DOI 10.1007/s00165-012-0227-6
BCS © 2012
Formal Aspects of Computing (2012) 24: 661–670

Formal Aspects of Computing

Probabilistic verification of Herman's self-stabilisation algorithm

Marta Kwiatkowska[1], Gethin Norman[2] and David Parker[1]

[1] Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford OX1 3QD, UK.
E-mail: david.parker@cs.ox.ac.uk
[2] School of Computing Science, University of Glasgow, Glasgow, UK

Abstract. Herman's self-stabilisation algorithm provides a simple randomised solution to the problem of recovering from faults in an N-process token ring. However, a precise analysis of the algorithm's maximum execution time proves to be surprisingly difficult. McIver and Morgan have conjectured that the worst-case behaviour results from a ring configuration of three evenly spaced tokens, giving an expected time of approximately $0.15N^2$. However, the tightest upper bound proved to date is $0.64N^2$. We apply probabilistic verification techniques, using the probabilistic model checker PRISM, to analyse the conjecture, showing it to be correct for all sizes of the ring that can be exhaustively analysed. We furthermore demonstrate that the worst-case execution time of the algorithm can be reduced by using a biased coin.

Keywords: Self-stabilisation, Herman's algorithm, Probabilistic model checking

1. Introduction

Self-stabilisation [Dij74] is an approach for designing fault-tolerant systems in a distributed setting. A self-stabilisation algorithm for a network of processes is a protocol that, starting from an arbitrary initial configuration, returns to a "stable" (legal) configuration within a finite number of steps and without any outside intervention. It is designed to allow the system to recover from transient faults, such as those caused by the states of individual processes becoming corrupted.

In this paper, we study the self-stabilisation algorithm of Herman [Her90], designed for a token ring of N synchronous processes. The algorithm is randomised and guarantees that the system eventually self-stabilises with probability 1. Another important property of the algorithm is the *expected time* required for stabilisation and, in particular, the *worst-case* expected time from an arbitrary system configuration. Despite the simplicity of Herman's algorithm, a precise analysis of this property turns out to be surprisingly difficult. An upper bound of $O(N^2)$ has been proven by several people [MM05, Nak05, FMP06] and recent work in [KMO+11] established the tightest known upper bound of approximately $0.64N^2$. McIver and Morgan [MM05] also provided an exact expression for the expected stabilisation time from any configuration containing three tokens and posed an interesting conjecture that the case of three tokens spaced evenly around the ring always yields the worst-case behaviour of the algorithm. However, a proof of this conjecture remains elusive.

Correspondence and offprint requests to: D. Parker, E-mail: david.parker@cs.ox.ac.uk

We analyse Herman's algorithm using *probabilistic model checking*, which provides a way to formally specify and verify quantitative system properties, based on the construction and analysis of a finite-state probabilistic model. We use the probabilistic model checker PRISM [KNP11] to perform an exhaustive analysis of Herman's algorithm on all system configurations up to a ring size of $N = 21$. We show that, in all these cases, the conjecture of McIver and Morgan does indeed hold. Furthermore, we then use PRISM to study the performance of the algorithm when using a biased coin, rather than a fair coin. Firstly, we discover that, in this case, three evenly spaced tokens does *not* always result in worst-case behaviour. More interestingly, we also find that, by selecting an appropriate coin bias, we can actually improve the performance of the algorithm, by reducing the worst-case expected stabilisation time.

Related Work. The problem of designing self-stabilising systems has received considerable attention; see for example [Dol00] for a detailed overview. Other self-stabilisation algorithms include those of Israeli and Jalfon [IJ90] and Beauquier, Gradinariu and Johnen [BGJ99]; however, both of these are designed for rings of asynchronous processes, rather than synchronous ones. A general approach for studying dependability properties of self-stabilisation algorithms, based on metrics that can be evaluated using probabilistic model checking was presented in [DTC+09].

As stated earlier, Herman's algorithm has been studied by several people. McIver & Morgan [MM05], in addition to making the conjecture described above, presented an elementary proof of the $O(N^2)$ upper bound and related this to more general notions of abstraction for probabilistic programs [MM04]. Fribourg et al. [FMP06] proved the same upper bound, but using the notion of coupling from probability theory. Nakata, on the other hand, derived a more precise bound (of approximately $0.93N^2$) based on techniques for the analysis of coalescing random walks. Most recently, Kiefer et al. [KMO+11] applied techniques from annihilating particle systems to show an upper bound of approximately $0.64N^2$. They also show that stabilisation is significantly faster for specific classes of faults and study an asynchronous version of the algorithm. Finally, probabilistic model checking of Herman's algorithm is used as an example in [KNP05], but only for a subset of the properties considered here, and without studying the effect of biased coins.

2. Herman's algorithm

Herman's self-stabilisation algorithm assumes a network of N identical processes in a ring, indexed 1 to N and ordered anticlockwise. The protocol operates synchronously, the ring is oriented and N must be odd. Processes can possess *tokens*, which are passed unidirectionally around the ring. At every step of the algorithm, each process with a token decides whether to keep it or pass it on to its left neighbour, with the decision being based on the outcome of a random coin toss. When two tokens meet (i.e. are held by the same process), they are both eliminated. Thus, if the number of tokens is initially odd, it always remains odd. A configuration of the ring is said to be *stable* if exactly one process has a token. Once a stable configuration is reached, the token should be passed around the ring forever in a fair manner.

