Parallelizing random-walk based model checking

• **Thang H. Bui**

Ho Chi Minh city University of Technology - Vietnam National University Ho Chi Minh City, Vietnam

(Manuscript Received on May 18nd, 2014, Manuscript Revised August 28nd, 2015)

ABSTRACT

In model checking, a formal methods technique to verify a system with some desired properties, the guidance techniques have been employed a long time ago in driving the verification into area of `error` in the state space. Another technique is to choose the next state to be explored in a walk randomly to avoid the `wrong` guidance. When the latter is a non-exhaustive technique in the sense that only a manageable number of walks are carried out before the search is terminated, it does scale well. In enhancing the technique to use recently powerful parallel/multi-core systems, research on parallelizing the algorithm shall be carried out. In this work, we propose a method that parallelizes the random-walk algorithm. It also increases the completeness of the non-exhaustive algorithm. The experimentation has shown the great improvement of the proposed algorithm compares to the original once.

Keywords: *Model checking, random-walk, parallelization, multi-core.*

1. INTRODUCTION

When software and hardware nowadays are used widely in society and human life, their correctness is the most challenge in system designing and implementing. Therefore, model checking, a formal methods approach, that verifies a given system with a desired property, is used recently in guarantee the correctness of the system. Traditionally, it is an exhaustive search on the state space of a system to ensure that no state violates the given property. Unfortunately, it faces the `state space explosion` problem in searching the state space. Although the use of heuristic guidance can improve the performance, model checking in reality is said not to scale well. Random-walk search, that draws walks through any large state space randomly to find errors, in contract, does scale well [1, 2, 3, 4, 5, 6].

Research on randomization has been carried out decades ago including random-walk [1, 14], guided random-walk [2, 3, 4, 5, 6]. In random-walk based model checking, the search verifies the desired property along a path (walk) through the state space, in which each state is chosen randomly among the available states of the previous one (in the path). This approach examines one path at the time then possibly it needs only a minimum storage to keep explored states. In some special case, we may need to remember only the current state in the walk. Of course, as it does not scan the whole state space (or we do not know it does), it is an incomplete solution [6] and suggested to be used in `bug hunting` only.

The guided random-walk based model checking is an application of heuristic search into random-walk [3]. In the kernel, the selection method to select the next state in a random walk is biased such that the `bester` states will have a higher probability to be chosen than the others. This approach has been shown to have impressive performance in model checking [3]. Once again, it is still an incomplete search.

Another model checking research in randomization is randomized state space search [15] and randomized heuristic search [16]. The first one applies a random selection into the exhausted search at every time the search selects a state to be explored. The latter is to avoid `wrong` heuristic (or local optimization) by applying randomization in selection the next state to be explored among the `best` states.

Research in paralleling model checking has also been carried out long time ago. It is divided into two main directions: parallel using distributed systems [9, 17] and using multi-core system [12]. The main difference between them is how the main memory is managed: distributed (in distributed systems) or centralized (in multi-core systems). Of course, the performance of the model checking based both on the number of the computation components (computer or core) and the cost of distributing data (explored states) among those components. When the multi-core systems are used widely nowadays, this research area has been focused more recently. Unfortunately, model checking using parallel/multi-core systems is still facing its `very old` problems such as state space explosion.

This work is to propose an application of parallelization and random walk in to the model checking for reducing the affect of the state space explosion (by using random walk) and increasing the overall performance (by using parallelization).

The rest of the paper is as follows: In Section 2, some background knowledge about model checking, random walk, parallelized model checking is introduced. The proposed approach is in Section 3. The experimentation is in Section 4. The summary and future work is in the last section.

2. BACKGROUND AND RELATED WORKS

2.1. Model checking

Model checking is a method to verify if a given (a model of) a system M satisfies a desired property φ, $M \models \varphi$. A model of a system is usually defined as a transition system and the property is given as a temporal logic expression.

