Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2014, Article ID 814159, 15 pages
http://dx.doi.org/10.1155/2014/814159

Research Article

Towards Light-Weight Probabilistic Model Checking

Savas Konur

Department of Computer Science, University of Sheffield, Sheffield S1 4DP, UK

Correspondence should be addressed to Savas Konur; s.konur@sheffield.ac.uk

Received 28 January 2014; Revised 19 May 2014; Accepted 3 June 2014; Published 10 July 2014

Academic Editor: Guiming Luo

Model checking has been extensively used to verify various systems. However, this usually has been done by experts who have a good understanding of model checking and who are familiar with the syntax of both modelling and property specification languages. Unfortunately, this is not an easy task for nonexperts to learn description languages for modelling and formal logics/languages for property specification. In particular, property specification is very daunting and error-prone for nonexperts. In this paper, we present a methodology to facilitate probabilistic model checking for nonexperts. The methodology helps nonexpert users model their systems and express their requirements without any knowledge of the modelling and property specification languages.

1. Introduction

Model checking [1] is a computational and algorithmic verification technique analysing if certain requirements hold in a system. These requirements are expressed as formal properties, such as *temporal logic* formulas. Model checking then exhaustively checks if these formal properties are satisfied by a structured model, for example, state transition system and finite state automaton, describing *all* system behaviours.

Model checking is an established research subject within computer science. There has been a vast amount of work carried out during the last two decades. Many tools have been devised and used in both academia and industry, for example, NuSMV [2], Spin [3], Uppaal [4], and Kronos [5]. Since model checking provides a comprehensive and an exhaustive computational analysis, it can reveal all possible system behaviours, which cannot be normally done by simulation or testing techniques. For this reason, it has been applied to various engineering problems, for example, hardware verification, software and programme verification, analysis of communication protocols, and safety-critical systems.

Standard model checking techniques are normally used to analyse *qualitative* temporal and dynamic properties. However, in order to have a deeper understanding of the system behaviour, some *quantitative* analysis is also required. Along with technological advances, systems are getting more complex. This requires the formal analysis to also include other aspects such as *uncertainy*, as the systems are getting more ubiquitous and probabilistic. In fact, while analysing many realistic systems, we need to consider uncertainty in system components and the reliability of communication [6] as well as uncertainty in planning and analysing performance. Unfortunately, standard model checking techniques are not sufficient to provide the level of analysis that we need in these cases.

Probabilistic model checking is a probabilistic variant of classical temporal model checking, which provides quantitative information regarding the likelihood that certain system behaviour is observed. This verification method has been successfully applied to formal analysis of a vast number of systems, for example, "bluetooth protocols, self-configuring protocols, self-organising systems, fault-tolerant algorithms, and scalable protocols" [7]. This is an active research subject within the model checking study. Various tools have been devised, such as Prism [8], Mrmc [9], Ymer [10], and Vesta [11]. Among these, Prism is the most widely used probabilistic model checking tool. Mrmc is also another popular tool with new improvements employed recently.

Although probabilistic model checking tools have been used to verify various systems, this usually has been done by experts who have a good understanding of model checking and who are familiar with the syntax of both modelling and property specification languages. Unfortunately, this is not an easy task for nonexperts to learn description languages for modelling and formal logics/languages for property specification. In particular, property specification is very daunting and error-prone for nonexperts. Research has shown that even experts make errors when they formally specify an informal requirement [12].

In this paper, we present a methodology to facilitate probabilistic model checking for nonexperts. As part of our methodology, we propose using some tools to achieve this. Namely,

(i) we propose a *property generator* tool which automatically generates formal properties using natural language statements;

(ii) we use a *front-end* tool which provides a graphical-user interface that allows constructing probabilistic state machines from which the model code is automatically generated;

(iii) we also devise a strategy which reduces the state space of probabilistic models to increase the performance of model checking and relieve the *state explosion* problem.

In this way, nonexperts can model their systems and express their requirements without any knowledge of the modelling and property specification languages and can analyse their systems using probabilistic model checking without having any expertise. We are not aware of any work which considers simplifying both modeling and property specification for model checking tool(s)—except very few studies introducing some simplifications for property specification, which will be reported in Section 5. We believe the idea and approach presented in this paper is novel, and this is an important step towards making model checking a more accessible computational technique for nonexpert users.

The paper is organised as follows: Section 2 summarises probabilistic model checking; Section 3 presents our methodology; Section 4 describes the model construction component; Section 5 describes the property generation component; Section 6 applies the methodology to an example system; Section 7 discusses how we will extend our methodology; and Section 8 concludes the paper.

2. Probabilistic Model Checking

In model checking, a finite system model (e.g., state transition system and finite state automaton), representing *all* system behaviours, is checked against a temporal logic formula. The entire state space represented by this model is then exhaustively analysed to verify if the formula is satisfied. For example, model checking allows us to check if the temporal logic formula,

$$\mathrm{AG}\neg\texttt{deadlock} \tag{1}$$

stating that "*deadlock* (`deadlock`) *never occurs* (G¬) *in any execution trace* (A)," is satisfied in a given system model.

In probabilistic model checking, probabilistic finite state machines are used as modelling language. The simplest probabilistic models are *Discrete-Time Markov Chains* (DTMCs), *Continuous-Time Markov Chains* (CTMCs), and *Markov Decision Processes* (MDPs). Formal properties are then expressed using *probabilistic logics*, a probabilistic extension of temporal logics. For discrete-time models (e.g., DTMCs and MDPs), the probabilistic extension of CTL, PCTL [13], is used and for continuous-time models (e.g., CTMCs) Continuous Stochastic Logic (CSL [14]) is used. Both languages can express quantitative expressions, such as *"the probability that deadlock never occurs is greater than 0.9,"* formally translated as

$$\mathrm{P}_{>0.9}\,[\mathrm{G}\neg\texttt{deadlock}]\,. \tag{2}$$

PCTL is defined according to the following grammar:

$$\begin{gathered}\varphi ::= \mathrm{true}\,|p|\,\neg\varphi\,|\varphi\wedge\varphi|\,\mathrm{P}_{\sim r}\,[\psi]\,,\\ \psi ::= \mathrm{X}\varphi\,|\varphi\mathrm{U}\varphi|\,\varphi\mathrm{U}^{\mathrm{I}}\varphi,\end{gathered} \tag{3}$$

where p is a set of atomic propositions, $0 \le r \le 1$ is a *probability bound,* and $\sim\in \{<,>,\le,\ge,=\}$. The *probabilistic operator* $\mathrm{P}_{\sim r}$ is the probabilistic extension of branch/path quantifiers A and E of CTL. Informally speaking, $\mathrm{P}_{\sim r}[\psi]$ is satisfied at a state s if, and only if, the probability of taking the path from the state s which satisfies ψ meets the bound "$\sim r$." The *path* formulas ψ are constructed using the *next* (X) *until* (U) and *bounded-until* (U^{I}) operators. We can derive other temporal operators, such as F (*eventually*) and G (*always*) from X and U. For example, $\mathrm{F}\varphi \equiv \mathrm{true}\,\mathrm{U}\varphi$ and $\mathrm{G}\varphi \equiv \neg\mathrm{F}\neg\varphi$. The informal meanings of path formulas are as follows:

(i) $\mathrm{X}\varphi$ holds at a state on a path if, and only if, φ holds in the next state on the path;

(ii) $\mathrm{F}\varphi$ holds at a state on a path if, and only if, φ holds eventually at some future state on the path;

(iii) $\mathrm{G}\varphi$ holds at a state on a path if, and only if, φ holds at all future states on the path;

(iv) $\varphi_1\mathrm{U}\varphi_2$ holds at a state on a path if, and only if, φ_1 holds on the path up until φ_2 holds; and

(v) $\varphi_1\mathrm{U}^{\mathrm{I}}\varphi_2$ holds at a state on a path if, and only if, φ_2 holds on the path at some time step within the interval I and φ_1 holds at all preceding states on the path.

The formal semantics of the operators X, U, and U^{I} are defined as follows:

(i) $\mathscr{M},\sigma \vDash \mathrm{X}\varphi$ if and only if $\mathscr{M},\sigma[1] \vDash \varphi$;

(ii) $\mathscr{M},\sigma \vDash \varphi_1\mathrm{U}\varphi_2$ if and only if $\exists i \ge 0$ s.t. $\mathscr{M},\sigma[i] \vDash \varphi_2$ and (for all $j < i$) $\mathscr{M},\sigma[j] \vDash \varphi_1$;

(iii) $\mathscr{M},\sigma \vDash \varphi_1\mathrm{U}^{\mathrm{I}}\varphi_2$ if and only if $\exists i \in \mathrm{I} = [d,d']$ s.t. $\mathscr{M},\sigma[i] \vDash \varphi_2$ and (for all $d \le j < i$) $\mathscr{M},\sigma[j] \vDash \varphi_1$,

where $\mathscr{M}$ is a Markov chain (or an Markov decision process) and σ is a *path*, that is, a (possible infinite) sequences of states. The ith element of a path σ is denoted by $\sigma[i]$.