Following the presentation of Herman [Her90], the algorithm can be implemented using a single bit for each process. More precisely, the state of process i is represented by a bit x_i and $i \ominus 1$ denotes the index of process i's left (clockwise) neighbour. Process i is said to have a token if its bit x_i equals $x_{i\ominus 1}$. At each step of the algorithm, process i checks the values of x_i and $x_{i\ominus 1}$ for equality. If the values differ, then process i inverts its bit, making it the same as its neighbour ($x_i := x_{i\ominus 1}$); if the values are the same, then process i randomly sets x_i to either 0 (with probability p) or 1 (with probability $1-p$).

Figure 1 shows an example of two successive steps of the algorithm for $N = 9$ processes. We use x_i and $\overline{x}_i$ to indicate that process i's bit is 1 or 0, respectively, and draw shaded grey circles to indicate processes that have tokens. Initially, processes 1, 4 and 6 have tokens; thus, each of the other six processes set their bit equal to their left neighbour's, while processes 1, 4 and 6 set their bits randomly. In the example, they select values 0, 1 and 0, respectively. The resulting configuration is shown in the centre of Fig. 1. Now, processes 1, 4 and 5 have tokens, i.e. process 6 has passed its token to process 5. In the second step, processes 1, 4 and 5 randomly select values 1, 0 and 1, respectively. This means that process 4 keeps its token, but process 5's token is passed to the left, resulting in the elimination of both them. In the subsequent (stable) configuration, only process 1 has a token.

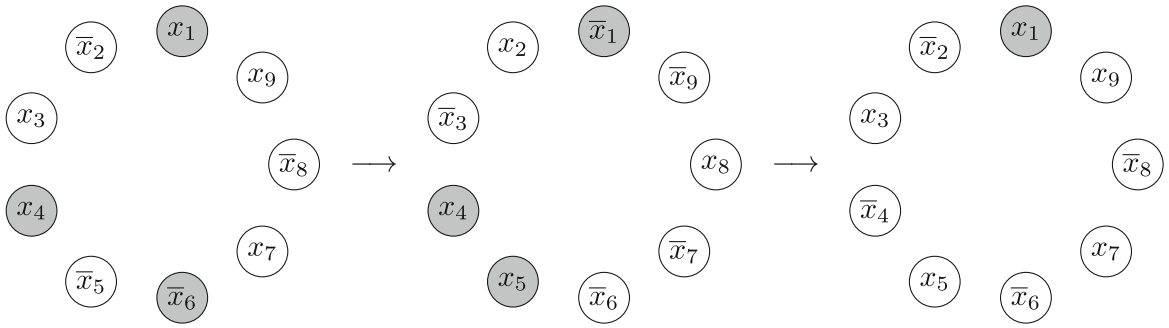


Fig. 1. Example successive configurations of Herman's ring for $N = 9$; we write x_i and $\overline{x}_i$ to denote the fact that bit x_i is 1 or 0, respectively; processes with a token are shaded *grey*

3. Probabilistic model checking

Probabilistic model checking is an approach for formally verifying systems with stochastic behaviour. It works by constructing and analysing a probabilistic model of the system. In this paper, we assume that models are *discrete-time Markov chains* (DTMCs), which specify the probability of making a transition from one system state to another. Formally, we have the following definition.

Definition 1 (Discrete-time Markov chain) A *discrete-time Markov chain* (DTMC) is a tuple $\mathsf{D} = (S, \bar{S}, \mathbf{P}, L)$, where S is a (finite) set of states, $\bar{S} \subseteq S$ is a set of possible initial states, $\mathbf{P} : S \times S \to [0, 1]$ is a transition probability matrix satisfying $\sum_{s' \in S} \mathbf{P}(s, s') = 1$ for all $s \in S$, and $L : S \to 2^{AP}$ is a labelling function mapping each state to a set of atomic propositions taken from a set AP.

An execution of a DTMC $\mathsf{D} = (S, \bar{S}, \mathbf{P}, L)$ is represented by a *path*, which is an infinite sequence of states $\omega = s_0 s_1 s_2 \ldots$ such that $\mathbf{P}(s_i, s_{i+1}) > 0$ for all $i \geq 0$. To reason about the likelihood of certain events occurring, we define, in standard fashion [KSK76], a probability measure $\Pr_s$ over the set of all paths starting from a given state s. Events of interest are then defined as measurable sets of paths.

