Searching for Pareto-optimal Randomised Algorithms

Alan G. Millard[1], David R. White[2], and John A. Clark[1]

[1] Department of Computer Science, University of York, UK
{millard, jac}@cs.york.ac.uk
[2] School of Computing Science, University of Glasgow, UK
david.r.white@glasgow.ac.uk

Abstract. Randomised algorithms traditionally make stochastic decisions based on the result of sampling from a uniform probability distribution, such as the toss of a fair coin. In this paper, we relax this constraint, and investigate the potential benefits of allowing randomised algorithms to use non-uniform probability distributions. We show that the choice of probability distribution influences the non-functional properties of such algorithms, providing an avenue of optimisation to satisfy non-functional requirements. We use Multi-Objective Optimisation techniques in conjunction with Genetic Algorithms to investigate the possibility of trading-off non-functional properties, by searching the space of probability distributions. Using a randomised self-stabilising token circulation algorithm as a case study, we show that it is possible to find solutions that result in Pareto-optimal trade-offs between non-functional properties, such as self-stabilisation time, service time, and fairness.

1 Introduction

Search-Based Software Engineering (SBSE) research has traditionally focused on using metaheuristic search techniques to tackle problems related to deterministic algorithms [1]. Somewhat surprisingly, problems associated with randomised algorithms have received little attention, despite tremendous growth in the area over the past twenty five years. A *randomised algorithm* is one that makes stochastic choices during its execution, based on the outcome of a random number generator [2]. Since Rabin's seminal paper [3] randomised algorithms have gained universal acceptance, mostly due to their speed and simplicity. In fact, many randomised algorithms, such as random-pivot Quicksort, provide significant efficiency gains over the best known deterministic solutions, and are easier to implement than their deterministic counterparts [2]. Consequently, randomised algorithms are now becoming more commonplace, and have seen widespread adoption in practical applications.

To date, randomised algorithm research has typically placed emphasis on formally proving the correctness of the algorithms. Whilst functionality is clearly important, we are not only interested in provable functional correctness, but also in eliciting guarantees about non-functional properties. The SBSE community has recognised that the satisfaction of non-functional requirements is an important consideration in algorithm design, and the problem is starting to attract greater attention [4]. In practice, we find that the non-functional properties of randomised algorithms can be influenced through the choice of probability distributions upon which they base their stochastic decisions. This is particularly interesting in the context of randomised distributed algorithms, where the decisions made by each processor in the system may affect the state of others.

The non-functional properties of randomised distributed algorithms are an emergent result of local interactions between processors. Thus, the choice of probability distribution to be used for each processor has consequences in terms of the emergent non-functional properties, and we wish to explore the effect of different choices. However, due to the complex interplay between the probabilistic behaviour of networked processors, the relationship between the emergent non-functional properties of the system and the probability distributions used are unlikely to be intuitive, and therefore difficult to predict *a priori*. The space of possible solutions is also infinitely large for continuous probability distributions, exacerbating the problem. The search for probability distributions that satisfy multiple objective criteria is therefore an appropriate application of SBSE.

Instead of searching for entirely novel algorithms, we seek to optimise pre-existing algorithms for some target criteria, by searching over the space of probability distributions. We have chosen to examine the problem of self-stabilising token circulation in a ring of networked processors as a case study, because simple randomised distributed algorithms already exist to solve this problem. We use Genetic Algorithms (GAs) in conjunction with Multi-Objective Optimisation (MOO) techniques, to find sets of solutions to this problem that approximate underlying Pareto fronts for pairs of conflicting non-functional objectives. However, the focus of this paper is not upon finding optimal solutions for this case study, we are simply interested in the discovery of trade-offs, primarily as an indication of whether similar trade-offs might exist for other randomised algorithms.

1.1 Contributions

The main contributions of this paper are to demonstrate that:

- GAs can be used in conjunction with MOO to explore the trade-offs between non-functional objectives associated with pre-existing randomised algorithms, by searching over the space of probability distributions.
- Such trade-offs exist for the randomised self-stabilising token circulation protocol used as a case study, and potentially for other randomised algorithms.
- The characteristics of these trade-offs vary with problem size (in this case, the number of processors in the system).

2 Related Work

In order to verify properties of randomised algorithms, a probabilistic model checking tool is required. Model checking is often used to ensure the correctness of a system, by constructing a finite state model of the system and then exhaustively checking this model against some specification [5]. Our approach uses the Probabilistic Symbolic Model Checker (PRISM) [6] language to describe a system of interest, the non-functional properties of which can then be measured using the PRISM model checker. The benefit of model checking over traditional testing, is that we are able to formally prove properties about solutions found through search, by checking every possible execution of an algorithm's code, rather than exploring only a sample of all possible execution paths.