In addition to the operators of PCTL, the logic CSL also contains the *steady-state* (*long-run*) $\mathrm{S}_{\sim r}$ operator. Namely, the formula $\mathrm{S}_{\sim r}[\psi]$ holds in a state s if, and only if, the steady-state

```
module module_name

    //variable declarations
    s : [0 ... 10] init 0;

    //state transitions
    []s = 0 → 0.4 : (s' = 1) + 0.6 : (s' = 2);

endmodule
```

Algorithm 1

probability of being in a state which satisfies ψ is bounded by "$\sim r$".

A probabilistic model checker can be used to verify if a PCTL or CSL formula—depending on the time semantics—holds in a given probabilistic model. In this paper, we particularly consider the model checkers Prism, Mrmc, and Ymer because of their support for these languages and useful features which they employ.

2.1. Prism. Prism is the most widely used probabilistic model checking tool. It provides formal verification for different probabilistic models such as DTMCs, CTMCs, and MDPs, which are coded as the Prism's *reactive modules*, a simple state-based language. Prism is a *symbolic* model checker; namely, it uses a compact and structured representation of "data structures based on binary decision diagrams (BDDs) and multiterminal binary decision diagrams (MTBDDs)" in order to reduce the size of probabilistic models [15].

The Prism modelling language can be described as follows (summarised from [16]). The language is composed of a set of components, called *modules*, representing an encapsulation for different parts of the system and *variables* representing the state of each module. The global state of the model at any time point is determined by the values of all variables (and optionally global variables). Variables can be declared using the syntax

$$s : [0 \cdots 10]\ \mathbf{init}0; \tag{4}$$

meaning that s is an integer variable whose values range between 0 and 10; that is, 0 is the *lower bound* and 10 is the *upper bound*. State transitions can be modeled using the following syntax:

$$[\text{act}]\ \text{guard} \longrightarrow \text{rate} : \text{update}, \tag{5}$$

where "act is an (optional) label, guard is a predicate over the variables of the model, rate is a (nonnegative) real-valued expression, and update is the values of variables in the next state" [15]. For example,

$$[\,]\ s = 0 \longrightarrow 0.4 : \left(s' = 1\right) + 0.6 : \left(s' = 2\right); \tag{6}$$

states that if the variable s is 0 then s will be 1 in the next state with probability 0.4 and will be 2 with probability 0.6, otherwise. A module is then described as shown in Algorithm 1.

The property specification languages supported by the tool include PCTL and CSL. In addition to probabilistic properties, Prism also supports *rewards* structures, providing reward properties based on quantitative information such as "expected number of deadlocks" and "expected time taken to reach a state where a deadlock occurs." This is done using "R" operator. Prism provides the following reward types: the *reachability* reward ($\mathrm{R}_{\sim r}[\mathrm{F}\varphi]$), *cumulative* reward ($\mathrm{R}_{\sim r}[\mathrm{C} \leq t]$), *instantaneous* reward ($\mathrm{R}_{\sim r}[\mathrm{I} = t]$), and *steady-state* reward ($\mathrm{R}_{\sim r}[\mathrm{S}]$). These formulas can be intuitively described as follows [15]:

(i) $\mathrm{R}_{\sim r}[\mathrm{F}\varphi]$ asserts that the expected reward accumulated until φ is satisfied is bounded by $\sim r$;

(ii) $\mathrm{R}_{\sim r}[\mathrm{C} \leq t]$ asserts that the expected reward accumulated until time t is bounded by $\sim r$;

(iii) $\mathrm{R}_{\sim r}[\mathrm{I} = t]$ asserts that the expected value of the state reward at time instant t is bounded by $\sim r$;

(iv) $\mathrm{R}_{\sim r}[\mathrm{S}]$ asserts that the long-run average expected reward is bounded by $\sim r$.

We remark that rewards are added to model files. The reward structures denote labellings of states and transitions with associated real values. These values are then used when calculating the accumulated or instantaneous rewards.

Prism provides both a command-line tool and a graphical user interface. The latter provides (i) a model editor for the description/modelling language; (ii) an editor for property specification langauge; (iii) a simulator tool to debug model execution traces; and (iv) graph-plotting tools [16]. Prism also provides an approximate/statistical model checking feature, but this is only allowed to a subset of PCTL/CSL formulas. For example, statistical model checking cannot be used in verification of steady-state properties.

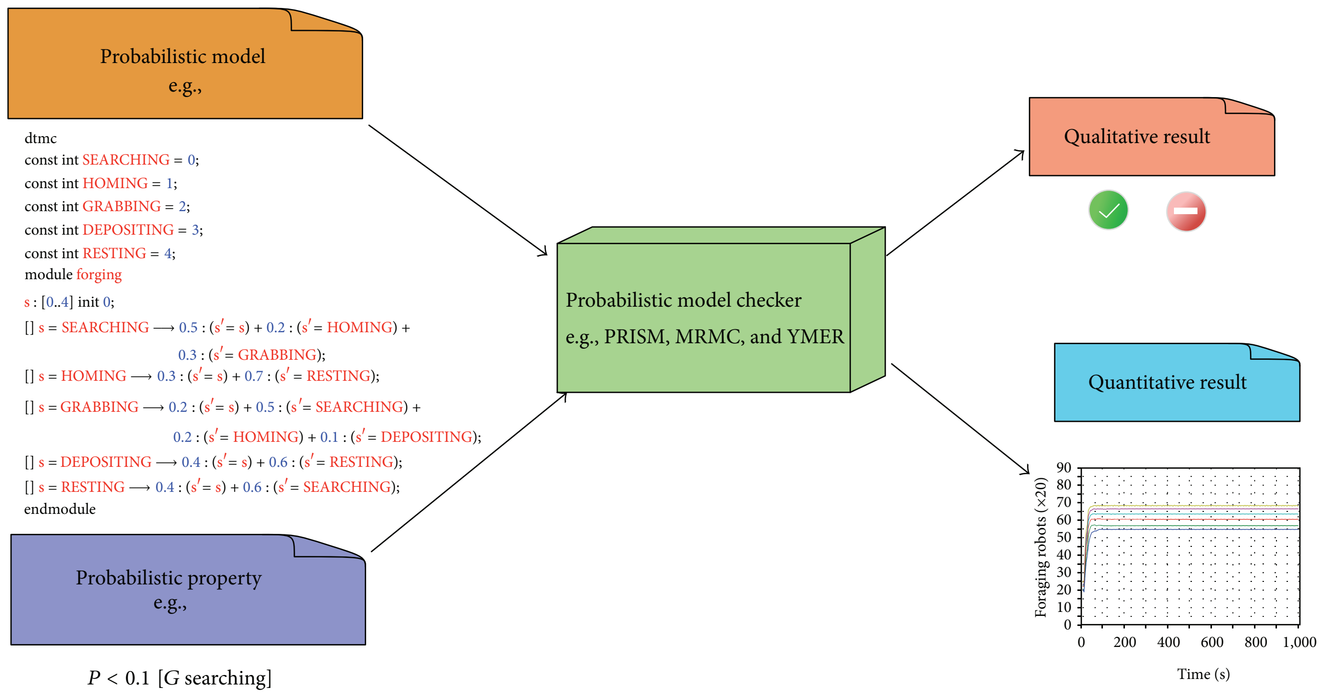
2.2. Mrmc. Mrmc is another probabilistic model checking tool, which allows formal verification for DTMCs and CTMCs (as well as Continuous-Time Markov Decision Processes, a nondeterministic variant of CTMCs). As in Prism, DTMCs and CTMCs are verified against PCTL and CSL formulas, respectively.

Mrmc also features the logics PRCTL and CRSL, which are an extension of PCTL and CSL, respectively, with rewards structures. The syntax of the reward formulas are slightly different than Prism's reward formulas [17]: *reachability* reward: $\mathrm{E}[t][r_1, r_2][\varphi]$, *cumulative* reward: $\mathrm{Y}[t][r_1, r_2][\varphi]$, *instantaneous* reward: $\mathrm{C}[t][r_1, r_2][\varphi]$, and *steady-state* reward: $\mathrm{E}[r_1, r_2][\varphi]$. The rewards represent expected rewards per time unit and are checked against a reward bound $[r_1, r_2]$. For example,

$$\mathrm{Y}\,[t]\,[r_1, r_2]\,[\varphi] \tag{7}$$

meaning that the expected accumulated reward rate per time unit in φ states until the tth transition will be within the interval $[r_1, r_2]$.

