Improving Parallel State-Space Exploration Using Genetic Algorithms

Etienne Renault

LRDE, EPITA, Kremlin-Bicêtre, France

Abstract. The verification of temporal properties against a given system may require the exploration of its full state space. In explicit model-checking this exploration uses a Depth-First-Search (DFS) and can be achieved with multiple randomized threads to increase performance.
Nonetheless the topology of the state-space and the exploration order can cap the speedup up to a certain number of threads. This paper proposes a new technique that aims to tackle this limitation by generating artificial initial states, using genetic algorithms. Threads are then launched from these states and thus explore different parts of the state space.
Our prototype implementation runs 10% faster than state-of-the-art algorithms. These results demonstrate that this novel approach worth to be considered as a way to overcome existing limitations.

1 Introduction and Related Work

Model checking aims to check whether a system satisfies a property. Given a model of the system and a property, it explores all the possible configurations of the system, i.e., the *state space*, to check the validity of the property. Typically two kind of properties are distinguished, *safety* and *liveness* properties. This paper focus on safety properties that are of special interest since they stipulate that some "bad thing" does not happen during execution. Nonetheless the adaptation of this work for liveness properties is straightforward.

The state-space exploration techniques for debugging and proving correctness of concurrent reactive systems has proven their efficiency during the last decades [13, 18, 21, 3]. Nonetheless they suffer from the well known *state space explosion problem*, i.e., the state space can be far too large to be stored and thus explored in a reasonable time. This problem can be addressed using *symbolic* [4] or *explicit* techniques even if we only consider the latter one in this paper.

Many improvements have been proposed for explicit techniques. *On-the-fly exploration* [5] computes the successors of a state only when required by the algorithm. As a consequence, if the property does not hold, only a subset of the state space is constructed. *Partial Order Reductions (POR)* [23, 19, 15] avoid the systematic exploration of the state space by exploiting the interleaving semantic of concurrent systems. *State Space Caching* [9] saves memory by "forgetting" states that have already been visited causing the exploration to possibly revisit a state several times. *Bit-state Hashing* [11] is a semi-decision procedure in which each state is associated to a hash value. When two states share the same hash value, one of this two states (and thus its successors) will be ignored.

The previous techniques focus on reducing the memory footprint during the state-space exploration. Combining these techniques with modern computer architectures, i.e., many-core CPUs and large RAM memories, tends to shift from a memory problem to an execution time problem which is: how this exploration can be achieved in a reasonable time?

To address this issue multi-threaded (or distributed) exploration algorithms (that can be combined with previous techniques) have been developed [12, 2, 7, 18]. Most of these techniques rely on the *swarming* technique presented by Holzmann et al. [13]. In this approach, each thread runs an instance of a verification procedure but explores the state space with its own transition order.

Nowadays, best performance is obtained when combining swarming with *Depth-First Search* (DFS)[1] based verification procedures [21, 3]. In these combinations, threads share information about states that have been *fully explored*, i.e. states where all successors have been visited by a thread. Such states are called *closed states*. These states are then avoided by other threads explorations since they can not participate in invalidating the property. These swarmed-DFS algorithms are linear but their scalability depends on two factors:

Topology problems. If the state space is linear (only one initial state, one successor per state), using more than one thread cannot achieve any speedup. This issue can be generalized to any state space that is deep but not wide.

Exploration order problems. States are tagged *closed* following the DFS postorder of a given thread. Thus, a state s can only be marked *closed* after visiting at least N states, where N is the minimal distance between the initial state and s.

	1 thread	2 threads	4 threads	8 threads	12 threads
Time in milliseconds	2 960 296	1 796 418	118 6344	981 222	978 711
Speedup	1	1.65	2.50	3.016	3.025

The table above highlights this scalability problem over the benchmark[2] used in this paper. It presents the cumulated exploration time (in a swarmed DFS fashion) for 38 models extracted from the literature. It can be observed that this algorithm achieves reasonable speedup up to 4 threads but is disappointing for 8 threads and 12 threads (the maximum we can test).

This paper proposes a novel technique that aims to keep improving the speedup as the number of threads increases and which is compatible with all memory reduction methods presented so far.

The basic idea is to use genetic algorithms to generate artificial initial states (Sections 2 and 3). Threads are then launched with their own verification procedure from these artificial states (Sections 4 and 5). We expect that these threads

[1] It should be noted that even if DFS-based algorithms are hard to parallelize [20] they scale better in practice than parallelized Breadth-First Search (BFS) algorithms.

[2] See Section 6 for more details about the benchmark.

will explore parts of the state space that are relatively deep regarding to (many) DFS order(s). Thus, some states may be marked as *closed* without processing some path between the original initial state to these states.

Our prototype implementation (Section 6) has encouraging performances: the proposed approach runs 10% faster (with 12 threads) than state-of-the-art algorithms (with 12 threads). These results are encouraging and show that this novel approach worth to be considered as a way to overcome existing limitations.