Definition 1 [Transition system] *A transition system is a 3-tupe $M = (S, S_0, \Delta)$, in which S is a set of states, $S_0 \subseteq S$ is a set of initial states, $\Delta = S \times S$ is a set of transition.*

An *execution path* is defined as *s0s1…*, in which each $<s_i,s_{i+1}> \in \Delta$. An execution path is a *counter-example* if it is an example to show a violation of system *M* to the property φ.

For reasoning about time, a property can be expressed in temporal logic, an extension of propositional logic to consist of temporal operators such as [] (always), <> (eventually), etc. (Linear Temporal Logic - LTL) and may consists of some qualifier such as ∀ (for all) and ∃ (exists) (Computational Tree Logic - CTL).

There are two main approaches in checking a model: explicit-state, which uses theories of automata, and symbolic, which uses symbolic representation for checking. In this work, we focus on the first approach only.

2.2. Automata-theoretic model checking

In this approach, both model and property are converted into Büchi automata, a kind of automata which accepts infinity languages. A (infinity) word is accepted in this automata if and only if an accepting/final state occurs infinity often in the word.

Suppose that B_M and $B_{\neg\varphi}$ are Büchi automata for the model and the negation of the property, respectively. If the language of the production of those automata is empty, $L(B_M \times B_{\neg\varphi}) = \emptyset$, the model does satisfy the property. In reverse, if it is not empty, a accepting word of that language will be a counter-example.

To do so, a search in the state space of the production $B = B_M \times B_{\neg\varphi}$ is performed. If the current execution path is accepted, it will be returned as a counter-example. The famous search algorithm in this approach is Nested Depth First Search - NDFS [7]. In general, the algorithm has two Depth First Search (DFS) loops, the outer searches for the first accepting state, and then the inner starts its own search from that state back to that state again (a strong connected component in the path). In this case, the current path, which ensures that a final state will be appeared infinity often, is a counter-example.

2.3. Randomization in model checking

The NDFS is simply an exhausted search, so it faces the state space explosion problem. To avoid that problem, [1, 14] applies random-walk to search the state space. There is only one loop: explore next state randomly until a strong connected component, which contains a final state, in the current path is confirmed. Each path can be limited in length (*cut off* point) and the search can be restarted. Research in [10] has shown that, the number of walks can be determined by $N = ln(\delta) / ln(1\text{-}\varepsilon)$, in which the probability that an accepting path can be found if make more walk is less than δ, given that the probability of a walk is greater than or equal ε. The *cut off* can be the point when the probability of a walk drops below ε. The algorithm is illustrated in Listing 1.

```
1. for N times
2.    s = rand(S0)
3.    while (!is_goal(s) and !cut_off)
4.       s = rand(successors(s))
5.       if is_goal(s)
6.          return counter-example(s)
7. return nill
```

Listing 1. Random-walk based algorithm

In this algorithm, the function *rand* returns one element of the input set randomly, *successors* returns all successors of a input state, *is_goal* checks for an accepted path, and *counter-example* builds the counter-example from the initial state to the input state.

To due with the incompleteness of the random-walk based algorithm, a `nearly-complete` random algorithm has been proposed. It remembers all explored states and keeps a fringe of to-be-explored states. At every step, a state is the fringe will be picked up randomly to be explored [11]. The algorithm is in Listing 2. It is actually a complete algorithm that explores the whole state space, but one can stop earlier at any time. In this algorithm, function *rand_remove* picks up one element from the input set randomly.

To avoid memory overload, we can choose to not to store all explored states (as in line 8). In this case, the walk may re-scan visited states and then reduces the performance.