Some important features of Mrmc are summarised as follows [9]. Unlike Prism, Mrmc is a command-line explicit



```
dtmc
const int SEARCHING = 0;
const int HOMING = 1;
const int GRABBING = 2;
const int DEPOSITING = 3;
const int RESTING = 4;
module forging
s : [0..4] init 0;
[] s = SEARCHING ⟶ 0.5 : (s′= s) + 0.2 : (s′= HOMING) +
                         0.3 : (s′= GRABBING);
[] s = HOMING ⟶ 0.3 : (s′= s) + 0.7 : (s′ = RESTING);
[] s = GRABBING ⟶ 0.2 : (s′= s) + 0.5 : (s′= SEARCHING) +
                         0.2 : (s′= HOMING) + 0.1 : (s′= DEPOSITING);
[] s = DEPOSITING ⟶ 0.4 : (s′= s) + 0.6 : (s′= RESTING);
[] s = RESTING ⟶ 0.4 : (s′= s) + 0.6 : (s′= SEARCHING);
endmodule
```

$P < 0.1$ [G searching]

Figure 1: Standard approach for probabilistic model checking.

state model checker employing a "numerical solution engine." It has been therefore used as a back-end model checking tools for various systems including Petri nets, process algebras, and stochastic hybrid systems. Mrmc has recently been improved with a better memory management and implementation of the sparse matrices and support for bisimulation minimization. This increases its efficiency and performance on large models.

Mrmc also employs a "discrete-event simulation engine" for statistical CSL model checking. In statistical model checking, rather than exhaustively exploring the entire state space, the system is simulated finitely many times to obtain execution traces and statistical evidence is provided for the verification of a property [18]. Unlike Prism, Mrmc's statistical model checking covers the entire formula set of CSL, including steady-state properties.

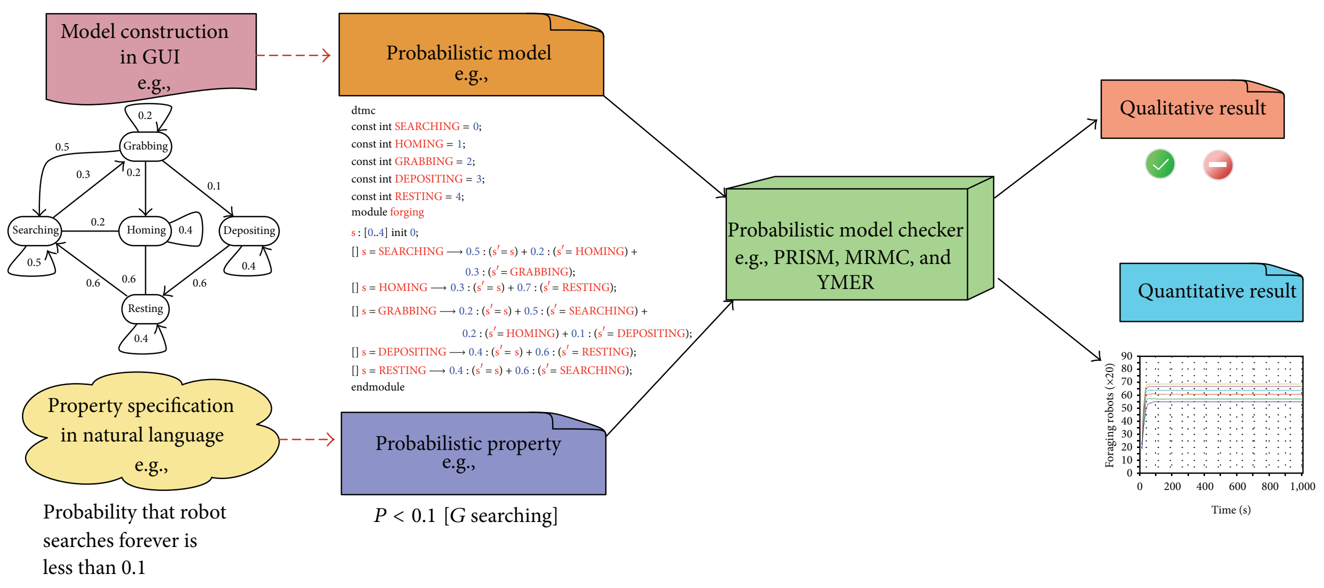
2.3. Ymer. Ymer is a *statistical* model checking tool, used to verify transient properties of CTMCs [10]. So, unlike Prism and Mrmc, it does not exhaustively analyse all system behaviour. Instead, it employs statistical techniques relying on "discrete event simulation and sequential acceptance sampling" [10]. The modelling language of Ymer is very similar to that of Prism with a difference that Ymer also supports *generalized semi-Markov processes*. The property specification language of the tool is CSL, but Prism and Mrmc support a richer set of properties than Ymer.

3. Light-Weight Approach to Probabilistic Model Checking

As in classical model checking, a probabilistic model checker requires two inputs: (i) a probabilistic model of the system to be analysed and (ii) a probabilistic property. As discussed above, a probabilistic model can be a DTMC, CTMC, or MDP, which is coded according to the high-level modelling language of the model checker, for example, reactive modules in case of Prism. A probabilistic property is the formal specification of a requirement to be checked, which is expressed in one of the probabilistic logics described above, for example, PCTL when the system is modeled in DTMC. The model checker then automatically checks if the model satisfies the given specification. Based on the type of the property, it produces either a qualitative answer (a "yes" or "no") or a quantitative result. If the property is not satisfied, the model checker also produces a counterexample to help the modellers debug the output and find the bug. The overall process is illustrated in Figure 1.

In standard approach, models are created manually using a high-level modelling language tailored to a particular model checker and properties are specified in a specific formal logic/language. This, however, requires having a good understanding of both modeling and property specification and being familiar with the syntax of both high-level description and specification languages. Unfortunately, this is not an easy task for nonexperts to learn description languages for modelling and formal logics for property specification. In particular, property specification is very daunting and error-prone for nonexperts and even for experts in case of some complex properties.

To make this process easier for nonexperts, we propose an approach facilitating probabilistic model checking by providing an abstraction over high-level modeling and property specification languages. Namely, we suggest creating a system model using a *graphical user interface* and generate



```
dtmc
const int SEARCHING = 0;
const int HOMING = 1;
const int GRABBING = 2;
const int DEPOSITING = 3;
const int RESTING = 4;
module forging
s : [0..4] init 0;
[] s = SEARCHING ⟶ 0.5 : (s′ = s) + 0.2 : (s′ = HOMING) +
                    0.3 : (s′ = GRABBING);
[] s = HOMING ⟶ 0.3 : (s′ = s) + 0.7 : (s′ = RESTING);
[] s = GRABBING ⟶ 0.2 : (s′ = s) + 0.5 : (s′ = SEARCHING) +
                   0.2 : (s′ = HOMING) + 0.1 : (s′ = DEPOSITING);
[] s = DEPOSITING ⟶ 0.4 : (s′ = s) + 0.6 : (s′ = RESTING);
[] s = RESTING ⟶ 0.4 : (s′ = s) + 0.6 : (s′ = SEARCHING);
endmodule
```

Model construction in GUI e.g.,
0.2
Grabbing
0.5
0.3
0.2
0.1
Searching
0.2
Homing
0.4
Depositing
0.5
0.6
0.6
0.6
0.4
Resting
0.4
Probabilistic model e.g.,
Probabilistic model checker e.g., PRISM, MRMC, and YMER
Qualitative result
Quantitative result
Foraging robots (×20)
90
80
70
60
50
40
30
20
10
0
0
200
400
600
800
1,000
Time (s)
Property specification in natural language e.g.,
Probability that robot searches forever is less than 0.1
Probabilistic property e.g.,
$P < 0.1$ [G searching]

FIGURE 2: A light-weight approach for probabilistic model checking.

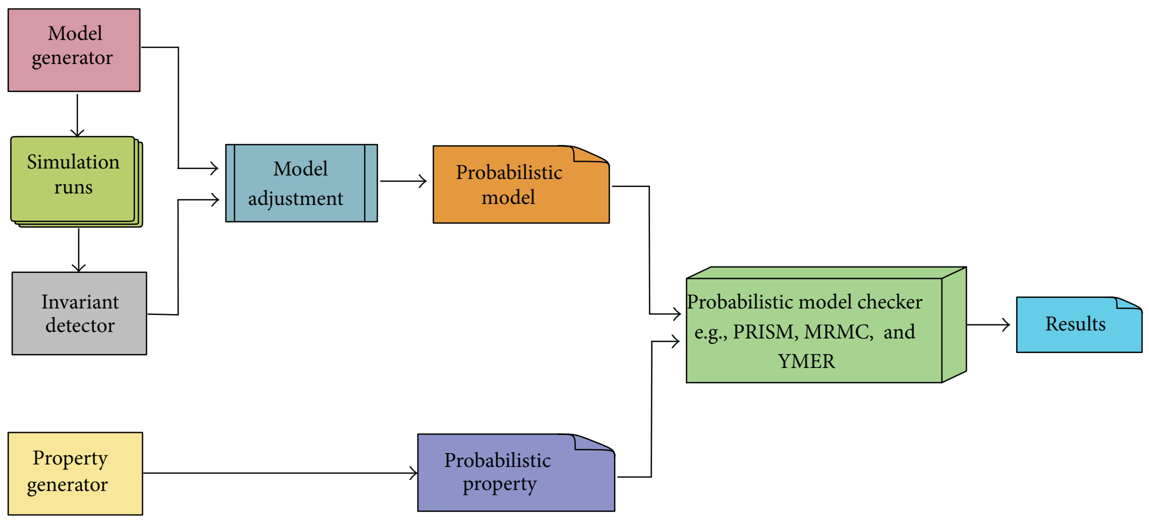


FIGURE 3: System overview.

informal properties using *natural language* statements. The constructed model is then translated to the corresponding high-level modelling language, and the informal properties are translated into their formal counterparts. Our approach is illustrated in Figure 2. Here, we consider the models constructed as probabilistic state machines and natural language statements as probabilistic properties, because in this paper we focus on probabilistic model checking.