Related Work. To our knowledge, the combination of parallel state space exploration algorithms with the generation of artificial initial states using genetic algorithms has never been done. The closest work is probably the one of Godefroid and Khurshid [8] that suggests to use genetic programming as an heuristic to help random walks to select the *best* successor to explore. The generation of other initial states have been proposed to maximize the coverage of random walks [22]: to achieve this, a bounded BFS is performed to obtain a pool of states that can be used as seed states. This approach does not help the scalability when the average number of successors is quite low (typically when mixing with POR).

In the literature there are some work that combine model checking with genetic programming but they are not related to the work presented here: Katz and Peled [14] use it to synthesize parametric programs, while all the other approaches are based on the work of Ammann et al. [1] and focus on the automatic generation of *mutants* that can be seen as particular "tests cases".

2 Parallel State Space Exploration

Preliminaries. Concurrent reactive systems can be represented using *Transitions Systems* (TS). Such a system $T = \langle Q, \iota, \delta, V, \gamma \rangle$ is composed of a finite set of states Q, an initial state $\iota \in Q$, a transition relation $\delta \subseteq Q \times Q$, a finite set of integer variables V and $\gamma : Q \rightarrow \mathbb{N}^{|V|}$ a function that associates to each state a unique assignment of all variables in V. For a state $s \in Q$, we denote by $\texttt{post}(s) = \{d \in Q \mid (s, d) \in \delta\}$ the set of its direct successors. A *path* of length $n \geq 1$ between two states $q, q' \in Q$ is a finite sequence of transitions $\rho = (s_1, d_1) \dots (s_n, d_n)$ with $s_1 = q$, $d_i = q'$, and $\forall i \in \{1, \dots, n-1\}$, $d_i = s_{i+1}$. A state q is *reachable* if there exists a path from the initial state ι to q.

Swarming. Checking temporal properties involves the exploration of (all or some part) of the state space of the system. Nowadays, best performance is obtained by combining on-the-fly exploration with parallel DFS reachability algorithms. Algorithm 1 presents such an algorithm.

This algorithm is presented recursively for the sake of clarity. Lines 4 and 5 represent the main procedure: `ParDFS` takes two parameters, the transition system and the number n of threads to use for the exploration. Line 5 only launches n instances of the procedure `DFS`. This last procedure takes three parameters, s the state to process, *tid* the current thread number and a *color* used to tag new visited states. Procedure `DFS` represents the core of the exploration. This exploration relies on a shared hashmap *visited* (defined line 2) that stores all states discovered so far by all threads and associate each state with a color (line 1):

Algorithm 1: Parallel DFS Exploration.

```
1  enum $color$ = { OPEN, CLOSED }
2  $visited$: hashmap of $(Q, color)$                         // Shared variable
3  $stop \leftarrow \bot$                                     // Shared variable
4  Procedure ParDFS($\langle Q, \iota, \delta, V, \gamma \rangle$ : TS, $n$ : Integer)
5      DFS($\iota, 1$, OPEN) || ... || DFS($\iota, n$, OPEN)
6  Procedure DFS($s \in Q$, $tid$ : Integer, $status : color$)
7      if $s \notin visited$ then $visited$.add($s, status$)
8      else if $visited[s]$ = CLOSED then return
9      $todo \leftarrow$ shuffle(post($s$), $tid$)   // Shuffle successors using $tid$ as seed
10     while ($\neg stop \wedge \neg todo.isempty()$) do
11         $s' \leftarrow todo.pick()$
12         if $s'$ is in the current recursive DFS stack then continue
13         if ($s' \notin visited$) $\vee$ $visited[s'] \neq$ CLOSED) then
14             DFS($s', tid, status$)
15     $visited[s] \leftarrow$ CLOSED
16     if ($s = \iota$) then $stop \leftarrow \top$
```

- OPEN indicates that the state (or some of its successors) is currently processed by (at least) a thread,
- CLOSED indicates that the states and all its successors (direct or not) have been visited by some thread.

The DFS function starts (lines 7 to 8) by checking if the parameter s has already been inserted, by this thread or another one, in the *visited* map (line 7). If not, the state is inserted with the color OPEN (line 7). Otherwise, if s has already been inserted we have to check whether this state has been tagged CLOSED. In this case, s and all its successors have been visited: there is no need to revisit them. Line 9 grabs all the successors of the state s that are then shuffled to implement the swarming. Finally lines 10 to 14 perform the recursive DFS: for each successor s' of the current state, if s' has not been tagged CLOSED a recursive call is launched. When all successors have been visited, s can be marked CLOSED.

One can note that a shared Boolean *stop* is used in order to stop all threads as soon as a thread closes the initial state. This Boolean is useless for this algorithm since, when the first threads ends, all reachable states are tagged CLOSED and every thread is forced to backtrack. Nonetheless this Boolean will be useful later (see Section 4). Moreover the *visited* map is thread safe (and lock-free) so that it does not degrade performances of the algorithm.

Problem statement. The previous algorithm (or some adaptations of it [21, 3]) obtains the best performance for explicit model checking. Nonetheless this swarmed algorithm suffers from a scalability problem. Figure 1 describes a case where augmenting the number of threads will not bring any speedup[3]. This figure

[3] This particular case will certainly degrade performance due to contention over the shared hashmap.

describes a transition system that is linear. The dotted transitions represent long paths of transitions. In this example, state x cannot be tagged CLOSED before state y and all the states between x and y have been tagged CLOSED. The problem here is that all threads start from state s. Since threads have similar throughput they will discover x and y approximately at the same time. Thus they cannot benefit from the information computed by the other threads. This example is pathological but can be generalized to any state space that is deep and narrow.