We formally specify properties to be verified against DTMCs using probabilistic temporal logics such as PCTL [HJ94] or PCTL* [ASB+95]. A key element of these logics is the probabilistic operator $\mathtt{P}$. A formula $\mathtt{P}_{\bowtie p}[\psi]$, where $\bowtie \in \{\leq, <, \geq, >\}$, p is a probability and ψ is a path formula, asserts that the probability of ψ being true (starting from some state s) satisfies the bound $\bowtie p$. Typical examples are:

- $\mathtt{P}_{\geq 1}[\mathtt{F}$ "*terminate*"$]$ – "the algorithm eventually terminates with probability 1";
- $\mathtt{P}_{\geq 0.98}[\,\mathtt{G}^{\leq 60}\,\neg$"*fail*"$]$ – "with probability at least 0.98, no failures occur within the first 60 steps".

We also use *rewards* (or, equivalently, *costs*) to specify additional system properties, such as those concerning resource usage. A *reward structure* $r : S \times S \to \mathbb{R}_{\geq 0}$ augments the transitions of a DTMC with (non-negative) reward values. We then specify properties using an $\mathtt{R}$ operator [KNP07], which takes a similar form to the $\mathtt{P}$ operator from above. This refers to the *expected* amount of reward accumulated over a system's execution. For example, using a simple reward structure $time$ that assigns a reward of 1 to all transitions, we have:

- $\mathtt{R}^{time}_{<20}[\mathtt{F}$ "*terminate*"$]$ – "the expected termination time of the algorithm is less than 20 s";

For both the $\mathtt{P}$ and $\mathtt{R}$ operators, we also adopt a *quantitative* (numerical) form, which simply returns the actual probability or expected reward value, rather than comparing it to a specified threshold:

- $\mathtt{P}_{=?}[\mathtt{F}$ "*terminate*"$]$ – "what is the probability of the algorithm eventually terminating?";
- $\mathtt{R}^{time}_{=?}[\mathtt{F}$ "*terminate*"$]$ – "what is the expected time taken for the algorithm to terminate?".

Model checking for the properties given above reduces to a combination of graph-based algorithms (e.g., determining the set of states from which a target set can be reached) and numerical computation (typically solving linear equation systems). See, for example, [KNP07, BK08] for details.

4. Probabilistic verification of Herman's algorithm

We now describe how to model and analyse Herman's algorithm using probabilistic model checking. In particular, we will use the probabilistic model checker PRISM [KNP11]. We first show how the algorithm can be modelled and then describe the verification of various properties, focusing first on the correctness of the algorithm and then its performance, specifically its expected execution time. Finally, we use PRISM to evaluate the influence that a biased coin has on the performance of the algorithm.

4.1. Modelling

It is straightforward to model Herman's self-stabilisation algorithm as a DTMC. For N processes, and assuming that the algorithm works with a fair coin (i.e. $p = \frac{1}{2}$), we use a DTMC $\mathsf{D} = (S, \bar{S}, \mathbf{P}, L)$ where:

- $S = \{0, 1\}^N$, i.e. each state is a vector $\underline{x} = (x_1, \ldots, x_N)$, where the ith element corresponds to the value of process i's bit;
- $\bar{S} = S$, i.e. the system can start in any of its possible states;
- $\mathbf{P} : S \times S \to [0, 1]$ is such that, for any states $\underline{x} = (x_1, \ldots, x_N)$ and $\underline{y} = (y_1, \ldots, y_N)$ we have $\mathbf{P}(\underline{x}, \underline{y}) = \prod_{i=1}^{N} \delta((x_{i\ominus 1}, x_i), y_i)$ where, for $x, x', y \in \{0, 1\}$:

$$\delta((x', x), y) = \begin{cases} 1 & \text{if } x \neq x' \quad \text{and} \quad y = x' \\ \frac{1}{2} & \text{if } x = x' \\ 0 & \text{otherwise;} \end{cases}$$

- the set of atomic propositions is $AP = \{token_1, \ldots, token_N, stable\}$ and, for any state $\underline{x} = (x_1, \ldots, x_N)$, we have $token_i \in L(\underline{x})$ if $x_{i\ominus 1} = x_i$ and $stable \in L(\underline{x})$ if exactly one of $\{token_1, \ldots, token_N\}$ is in $L(\underline{x})$.

Returning to the example steps of the algorithm shown in Fig. 1 (where $N = 9$), the corresponding path fragment of the DTMC is (101100101)(010110010)(101010101) where:

$$\begin{aligned} \mathbf{P}((101100101), (010110010)) &= \tfrac{1}{2}\cdot 1\cdot 1\cdot \tfrac{1}{2}\cdot 1\cdot \tfrac{1}{2}\cdot 1\cdot 1\cdot 1 = \tfrac{1}{8} \\ \mathbf{P}((010110010), (101010101)) &= \tfrac{1}{2}\cdot 1\cdot 1\cdot \tfrac{1}{2}\cdot \tfrac{1}{2}\cdot 1\cdot 1\cdot 1\cdot 1 = \tfrac{1}{8}. \end{aligned}$$

We next introduce a PRISM language description for this DTMC model of Herman's algorithm. Models in PRISM are described in a high-level language based on guarded commands. The basic components of the modelling language are *modules* and *variables*. A system model is defined by specifying a set of modules, each of whom's state is represented by a finite number of variables. The behaviour of each module is given by a set of guarded commands of the form:

[<action>] <guard> → <prob> : <update> + ⋯ + <prob> : <update>;

The action label is used to force modules to synchronise (i.e. execute their commands simultaneously) and the guard is a predicate over all the variables of the model, indicating when the command is enabled. The updates describe the probabilistic transitions that the module can make when the command is executed, i.e. the changes made to its own variables; primed variables indicate the next values of variables.