The system overview of our approach is shown in Figure 3, summarised as follows.

Model Construction. A system model is constructed using the *Model Generator*, which is a graphical user interface providing drawing features to construct probabilistic state machines. The constructed model is then translated into a high-level modelling language, which then becomes an input to the corresponding model checker.

Reducing Model Size. A well-known problem with model checking is the *state explosion* problem. Namely, the number of states increases so rapidly that model checking cannot become feasible any more. The size of model, mostly depending on the number of states, is an important factor for the model checking efficiency, because the resources required to perform model checking are very sensitive to the model size. Unfortunately, it is very easy to create huge models when variables are unnecessarily assigned to very large ranges. A good strategy to tackle the state explosion problem and increase the performance of model checking is therefore to reduce the number of states. As discussed in Section 2.1, variable ranges are defined using a *lower* and an *upper* bound. In most cases, users might not be able to know the exact lower and upper bounds. If the estimation of variable ranges is not realistic, then the resulting model size will be very large.

In order to tackle this issue, we devise a method automatically estimating a reasonable lower and upper bound for model variables. In order to do this, we simulate the model generated by the Model Generator, use an *invariant detector* to analyse the simulation results, and acquire the lower and upper bounds for variables. In the *Model Adjusment* process we then update the model variables with the bounds acquired from the invariant detector. In this way, even if the users provide very large bounds, we can reduce them and update the model accordingly. The updated model is then given to the model checker as an input.

Property Construction. The *Property Generator* tool provides a graphical user interface allowing users to create properties easily by manipulating a configurable form with some interfaces such as combo and text boxes. So, by only selecting natural language statements, users can generate a set of informal properties which are automatically translated into their formal counterparts. The formal properties can be directly given to the corresponding model checker as an input.

4. Model Construction

As part of the proposed methodology, we aim to use various tools to make the modeling task less complicated for nonexperts. In this section, we provide a more detailed account for the tools and methods used.

4.1. Model Generator. The *Model Generator* component relies on the prototype front-end tool, called *Drawing* Prism, developed in [19]. The Drawing Prism tool, DP in short, provides drawing features to construct probabilistic state machines, which are then translated into a Prism model file.

DP implements two interfaces: (i) *drawing interface* allowing users to draw probabilistic state machines—see Figure 4—and to populate their state machines as a data model; (ii) *translator*, which translates the data model representing the probabilistic state machine into the Prism's high-level modelling language.

DP employs ArgoUML [20] as the drawing tool. ArgoUML is an open source project, and it supports most popular standards, for example, UML, XMI, and SVG. It is written in Java; it is therefore supported by any platform with a Java environment.

ArgoUML converts a state diagram into an XMI data. XMI, standing for XML-based Metadata Interchange Format, is mainly used as a model interchange format for UML. Although XMI is not an easy format to read, it has some advantages, for instance, its abstract tree representation which makes parsing easy. XMI stores all the necessary information in a state machine, for example, states, transitions, labels, and probability values. The translator extracts all

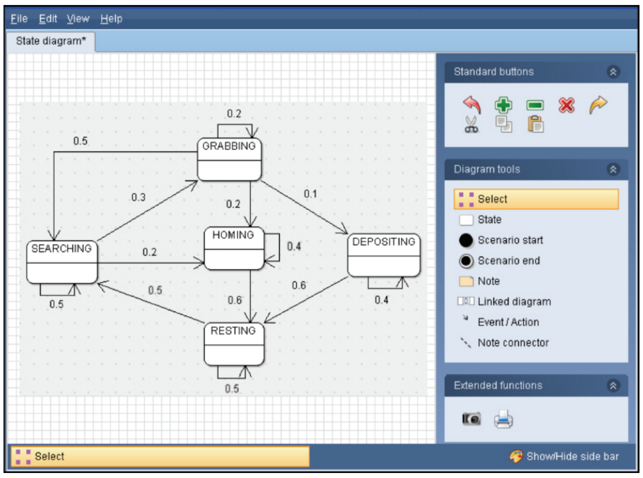


Figure 4: A probabilistic state machine constructed in DP [19].

information from a state machine in XMI and then translates it into the format accepted by Prism.

The DP function design is presented in Figure 5. The functional components can be summarised as follows [19].

(i) *User Interface*. "It has a control and drawing panel through which users access DP and interact closely with system operators. Control components provide a common control surface used by GUI, scripts, and programmatic access. The Interface is used to integrate the rest of these subsystems."

(ii) *Drawing System*. It classifies every component of the state machine drawn using the user interface and then integrates these components in the data structure.

(iii) *XMI Data*. It is the data structure acquired from the Drawing System and translated to the XMI format.

(iv) *Translator*. It translates state machine information in the XMI data structure to a Prism model file using specific algorithms.

4.2. Invariant Detector. The *Invariant Detector* component generates reasonable lower and upper bounds for model variables by automatically detecting invariants through analysing the simulation traces produced by *Model Generator*. For this task we employ the Daikon tool [21].

Daikon is an invariant detector tool which dynamically reports invariants in a program. An *invariant* is simply a mathematical property, such as $x = 3$, $y \leq 2x$, z is one of $\{0, 1\}$, holding at specific execution points of a program. Daikon executes a code, analyses the values of program variables, and then detects invariants which are true over certain points.

Daikon is originally developed to support high-level programming languages, for example, C, C++, Java, and Perl. However, its source code is freely available and can be modified to be able to use it in other formats. In order to facilitate its usage in detecting invariants from simulation traces, we have extended its functionality. After analysing the simulations, it produces some mathematical relations

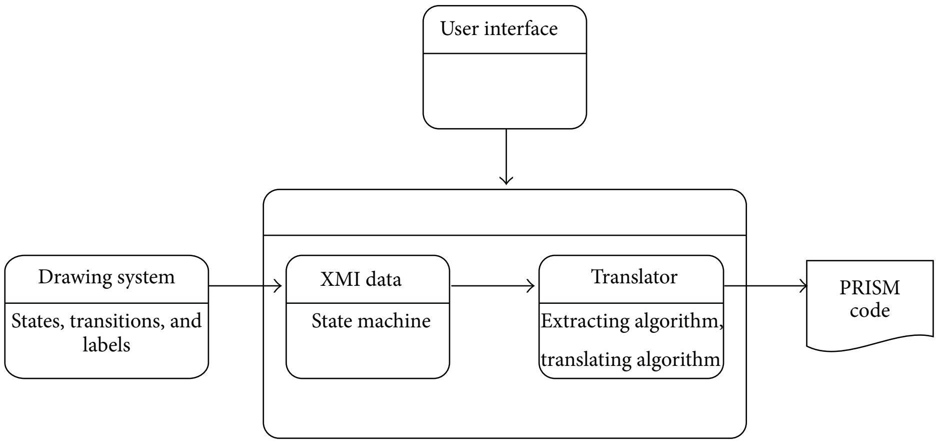
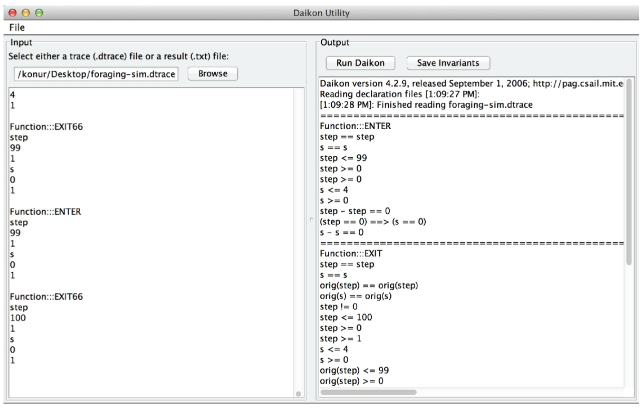

User interface
Drawing system
States, transitions, and labels
XMI data
State machine
Translator
Extracting algorithm, translating algorithm
PRISM code

Figure 5: Functional components of DP [19].