In contract, work of [15] just applies the randomization into an exhausted search to avoid `model-bias`, a special cases in DFS in which the direction to the counter-example is always explored last. This leads to the case that, the error state can only be found when almost all states have been explored [1]. In that work, at every search step, a next state is chosen randomly to be explored. They later proposes that the whole algorithm can be paralleled if the main algorithm is duplicated into some other machines [8]. The drawback is that, each machine runs separately without any communication, so some machines re-do what the other machine already done.

```
1. explored = Ø
2. fringe = S0
3. while (fringe ≠ Ø and !cut_off)
4.   s = rand_remove(fringe)
5.   if is_goal(s)
6.     return counter-example(s)
7.   fringe = fringe∪(successors(s) \ explored)
8.   explored = explored ∪ {s}
9. return nill
```

Listing 2. Nearly-complete random-walk based algorithm

2.4. Multi-core model checking

As modern computer systems have been armed multi-core CPUs for performance improvement, some model checking algorithms have been proposed to gain advantages from the multi-core architecture.

The most famous multi-core approach is to separate the outer and inner DFS into different concurrent processes, called *dfs_blue* and *dfs_red*, respectively [12] and using a set of colors (red, blue, cyan) to indicates visiting status of a state. By this approach, the outer DFS thread (dfs_blue) does not wait for the inner DFS thread (dfs_red) to be finished.

[13] later proposed another way to increase the number of threads that the same time by adding a new color (pink) and sharing all status of each state (its colors). The performance is improved dramatically.

3. RANDOM-WALK BASED MULTI-CORE MODEL CHECKING

As mentioned earlier in this text, in bug-hunting manner, we have to consider two problems: state space explosion and `model-bias`. The random-walk have been shown to solve both problems well [1, 6]. To improve the performance of this approach, a parallelization will be applied.

3.1. System architecture

The architecture of the whole algorithm has been illustrated in Figure 1. The whole system is a network of parallel workers (depicted as *worker*, …). Each worker has its own local queue to store the to-be-explored states and to receive the to-be-explored states that comes from other workers in the system.

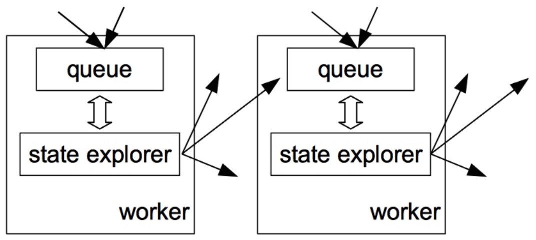

Figure 1. System architecture

In this approach, the *state explorer* component in each worker gets a state from its queue randomly and generates all successors of that state. Those generated states can be put back to the queue or dispatched to other workers randomly. If each worker is initialized by different random seeds, the degree of randomization is increased.

By this design, our proposed method can be executed in both random-walk modes: (1) basic random-walk mode (*rw*), in which only a few successors of the current state is explored at every step (see Listing 1); and (2) nearly-complete mode (*rwnc*), in which all successors of the current state are put in the fringe (see Listing 2).

In the first mode, the state explorer in each worker will put a successor to its queue and then may randomly send another successor, if any, to another worker. In some case, a worker may have more than one state in its queue (it put one in its queue when receiving another one from the network, simultaneously). In this case, it likely become a light-weight version of the second mode. As mentioned earlier in this text, a number of trials will be performed before terminating the whole search if no accepting random walk was found.

In the second mode, the fringe is partitioned and distributed in all workers (such that the queues of all workers form the fringe). When each worker only picks a state from its own queue (partial fringe), there is no mutual exclusion when the workers intercept each others in selecting a state (from the fringe). Of course, there is still problem when a worker receives more than one state dispatched from the network at the same time. But that problem can be solved independently in each worker.

This architecture allows us to install this system in a distributed system (distributed memory) when each worker can work independently. To reduce the bandwidth of the network, data may be redirected to the local queue more often than to be delivered over the network. The dispatching (of states) is not a problem in a share-memory/multi-core system.

In case of completeness, the network needs a share storage to store all explored states. That could be a problem in a distributed system.