A PRISM model of Herman's algorithm, for the case $N = 5$, is shown in Fig. 2 (left). We have one module for each process, and one variable for each module. Since the algorithm is synchronous, each command is labelled with the same action *step*, ensuring that all modules synchronise at each stage of the algorithm. The model description uses *module renaming* since the modules of processes $2, \ldots, 5$ are identical in structure to the module for process 1. The fact that all possible system configurations can be initial states is expressed using the `init ... endinit` construct. The model includes a reward structure *time*, which we will use to reason about the expected execution time of the algorithm. This is measured in terms of the number of steps performed, so it assigns a reward of 1 to every transition of the model. We also include a formula that calculates the number of tokens in a state, and then use this to define a label (an atomic proposition) for the stable configurations, i.e. states in which there is precisely one token. Figure 2 also shows the size of the resulting DTMC when constructed by PRISM, for varying N.

```
dtmc

const double p = 0.5;

module process1

    x1 : [0..1];

    [step] (x1=x5) → p : (x1'=0) + 1 − p : (x1'=1);
    [step] !(x1=x5) → (x1'=x5);

endmodule

module process2 = process1 [ x1=x2, x5=x1 ] endmodule
module process3 = process1 [ x1=x3, x5=x2 ] endmodule
module process4 = process1 [ x1=x4, x5=x3 ] endmodule
module process5 = process1 [ x1=x5, x5=x4 ] endmodule

// Set of initial states: all possible configurations
init true endinit

// Reward structure for time (number of steps)
rewards "time"
    [step] true : 1;
endrewards

// Formula/label for use in properties
formula num_tokens = (x1=x5?1:0) + (x2=x1?1:0)+
             (x3=x2?1:0) + (x4=x3?1:0) + (x5=x4?1:0);
label "stable" = (num_tokens=1);
```

N	States	Transitions
3	8	28
5	32	244
7	128	2,188
9	512	19,684
11	2,048	177,148
13	8,192	1,594,324
15	32,768	14,348,908
17	131,072	129,140,164
19	524,288	1,162,261,468
21	2,097,152	10,460,353,204

Fig. 2. *Left* PRISM code for $N = 5$; *right* statistics showing model size as N varies

Table 1. Summary of the PRISM properties used in the paper

Name	PRISM notation	Meaning
"*terminate*"	$\mathtt{P}_{\geq 1}[\mathtt{F}$ "*stable*"$]$	A stable configuration is eventually reached with probability 1
"*fairness*"	$\mathtt{P}_{\geq 1}[\wedge_{i=1}^{N}(\mathtt{G}\,\mathtt{F}$ "*token*$_i$"$)]$	Each process gets a token infinitely often with probability 1
"*exp_time*"	$\mathtt{R}^{time}_{=?}[\mathtt{F}$ "*stable*"$]$	Expected time (number of steps) until stabilisation
"*worst_time*"	*filter*(*max*, $\mathtt{R}^{time}_{=?}[\mathtt{F}$ "*stable*"$]$, "init")	Worst-case expected time until stabilisation, from any initial configuration
"*worst_time_k*"	*filter*(*max*, $\mathtt{R}^{time}_{=?}[\mathtt{F}$ "*stable*"$]$, *num_tokens* $= k$)	Worst-case expected time until stabilisation, from any initial configuration with k tokens
"*worst_states*"	*filter*(*argmax*, $\mathtt{R}^{time}_{=?}[\mathtt{F}$ "*stable*"$]$, "init")	True if a state results in the worst-case expected time until stabilisation
"*worst_tokens*"	*filter*(*range*, *num_tokens*, "*worst_states*")	Range of values of the number of tokens in states satisfying property

4.2. Correctness

We begin by verifying that the protocol is correct, namely that, from any initial state, the ring eventually reaches a stable configuration with probability 1. In fact, it is relatively straightforward to prove that this property holds for any value of N (and any value of p) [Her90]. However, for completeness, we illustrate how this can checked using PRISM (for fixed N). The required property is given by:

$\mathtt{P}_{\geq 1}[\mathtt{F}$ "*stable*"$]$

where ; as shown in Fig. 2, identifies states of the model with one token. This property, along with the others that we will use in this section are also summarised in Table 1 for reference.

Since this a *qualitative* property (meaning that the probability bound is 0 or 1), model checking reduces to an analysis of the underlying graph of the DTMC, i.e., the exact probability values labelling its transitions are not important. This makes verification relatively efficient (compared to properties that also require numerical com-

putation). Furthermore, PRISM's *symbolic* implementation (using binary decision diagrams) allows the analysis to be performed for a large range of values of N. For example, in the case where $N = 31$, and the DTMC has $2^{31} = 2,147,483,648$ states, the model can be built and verified in under a second.