Daikon Utility
File
Input
Select either a trace (.dtrace) file or a result (.txt) file:
Browse
Output
Run Daikon
Save Invariants

```
/konur/Desktop/foraging-sim.dtrace
```

```
4
1

Function:::EXIT66
step
99
1
s
0
1

Function:::ENTER
step
99
1
s
0
1

Function:::EXIT66
step
100
1
s
0
1
```

```
Daikon version 4.2.9, released September 1, 2006; http://pag.csail.mit.e
Reading declaration files [1:09:27 PM]:
[1:09:28 PM]: Finished reading foraging-sim.dtrace
===========================================================
Function:::ENTER
step == step
s == s
step <= 99
step >= 0
step >= 0
s <= 4
s >= 0
step - step == 0
(step == 0) ==> (s == 0)
s - s == 0
===========================================================
Function:::EXIT
step == step
s == s
orig(step) == orig(step)
orig(s) == orig(s)
step != 0
step <= 100
step >= 0
step >= 1
s <= 4
s >= 0
orig(step) <= 99
orig(step) >= 0
```

Figure 6: Daikon invariant detector.

between various model variables. The ranges provided by Daikon then become a realistic lower and upper bound.

Daikon was previously used to detect invariants to construct formal specifications [22, 23]. In contrast, here we use the invariants to estimate reasonable bounds for model variables, as discussed above. In [24], Daikon was used to formulate temporal logic formulas to be model checked by NuSMV. Here, we extend the work done in [24] by adjusting it to accept the outputs of Prism's discrete event simulator. A screen shot of the tool is shown in Figure 6.

Depending on the quality of information contained in execution traces, Daikon can report many types of invariants, such as *arithmetic* ($y \leq 2x$), *non-zero* (e.g., $x \neq 0$), *element of* (e.g., x is one of $\{0, 1\}$), and *interval* (e.g., $0 \leq x \leq 5$). Daikon might also report some redundant invariants (e.g., $x == x$). However, it employs a filtering mechanism allowing to omit redundant and unwanted invariants to be returned. In this way, we can only obtain *upper bound* (e.g., $x \leq 2$) and *lower bound* (e.g., $x \geq 0$) invariants.

Remark. The Daikon tool is originally developed to detect invariants within a program written in a high-level programming language. In a typical programme, there can be hundreds of variables. Daikon is capable of reporting invariants for such large numbers of variables over thousands of executions. A typical system for which model checking is feasible probably contains tens of model variables in the model description (which might lead to millions/billions of states/transitions when the model is constructed by the model checker). It will be sufficient to obtain around a hundred simulation traces to find the bounds for model variables. This suggests that the resources required to use the Daikon tool to obtain variable bounds are not more than those required by its original usage.

4.3. Model Adjustment. In the *Model Adjustment* process we perform two tasks as follows. (i) We update the bounds of model variables in the Prism model file with the bounds acquired from the invariant detector. In this way, even if the users provide very large bounds, we can reduce them, and update the model accordingly. The updated model is then given to Prism as an input. (ii) We also translate the same model to the corresponding Mrmc language. At the moment, we are using Prism's built-in export facility to translate Prism models to the Mrmc format. The modelling language of Ymer is very similar to that of Prism, which requires a few minor changes. Currently, we make these changes manually, but an automatic translation is a very primitive process.

5. Property Construction

Property specification is very daunting and error-prone for nonexperts. It is even cumbersome for experts to specify complex properties. In most cases, properties are not intuitive and self-explanatory, so their meanings will not be clear to those not familiar with formal methods. This unfortunately hinders reusability of properties already studied in the literature and makes them inaccessible to a wide audience. Another issue that we would like to address is that although a formal property can be syntactically correct it might not be a valid representation of the requirement we wish to verify [12]. This is especially the case when the formal specification is complex and long.

In order to tackle these issues and facilitate property specification, we propose the *Property Generator* tool, which provides guidance to construct properties using the *natural* *language statements* representing informal properties and then automatically translates them to their formal counterparts. Users are able to construct properties easily by manipulating a configurable form with some interfaces such as combo and text boxes, which makes the property specification a very simple and intuitive task.

The Property Generator tool features a set of *property patterns* based on most recurring properties studied in the literature. These patterns provide a systematical classification, guiding users to construct natural language expressions representing the properties that they want to express. The tool works based on a *structured grammar* defined for both natural language representation of these patterns and their formal translations.

The idea of categorising properties into a set of patters initially started in [12], where several hundreds of properties were analysed. Since this seminal work, there have been several studies in this direction. [25] extended the pattern classes of [12] with more time related patterns and their "associated observer automata." [26] provided a set of patterns in the context of real-time specification. [27] introduced a unified pattern system extended with a new set of real-time pattern classes. [28] analysed probabilistic properties and provided a probabilistic category system along with a structured grammar. In [29] we proposed a set of query templates similar to the one presented in this paper which targeted biological models.

A subset of the patterns which we used in the Property Generator tool and their formal translations in Prism and Mrmc are shown in Table 1. In the table, *phi* and *psi* are state expressions returning true or false (e.g., $x \leq 4$), $\bowtie\ \in \{<, >, \leq, \geq\}$ is a relational operator, $p \in [0, 1]$ is a real, $r, r1, r2$ are real numbers, and $t, t1, t2$ are integers. Here, where applicable, we provide both the *unbounded* and *bounded* versions of the patterns. Note that Ymer can express only a subset of these patterns, and we therefore do not include it in the table.

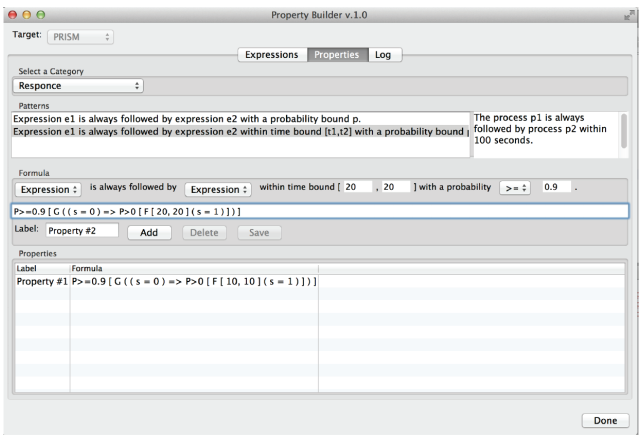
As Table 1 illustrates, properties in natural language are much more intuitive and comprehensive than their formal translations. For example, the natural language representation of the *Bounded Response* pattern

$$\text{“}\mathit{phi} \text{ is always followed by } \mathit{psi} \text{ within time bound } [t1, t2] \text{ with a probability } \bowtie p\text{”} \tag{8}$$

is much more easier than its formal translation, for example, Mrmc,

$$\mathtt{P\,\{\bowtie p\}\,[!\,(tt\ U\ !(phi => P >= 1\,[tt\ U\,[t1, t2]\,psi]))]}\,. \tag{9}$$

The Property Generator tool makes the property construction a very easy and effortless task without requiring the knowledge about the formal syntax of the target model checker. For example, to build this property, the user selects the *bounded* version of the *Response* pattern using the graphical user interface provided. The user only needs to provide the values for *phi*, *psi*, $t1$, $t2$, $\bowtie$, and p. The formal translation is then done automatically.



Target: PRISM

Select a Category: Responce

Patterns:
Expression e1 is always followed by expression e2 with a probability bound p.
Expression e1 is always followed by expression e2 within time bound [t1,t2] with a probability bound

The process p1 is always followed by process p2 within 100 seconds.

Formula: Expression is always followed by Expression within time bound [20 , 20] with a probability >= 0.9 .

P>=0.9 [G ((s = 0) => P>0 [F [20, 20] (s = 1)])]

Label: Property #2

Properties

Label	Formula
Property #1	P>=0.9 [G ((s = 0) => P>0 [F [10, 10] (s = 1)])]

Figure 7: Property Generator tool.

A screen shot of the Property Generator tool is illustrated in Figure 7. We developed a previous version of the tool to construct (linear) temporal properties used in formal verification of *kP System* models [30].

The operation of the tool during a property construction process is as follows. The user first selects a target, for example, Prism, Mrmc, and Ymer, to which the property is translated. When a pattern is selected from the category combo box, a template (for bounded and unbounded cases) is displayed in the text field. When the appropriate template is clicked, both the informal representation and its formal translation appear in the corresponding text box. Also, a typical example of the selected pattern is displayed at another text field. In order to complete the formula, the user is required to select and fill in the missing values. The selection and typing are interactively shown in the text box displaying the formal translation. When the formula is complete, the user can save it. If there are still missing fields to be completed, he/she receives a warning. The user can edit a saved property at any time.

The main components of the Property Generator tool are shown in Figure 8. The *Expression Builder* component of the *Property Generator GUI* constructs atomic expressions, that is, state formulas returning *true* or *false*, using model variables and constants, and Boolean expressions using atomic expressions. The grammar for building expression is defined in the *Expressions* file. The *Property Builder* of the GUI constructs properties using the generated expressions and a set of patterns whose grammar is defined in the *Patterns* file. The *Cached Data* component keeps the expressions and properties created or managed throughout a user session. It works as a repository to keep the generated data until they are saved to a file and the application closes.

Remark. The performance of the Property Generator tool does not depend on the model size. The properties are constructed from a set of patterns. Once the user fills in the empty fields, the construction is done instantly. So, it does not matter whether one uses this tool for a small example or a large example.