Johnson [7] has previously used model checking to guide the synthesis of finite state automata with Genetic Programming. Each individual was evaluated against a formal specification given in temporal logic, and assigned a fitness value directly proportional to the number of functional properties satisfied. In an attempt to smooth the fitness landscape, Johnson included properties that check for partially correct programs. However, this method still suffers from discontinuities in the fitness landscape, making it hard for evolution to make gradual improvements towards the target specification.

Katz and Peled [8] have also used model checking to generate fitness values, for the rediscovery of classical two-process mutual exclusion algorithms using Genetic Programming, and later the synthesis of novel algorithms for mutual exclusion [9]. Of greater relevance to our work, is their synthesis of leader election protocols for unidirectional token rings [10], however only deterministic algorithms were evolved. The fitness landscape was smoothed by analysing the graph generated during model checking, to extract further information to quantify the degree of satisfaction, rather than just checking whether properties were satisfied or not, giving a finer-grained fitness measure.

In contrast to the work of Johnson, Katz and Peled, our goal is not to synthesise provably correct programs from scratch. Rather, we seek to modify existing programs that are already known to be provably correct, to trade-off non-functional properties by optimising a vector of probabilities. In addition to discrete true/false values returned by correctness properties, PRISM may return continuous values for non-functional properties, such as the expected number of steps to reach a goal state. The return values from such non-functional properties can be used as input to a fitness function, resulting in a much smoother fitness landscape than one based on correctness properties.

3 Distributed Systems

We consider a *distributed system* to be modelled by a connected directed graph $G = (V, E)$, where V denotes the set of nodes representing the processors, and E denotes the set of edges representing communication links between pairs of processors. In general, the topology of a distributed system is unrestricted. However, in this paper we focus on systems with unidirectional ring topologies, which consist of $|V| = N$ connected processors $P_1, P_2, \ldots, P_N$. Adopting the notation of Dolev [11], the *predecessor* of each processor P_i, $1 < i \leq N$, is P_{i-1}, and the predecessor of P_1 is P_N. Similarly, the *successor* of each processor P_i, $1 \leq i < N$, is P_{i+1}, and the successor of P_N is P_1. In such a system, each processor may only receive information from its predecessor, and send information to its successor.

3.1 Protocols and Configurations

Each processor in the ring executes a *protocol*, which comprises a set of variables and guarded actions. The guard of each action is a boolean expression involving the state of processor P_i and its predecessor. The corresponding statement of a guarded action atomically updates the state of the processor that executed it. A *configuration* of a distributed system is given by a vector containing the state of every processor at a particular time. A processor is said to be *enabled* in some system configuration, if any of the processor's action guards are satisfied by that configuration [12]. In the protocol considered here, no configuration of the distributed system will satisfy multiple guards within a single processor.

3.2 Daemons

The *daemon* is a scheduler that selects a subset of the enabled processors to be activated at each computation step, which then perform the actions corresponding to their satisfied guards. A *central daemon* is a scheduler that may only select a single processor to schedule from the set of enabled processors. In contrast, a *distributed daemon* may choose one or more enabled processors at each computation step [13]. A daemon is said to be *fair* if it will eventually schedule a continuously enabled processor. On the other hand, the only restriction on an *unfair* daemon is that it must schedule processors that can perform an action [13]. For reasons of computational tractability, we consider a protocol that assumes a *synchronous daemon* [14]. This special case of a fair distributed daemon schedules *all* enabled processors at each computation step, thus removing any choice and therefore non-determinism from the scheduling.

4 Self-stabilising Token Circulation

A fundamental problem in distributed systems is that of ensuring mutually exclusive access to shared resources. For systems consisting of a networked ring of processors, one solution to this problem is to implement a token circulation scheme. When a processor gains possession of the token it is considered privileged, and is granted access to the shared resource. However, mutual exclusion is only ensured if there exists just a single token in the network at any one time.