A failure recovery component can also be included in this network to discover crashed worker. In this case, the local queue of the crashed one will be re-distributed to other workers.

3.2. The algorithm

The algorithm is illustrated in Listing 3 and 4. The algorithm is at follows.

The main procedure starts all the workers (function *start_all_workers* in line 2). Each worker will do nothing when its queue is initialized empty (line 10, 11). Then, depend on the mode of the algorithm (mode=*rw* for basic random-walk and mode=*rwnc* for nearly-complete random-walk), it initializes the queue of the worker$_0$ (line 3-8). In *rw* mode, the algorithm carries out a number of trials (N times) if no accepted path is found (*goal_found*). Of course, it has to clear all queue of all workers before starting the next trail (function *clear_all_queues* in line 5). In this mode, the queue of the first worker (queue$_0$) is initialized by only one initial state randomly. In the second mode, that queue is initialized by the set of all initial states. Note that, all the queues together act as the fringe. At the end, all workers should be terminated (function *stop_all_workers* in line 9).

```
proc main
1.  explored = Ø
2.  start_all_workers()
3.  if mode=rw
4.    for N times and !goal_found
5.      clear_all_queues()
6.      queue[0] = {rand(S0)}
7.  else if mode=rwnc
8.    queue[0] = S0
9.  stop_all_workers()
```

Listing 3. Parallel random-walk based algorithm (1)

All worker works exactly the same without any priority. It scans its own queue for any activity. When the queue is empty, it is inactive (line 11). We can also modify this line to put this worker into sleeping mode for a while when necessary. Similar to the algorithm in Listing 2, when the current path is at the cut off point, the current path can be rejected.

The main different to the algorithm listing in Listing 1 & 2 is in line 17-26. In *rw* mode, a successor state will be selected randomly to put into the queue of the worker (line 19). A randomly decision will be made (function *random_decide* in line 20) to send another successor state (function *send* in line 22) to a random worker (function *rand_select* in line 21 return a worker randomly).

At this step, we can only select an inactive worker that has no state in its queue, to send a state to. This will make all the workers work as much as possible. And hence, we can use the network effectively.

In mode *rwnc* mode, all successor states will be dispatched (line 23-26) to all workers, including itself, randomly (function *rand_select* in line 25 picks one worker randomly).

The function *send(s, j)*, that distributes state around the network, should be *synchronized* when there may be more than one other workers send states to a worker at the same time.

```
proc worker(i)
10. while (true)
11.   if (queue[i] = Ø) continue  /* inactive */
12.   s = rand_remove(queue[i])
13.   if is_goal(s)
14.     goal_found = true
15.     return counter-example(s)
16.   if (cut_off) continue
17.   Sc = successors(s) \ explored
18.   if mode=rw
19.     queue[i] = queue[i]∪(rand_remove(Sc))
20.     if random_decide()
21.       if (workerj = rand_select(workers))
22.         send(rand_remove(Sc), workerj)
23.   else if mode=rwnc
24.     for all s' ∈ Sc
25.       if (workerj = rand_select(workers))
26.         send(s', workerj)
27.   explored = explored ∪ {s}
28. return 0

proc send(s, j) synchronized
29.   queue[j] = queue[j] ∪ {s}
```

Listing 4. Parallel random-walk based algorithm (2)

3.3. Compare to related works

This proposed approach is totally different with all works discussed in section 2.3 and 2.4. The work of [8] different with our work in two manners: the random-walk approach and the parallelization approach. When applying randomization, they randomly shuttle the queue in an exhausted search. When applying the parallelization, they duplicate one worker multiply with different random seed. All workers have no communication with each other. We also have a feature that allow us to reduce the coupling if we reduce the frequency of sending states around (in line 21, 22 and 25).

