from RoboChart to PRISM's Reactive Modules formalism via probabilistic Statecharts [10], and will give an alternative way of establishing probabilistic temporal properties. This translation is being built from metamodels of RoboChart, probabilistic Statecharts, and Reactive Modules, with the translation carried out using the Epsilon model transformation tool[7]. Our translation

[7] www.eclipse.org/epsilon/doc/book/

can also be expressed in Unifying Theories of Programming [9] as a Galois connection between Statecharts and Reactive Modules, suggesting a bidirectional transformation in Epsilon, supporting traceability of analysis results and counterexamples, and giving a formal way of verifying the translation using a semantics based on probabilistic predicate transformers in the style of McIver [19]. An interesting question is whether this more direct route will lead to models with different analysis performance in the PRISM tool.

Our translations will allow the use of more than just PRISM: the Reactive Modules formalism is also used for input to the MRMC and Storm model checkers, amongst others, and a dialect of probabilistic CSP is used for input to the PAT model checker [24]. We plan to explore the use of these alternatives and compare their performance on benchmarks that we will establish in robotic and autonomous control.

Finally, we plan to verify the results used in this paper, and in particular, the *WatchDog Transformation* as an implementation technique, using the CSP theories in Isabelle/UTP [5, 7, 6].

Acknowledgements.
This research was partially supported by INES 2.0, CAPES, FACEPE (grants PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17), and CNPq (grant 465614/2014-0). We would like to thank André Didier and Matheus Santana.

References

1. C. Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008.
2. F. Ciocchetta and J. Hillston. Bio-pepa: a framework for the modelling and analysis of biological systems, 2008.
3. C. Dehnert, S. Junges, Joost-Pieter Katoen, and M. Volk. A storm is coming: A modern probabilistic model checker. In *CAV (2)*, volume 10427 of *LNCS*, pages 592–600. Springer, 2017.
4. L. Fernandes, V. Custodio, G. Alves, and M. Fisher. A rational agent controlling an autonomous vehicle: Implementation and formal verification. In *EPTCS*, pages 35–42, 2017.
5. S. Foster and J. Woodcock. Unifying theories of programming in isabelle. In *ICTAC Training School on Software Engineering*, volume 8050 of *LNCS*, pages 109–155. Springer, 2013.
6. S. Foster and J. Woodcock. Mechanised theory engineering in isabelle. In *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 246–287. IOS Press, 2015.
7. S. Foster, F. Zeyda, and J. Woodcock. Isabelle/utp: A mechanised theory engineering framework. In *UTP*, volume 8963 of *LNCS*, pages 21–41. Springer, 2014.
8. M. Goldsmith. Csp: The best concurrent-system description language in the world-probably! In *Communicating Process Architectures*, pages 227–232, 2004.
9. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice Hall, 1998.

10. David N. Jansen, Holger Hermanns, and Joost-Pieter Katoen. A probabilistic extension of UML statecharts. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium, FTRTFT 2002, Co-sponsored by IFIP WG 2.2, Oldenburg, Germany, September 9-12, 2002, Proceedings*, volume 2469 of *Lecture Notes in Computer Science*, pages 355–374. Springer, 2002.
11. Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen. The ins and outs of the probabilistic model checker mrmc. *Perform. Eval.*, 68(2):90–104, February 2011.
12. D. Knuth and A. Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press, 1976.
13. Fisher M. Konur S., Dixon C. Formal verification of probabilistic swarm behaviours.
14. M. Kwiatkowska, G. Norman, and D. Parker. Prism 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 585–591. Springer Berlin Heidelberg, 2011.
15. W. Liu, Alan F. T. Winfield, and Jin Sa. Modelling swarm robotic systems : A case study in collective foraging. 2007.
16. Gavin Lowe. Probabilistic and prioritized models of timed CSP. *Theoretical Computer Science*, 138(2):315 – 352, 1995.
17. D. Latella M. Dorigo M. Massink, M. Brambilla and M. Birattari. On the use of bio-pepa for modelling and analysing collective behaviours in swarm robotics. page 201228.
18. M. Fisher M. Salem J. Saunders K. L. Koay M. Webster, C. Dixon and K. Dautenhahn. Formal verification of an autonomous personal robotic assistant. In *AAAI Spring Symposium Series*, page 7479, 2014.
19. A. McIver. Quantitative refinement *and* model checking for the analysis of probabilistic systems. In *FM*, volume 4085 of *LNCS*, pages 131–146. Springer, 2006.
20. A. Miyazawa, A. Cavalcanti, P. Ribeiro, W. Li, J. Woodcock, and J. Timmis. *RoboChart Reference Manual*. University of York, 2016. https://www.cs.york.ac.uk/circus/RoboCalc/assets/RoboChart-manual.pdf.
21. A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, and J. Timmis. Automatic property checking of robotic applications. In *IROS*, pages 3869–3876, 2017.
22. P. Ribeiro, A. Miyazawa, W. Li, A. Cavalcanti, and J. Timmis. Modelling and verification of timed robotic controllers. In *IFM 2017*, pages 18–33, 2017.
23. A.W. Roscoe. *Understanding Concurrent Systems*. Springer-Verlag.
24. Songzheng Song, Jun Sun, Yang Liu, and Jin Song Dong. A model checker for hierarchical probabilistic real-time systems. In *CAV*, volume 7358 of *LNCS*, pages 705–711. Springer, 2012.
25. A. Boulgakov A.W. Roscoe T. Gibson-Robinson, P. Armstrong. FDR3 — A Modern Refinement Checker for CSP. In Erika brahm and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.
26. M. Tribastone. The pepa plug-in project, 2009.
27. M. Webster, M. Fisher, N. Cameron, and M. Jump. Formal methods for the certification of autonomous unmanned aircraft systems. SAFECOMP, pages 228–242. Springer, 2011.