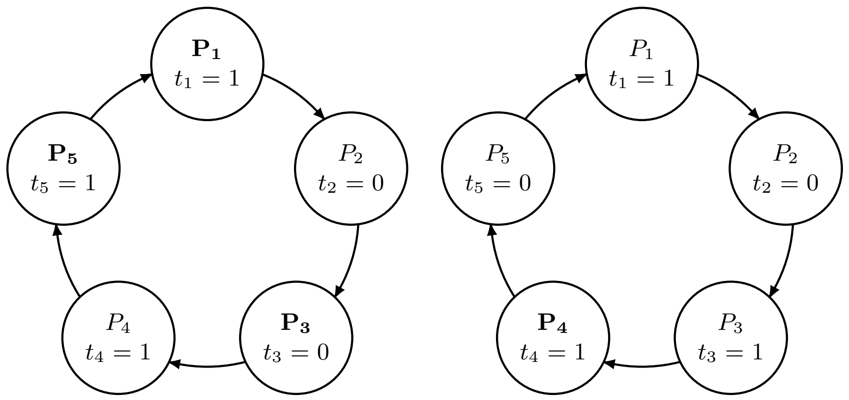
The concept of a *self-stabilising* system was first introduced by Dijkstra [15] in 1974, to describe a system that converges to a *stable* state within a finite number of steps, regardless of its initial state. This is a desirable trait, since it allows the system to automatically recover from the occurrence of transient faults [11]. In the context of token circulation, the system is considered to be in a stable state when exactly one processor holds a token. Dijkstra [15] gave three deterministic protocols that solve the problem of self-stabilising token circulation in a ring, which each require a distinguished processor to control the system. However, for distributed systems comprising identical anonymous processors, it is known that there exists no deterministic algorithm for self-stabilising token circulation, due to the inability to break symmetry [16]. Randomisation is a powerful tool in algorithm design, and is often used to break symmetry in such systems [11].

The protocol presented by Herman [14] is a classic example of randomised self-stabilising token circulation in unidirectional rings of anonymous identical processors, which assumes the presence of a synchronous daemon. Several other randomised self-stabilising protocols exist [11], but we have chosen the protocol presented by Beauquier et al. [17] as a case study due to its simplicity, and because the probability of each processor passing a token to its successor is directly controlled by the probability distribution it uses.

4.1 Case Study

Beauquier et al. [17] present a randomised self-stabilising token circulation protocol for unidirectional rings of anonymous processors. Each processor is identical, and executes the same protocol. The state of processor P_i is determined by the value of a single binary variable t_i. The configuration of the system at a particular time is therefore given by the vector $(t_1, t_2, \ldots, t_N)$. The locations of tokens in the ring are implicitly defined by the states of individual processors. A processor P_i holds a token if its state is equal to that of its predecessor. It is instructive to consider the example system configuration shown in Figure 1(a). Here, $t_1 = t_5$, therefore processor P_1 holds a token. Similarly, processors P_3 and P_5 each possess a token, because $t_3 = t_2$ and $t_5 = t_4$, respectively.

The system may begin in any state, and may therefore start with up to N tokens. In order to reach a stable state, the number of tokens in the ring must be reduced to one. For token rings comprising an odd number of processors,



(a) System configuration where processors P_1, P_3 and P_5 hold tokens (b) System configuration where only processor P_4 holds a token

Fig. 1: Example token rings of size $N = 5$

the protocol guarantees convergence to a stable state under a synchronous daemon. This is achieved through the random delay of tokens — a processor with the token randomly decides whether or not to pass it on to its successor. Inevitably, some tokens will temporarily remain stationary while others continue to circulate, eventually catching up with the others and eliminating them [18].

The synchronous daemon will schedule any processor in possession of a token at each computation step. Each scheduled processor tosses a fair coin, to decide whether or not it should pass the token it holds to its successor. This is achieved by negating the value of its binary state variable t_i. Considering the system configuration in Figure 1(a) again, P_1, P_3, and P_5 will all be scheduled synchronously by the daemon, because they each hold a token. Supposing that P_3 and P_5 decide to pass on their tokens, but P_1 does not, the system configuration will change to the stable state shown in Figure 1(b), where only P_4 holds a token. Once the system has reached a stable state, the token will continue to be passed around the ring in a fair manner, until the system is perturbed due to the occurrence of a fault, and must self-stabilise again.

5 PRISM Model

We now consider the PRISM model for a token ring of size $N = 3$ as an example, since it is the simplest system we can construct that comprises multiple processors. The concepts covered here generalise to larger token rings, which will be examined later. A system is constructed in the PRISM language by defining a number of interacting modules. These modules contain variables, that describe the state of the module, and a set of guarded commands of the form:

```
[] <guard> -> <prob_1> : <action_1> + ... + <prob_n> : <action_n>;
```

The guard is a predicate over the variables in the system. Each action describes a transition that will update the system if the guard is true, and is selected for execution with the corresponding probability [19]. The code snippet below, taken from our model for $N = 3$, shows the module definition for processor P_1.

```
module P1
    t1 : bool;
    [update]  t1=t3 -> p1 : (t1'=t1) + 1-p1 : (t1'=!t1);
    [update] !t1=t3 -> true;
endmodule
```

This code first declares a boolean variable which represents the state variable t_1 belonging to processor P_1. Following this is a guarded command which models the action of probabilistically deciding whether to pass on the token. Our PRISM model of the algorithm differs slightly from the original Beauquier et al. [17] protocol, in that it is parameterised by the probability p_1 of processor P_1 holding on to a token (by leaving its state unchanged), which leaves a probability of $1-p_1$ of passing it on (by negating its state variable). In the original algorithm, the value of both these probabilities is 0.5, simulating the toss of a fair coin. The module definitions for processors P_2 and P_3 are defined in a similar way.