Our proposed work also different with the works of [12] and [13]. Their works play around with sharing the status of states (colors) in NDFS (or blue-red search). The latter increase the number of workers by adding some more colors. Importantly, their algorithms are complete as in the worst case, they scan the whole state space. Our approach in general is incomplete as it is a random-walk based algorithm. Although we have one mode in our algorithm to allow the completeness, we actually do not focus in that case. We work for \`bug-hunting\`, so the random, fast and lightweight features are very important. In parallelizing the algorithm, we distribute states around the workers to increase the balance workload of the whole network. Moreover, as mentioned earlier in this text, our algorithm has a very nice feature to reduce the network workload by not sending states over the network frequently.

4. EXPERIMENTATION

4.1. Experimentation setup

The system: the whole algorithm has been built in Spin model checker[1].

Benchmarks: we re-uses benchmark using in previous works published in [1, 6]. They are \`error\` versions of leader-election (*l3_error*,

[1] http://spinroot.com

l4_error, *l5_error*), peterson (*peterson3*, *peterson4*), and mutual-exclusion (*mutex8*, *mutex12*) protocols. We are not going to use `correct` versions when our work is not an exhausted search, then in the worst case, it may return no counter-example and conclude that there is no error with a probability [1].

Environment: all experiments are carried out in a 16-core server 3.27GHz with 16GB of RAM running CentOS 6.5. All data are averaged over 10 executions.

4.2. Experimentation result

The experimental results for all case studies with different modes are shown in Table 1 and illustrated in Figure 2, Figure 3 and Figure 4. We use two metrics in measuring the performances of all the algorithm: running time in seconds and the length of the counter-example found by the algorithms. They are named **time (s)** and **cxl** in Table 1, respectively.

The algorithms are as follow:

- Parallel Breadth First Search (BFS_P): parallel version of the Breadth First Search provided by Spin.

- Parallel BFS with randomization approach (BFS_P_RW): an application of random to the BFS_P, similar to the randomization in the work of [8].

- Parallel Nested Depth First Search (NDFS_P): the parallel version of the core algorithm (NDFS) using in Spin.

- Parallel NFDS with our nearly-complete random-walk approach (NDFS_P_RW): the main contribution in this work.

Table 1. Experimental results

	BFS_P		BFS_P_RW		NDFS_P		NDFS_P_RW	
	time (s)	cxl	time (s)	cxl	time (s)	cxl	time (s)	cxl
l3_error	1.2383	83.0	1.3183	97.8	0.7780	151.0	0.5020	164.6
l4_error	9.0160	91.0	10.2040	107.8	0.6860	141.2	0.6680	421.8
l5_error	out of mem		out of mem		0.7000	196.8	0.7025	174.3
mutex8	0.0720	31.0	0.0800	35.0	0.5600	10015.0	0.4560	10019.0
mutex12	0.4220	39.0	0.5160	43.8	0.7500	10019.0	0.6220	10019.0
peterson3	0.0360	141.0	0.0460	148.6	0.0040	135.8	0.0020	136.2
peterson4	0.0400	141.0	0.0460	147.4	0.0080	145.0	0.0040	137.0

It is easy to see that, the randomization applied to Breadth First Search (BFS) slows down the system (around 7~13%). It is reasonable when the randomization takes time in shuttling the states in the queue. Of course, we can believe that the randomization may overcome the `bias` of the system-under-test. In this experiment, it is not that case. We suggest that, the work of [8] may have no advantage in applying into BFS.

Another observable issue in this experiment is that the NDFS_P takes advantages to that of BFS_P in some cases, but some others. For example, it reduces the execution time in checking the leader-election protocol (l4_error) and peterson protocol (peterson4) around ~90%. However, it takes 70% more in the mutual-exclusion (mutex12) protocol. It is illustrated in Figure 2, Figure 3 and Figure 4 (BFS_P and NDFS_P).