Table 1: Property patterns and their PRISM and MRMC translations.

Property in natural language	PRISM translation	MRMC translation
phi will eventually hold, until then *psi* holds with a probability ⋈*p*	`P⋈p [psi U phi]`	`P{⋈p} [psi U phi]`
phi will hold within time bound [*t*1, *t*2], until then *psi* holds with a probability ⋈*p*	`P⋈p [psi U[t1, t2] phi]`	`P{⋈p} [psi U[t1, t2] phi]`
phi always holds with a probability ⋈*p*	`P⋈p [G phi]`	`P{⋈p} [!(tt U !(phi))]`
phi always holds within time bound [*t*1, *t*2] with a probability ⋈*p*	`P⋈p [G[t1, t2] phi]`	`P{⋈p} [!(tt U[t1, t2]!(phi))]`
phi is always followed by *psi* with a probability ⋈*p*	`P⋈p [G (phi => P >= 1 [true U psi])]`	`P{⋈p} [!(tt U !(phi => P >= 1 [tt U psi]))]`
phi is always followed by *psi* within time bound [*t*1, *t*2] with a probability ⋈*p*	`P⋈p [G (phi => P >= 1 [true U[t1, t2] psi])]`	`P{⋈p} [!(tt U !(phi => P >= 1 [tt U[t1, t2] psi]))]`
phi holds infinitely often with a probability ⋈*p*	`P⋈p [G (P >= 1 [true U phi])]`	`P{⋈p} [!(tt U !(P >= 1 [tt U phi]))]`
The expected accumulated reward until *phi* holds is ⋈*r*	`R⋈r [true U phi]`	
The expected accumulated reward until *phi* holds within *t* time units is between *r*1 and *r*2		`E[t][r1, r2][phi]`

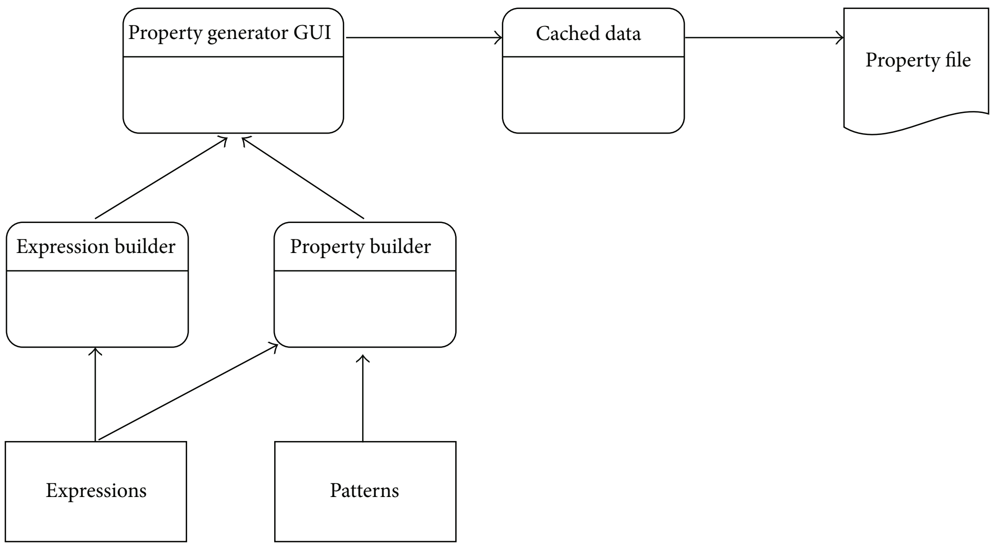


Figure 8: Main components of the Property Generator tool.

6. An Example: Foraging Robots

In this section, we illustrate our approach on an example system. Here, we choose the *foraging robot scenario*, presented in [31]. We analysed the system in [32, 33], which can be described as follows:

> "Within a fixed size arena, there are a number of foraging robots; that is, each robot must search a finite area and bring food items back to the common nest. Food is placed randomly over the arena and more may appear over time. There is no guarantee that robots will actually find any food.
>
> The behaviour of each robot in the system is represented by the probabilistic state machine in Figure 9, comprising the states: (i) SEARCHING, wherein the robot is searching for food items; (ii) GRABBING, wherein the robot attempts to grab a food item it has found; (iii) DEPOSITING, wherein the robot moves home with the food item; (iv) HOMING, wherein the robot moves home without having found food; and (v) RESTING, wherein the robot rests for a particular time interval."

In this scenario, we assume that all transitions occur with a probability. The state machine in Figure 9 operates on the following probabilities: γ_f (the probability of finding a food item), γ_g (the probability of grabbing a food item), γ_h (the probability of moving to HOMING state), γ_r (the probability of moving to RESTING state), and γ_s (the probability of moving to SEARCHING state).

We can draw this machine using the Drawing PRISM tool as in Figure 4. Here we assume $\gamma_f = 0.3$, $\gamma_g = 0.1$, $\gamma_h = 0.2$, $\gamma_r = 0.6$, and $\gamma_s = 0.5$. Note that this state machine represents the behaviour of an individual robot; however, a robot swarm is a collection of (often) identical robots working together. We therefore need a set of state machines to model the dynamic behaviour of the overall swarm. Fortunately, the Drawing PRISM tool has a feature allowing us to have a multiple copies of a probabilistic model, working as a parallel composition. Using this feature we can model the overall swarm.

The DP tool automatically translates the probabilistic state machine in Figure 4 to the PRISM's high-level modelling language, given in Algorithm 2.

After obtaining the model code, we simulate the model and have a number of simulation traces. We then run the invariant detector tool over these traces to estimate a lower and upper bound for the model variable `s`. Not surprisingly, we obtain the following constraints:

$$\texttt{s} >= 0, \qquad \texttt{s} <= 4. \tag{10}$$

The model therefore will not be adjusted. We remark that, since the model is very intuitive, we could give the exact range of the variable `s` because we know the values thats will take. Daikon has then returned the precise same bounds. However, in realistic cases, users might not be able to know the exact lower and upper bounds, and range estimations might not be realistic. The invariant detector will then provide more realistic bounds and reduce the state space.

We now construct some properties using the Property Generator tool and property patterns provided. Our strategy to build properties is as follows:

(i) first specify the properties that we are interested informally;

(ii) construct the corresponding natural language representations using the Property Generator tool;

(iii) select the target language to which the properties are converted automatically;

(iv) run model checking experiments using the translated formal properties.

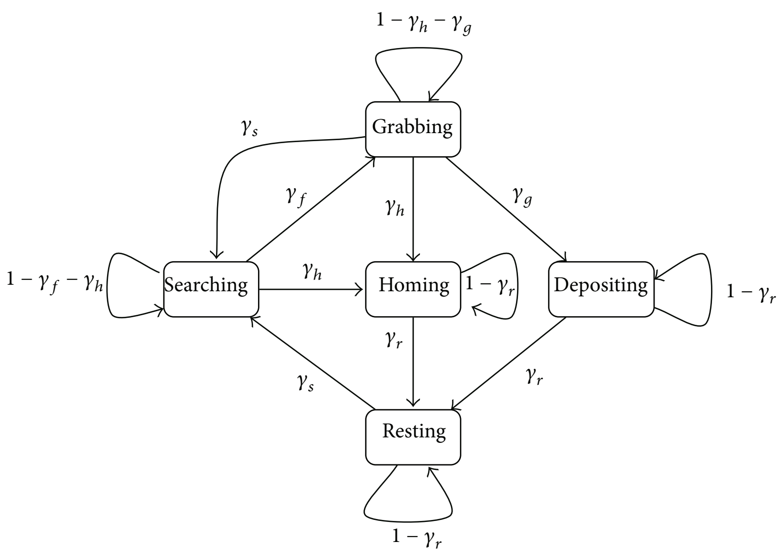


Figure 9: Probabilistic state machine for a foraging robot.

```
dtmc

const int SEARCHING = 0;
const int HOMING = 1;
const int GRABBING = 2;
const int DEPOSITING = 3;
const int RESTING = 4;

module foraging
  s : [0...4] init 0;
[] s = SEARCHING -> 0.5 : (s' = s) + 0.2 : (s' = HOMING) + 0.3 : (s' = GRABBING);
[] s = HOMING -> 0.4 : (s' = s) + 0.6 : (s' = RESTING);
[] s = GRABBING -> 0.2 : (s' = s) + 0.5 : (s' = SEARCHING) + 0.2 : (s' = HOMING)
     + 0.1 : (s' = DEPOSITING);
[] s = DEPOSITING -> 0.4 : (s' = s) + 0.6 : (s' = RESTING);
[] s = RESTING -> 0.5 : (s' = s) + 0.5 : (s' = SEARCHING);
endmodule
```

Algorithm 2

Table 2 shows informal, natural language and formal specifications of some properties. Here, we translate the properties to the Prism's property language. The results of the verification experiments are also illustrated in the table. We remark that a property can be constructed as a query using ? symbol. In this case, Prism returns the corresponding probability result after verifying the query. In Table 2, we show some query samples.