We model the algorithm as a Discrete Time Markov Chain (DTMC), and use the action label `update` to implement a synchronous daemon. This forces each module to make transitions simultaneously, resulting in a deterministic choice over which of the enabled processors should be scheduled at each time step. The second guarded command in the code snippet above causes the processor to do nothing when it does not hold a token, and is included simply to prevent deadlocks from occurring, as PRISM requires every module containing the action label `update` to take some action at each time step.

PRISM allows rewards (which can also be thought of as costs) to be assigned to specific states or transitions of the model. So that we may check non-functional properties of the system, we use a PRISM reward structure to assign every state in the model a reward value of 1. The expected value of these rewards can be checked using PRISM properties, to count the expected number of steps taken to reach a particular state. We also define a PRISM formula called `num_tokens`, which calculates the number of tokens present in the system at the current time step, given by the sum of the number of processors which have the same state as their predecessor. This then allows us to define the boolean label `stable`, for use in PRISM properties, which evaluates to true when `num_tokens=1`.

6 Objective Measures

The PRISM property specification language allows us to check PRISM models against specifications written in probabilistic temporal logics such as PCTL [6]. We will now discuss the PRISM properties we have written in this language, and how they are used to create objective functions.

6.1 Self-stabilisation Time

A non-functional objective of particular interest is the expected self-stabilisation time of the system, which we would like to minimise, since the system must be able to converge to a stable state faster than its expected failure rate if it is to make progress [20]. The following two PRISM properties check the average and maximum expected number of steps the system takes to reach a stable state, the values of which can be minimised directly, giving us the objective functions f_1 and f_2, respectively.

```
filter(avg, R=? [F "stable"], "init")
filter(max, R=? [F "stable"], "init")
```

The `R=?` operator can be used to analyse properties that relate to the expected value of rewards accumulated along an execution path until a certain state is reached. The path property `F` (meaning "future") is used here to check the probability of the system eventually reaching a state where the label `"stable"` evaluates to true, from a given initial state. Since we assign a reward of 1 to every state in the model, the property `R=? [F "stable"]` calculates the expected number of steps the system takes to reach a stable state, from a specified initial state. We instruct PRISM to calculate the expected number of steps to self-stabilise from every initial state by using a filter, which is specified using the syntax `filter(operator, property, states)`. The `avg` and `max` operators are used to calculate the average and maximum value of `property` over states satisfying `states`, respectively. The set of states specified here is the set of all possible initial states, defined in the PRISM model using the label `"init"`.

6.2 Fairness

Another interesting non-functional property of the system is the *fairness* of token circulation. In this paper, we consider a completely "fair" system to be one where each processor receives an equal time share of token possession. The following properties check steady-state *token residency* — the proportion of time each processor holds a token in the long-run — using the `S=?` operator.

```
filter(max, S=? [t1=t3], "init")
filter(max, S=? [t2=t1], "init")
filter(max, S=? [t3=t2], "init")
```

$$f_3 = 1 - \frac{\min\{R_1, R_2, \ldots, R_N\}}{\frac{1}{N}\sum_{i=1}^{N} R_i} \tag{1}$$

PRISM returns the minimum and maximum of a range of values over initial states, which are identical for these kinds of properties. We simply use a `max` filter operator to ensure a single value is returned, to save parsing the output. Let R_i be the steady-state token residency for processor P_i. Then, the above properties return the values R_1, R_2 and R_3, respectively. The objective measure for fairness can then be formally defined as shown in Equation 1. The second term of the equation evaluates to 1 when the token residency for each individual processor equals $1/N$, which only occurs when every processor P_i in the system shares the same value of p_i. Note that we attempt to maximise fairness, by minimising the value of f_3. This is simply because the evolutionary toolkit we are using attempts to minimise the value of objective measures by default.