As expected, our algorithm (NDFS_P_RW) overcomes NDFS_P in all the cases. It reduces the running time in checking the peterson3 protocol to 100% (0.0040s down to 0.0020s). We believe that, when the random-walk is applied, the system has chances to avoid the `bias` of the system-under-test then it may reach the `error` faster in some case. Moreover, the randomization makes the checking stable, no matter how `bias` the system-under-test is. Of course, naturally, the NDFS_P_RW, a version of NDFS, cannot guarantee about the optimal counter-example, if any, as in BFS.

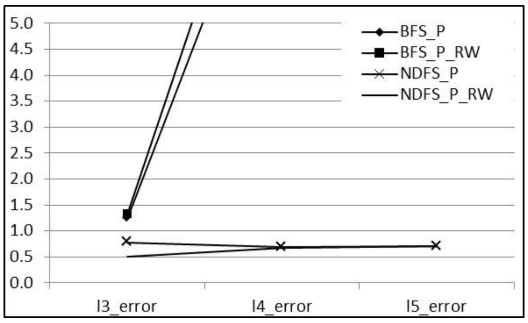

Figure 2. Running times for leader-election protocol

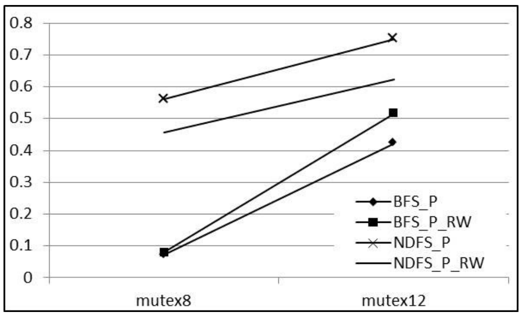

Figure 3. Running times for mutual-exclusion protocols

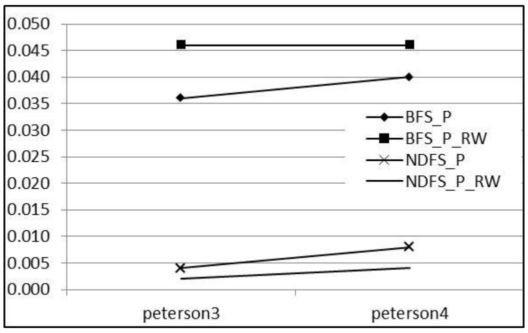

Figure 4. Running times for peterson protocols

4.3. Thread to validity

In this work, we propose an algorithm with two execution modes: random-walk and nearly-complete random-walk. The experimentation is for the nearly-complete mode only. The `purely` random-walk has not be experimented yet and may has a better or worse performance. That experimentation will be carried out in the near future.

We also mentioned that, our algorithm can work on the real distributed system over a network of separated computers using separated memory. That experimentation will also be carried in the future.

The implementation of our algorithm is in the Spin tool, then the correctness of the related features such as analysing the source models, the kernel of the model checking, etc. is trustable.

5. SUMMARY AND FUTURE WORK

In this work, we have proposed a new approach in parallelizing the random-walk based model checking. This approach allows a network of workers work simultaneously to explore the state space. It has a feature to share state around the network to increase the chance in finding error in the system-under-test and increase the overall performance. It also has a mode to consider the complete search through the state space by maintaining the explored states. Our proposed approach is shown to be different with related works.

The performance of the proposed method is quite extremely good compare to the parallel version of NDFS algorithm in the Spin itself. That is expected when we do not suffer from the `bias` of the system-under-test as in Spin. Of course, in some circumstance, Spin will be best as it may luckily reach the goal on it first few attempt.

There are still works to do with our approach: (1) install it into a distributed system (and distributed memory) to find out the configuration of dispatching states around the network; (2) study about the cut off points as it may affect the completeness of the algorithm; (3) applying heuristic into random-walk as it has been research years ago in [6].