We also consider a second correctness property, namely that, after a stable configuration is reached, the token is passed around the ring in a fair manner. Since we already know that a stable configuration is reached with probability 1, to verify this property, it suffices to check that (from any possible initial state) each process gets a token infinitely often with probability 1. Formally this can be expressed by:

$$\mathtt{P}_{\geq 1}[\wedge_{i=1}^{N}(\mathtt{G}\,\mathtt{F}\text{ ``}token_i\text{''})]$$

where ìs true for states in which $x_i = x_{i\ominus 1}$.

As already mentioned, for qualitative properties such as the ones given here, the actual probability values of the DTMC are not used during the analysis. Therefore, although we have to specify a fixed value of the probability p to PRISM, it in fact follows from the definition of the model that checking for one value of $p \in (0, 1)$ implies that the above two properties hold for all values of $p \in (0, 1)$.

4.3. Execution time

Next, we consider a more interesting property of Herman's algorithm: the expected amount of time required for its execution, i.e. the expected number of steps required until stabilisation occurs. From a fixed initial configuration of the ring, this corresponds to the PRISM query:

$$\mathtt{R}^{time}_{=?}[\,\mathtt{F}\text{ ``}stable\text{''}],$$

where *time* denotes the reward structure shown in Fig. 2 that assigns 1 to every transition of the model. Unlike the formulas used in the previous section, this is a *quantitative* property, whose verification requires numerical computation to be performed; here, this amounts to the solution of a linear equation system.

Of particular interest is the *worst-case* expected stabilisation time, from any possible initial configuration. We can check this with PRISM in a similar fashion. In fact, computationally, this is no more expensive than the previous property, since verification of that property proceeds by computing the expected time from all states of the model. In PRISM's property specification language, the required query uses a *filter*:

$$filter(max, \mathtt{R}^{time}_{=?}[\mathtt{F}\text{ ``}stable\text{''}], \text{``}\mathtt{init}\text{''}).$$

A filter takes the form $filter(\mathtt{op}, \mathtt{prop}, \mathtt{states})$, where op is a (commutative and associative) operator, prop is an (arbitrary) PRISM property and states is a Boolean-valued PRISM property. The filter applies operator op to values of prop over all states of the model that satisfy states.

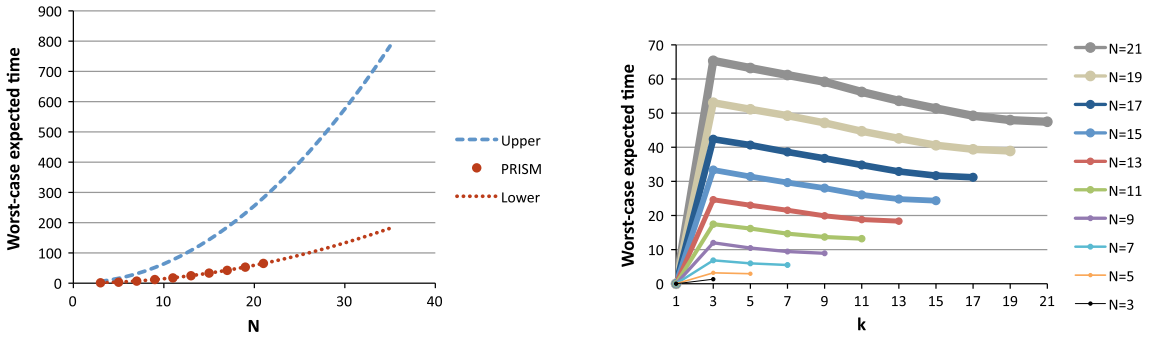
A *lower* bound on the worst-case expected time for Herman's algorithm can be established from the results of McIver and Morgan [MM05]. They show that, for an initial configuration containing 3 tokens, the expected stabilisation time is $4abc/N$, where a, b and c are the distances between the three tokens. The worst three-token configuration, i.e. the one from which the execution time is the highest, is where the tokens are distributed evenly (or as evenly as possible) throughout the ring. In this case, since $a+b+c = N$, the execution time is $\frac{4}{27}N^2 \approx 0.15N^2$, which provides us with a lower bound on the worst-case expected time. McIver and Morgan also conjecture that this scenario actually constitutes the worst case over *all* possible initial configurations. However, a proof of this statement remains elusive.

The tightest *upper* bound on the worst-case time provided so far comes from recent work by Kiefer et al. [KMO+11], who prove a bound of $(\frac{\pi^2}{2} - \frac{116}{27})N^2 \approx 0.64N^2$. In Fig. 3a, we show results obtained from PRISM using the query given above, for values of N between 3 and 21, along with the lower and upper bounds from the results of [MM05] and [KMO+11], respectively. Notice that the worst-case times coincide precisely with the lower bound, corroborating the conjecture of [MM05].

The fact that worst-case expected times always result from configurations with three tokens (for N=3, ..., 21) is more clearly illustrated using the following query:

$$filter(max, \mathtt{R}^{time}_{=?}[\mathtt{F}\text{ ``}stable\text{''}], num\_tokens{=}k)$$

which gives the maximum expected time from k-token configurations. Fig. 3b shows the results as the value of k varies.