Suppose now that there are 2 threads and that the distance between s and x is the same than the distance between x and y. The only way to obtain the maximum speedup is to launch one thread with a DFS starting from s and launch the other thread from x. In this case, when the first thread reaches state x, x has just been tagged CLOSED: the first thread can backtrack and stop.

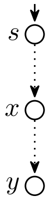

Fig. 1. Using more than one thread for the exploration is useless.

A similiar problem arise when performing on-the-fly model checking since (1) there is only one initial state and (2) all states are generated during the exploration. Thus a thread cannot be launched from a particular state. Moreover, the system's topology is only known after the exploration: we need a technique that works for any kind of topology.

The idea developed in this paper is the automatic generation of state x using genetic algorithms. The generation of *the perfect state* (the state x in the example) is a utopia. Nonetheless if we can generate a state relatively deep regarding to many DFS orders, we hope to avoid redundant work between threads, and thus maximize the information shared between threads. In practice we may generate states that do not belong to the state space, but Section 6 shows that more than 84% of generated states belongs to it.

3 Generation of Artificial Initial State

Genetic algorithms. For many applications the computation of an optimal solution is impossible since the set of all possible solutions is too large to be explored. To address this problem, Holland [10] proposed a new kind of algorithms (now called genetic algorithms) that are inspired by the process of natural selection. These algorithms are often considered as optimizer and used to generate high-quality solutions to search problems. Basically, genetic algorithms start by a population of candidate solutions and improve it using bio-inspired operators:

- *Crossover*: selects multiple elements in the population (the parents) and produces a child solution from them.
- *Mutation*: selects one element in the population and alters it slightly.

Applying and combining these operators produces a new generation that can be evaluated using a *fitness* function. This fitness function allows to select the best elements (w.r.t the considered problem) of this new population. These best elements constitute a new population on which mutation and crossover operations can be re-applied. This process is repeated until some satisfying solution is found (or until a maximal number of generations has been reached).

Genetic algorithms rely on a representation of solutions that is chromosome-like. In the definition of a transition system we observe that every state can be seen as a tuple of integer variables using the γ function. Each variable can be considered as a gene and the set of variables can be considered as a chromosome composed of 0 and 1. For instance, if a state is composed of two variables $a = 42$ and $b = 51$ the resulting chromosome (considering 8 bits integers) would be the one described Figure 2.

a	b
00101010	00110011

Fig. 2. Chromosome representation.

Crossover. Concurrent reactive systems are generally composed of a set of N_p processes and a set of shared variables (or channels). Given a transition system $T = \langle Q, \iota, \delta, V, \gamma \rangle$ we can define $E : V \rightarrow [0, N_p]$, such that if v is a shared variable, $E(v)$ returns 0 and otherwise $E(v)$ returns the identifier of the process where the variable v is defined.

Algorithm 2 defines the crossover operation we use. This algorithm takes a parameter S which represents the population to use for generating a new state. Line 2 instantiates a new state s that will hold the result of the crossover operation. Lines 3 to 5 set up the values of the shared variables of s: for each shared variable v, an element of S is randomly selected to be the parent. Then, at line 5, one can observe that $\gamma(s)[v]$ (the value of v in s) is set according to $\gamma(parent)[v]$ (the value of v in the *parent*). Lines 6 to 9 perform a similar operation on all the remaining variables.

These variables are treated by batch, i.e., all the variables that belong to a same process are filled using only one parent (line 7). This choice implies that in our `Crossover` algorithm the local variable of a process cannot have two different parents: this particular processing helps to exploit the concurrency of underlying system. A possible result of this algorithm is represented Figure 3 (with 8 bits integer variables, only one process, no shared variables, $S = \{parent1, parent2\}$ and *child* the state computed by `Crossover`(S)).

	Process 1	
	a	b
*parent*1	00000000	00000000
*parent*2	11111111	11111111
`Crossover`(S)	00000000	11111111

Fig. 3. Possible Crossover.

Algorithm 2: Crossover.

```
1  Procedure Crossover($S \subseteq Q$)
2      $s \leftarrow newState()$
3      for $v \in V$ s.t. $E(v) = 0$ do
4          $parent \leftarrow$ pick randomly one of $S$
5          $\gamma(s)[v] \leftarrow \gamma(parent)[v]$
6      for $i \in [0, N_p]$ do
7          $parent \leftarrow$ pick randomly one of $S$
8          for $v \in V$ s.t. $E(v) = i$ do
9              $\gamma(s)[v] \leftarrow \gamma(parent)[v]$
10     return $s$
```

Algorithm 3: Mutation.

```
Procedure Mutation($s \in Q$)
    for $v \in V$ do
        $r \leftarrow random(0..1)$
        if $r >$ THRESHOLD then
            $\gamma(s)[v] =$
              $randomly\_flip\_one\_bit\_in(\gamma(s)[v])$
            $\gamma(s)[v] = bound\_project(\gamma(s)[v])$
```

Mutations. The other bio-inspired operator simulates alterations that could happen while genes are combined over multiples generations. In genetic algorithms, these mutations are performed by switching the value of a bit inside of a gene. Here, all the variables of the system are considered as genes.