6.3 Service Time

The last non-functional property we consider, is *service time* — how long it takes the token to circulate the ring of processors once a stable configuration is reached [18]. Minimising service time is desirable, as this reduces the time a processor must wait to regain privileged status. Each of the following properties checks, from every state where a processor P_i does not hold the token, the expected average number of steps until it obtains the token. Let A_i be the average expected service time for processor P_i, then the properties below return the values A_1, A_2 and A_3, respectively.

```
filter(avg, R=? [F "stable" & t1=t3], "stable" & t1!=t3)
filter(avg, R=? [F "stable" & t2=t1], "stable" & t2!=t1)
filter(avg, R=? [F "stable" & t3=t2], "stable" & t3!=t2)
```

Similarly, let M_i be the maximum expected service time for processor P_i. Then, the values of M_1, M_2 and M_3 are given by properties identical to those above, except with each filter's `avg` operator replaced with a `max` operator. The service time value is different for each processor when non-uniform probabilities are used, because the property does not take into account the behaviour of the processor being checked. We therefore find the average and maximum service times of every processor in the system, giving two more objective measures:

$$f_4 = \frac{1}{N}\sum_{i=1}^{N} A_i \qquad (2) \qquad\qquad f_5 = \max\{M_1, M_2, \ldots, M_N\} \qquad (3)$$

The objective measures f_4 and f_5 represent the average-case and worst-case service time of the system — the average and maximum time a processor must wait to receive the token again, respectively. These will both be minimised in system configurations where every processor P_i uses a value of $p_i = 0$, causing it to pass on any token it receives deterministically. If such a system is in a stable state, it will take $N - 1$ steps for each processor to receive the token again.

7 Multi-Objective Optimisation

Instead of having each processor in the system use an identically biased coin, we consider the case where each processor uses a coin with a different bias. In order to parameterise the system with a different probability distribution for each processor, we define a vector $(p_1, p_2, \ldots, p_N)$ of probabilities assigned to processors $(P_1, P_2, \ldots, P_N)$, respectively. Note that this assignment implicitly determines the values of $(1-p_1, 1-p_2, \ldots, 1-p_N)$, which control the probability of each processor passing on a token. The variables `p1`, `p2`, and `p3` in the PRISM model for $N = 3$ are defined as uninitialised constants, the values for which are passed to PRISM via the command line. Due to the symmetrical ring network topology, the exact assignment of probabilities to processors is unimportant, so long as the order of their assignment is preserved. There exist N duplicate solutions for each probability vector, in which the probabilities are simply shifted along and assigned to different processors.

The PRISM model is parameterised by the vector $(p_1, p_2, \ldots, p_N)$ of probabilities, which may be searched over, to find solutions that trade-off non-functional objectives. In order to search for Pareto fronts we use a Genetic Algorithm to evolve individuals comprising a genome of N real numbers, each of which corresponds to a probability p_i in the parameter vector. The evolutionary computation toolkit ECJ [21] was used to evolve Pareto-optimal solutions using the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [22], the main parameter settings for which are given in Table 1.

Parameter	**Value**
Population size	100
Generations	250
Crossover method	Simulated Binary Crossover
Mutation method	Polynomial
Crossover probability	0.9
Mutation probability	$1/N$
Crossover distribution index	20
Mutation distribution index	20
Selection method	Tournament selection, size 2

Table 1: Genetic Algorithm parameters

These parameters are the same as those originally used by Deb et al. [22], when NSGA-II was first presented. Any parameters not listed here are defaults inherited from the parameter files `ec.params` and `simple.params` which come packaged with ECJ. We have not attempted any parameter tuning, since the aim of these experiments was simply to confirm the existence of non-functional trade-offs for randomised algorithms in general, rather than to accurately approximate the underlying Pareto fronts for this particular case study.

7.1 Fitness Evaluation

The fitness evaluation process for a single individual is as follows: ECJ invokes the PRISM model checker via a command-line interface, specifying as input a PRISM model containing undefined constants corresponding to the probabilities used by each processor, along with the probability vector $(p_1, p_2, \ldots, p_N)$ representing the individual. PRISM uses these values to instantiate the model, assigning the probabilities to the corresponding constants. PRISM also takes as input a properties file, which contains a list of properties that we wish to check the model against. Model checking is computationally expensive, so instead of checking every PRISM property listed in Section 6, we only select the subset corresponding to the pair of non-functional properties we wish to optimise. The results from the model checking are output to a text file, which is then parsed by ECJ. The objective functions $f_1, f_2 \ldots, f_5$ are calculated from the return values of the individual PRISM properties, as described in Section 6, and their values are fed into the multi-objective fitness function for NSGA-II.