Multiple Robots. Figure 9 corresponds to a single robot behaviour, comprising five states that the robot visits during the execution of the system. However, a foraging swarm contains several robots. To model such a swarm, we can construct a state machine for each robot in the swarm and then take the *product of* all these to provide the behaviour of the overall swarm. To calculate the product, we take a simple and synchronous view. Namely, for each state machine A, say $A_1, A_2, A_3, \ldots$, and the state machine B, say $B_1, B_2, B_3, \ldots$, given that the transitions $A_1 \rightarrow A_2$ are labelled by α and $B_1 \rightarrow B_2$ are labelled by β, we can have a transition $A_1B_1 \rightarrow A_2B_2$ labelled by $\alpha\beta$ where A_1B_1, A_2B_2, and so forth are states in the product state machine and $\alpha\beta$ is a consistent label. This is done for every possible pair of transitions. The overall swarm system as the product of individual robots is shown in Figure 10. The DP tool permits creating multiple instances of modules, allowing composing individual state machines.

Table 3 compares the state spaces for different swarm populations. The table shows that the model size exponentially grows when the swarm size increases. We note that the construction times illustrated in Table 3 correspond to the building models by the model checker. The translation of models into the model checking language is not significantly affected by the size of the model. Namely, the time to translate

Table 2: Sample properties for a single robot.

Property	Informal, Natural language and Prism specifications	Result
1	*The robot eventually grabs food with a probability greater than 0.9* `s = 3` will eventually hold, until then `true` holds with a probability >0.9 `P > 0.9 [true U s = 3]`	TRUE
2	*What is the probability that the robot grabs food within 50 s?* `s = 3` will eventually hold within time bound [0, 50], until then `true` holds with a probability? `P = ? [true U[0, 50] s = 3]`	0.55
3	*What is the probability that the robot searches for food for 50 s. before going to home?* `s = 1` will hold within time bound [0, 50], until then $s = 0$ holds with a probability? `P = ? [s = 0 U[0, 50] s = 1]`	0.39
4	*The robot never goes to home within 50 s* `s! = 1` always holds within time bound [0, 50] with a probability >=1 `P >= 1 [G[0, 50] s! = 1]`	FALSE
5	*The robot does not continuously search for food forever* `s = 0` always holds with a probability <=0 `P <= 0 [G s = 0]`	TRUE
6	*When searching starts, the robot eventually grabs food with a probability greater than 0.6* `s = 0` is always followed by `s = 3` with a probability >0.6 `P > 0.6 [G (s = 0 => P >= 1 [true U s = 3])]`	TRUE
7	*The robot repeats its behaviour forever (e.g. searches for food)* `s = 0` holds infinitely often with a probability >=1 `P >= 1 [G (P >= 1 [true U s = 0])]`	TRUE

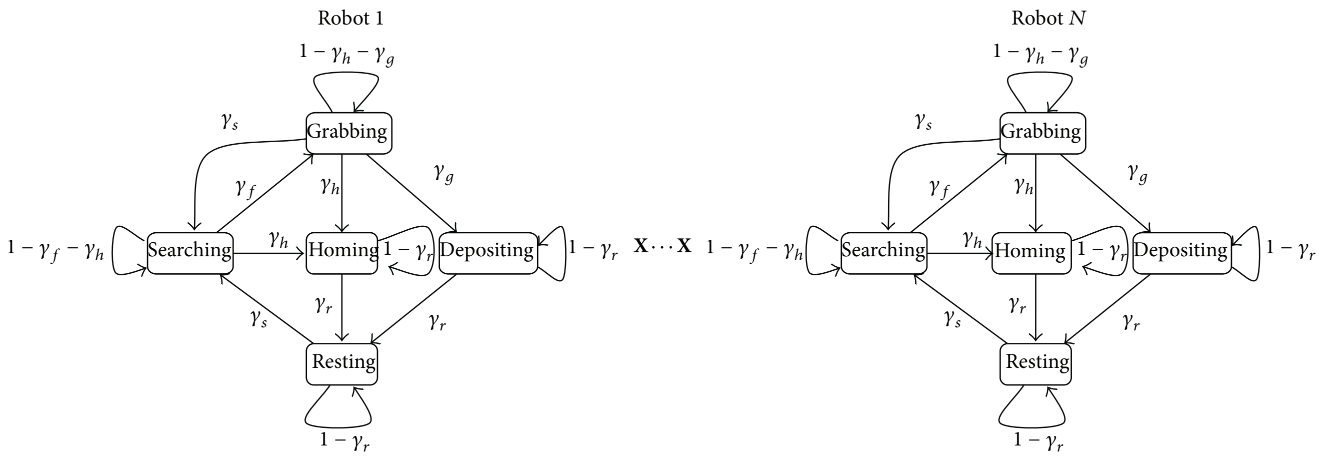


Figure 10: Probabilistic state machine for a foraging swarm.

Table 3: State space for different swarm populations.

Number of robots	States	Transitions	Model construction
1	5	13	0.001 seconds
2	25	105	0.002 seconds
3	125	725	0.003 seconds
4	625	4625	0.006 seconds
5	3125	28125	0.008 seconds
10	9765625	166015625	0.024 seconds
20	95367431640625	3147125244140625	0.733 seconds

a model into the model checking format can be neglected with respect to the resources required for model checking.

7. Discussion

Standard model checking methods work on a basic principle: a model is described in a certain modelling paradigm, for example, a probabilistic model; a property is specified in a certain logic, for example, a probabilistic temporal logic; and a suitable model checker is then used, for example, a probabilistic model checker. This approach suggests that whatever model checker the user selects, he/she has to analyse his/her system based on the aspects of modelling and specification languages. If he/she wants to use a different model checker, the user must start all over again. This, of course, means learning the syntax of all model checkers to be used, which is surely a big cumbersome for nonexperts (even for experts in some cases).

Another drawback is that existing specification languages are limited to a certain dimension, for example, time and probability. This unfortunately prevents specifying and verifying multidimensional behaviour, which can reveal more novel information about system behaviour.

We believe next generation model checkers should be able to do more than standard model checkers and hence should employ some new features in order to eliminate these drawbacks. Here, we discuss two of such features that we can integrate into our methodology.

Generic Model Checking. Although we have presented our approach for probabilistic model checking, it can be generalised to cover model checking for temporal, real-time, and probabilistic temporal logics. The property generator tool we have proposed is very feasible as it supports the translation of natural language statements to any logic. We can also extend the pattern list presented in Table 1 to cover temporal and real-time logics.

As we already know, the semantics of different logics are defined over different structures. For example, temporal, real-time, and probabilistic temporal logics are usually interpreted over Kripke structures, timed automata (TA) [34], and probabilistic models (e.g., DTMCs, CTMCs, and MDPs), respectively. We remind that a timed automaton models real-time behaviour using a finite set of real-valued *clocks* associated with states and transitions.

In order to allow model checking for these logics in the same platform, we therefore need a generic structure to cover all these models. Probabilistic timed automata (PTA) [35] can be used as a generic structure, because PTA extend TA with discrete probability distributions and embed Kripke structures, timed automata, and probabilistic models. We can construct PTA using the Drawing PRISM tool because it already supports labelling the states and transitions. If we allow clock constraints in labels, we can obtain PTA, which can then be instantiated to construct corresponding models. Namely, to construct a probabilistic model we disable clocks (i.e., state and transition labels do not include clock constraints); to construct a timed automaton we disable probability distributions (i.e., transition labels do not include probabilities); and to contract Kripke structures we disable both clocks and probability distributions. Models constructed using the tool can then be translated to the modelling languages of corresponding model checkers. For example, Kripke structures are translated to the PROMELA language for SPIN, timed automata are translated to the UPPAAL's description language, and probabilistic state machines and probabilistic timed automate are translated to the PRISM's reactive modules.

This general approach will allow us to use model checkers for temporal logics, for example, SPIN and NuSMV, real-time model checkers, for example, UPPAAL and KRONOS, and probabilistic model checkers, for example, PRISM and MRMC, in the same platform.

Combined Model Checking. Standard model checking is defined for standalone logics. We can also develop a model checking strategy for *combined logics*. A combined logic synthesizes different aspects using constituent logics, for example, (i) classic temporal logics (CTL, LTL, etc.); (ii) belief/knowledge logics (modal logics KD45, S5, etc.); (iii) probabilistic temporal logics (PCTL, etc.); and (iv) real-time temporal logics (TCTL, etc.). We can then write statements specifying multidimensional behaviour. For example, assume that, in our foraging scenario, we want to combine the belief and probability aspects, which can be done by combining PCTL and KD45. The statement

The robot i believes that the probability of robot j's grabbing a food item within 50 s. is greater than 0.9

is expressed in the combined logic as

$$\mathrm{B}_i \left[\mathrm{P}_{>0.9} \left(\mathrm{true}\, \mathrm{U}^{[0,50]} s_j = 3 \right) \right] \tag{11}$$

which cannot be stated using standard logics.