Algorithm 3 describes this mutation. For each variable in the state s (line 2), a random number is generated. A mutation is then performed only if this number is above a fixed threshold (line 4): this restriction limits the number of mutations that can occur in a chromosome. We can then select randomly a bit in the current variable v and flip it (line 5). Finally, line 6 exploits the information we may have about the system by restricting the mutated variable to its bounds.

Indeed, even if all variables are considered as integer variables there are many cases where the bounds are known a priori: for instance Boolean, enumeration types, characters, and so on are represented as integers but the set of value they can take is relatively small regarding the possible values of an integer. A possible result of this algorithm is represented Figure 4 (with 8 bits integer variables and only two character variables, i.e., that have values between [0..255]).

	Process 1	
	a	b
s	00000100	00001000
`Mutation`(s)	00000101	00001000

Fig. 4. Possible Mutation.

Fitness. As mentioned earlier, every new population must be restricted to the only elements that help to obtain a better solution. Here we want to generate states that are (1) reachable and (2) deep with respect to many DFS orders. These criteria help the swarming technique by exploring parts of the state space before another thread (starting from the real initial state) reaches them.

We face here a problem that is: for a given state it is hard to decide whether it is a *good* candidate without exploring all reachable states. For checking *deadlocks* (i.e., states without successors) Godefroid and Khurshid [8] proposed a fitness function that will only retains state with few transitions enabled[4].

Since we have different objectives a new fitness function must be defined. In order to maximize the chances to generate a reachable state, we compute the average outgoing transitions (T_{avg}) of all the states that belong to the initial population. Then the fitness function uses this value as a threshold to detect *good* states. Many fitness function can be considered:

- **equality**: the number of successors of a *good* state is exactly equal to T_{avg}. The motivation for this fitness function is that if there are $N > 1$ independent processes that are deterministic then at every time, any process can progress. Thus a good state has exactly N (equal to T_{avg}) outgoing transitions.
- **lessthan**: the number of successors of a *good* state is less than T_{avg}. The motivation for this fitness function is that if there are $N > 1$ independent deterministic processes that communicate then at any time each process can progress or two processes can be synchronized. This latter case will reduce the number of outgoing transitions

[4] Godefroid and Khurshid [8] do not generate states but finite paths and their fitness fonction analyzes the whole paths to keep only those with few enabled transitions.

- **greaterthan**: the number of successors of a *good* state is greater than T_{avg}. The motivation for this fitness function is that if there are $N > 1$ independent and non-deterministic processes then at any time each processes can perform the same amount of actions or more.

Generation of artificial state. Algorithm 4 presents the genetic algorithm used to generate artificial initial states using the previously defined functions.

The only parameter of this algorithm is the initial population S we want to mutate: S is obtained by performing a swarmed bounded DFS and keeping trace of all encountered states. From the initial population S, a new generation can be generated (lines 4 to 8). At any time the next generation is stored in S' (lines 7 and 3). The algorithm stops after NB_GENERATION generations (line 2). Note that this algorithm can report an empty set according to the fitness function used.

Algorithm 4: The generation of new states.

```
1 Procedure Generate($S \subseteq Q$)
2     for $i \leftarrow 0$ to NB_GENERATION do
3         $S' \leftarrow \emptyset$
4         for $j \leftarrow 0$ to POP_SIZE do
5             $s \leftarrow$ Crossover($S$)
6             Mutation($s$)
7             if Fitness($s$) then
                  $S' \leftarrow S' \cup \{s\}$
8         $S \leftarrow S'$
9     return $S$
```

4 State-Space Exploration with Genetic Algorithm

This section explains how Algorithm 1 can be adapted to exploit the generation of artificial initial states mentioned in the previous section. Algorithm 5 describes this parallel state-space exploration using genetic algorithm. The basic idea is to have a *collaborative portfollio* approach in which threads will share information about CLOSED states. In this strategy, half of the available threads runs a the DFS algorithm presented Section 2, while the other threads perform genetic exploration. This exploration is achieved by three steps:

1. Perform swarmed bounded depth-first search exploration that stores into a set $\mathcal{P}$ all encountered states (line 7). This exploration is *swarmed*, so that each thread has a different initial population $\mathcal{P}$. (Our *bounded*-DFS differs from the literature since it refers DFS that stops after visiting N states.).
2. Apply Algorithm 4 on $\mathcal{P}$ to obtain a new population $\mathcal{P}'$ of artificial initial states (line 8).
3. Apply the DFS algorithm for each element of $\mathcal{P}'$ (lines 9 to 11). When the population $\mathcal{P}'$ is empty, just restart the thread with the initial state ι (line 12).