Although the original Beauquier et al. [17] protocol is provably correct, in our PRISM model the probability p_i of a processor P_i holding on to a token may potentially be set to 1, which would cause the processor to deterministically retain possession of any token it receives. While this would still guarantee convergence to a stable state, the single remaining token would not continue to circulate the ring of processors, and would instead be held forever by the same processor, causing others in the ring to wait indefinitely for privileged status. We automatically identify such invalid solutions during evolution, by checking values in the probability vector within ECJ, and discard them immediately.

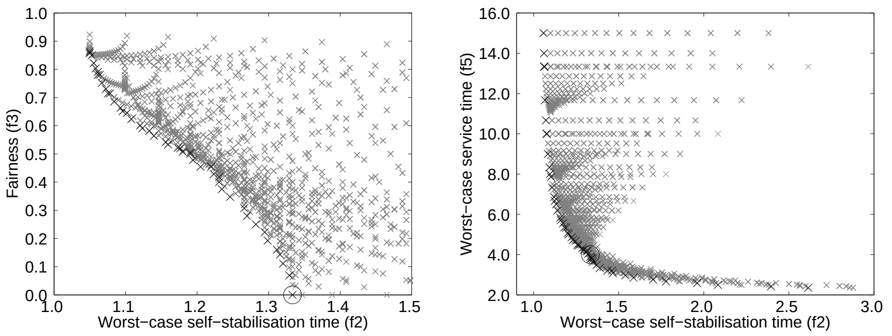
8 Results

In summary, the problem to be solved was to determine which pairs of non-functional objectives were conflicting, and could therefore be traded-off, allowing us to find a Pareto-optimal set of solutions. The five objective functions $f_1, f_2, \ldots, f_5$ give rise to ten possible unique pairs of objectives to optimise.

8.1 Initial Exploration

Since there was no *a priori* indication of whether trade-offs would exist for any given pair of objectives, we enumerated the space of solutions for $N = 3$ with a fine granularity. Models were generated for every possible vector (p_1, p_2, p_3), where each probability was limited to increments of 0.01 in the closed interval [0.05, 0.95] — essentially a 90-level factorial experiment with 3 factors. For each model, the values of $f_1, f_2, \ldots, f_5$ were checked using PRISM, and plotted against each other in pairs. Out of the ten possibilities, only six pairs of objective measures were found to be conflicting. Average and worst-case expected time to self-stabilise (f_1 vs f_2) were found to be non-conflicting, as were fairness and average/worst-case service time (f_3 vs f_4 and f_3 vs f_5), and average-case and worst-case service time (f_4 vs f_5). For each of the remaining six conflicting objective pairs, Pareto fronts were discovered. However, due to their similarity, we only show the results for worst-case properties here, for the sake of brevity.

Figure 2(a) shows the values of f_2 and f_3 measured for every individual in the factorial experiment, where each point corresponds to an individual's non-functional properties in objective space. The original solution, which uses uniform probability distributions, is circled in this and every subsequent plot. This graph clearly demonstrates that trade-offs between non-functional properties exist for this protocol in rings of size $N = 3$. The individuals on the Pareto front with the lowest values of f_3 (the most fair — including the original solution) are



(a) Fairness vs Worst-case self-stabilisation time (f_3 vs f_2)

(b) Worst-case service time vs Worst-case self-stabilisation time (f_5 vs f_2)

Fig. 2: Results of enumeration for $N = 3$

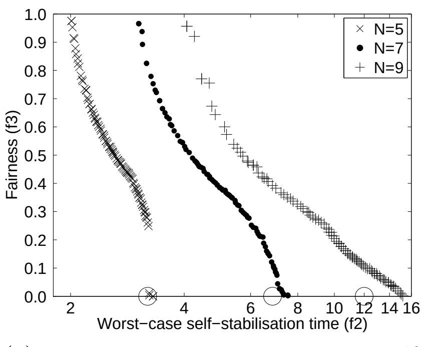
the slowest to converge to a stable state, and we can see that self-stabilisation time can be improved at the expense of fairness. Note that unfairness does not directly result in faster self-stabilisation, as there exist many unfair solutions with far worse stabilisation times than those that are completely fair.

Figure 2(b) shows the Pareto front found for the conflicting objectives of worst-case expected self-stabilisation time and worst-case service time (f_2 vs f_5). Here, we can see that in order to improve the speed of self-stabilisation, we must sacrifice service time. These Pareto fronts allow us to visualise the relationship between pairs of non-functional properties, and we can see that the relationship for (f_2, f_3) compared to (f_2, f_5) is qualitatively very different for $N = 3$.