Based on the combination strategy, a model checking procedure can be defined. In [36] we provide a generic model checking method, which synthesizes a combined model checker from the model checkers of simpler constituent logics. We also "show that the complexity of the synthesized model checker is essentially the supremum of the complexities of the component model checkers" [36].

This result suggests that we can devise a generic model checking platform based on the method proposed in [36]. If we consider combining the aspects discussed above, a probabilistic timed automaton will be sufficient to construct the combined model. In this case, we only need to introduce special tags to distinguish the transitions of component logics. We can then verify properties expressing multidimensional behaviour using component model checkers of constituent logics. In this way, we do not need to implement a new model checker. We can devise the model checking engine based on individual model checkers, for example, SPIN, UPPAAL, and PRISM.

8. Conclusion

In this paper, we have presented a methodology to make probabilistic model checking more intuitive and accessible for nonexpert users. As part of our methodology, we have proposed the Property Generator tool automatically generating formal properties using natural language statements. As for the model construction, we have used the Drawing Prism tool, providing a GUI that allows us to construct probabilistic state machines from which the model code is automatically generated. In this way, nonexperts can model their systems and express their requirements without any knowledge of the modelling and property specification languages and can analyse their systems without having any expertise.

In our methodology, we have also devised a strategy which reduces the state space of probabilistic models, using the Daikon invariant detector, to increase the performance of model checking and relieve the state explosion problem.

At the moment, the tools discussed and presented in this paper are standalone. As future work, we will integrate these tools to create a software suit. We already have the methods for two features discussed in Section 7. They will also be integrated into the toolset.

Conflict of Interests

The author declares no conflict of interests regarding the publication of this paper.

Acknowledgments

This work is partially supported by the EPSRC (UK) funded ROADBLOCK Project, a joint Project with Dr. M. Gheorghe at the University of Sheffield (EP/I031812/1), Dr. N. Krasnogor at the University of Nottingham (EP/I031642), and Dr. S. Kalvala at the University of Warwick (EP/I03157X).

References

[1] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 1999.

[2] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, " Nusmv : a new symbolic model verifier," in *Proceedings of International Conference on Computer-Aided Verification (CAV '99)*, pp. 495–499, Trento, Italy, 1999.

[3] G. J. Holzmann, *The Spin Model Checker*, Addison-Wesley, Boston, Mass, USA, 2003.

[4] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL—a tool suite for automatic verification of real time systems," in *Proceedings of Workshop on Verification and Control of Hybrid Systems III*, vol. 1066 of *Lecture Notes in Computer Science*, pp. 232–243, Springer, New York, NY, USA, 1995.

[5] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: a model-checking tool for realtime systems," in *Proceedings of the 10th International Conference on Computer Aided Verification (CAV '98)*, pp. 546–550, Springer, New York, NY, USA, 1998.

[6] S. Konur, M. Fisher, S. Dobson, and S. Knox, "Formal verification of a pervasive messaging system," *Formal Aspects of Computing*, vol. 2013, 18 pages, 2013.

[7] M. Kwiatkowska, G. Norman, and D. Parker, "Quantitative analysis with the pro babilistic model checker PRISM," *Electronic Notes in Theoretical Computer Science*, vol. 153, no. 2, pp. 5–31, 2006.

[8] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: a tool for automatic verification of probabilistic systems," in *Tools and Algorithms for the Construction and Analysis of Systems: Proceedings of the 12th International Conference, TACAS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25–April 2, 2006*, vol. 3920 of *Lecture Notes in Computer Science*, pp. 441–444, Springer, Berlin, Germany, 2006.

[9] J. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen, "The ins and outs of the probabilistic model checker MRMC," *Performance Evaluation*, vol. 68, no. 2, pp. 90–104, 2011.

[10] H. L. Younes, "Ymer: a statistical model checker," in *Computer Aided Verification*, vol. 3576 of *Lecture Notes in Computer Science*, pp. 429–433, Springer, Berlin, Germany, 2005.

[11] K. Sen, M. Viswanathan, and G. Agha, "On statistical model checking of stochastic systems," in *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, vol. 3576 of *Lecture Notes in Computer Science*, pp. 266–280, Springer, 2005.

[12] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the International Conference on Software Engineering (ICSE '99)*, pp. 411–420, ACM, May 1999.

[13] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal Aspects of Computing*, vol. 6, no. 5, pp. 512–535, 1994.

[14] C. Baier, B. Haverkort, H. Hermanns, and J. Katoen, "Model-checking algorithms for continuous-time Markov chains," *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 524–541, 2003.

[15] PRISM, *Probabilistic Symbolic Model Checker*, 2013, http://www.prismmodelchecker.org/.

[16] M. Kwiatkowska, G. Norman, and D. Parker, "Prism: probabilistic model checking for performance and reliability analysis," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 40–45, 2009.

[17] MRMC, "Markov Reward Mmodel Checker, Version 1.5," 2011, http://www.mrmc-tool.org/downloads/MRMC/Specs/MRMC_Manual_1.5.pdf.

[18] A. Legay, B. Delahaye, and S. Bensalem, "Statistical model checking: an overview," in *Runtime Verification*, vol. 6418 of *Lecture Notes in Computer Science*, pp. 122–135, Springer, 2010.

[19] F. Zhang, *A swarm-checker, a robot swarm front-end for PRISM [M.S. thesis]*, Department of Computer Science, University of Liverpool, Liverpool, UK, 2010.

[20] A. Ramirez, P. Vanpeperstraete, A. Rueckert et al., *ArgoUML User Manual*, 2010, http://argouml-stats.tigris.org/documentation/manual-0.30/.

[21] M. D. Ernst, J. H. Perkins, P. J. Guo, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 35–45, 2007.

[22] F. Bernardini, M. Gheorghe, F. J. Romero-Campero, and N. Walkinshaw, "A hybrid approach to modeling biological systems," in *Membrane Computing*, vol. 4860 of *Lecture Notes in Computer Science*, pp. 138–159, Springer, Berlin, Germany, 2007.

[23] M. Gheorghe, F. Ipate, R. Lefticaru, and C. Dragomir, "An integrated approach to P systems formal verification," in *Membrane Computing*, vol. 6501, pp. 226–239, Springer, Berlin, Germany, 2011.

[24] R. Lefticaru, F. Ipate, L. Valencia-Cabrera et al., "Towards an integrated approach for model simulation, property extraction and verification of P systems," in *Proceedings of the 10th Brainstorming Week on Membrane Computing*, vol. 1, pp. 291–318, Sevilla, Spain, January 2012.

[25] V. Gruhn and R. Laue, "Patterns for timed property specifications," *Electronic Notes in Theoretical Computer Science*, vol. 153, no. 2, pp. 117–133, 2006.

[26] S. Konrad and B. Cheng, "Real-time specification patterns," in *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pp. 372–381, St. Louis, Mo, USA, May 2005.

[27] P. Bellini, P. Nesi, and D. Rogai, "Expressing and organizing real-time specification patterns via temporal logics," *Journal of Systems and Software*, vol. 82, no. 2, pp. 183–196, 2009.

[28] L. Grunske, "Specification patterns for probabilistic quality properties," in *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pp. 31–40, ACM, May 2008.

[29] J. Blakes, J. Twycross, S. Konur, F. Romero-Campero, N. Krasnogor, and M. Gheorghe, "Infobiotics workbench: A p systems based tool for systems and synthetic biology," in *Applications of Membrane Computing in Systems and Synthetic Biology*, vol. 7 of *Emergence, Complexity and Computation*, pp. 1–41, Springer, 2014.

[30] C. Dragomir, F. Ipate, S. Konur, R. Lefticaru, and M. Laurentiu, "Model checking Kernel P systems," in *Proceedings of the 14th International Conference on Membrane Computing (CMC '13)*, pp. 131–152, Chisinau, Moldova, 2013.

[31] W. Liu, A. Winfield, and J. Sa, "Modelling swarm robotic systems: a study in collective foraging," in *Proceedings of the Towards Autonomous Robotic Systems (TAROS '07)*, pp. 25–32, 2007.

[32] S. Konur, C. Dixon, and M. Fisher, "Formal verification of probabilistic swarm behaviours," in *Swarm Intelligence*, vol. 6234 of *Lecture Notes in Computer Science*, pp. 440–447, Springer, Berlin, Germany, 2010.

[33] S. Konur, C. Dixon, and M. Fisher, "Analysing robot swarm behaviour via probabilistic model checking," *Robotics and Autonomous Systems*, vol. 60, no. 2, pp. 199–213, 2012.

[34] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking in dense real-time," *Information and Computation*, vol. 104, no. 1, pp. 2–34, 1993.

[35] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston, "Automatic verification of real-time systems with discrete probability distributions," *Theoretical Computer Science*, vol. 282, no. 1, pp. 101–150, 2002.

[36] S. Konur, M. Fisher, and S. Schewe, "Combined model checking for temporal, probabilistic, and real-time logics," *Theoretical Computer Science*, vol. 503, pp. 61–88, 2013.