One can note (line 1) that the *color* enumeration has been augmented with OPEN_GP. This new status may seem useless for now but allows to distinguish states that have been discovered by the genetic algorithm from those discovered by the traditional algorithm. In this algorithm OPEN_GP acts and means exactly the same than OPEN but: (1) this status is useful for the sketch of termination proof below and, (2) the next section shows how we can exploit similar information.

Algorithm 5: Parallel DFS Exploration using Genetic Algorithm.

```
1  enum $color$ = { OPEN, OPEN_GP, CLOSED }
2  $visited$: hashmap of $(Q, color)$
3  $stop \leftarrow \bot$
4  Procedure ParDFS_GP($\langle Q, \iota, \delta, V, \gamma \rangle$ : TS, $n$ : Integer)
5      DFS($\iota$, 1, OPEN) || ... || DFS($\iota$, $\lfloor \frac{n}{2} \rfloor$, OPEN) || DFS_GP($\iota$, $\lfloor \frac{n}{2} \rfloor + 1$) || ... || DFS_GP($\iota$, $n$)
6  Procedure DFS_GP($\iota \in Q$, $tid$ : Integer)
7      $\mathcal{P} \leftarrow$ Bounded_DFS($\iota$, $tid$)     // Swarmed exploration using $tid$ as a seed
8      $\mathcal{P}' \leftarrow$ Generate($\mathcal{P}$)                    // Described Algorithm 4
9      while $\mathcal{P}'$ not empty $\wedge \neg stop$ do
10         $s \leftarrow$ pick one of $\mathcal{P}'$
11         DFS($s$, $tid$, OPEN_GP)
12     if $\neg stop$ then DFS($\iota$, $tid$, OPEN)
```

Termination. Until now we have avoided mentioning one problem: there is no reason that a generated state is a reachable state. Nonetheless even if the state is not reachable, some of its successors (direct or not) may be reachable. Since the number of unreachable states is generally much larger than the number of reachable states, we have to ensure that Algorithm 5 terminates as soon as all reachable states have been explored.

First of all let us consider only threads running the DFS algorithm. Since this algorithm has already been prove (see. [21] for more details), only the intuition is given here. When all the successors of an OPEN state have been visited, this state is tagged as CLOSED. Since all CLOSED states are ignored during the exploration, each thread will restrict parts of the reachable state space. At some point all the states will be CLOSED: even if a thread is still performing its DFS procedure, all the successors of its current state will be marked CLOSED. Thus the thread will be forced to backtrack and stop.

The problem we may have with using genetic algorithm is that all the threads performing the genetic algorithm may be running while all the other ones are idle since all the reachable states have already been visited. In this case, a running thread can see only unreachable states, i.e. OPEN_GP, or CLOSED ones. To handle this problem, a Boolean *stop* is shared among all threads (line 2). When this Boolean is set to $\top$ all threads stop regardless the exploration technique used (line 10, Algorithm 1). We observe line 9 that the use of other artificial states is also stopped, and no restart will be performed (line 12). This Boolean is set to $\top$ only when all the successors of the real initial state have been explored (line 16, Algorithm 1). Thus, one can note that even if a thread using the genetic algorithm visits first all reachable states it will stop all the other threads.

5 Checking Temporal Properties

Safety properties cover a wide range of properties: *deadlock freedom* (there is no state without successors), *mutual exclusion* (two processes execute some critical section at the same time), *partial correction* (the execution terminates in a state that does not satisfies the postcondition while the precondition of the run was satisfied), *etc.* One interesting characteristic of safety properties is that they can be checked using a reachability analysis (as described Section 2). Nonetheless, our genetic reachability algorithm (Algorithm 5) cannot be directly used to check safety properties. Indeed, if a thread (using genetic programming) reports an error we do not know if this error actually belongs to the state space.

Algorithm 6 describes how to adapt Algorithm 5 to check safety properties. To simplify things we focus on checking deadlock freedom, but our approach can be generalized to any safety property. This algorithm[5] relies on both Algorithms 1 and 5 The basic idea is still to launch half of the threads from the initial state ι and the remaining ones from some artificial initial state (line 7).

- For a thread performing reachability with genetic algorithm the differences are quite few. When a deadlock state is detected (line 24) we just tag this state as DEADLOCK_GP rather than CLOSED. This new status is used to mark all states leading to a deadlock state. Indeed since we do not know if the state is a reachable one we cannot report immediately that a deadlock has been found. Moreover we cannot mark this state CLOSED otherwise a counterexample could be lost. This new status helps to solve the problem: when such a state is detected to be reachable, a deadlock is immediately reported. The other modifications are lines 20 and 22: when backtracking, if a deadlock has been found no more states will be explored.
- For a thread performing reachability without genetic algorithm the differences are also quite few. Lines 16 to 18 only check if the next state to process has been marked DEADLOCK_GP. In this case this state is a reachable one and it leads to a deadlock state. We can then report that a deadlock has been found and stop all the other threads. A deadlock can also be reported directly (line 23), if the current state is a deadlock.

Deadlock – sketch of proof. Due to lack of space only the schema of a proof, that the algorithm will report a deadlock if and only if there exists a reachable state that has no successors, is given here.

Theorem 1. For all systems S, the algorithm terminates.