(a) Worst-case expected times from PRISM; with lower and upper bounds from [MM05] and [KMO+11], respectively.

(b) Worst-case expected times, over initial configurations containing k tokens

Fig. 3. Worst-case expected time for self-stabilisation

Lastly, we note that, since PRISM performs an exhaustive search and analysis of the models it constructs, it can also report precisely which states result in the worst-case times. For all models analysed, these states are the ones proposed in [MM05], i.e. those in which the three tokens are evenly spaced around the ring.

4.4. Biased coins

To complete our analysis of Herman's algorithm, we consider how its performance varies for different values of the probability p, used to randomise the processes' choices at each step. Herman's original presentation of the algorithm includes p as a parameter, but the analysis of expected time assumes a fair coin (i.e. $p = 0.5$). We will show in this section that the worst-case performance can actually be improved using a biased coin.

Some previous analyses of Herman's algorithm do consider variations in p, including [KMO+11]. However, there is a subtlety to be aware of in the way that biased coins are introduced. Our presentation of the algorithm, and our corresponding model in PRISM, follow the original description of Herman [Her90]: when process i has a token (i.e. $x_i = x_{i\ominus 1}$), he sets his bit x_i to 0 with probability p and to 1 with probability $1 - p$. If $p = 0.5$, as used in the analysis of [Her90], then an equivalent interpretation of the algorithm is as follows: when process i has a token, he passes it to his neighbour with probability p; if this results in a process possessing two tokens, then both are removed. We will refer to these two interpretations of the algorithm as the "random-bit" model and "random-pass" model, respectively.

In the case where $p \neq 0.5$, however, these two interpretations do *not* coincide. When using the "random pass" model, in each state where a process has a token, the probability of him passing it to his neighbour is fixed and equals probability p. On the other hand, when using the "random-bit" model, his actions depend on both the relationship between his bit and his neighbour's (i.e. whether he has a token) and the actual current value of his bit. For example, in the case where a process's bit does equal that of his neighbour, the probability of him passing his token is:

$$\mathit{Prob}[\text{passes token}] = \begin{cases} (1-p)^2+p^2 & \text{if left neighbour has a token} \\ p & \text{if left neighbour does not have a token and current bit value is 1} \\ 1-p & \text{if left neighbour does not have a token and current bit value is 0.} \end{cases}$$

From the above, it is clear that, for $p = 0.5$, the probability of passing a token is fixed at 0.5; however, for any other value of $p \in (0, 1)$, the three probability values in the expression are distinct. It is also worth noting that, for any $p \neq 0.5$, when its left neighbour has a token, the probability of passing on the token is strictly greater than 0.5.

Our analysis will focus mainly on the "random bit" model (which is the one shown in the PRISM code in Fig. 2), however, we briefly also consider the "random pass" version, which is the one studied in [KMO+11]. Our PRISM model is easily adapted to this version by replacing the first command in module *process1* with:

$$[\mathit{step}]\ (x1 = x5) \rightarrow p : (x1' = 1 - x1) + 1 - p : (x1' = x1);$$

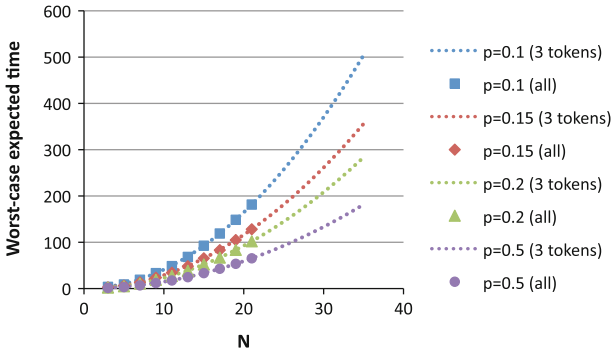
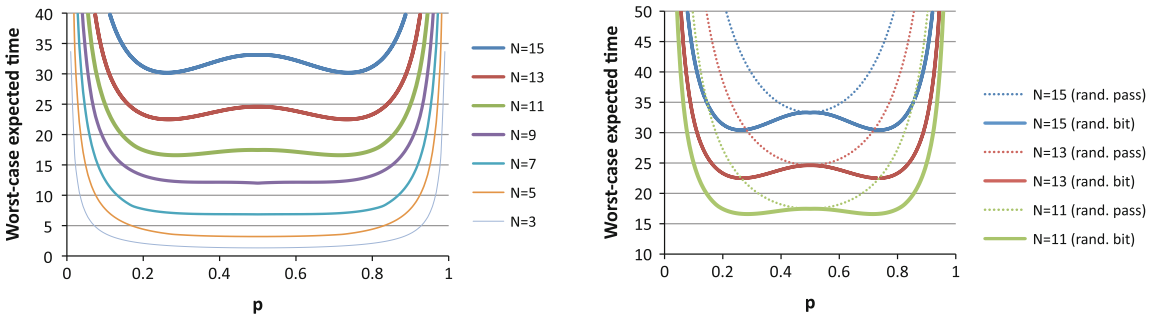


Fig. 4. Worst-case expected times in the "random pass" model for various values of p; individual points are obtained from PRISM and *dotted lines* show times for the worst three-token configuration, from [KMO+11]



(a) Worst-case expected time for $N = 3, \ldots, 15$ (assuming the "random bit" model).