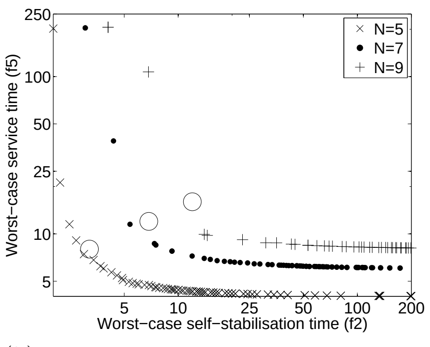
8.2 Larger Token Rings

After confirming the existence of trade-offs for token rings of size $N = 3$, we sought to investigate whether these trade-offs were restricted to this particular problem size. However, enumeration of the search space at any useful granularity is infeasible for problem sizes larger than $N = 3$, due to the computational effort required by PRISM to verify properties of larger systems. This is where the application of evolutionary search becomes useful, since it allows us to approximate the underlying Pareto fronts with far fewer fitness evaluations.

In order to provide some assurance that our evolutionary framework would be able to reasonably approximate any Pareto fronts that may have existed for larger problem sizes, we attempted to rediscover the trade-offs found for $N = 3$, but this time using evolutionary search. The rediscovered Pareto fronts were found to be almost identical to those originally discovered through enumeration, so we proceeded to apply our evolutionary method to search for trade-offs in token rings of sizes $N = 5, 7, 9$. As shown in Figures 3(a) and 3(b), Pareto fronts were found for these larger rings, demonstrating that trade-offs of non-functional properties are not limited to rings of size $N = 3$. In addition, we checked whether



(a) Fairness vs Worst-case self-stabilisation time (f_3 vs f_2)



(b) Worst-case service time vs Worst-case self-stabilisation time (f_5 vs f_2)

Fig. 3: Pareto fronts for larger token rings, found through evolution

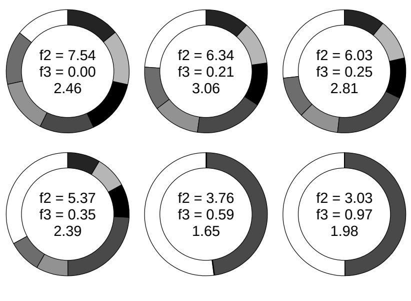
the non-conflicting objectives became conflicting at these higher values of N, but they did not.

The shape of the Pareto fronts for $N = 5, 7$ in Figure 3(a) are similar to that of $N = 3$ in Figure 2(a), except that the points of inflection occur at lower values of f_3, indicating that characteristics of the trade-off change with problem size. There does not even appear to be a point of inflection in the Pareto front found for $N = 9$, however this may be because 250 generations were not sufficient for NSGA-II to converge completely on the underlying Pareto front, as indicated by the position of the original solution in objective space.

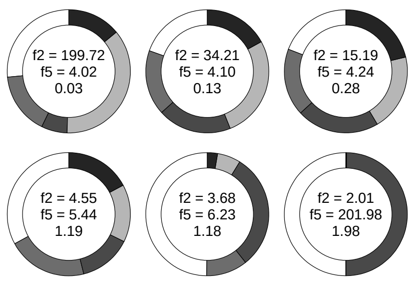
The Pareto fronts shown in Figure 3(b) are also qualitatively similar to those found for $N = 3$ in Figure 2(b), illustrating that the relationship between f_2 and f_5 does not change radically as the size of the problem increases. However, the sparsity of solutions in the evolved Pareto fronts does increase with problem size. We conjecture that this is due to the parameters used for NSGA-II, rather than a feature of the underlying solution space, but we cannot be certain.

8.3 Example Individuals

We now present graphically the probability vectors of selected individuals, to illustrate how the probability distributions used by the processors in the system impact each non-functional objective. Figure 4(a) shows example individuals from the $N = 7$ Pareto front in Figure 3(a). Each shaded segment represents the relative magnitude of the probability p_i for each processor P_i.The sum of the probability vector $(p_1, p_2, \ldots, p_N)$ for each individual is also given, beneath the values of the objective functions, to demonstrate the influence of the absolute probability values used. It can be seen that individuals with the worst self-stabilisation times are very fair, but as stabilisation time improves, individuals begin to use probabilities that result in unfair token circulation. Those individuals with the fastest self-stabilisation times contain two selfish processors,



(a) Individuals from the Pareto front for $N = 7$ in Figure 3(a) (f_3 vs f_2)



(b) Individuals from the Pareto front for $N = 5$ in Figure 3(b) (f_5 vs f_2)

Fig. 4: Example individuals from the Pareto fronts found through evolution

while the rest are almost completely selfless. Tokens will stop at the selfish processors with high probability, allowing those circulating behind them to catch up and eliminate them, resulting in fast convergence to a configuration with a single token. Notice that individuals with nearly identical relative distributions of probabilities, but a different sum of probabilities, can have quite different non-functional properties, indicating that both relative values and absolute probabilities have a significant influence. Figure 4(b) shows example individuals from the $N = 5$ Pareto front in Figure 3(b). Again, we see here that the absolute probability values are important — fair rings of very selfless processors (indicated by a small sum of probabilities) result in fast token circulation. As service time is traded-off for stabilisation time, we begin see individuals where pairs of selfish processors dominate.