Theorem 2. A thread reports a deadlock iff $\exists s \in Q, post(s) = \emptyset$.

To simplify the sketch of proof, we denote by *classical thread* a thread that does not perform genetic algorithm while the other threads are called *gp threads*. The following invariants hold for all lines of Algorithm 6:

Invariant 1. If *stop* is $\top$ then no new state will be discovered.

Invariant 2. A deadlock state can only be OPEN, OPEN_GP or DEADLOCK_GP.

Invariant 3. No direct successor of a CLOSED state is a deadlock state.

Invariant 4. A state is CLOSED iff all its successors that are not on the thread's recursive stack are CLOSED.

Invariant 5. Only *gp threads* can tag a state DEADLOCK_GP.

Invariant 6. A state is DEADLOCK_GP iff it is a deadlock state or if one of its successors (direct or not) is a deadlock state.

[5] Main differences have been highlighted to help the reader.

Algorithm 6: Parallel Deadlock Detection Using Genetic Algorithm.

```
enum $color$ = { OPEN, OPEN_GP, CLOSED , DEADLOCK_GP }
$visited$: hashmap of $(Q, color)$
$stop \leftarrow \bot$
$deadlock \leftarrow \bot$
Procedure ParDeadlockGP($\langle Q, \iota, \delta, V, \gamma \rangle$ : TS, $n$ : Integer)
    DeadlockDFS($\iota$, 1, OPEN) || ... || DeadlockDFS($\iota$, $\lfloor \frac{n}{2} \rfloor$, OPEN) ||
            DeadlockDFS_GP($\iota$, $\lfloor \frac{n}{2} \rfloor + 1$) || ... || DeadlockDFS_GP($\iota$, $n$)
Procedure DeadlockDFS($s \in Q$, $tid$ : Integer, $status$ : $color$)
    if $s \notin visited$ then $visited$.add($s$, $status$)
    else if $visited[s]$ = CLOSED then return
    $todo \leftarrow$ shuffle(post($s$), $tid$)
    while ($\neg stop \wedge \neg todo.isempty()$) do
        $s' \leftarrow todo.pick()$
        if $s'$ is in the current recursive DFS stack then continue
        if ($s' \notin visited \vee visited[s'] \neq$ CLOSED) then
            if $s' \in visited \wedge visited[s']$ = DEADLOCK_GP $\wedge$ $status$ = OPEN then
                $deadlock \leftarrow \top$; $stop \leftarrow \top$
                break
            DeadlockDFS($s'$, $tid$, $status$)
            if $visited[s']$ = DEADLOCK_GP $\wedge$ $status$ = OPEN_GP then
                $visited[s] \leftarrow$ DEADLOCK_GP
                return
    if post($s$) $= \emptyset$ $\wedge$ $status$ = OPEN then $deadlock \leftarrow \top$; $stop \leftarrow \top$
    if post($s$) $= \emptyset$ $\wedge$ $status$ = OPEN_GP then $visited[s] \leftarrow$ DEADLOCK_GP
    else $v[s] \leftarrow$ CLOSED
    if ($s = \iota$) then $stop \leftarrow \top$
Procedure DeadlockDFS_GP($\iota \in Q$, $tid$ : Integer)
    $\mathcal{P} \leftarrow$ Bounded_DFS($\iota$, $tid$) // Also check deadlock during this DFS
    $\mathcal{P}' \leftarrow$ Generate($\mathcal{P}$)
    while $\mathcal{P}'$ not empty $\wedge$ $\neg stop$ do
        $s \leftarrow$ pick one of $\mathcal{P}'$
        DeadlockDFS($s$, $tid$, OPEN_GP)
    if $\neg stop$ then DeadlockDFS($\iota$, $tid$, OPEN)
```

Invariant 7. Only *classical thread* can report that a deadlock has been found.
Invariant 8. If a state is reachable then all its direct successors are reachable.

Invariants, combined to the sketch of proof of the previous section, helps to prove Theorem 1: the algorithm stops either because a deadlock is detected or because all reachable states have been explored. These invariants establish both directions of Theorem 2: invariant 7 and 8 are the most important for correctness.

Discussion. The verification of complex temporal properties involves the exploration of an automaton which is the result of the synchronous product between the state space of the system and the property automaton. Thus a state is composed of two parts: the system state and the property state. Genetic algorithms presented so far can then be applied by considering that the property state is a variable just like the other system's variables. The adaptation of Algorithm 6 for checking liveness properties is straightforward: when a *gp thread* detects an accepting cycle, all the states forming it are tagged with an ACCEPTING_CYCLE status. When a *classical thread* detects such a state, a counterexample is raised.

6 Evaluation

Benchmark Description. To evaluate the performance of our algorithms, we selected 38 models from the BEEM benchmark [16] that cover all types of models described by the classification of Pelánek [17]. All the models where selected such that Algorithm 1 with one thread would take at most 40 minutes on Intel(R) Xeon(R) @ 2.00GHz with 250GB of RAM. This six-core machine is also used for the following parallel experiments[6]. All the approaches proposed here have been implemented in Spot [6]. For a given model the corresponding system is generated on-the-fly using DiVinE 2.4 patched by the LTSmin team[7].