(b) Worst-case expected time for both versions ("random bit" and "random pass"), where $N = 11, \ldots, 15$.

Fig. 5. Worst-case expected time for varying probability p

The analysis of [KMO+11] generalises the expression given in [MM05] for the expected time required from three-token configurations to $abc/p(1-p)N$, where a, b and c are the distances between the tokens. In Fig. 4, we plot the worst-case expected times over all configurations of the "random pass" model, computed by PRISM, against the worst-case expected timed for a three-token configuration, obtained from the expression above. It appears that, as for the case where $p = 0.5$, a three-token configuration again always results in the worst-case behaviour.

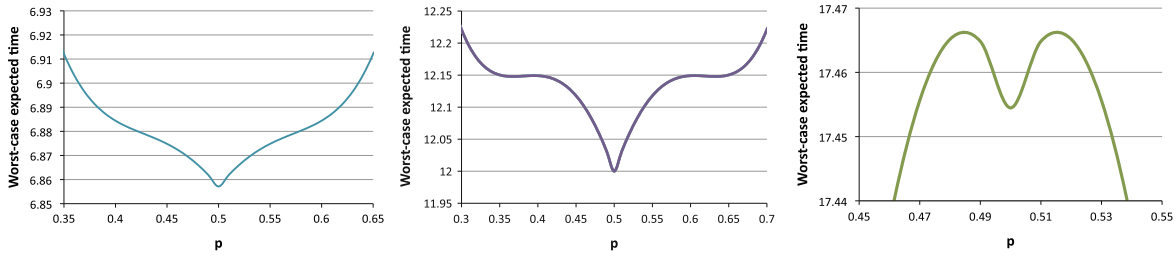
Now, we consider the "random bit" model. Figure 5a shows the worst-case expected time over all configurations as p varies and for a range of values of N. Interestingly, we observe that, for $N \leq 9$, setting p to 0.5 results in the minimum worst-case expected time, but for larger values of N, this is not the case. In fact, for $N > 9$, we can actually improve the worst-case behaviour of the algorithm by using a biased coin with an appropriate value of p. By contrast, in the "random pass" version of the algorithm, the worst-case time is always minimised by choosing $p = 0.5$ and is higher than for the "random bit" model. Fig. 5b illustrates this comparison for ring sizes 11, 13 and 15.

Also, whereas the worst-case time for the "random pass" model appears to be given by the simple expression $abc/p(1-p)N$ (i.e. assuming that the 3-token case is worst), the times for the "random bit" model follow a more complex pattern. This can be seen in Fig. 6, which magnifies several of the plots from Fig. 5a around the central point $p = 0.5$.

Finally, we observe that, in the "random bit" model, 3-token configurations do *not* always result in the worst-case behaviour. The PRISM query:

$$\mathit{filter}(\mathit{range}, \mathit{num\_tokens}, \mathit{filter}(\mathit{argmax}, \mathtt{R}^{\mathit{time}}_{=?}[\mathtt{F}\ \text{“}\mathit{stable}\text{”}], \text{“}\mathtt{init}\text{”}))$$

reveals the different possible numbers of the tokens that occur in the configurations yielding maximum expected times. For all values of N we considered, the worst-case number of tokens appears to be unique for each value of p, but varies between 3 and N for different values of p.



(a) $N = 7$ **(b)** $N = 9$ **(c)** $N = 11$

Fig. 6. Magnified portions of the plots from Fig. 5a for ring sizes 7, 9 and 11

5. Conclusions

In this paper, we applied probabilistic model checking to analyse a simple randomised self-stabilisation algorithm due to Herman [Her90]. In particular, we investigated the worst-case expected time required for the algorithm to execute, providing experimental evidence to corroborate the unproven conjecture of McIver and Morgan [MM05] that three tokens, spaced evenly around the ring, always yields the maximum execution time. We also studied the effect of using a based coin in the algorithm, discovering that this can actually improve the worst-case behaviour of the algorithm.

This example gives a good illustration of the benefits of probabilistic model checking. Firstly, it provides languages, techniques and tools that make it simple to model probabilistic systems and then analyse a variety of quantitative properties against them. Another key advantage of the approach is that it performs an *exhaustive* analysis: in this case, that amounts to computing values of a given property for all system configurations, then determining the maximum possible value and identifying the precise configurations that yield this value. By contrast, other analysis methods, such as those based on Monte Carlo simulation, are unable to perform an exhaustive analysis of this kind.

On the other hand, a disadvantage is that probabilistic model checking is typically restricted to analysing *finite-state* models. Furthermore, the size of such models that can be analysed is in practice always limited by the amount of time and space available. In this paper, the largest models for which we studied quantitative properties (that require numerical solution) contained about 2 million states and 10 billion transitions. In this instance, PRISM's symbolic engines can handle these (and slightly larger) models without problems; the limiting factor is actually the time required to solve the models. But, more typically, it is the model size that represents the bottleneck in terms of scalability.