9 Conclusions and Future Work

To our knowledge, this paper represents the first application of Search-Based Software Engineering to randomised algorithms. Although this work constitutes only a proof of principle, and the objective measures examined for this case study algorithm are simply a means to an end, it is conjectured that similar trade-offs between non-functional properties will exist for other randomised algorithms.

Unfortunately, since our approach is based on model checking, verifying properties of systems larger than those considered here is computationally very expensive. This is because the number of states in the model grows exponentially with problem size, due to the well known state explosion problem [23]. To combat this, future work may investigate the potential of using approximate model checking, or sampling methods, to guide evolutionary search, by providing imperfect information about the non-functional properties of individual solutions at a reduced computational cost.

10 Acknowledgements

This work is part of the Software Engineering by Automated SEarch (SEBASE) project and is funded by an EPSRC grant (EP/D050618/1).

References

1. Harman, M., Mansouri, S., Zhang, Y.: Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications. Department of Computer Science, Kings College London, Tech. Rep. TR-09-03 (2009)
2. Motwani, R.: Randomized Algorithms. Cambridge University Press (1995)
3. Rabin, M.: Probabilistic algorithms. Algorithms and Complexity **21** (1976)
4. Afzal, W., Torkar, R., Feldt, R.: A systematic review of search-based testing for non-functional system properties. Information and Software Technology **51** (2009) 957–976

5. Clarke, E.: Model checking. In: Foundations of software technology and theoretical computer science, Springer (1997) 54–56
6. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic Symbolic Model Checker. In: Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02). Volume 2324 of LNCS., Springer (2002) 200–204
7. Johnson, C.G.: Genetic Programming with Fitness based on Model Checking. In: Genetic Programming. Volume 4445 of Lecture Notes in Computer Science., Springer-Verlag (2007) 114–124
8. Katz, G., Peled, D.: Model Checking-Based Genetic Programming with an Application to Mutual Exclusion. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 4963 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2008) 141–156
9. Katz, G., Peled, D.: Genetic Programming and Model Checking: Synthesizing New Mutual Exclusion Algorithms. In: Automated Technology for Verification and Analysis. Volume 5311 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2008) 33–47
10. Katz, G., Peled, D.: Synthesizing Solutions to the Leader Election Problem Using Model Checking and Genetic Programming. In: Hardware and Software: Verification and Testing. Volume 6405 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2011) 117–132
11. Dolev, S.: Self-stabilization. The MIT Press (2000)
12. Beauquier, J., Gradinariu, M., Johnen, C.: Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. Distributed Computing **20** (2007) 75–93
13. Beauquier, J., Gradinariu, M., Johnen, C.: Memory space requirements for self-stabilizing leader election protocols. In: Proceedings of the eighteenth annual ACM Symposium on Principles of Distributed Computing, ACM (1999) 199–207
14. Herman, T.: Probabilistic Self-stabilization. Information Processing Letters **35** (1990) 63–67
15. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM **17** (1974) 643–644
16. Angluin, D.: Local and global properties in networks of processors. In: Proceedings of the twelfth annual ACM Symposium on Theory of Computing. (1980) 82–93
17. Beauquier, J., Cordier, S., Delaët, S.: Optimum probabilistic self-stabilization on uniform rings. In: Proceedings of the Second Workshop on Self-Stabilizing Systems. (1995) 15.1–15.15
18. Johnen, C.: Service Time Optimal Self-stabilizing Token Circulation Protocol on Anonymous Undirectional Rings. In: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems, IEEE (2002) 80–89
19. Norman, G.: Analysing Randomized Distributed Algorithms. Validation of Stochastic Systems (2004) 384–418
20. Higham, L., Myers, S.: Self-stabilizing token circulation on anonymous message passing rings. In: OPODIS98 Second International Conference On Principles Of Distributed Systems. (1999)
21. Luke, S.: (ECJ) `http://cs.gmu.edu/~eclab/projects/ecj/`.
22. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. Evolutionary Computation, IEEE Transactions on **6** (2002) 182–197
23. Valmari, A.: The State Explosion Problem. Lecture Notes in Computer Science (1998) 429–437