Reachability. To evaluate the performance of the algorithm presented Section 4 we conducted 9158 experiments, each taking 30 secondes on the average. Table 1 reports selected results to show the impact of the fitness function and the threshold over the performance of Algorithm 5 with 12 threads (the maximum we can test). For each variation, we provide *nb* the number of models computed within time and memory constraints, and *Time* the cumulated walltime for this configuration (to run the whole benchmark). For a fair-comparison, we excluded from *Time* models that cannot be processed. Table 1 also reports state-of-the-art and **random** (used to evaluate the accuracy of genetic algorithms by generating random states as seed state). This latter technique is irrelevant since it is five time slower than state-of-the-art and only process 32 models over 38.

If we now focus on genetics algorithms, we observe that the threshold highly impacts the results regardless the fitness function used: the more the threshold grows, the more models are processed within time and memory constraints.

The table also reports the best threshold[8] for all fitness function, i.e. 0.999. It appears that **greaterthan** only processed 37 models: this fitness function does not seem to be a good fitness function since (1) it tends to explore useless parts of the state-space and (2) the variations of the threshold highly impacts the performance of the algorithm. All the other fitness function provide similar results for a threshold fixed at 0.999. Nonetheless we do not recommend **equality** since a simple variation of the threshold (0.7) could lead to extremely poor

[6] For a description of our setup, including selected models, detailed results and code, see `http://www.lrde.epita.fr/~renault/benchs/VECOS-2018/results.html`

[7] See http://fmt.cs.utwente.nl/tools/ltsmin/#divine for more details. Also note that we added some patches (available in the webpage) to manage out-of-bound detection.

[8] We evaluate other thresholds like 0.9999 or 0.99999 but it appears that augmenting the threshold does not increase performance, see the webpage for more details.

	Threshold							
	0.7		0.8		0.9		0.999	
	nb	*Time (ms)*	*nb*	*Time (ms)*	*nb*	*Time (ms)*	*nb*	*Time (ms)*
greaterthan	35	1 041 015	35	970 248	35	1 000 184	37	900 468
equality	35	3 217 183	35	965 259	35	934 947	38	907 148
lessthan	35	972 038	35	951 767	35	928 978	38	904 776
lessstrict	35	970 668	35	983 225	35	935 319	38	894 131
	No threshold							
random	(trivial comparator to evaluate genetic algorithms)						32	5 079 869
Algorithm 1	(state-of-the-art with 12 threads)						38	978 711

Table 1. Impact of the threshold and the fitness function on Algorithm 5 with 12 threads (NB_GENERATION=3, INIT=1000, POP_SIZE=50). The time is expressed in millisecond and is the cumulated time taken to compute the whole benchmark (38 models); *nb* is the number of instances resolved with time and memory limits.

results. Our preference goes to **lessthan** and **lessstrict** since they seem to be less sensitive to threshold variation while achieving the benchmark 9% faster than state-of-the-art algorithm. Thus, while the speedup for 12 threads was 3.02 for state-of-the-art algorithm, our algorithm achieves a speedup of 3.31.

Note that the results reported Table 1 include the computation of the artificial initial states. On the overall benchmark, this computation take in average slightly less than 1 second per model (30 seconds for the whole benchmark). This computation has a negligible impact on the speedup of our algorithm.

We have also evaluated (not reported here, see webpage for more details) the impact of the size of the initial population and the size of each generation over the performance. It appears that augmenting (or decreasing) these two parameters deteriorate the performance. It is worth noting that the best value of all parameters are classical values regarding to state-of-the-art genetic algorithms. Finally, for each model (and **lessthan** as fitness), we compute a set of artificial initial states and run an exploration algorithm from each of these states. It appears that 84.6% of the 7 866 005 486 generated states are reachable states.

Safety properties. Now that we have detected the best values for the parameters of the genetic algorithm we can evaluate the performance of our deadlock detection algorithm. In order to evaluate the performance of our algorithm we conduct 418 experiments. The benchmark contains 21 models with deadlocks and 17 models without. Table 2 compares the relative performance of state-of-the-art algorithm and Algorithm 6. For this latter algorithm, we only report the two fitness functions that give the best performance for reachability. Indeed, since Algorithm 6 is based on Algorithm 5 we reuse the best parameters to obtain the best performance. Results for detecting deadlocks are quite disappointing since our algorithm is 15% to 30% slower. A closer look to these results show that deadlocks are detected quickly and Algorithm 6 has degraded performance due to the computation of artificial initial states.

	Algorithm 1		Algorithm 6			
	(state-of-the-art)		**lessthan**		**lessstrict**	
	Time (ms)	States	Time (ms)	States	Time (ms)	States
Deadlocks	2 888	7.01E^{6}	3 713	5.87E^{6}	3 414	5.47E^{6}
No deadlocks	516 152	5.79E^{8}	462 881	6.73E^{8}	468 683	6.82E^{8}

Table 2. Comparison of algorithms for deadlock detection. Each runs with 12 threads, and we report the variation of two different fitness functions: **lessstrict** and **lessthan**. Results presents the cumulated time and states visited for the whole benchmark.