Progress has been made in recent years on the development of techniques to automatically construct finite abstractions of large or infinite-state probabilistic models [HWZ08, KKNP10]. However, for a comprehensive analysis of systems like Herman's algorithm, we really require automated tools and techniques that can perform a *parametric* verification of the algorithm, i.e. for an arbitrary number of processes N. This represents a very challenging area of future work. On the other hand, promising steps have been taken [HHZ11] in the related problem of computing *symbolic* results expressed in terms of model parameters that affect probabilities (e.g. p in Herman's algorithm), rather than the size of the model.

Acknowledgments

This work was part supported by ERC Advanced Grant VERIWARE.

References

[ASB+95] Aziz A, Singhal V, Balarin F, Brayton R, Sangiovanni-Vincentelli A (1995) It usually works: The temporal logic of stochastic systems. In: Wolper P (ed) Proceedings of 7th Internatiol Conference on Computer Aided Verification (CAV'95), volume 939 of LNCS. Springer, pp 155–165

[BGJ99] Beauquier J, Gradinariu M, Johnen C (1999) Memory space requirements for self-stabilizing leader election protocols. In Proceedings of 18th ACM Symposium on principles of distributed computing (PODC'99). ACM, pp 199–208

[BK08] Baier C, Katoen J-P (2008) Principles of model checking. MIT Press, USA

[Dij74] Dijkstra E (1974) Self-stabilizing systems in spite of distributed control. Commun ACM, 17(11):643–644
[Dol00] Dolev S (2000) Self-stabilization. MIT Press, USA
[DTC+09] Dhama A, Theel O, Crouzen P, Hermanns H, Wimmer R, Becker B (2009) Dependability engineering of silent self-stabilizing systems. In: Proceedings of 11th International Symposium on stabilization, safety, and security of distributed systems, volume 5873 of LNCS. Springer, pp 238–253
[FMP06] Fribourg L, Messika S, Picaronny C (2006) Coupling and self-stabilization. Distrib Comput. 18(3):221–232
[Her90] Herman T (1990) Probabilistic self-stabilization. Inform Process Lett. 35(2):63–67. ftp://ftp.math.uiowa.edu/pub/selfstab/H90.html
[HHZ11] Hahn EM, Hermanns H, Zhang L (2011) Probabilistic reachability for parametric Markov models. Int J Soft Tools Technol Transfer (STTT). 13(1):3–19
[HJ94] Hansson H, Jonsson B (1994) A logic for reasoning about time and reliability. Formal Aspects Comput. 6(5):512–535
[HWZ08] Hermanns H, Wachter B, Zhang L (2008) Probabilistic CEGAR. In: Gupta A, Malik S (eds) Proceedings of 20th International Conference on Computer Aided Verification (CAV'08), volume 5123 of LNCS. Springer, pp 162–175
[IJ90] Israeli A, Jalfon M (1990) Token management schemes and random walks yield self-stabilizating mutual exclusion. In: Proceedings of 9th ACM Symposium on principles of distributed computing (PODC'90). ACM, pp 119–131
[KKNP10] Kattenbelt M, Kwiatkowska M, Norman G, Parker D (2010) A game-based abstraction-refinement framework for Markov decision processes. Formal Methods Sys Des. 36(3):246–280
[KMO+11] Kiefer S, Murawski A, Ouaknine J, Worrell J, Zhang L (2011) On stabilization in Herman's algorithm. In: Aceto L, Henzinger M, Sgall J (eds) Proceedings on 38th International Colloquium on Automata, Languages and Programming (ICALP'11), volume 6756 of LNCS. Springer, pp 466–477
[KNP05] Kwiatkowska M, Norman G, Parker D (2005) Quantitative analysis with the probabilistic model checker PRISM. Electron Notes Theor Comput Sci. 153(2):5–31
[KNP07] Kwiatkowska M, Norman G, Parker D (2007) Stochastic model checking. In: Bernardo M, Hillston J (eds) Formal methods for the design of computer, communication and software systems: performance evaluation (SFM'07), volume 4486 of LNCS. Springer, pp 220–270
[KNP11] Kwiatkowska M, Norman G, Parker D (2011) PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan G, Qadeer S (eds) Proceedings of 23rd International Conference on Computer Aided Verification (CAV'11), volume 6806 of LNCS. Springer, pp 585–591
[KSK76] Kemeny J, Snell J, Knapp A (1976) Denumerable Markov chains, 2nd edn. Springer, Berlin
[MM04] McIver A, Morgan C (2004) Abstraction, refinement and proof for probabilistic systems. Springer, Berlin
[MM05] McIver A, Morgan C (2005) An elementary proof that Herman's ring is $\Theta(N^2)$. Inform Process Lett. 94(2):79–84
[Nak05] Nakata T (2005) On the expected time for Herman's probabilistic self-stabilizing algorithm. Theor Comput Sci. 349(3):475–483

Received 20 December 2011
Accepted 21 March 2012 by Peter Höfner, Robert van Glabbeek and Ian Hayes
Published online 2 July 2012