On the contrary we observe that our algorithm is 10% faster (regardless whether we use **lessthan** or **lesstrict**) than the classical algorithm when the system has no deadlock. One can note that this algorithm performs better than simple reachability algorithm. Indeed, even if the system has no deadlock: the algorithm can find non-reachable deadlock. In this case, the algorithm backtracks and the next generation is processed. This early backtracking force the use of a new generation that will helps the exploration of the reachable states. To achieve this speedup, we observe an overhead of 13% for the memory consumption. The use of dedicated memory reduction techniques could help to reduce this footprint.

Discussion. Few models in the benchmark have a linear topology, which can be considered as the perfect one for the algorithms presented in this paper. Nonetheless, we observe a global improvement of state-of-the-art algorithm. We believe that other fitness function (based on interpolation or estimation of distribution) could help to generate better states, i.e. deep with respect to many DFS orders.

7 Conclusion

We have presented some first and new parallel exploration algorithms that rely on genetic algorithms. We suggested to see variables of the model as genes and states as chromosomes. With this definition we were able to build an algorithm that generates artificial initial states. To detect if such a state is relevant we proposed and evaluate various fitness functions. It appears that these seed states improve the swarming technique. This combination between swarming and genetic algorithms has never been proposed and the benchmark show encouraging results (10% faster than state-of-the-art). Since the performance of our algorithms highly relies on the generation of good artificial states we would like to see if other strategies could help to generate better states.

This work mainly focused on checking safety properties even if we proposed an adaptation for liveness properties. A future work would be to evaluate the performance of our algorithm in this latter case. We also want to investigate the relation between artificial state generation and POR, since both rely on the analysis of processes variables. Finally, we strongly believe that this paper could serve as a basis for combining parametric model-checking with neural network.

References

1. P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In ICFEM'98, pp. 46–54, december 1998.
2. J. Barnat, L. Brim, and P. Ročkai. Scalable shared memory LTL model checking. STTT, 12(2):139–153, 2010.
3. V. Bloemen and J. van de Pol. Multi-core SCC-Based LTL Model Checking, pp. 18–33. Lecture Notes in Computer Science. Springer, 2016.
4. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In Proc. of the Fifth Annual IEEE Symposium on Logic in Computer Science, pp. 1–33, 1990. IEEE.
5. C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithm for the verification of temporal properties. In CAV'90, vol. 531 of LNCS, pp. 233–242. Springer, 1991.
6. A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In ATVA'16, vol. 9938 of LNCS, pp. 122–129. Springer, Oct. 2016.
7. H. Garavel, R. Mateescu, and I. Smarandache. Parallel State Space Construction for Model-Checking. Technical Report RR-4341, INRIA, 2001.
8. P. Godefroid and S. Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms, pp. 266–280. Springer, Berlin, Heidelberg, 2002.
9. P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In CAV'92, vol. 663 of LNCS, pp. 178–191. Springer, 1992.
10. J. H. Holland. Genetic algorithms. Scientific American, 1992.
11. G. J. Holzmann. On limits and possibilities of automated protocol analysis. In PSTV'87, pp. 339–344. North-Holland, May 1987.
12. G. J. Holzmann and D. Bosnacki. The design of a multicore extension of the SPIN model checker. IEEE Transaction on Software Engineering, 33(10):659–674, 2007.
13. G. J. Holzmann, R. Joshi, and A. Groce. Swarm verification techniques. IEEE Transaction on Software Engineering, 37(6):845–857, 2011.
14. G. Katz and D. A. Peled. Synthesis of parametric programs using genetic programming and model checking. In INFINITY'13, pp. 70–84, 2013.
15. A. Laarman, E. Pater, J. Pol, and H. Hansen. Guard-based partial-order reduction. STTT, pp. 1–22, 2014.
16. R. Pelánek. BEEM: benchmarks for explicit model checkers. In SPIN'07, vol. 4595 of LNCS, pp. 263–267. Springer, 2007.
17. R. Pelánek. Properties of state spaces and their applications. International Journal on Software Tools for Technology Transfer (STTT), 10:443–454, 2008.
18. R. Pelánek, T. Hanžl, I. Černá, and L. Brim. Enhancing random walk state space exploration. In FMICS'05, pp. 98–105. ACM Press, 2005.
19. D. Peled. Combining partial order reductions with on-the-fly model-checking. In CAV'94, vol. 818 of LNCS, pp. 377–390. Springer, 1994.
20. J. H. Reif. Depth-first search is inherently sequential. Information Processing Letters, 20:229–234, 1985.
21. E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Variations on parallel explicit model checking for generalized Büchi automata. International Journal on Software Tools for Technology Transfer (STTT), pp. 1–21, Apr. 2016.
22. H. Sivaraj and G. Gopalakrishnan. Random walk based heuristic algorithms for distributed memory model checking. Electronic Notes in Theoretical Computer Science, 89(1):51 – 67, 2003.
23. A. Valmari. Stubborn sets for reduced state space generation. In ICATPN'91, vol. 618 of LNCS, pp. 491–515, 1991. Springer.