Song song hóa kiểm tra mô hình dựa trên đường đi ngẫu nhiên

• **Bùi Hoài Thắng**

Trường Đại học Bách Khoa, ĐHQG-HCM

TÓM TẮT

Phương pháp kiểm tra mô hình là một phương pháp hình thức được dùng rộng rãi trong thời gian gần đây để kiểm định tính đúng đắn của các hệ thống phần mềm và phần cứng. Các kỹ thuật dẫn hướng dùng heuristic đã được sử dụng trong một thời gian dài để dẫn hướng việc tìm kiếm vào trong các vùng có khả năng có sai sót. Một kỹ thuật khác là dùng đường đi ngẫu nhiên, nghĩa là chọn các trạng thái đi kiểm tra một cách ngẫu nhiên, để tránh việc dẫn hướng sai lầm. Kỹ thuật này là một kỹ thuật không phải vét cạn nên nó không (thực sự) cần bộ nhớ lớn và như vậy có thể tránh né sự bùng nổ tổ hợp trong bài toán kiểm tra mô hình. Để nâng cao hiệu quả của hướng nghiên cứu này và tận dụng sức mạnh của các hệ thống song song/đa nhân gần đây, việc áp dụng giải pháp song song hóa là rất cần thiết. Nghiên cứu này đề xuất một phương pháp song song hóa giải thuật đường đi ngẫu nhiên để nâng cao hiệu suất kiểm tra. Ngoài ra, nghiên cứu này cũng quan tâm đến việc tăng cường tính đầy đủ của giải thuật kiểu không vét cạn như thế này. Thực nghiệm cho thấy, giải pháp đề ra rất hiệu quả.

Từ khóa: *Kiểm tra mô hình, đường đi ngẫu nhiên, song song hoá, đa-nhân.*

REFERENCES

[1]. T.H. Bui and A. Nymeyer, *ranSPIN: a random-walk based model checker*, Workshop on Advanced Computing and Applications (ACOMP'08), HoChiMinh City, Vietnam, pp. 38-48, 2008.

[2]. T.H. Bui and A. Nymeyer, *The spin on guided random search in verification*, ICSTW 2008, pp. 170-177, 2008.

[3]. T.H. Bui and A. Nymeyer, *Formal verification based on guided random walks*, iFM'09, vol. 5423 of LNCS, pp. 72-87, Springer-Verlag, 2009.

[4]. T.H. Bui and A. Nymeyer, *Formal model simulation: Can it be guided?*, SSBSE'09, pp. 93-96, IEEE CS, 2009.

[5]. T.H. Bui and A. Nymeyer, *Integrated guided model checking and model simulation*, Submitted, 2009.

[6]. T.H. Bui and A. Nymeyer, *Heuristic sensitivity in guided random-walk based model checking*, SEFM'09, IEEE, 2009.

[7]. C. Courcoubetis and M. Y. Vardi and P. Wolper and M. Yannakakis, *Memory-Efficient Algorithms for the Verification of Temporal Properties*, Formal Methods in System Design 92, vol. 1(2/3), pp. 275-288, 1992.

[8]. M.B. Dwyer, S. Elbaum, S. Person, and R. Purandare, *Parallel randomized state-space search*, ICSE '07, pp. 3-12, IEEE CS, 2007.

[9]. H. Garavel, R. Mateescu, and I. Smarandache, *Parallel state space construction for model-checking*, SPIN '01, Springer-Verlag New York, Inc., USA, pp. 217-234, 2001.

[10]. R. Grosu and S.A. Smolka, *Monte Carlo model checking*, TACAS'05, vol. 3440 of LNCS, pp. 271-286, Springer, 2005.

[11]. R. Grosu, X. Huang, S.A. Smolka, W. Tan, and S. Tripakis, *Deep random search for efficient model checking of timed automata*, vol. 4888 of LNCS, pp. 111-124, Springer, 2006.

[12]. G.J. Holzmann, D. Bosnacki, *Multi-Core Model Checking with SPIN*, Parallel and Distributed Processing Symposium 2007 - IPDPS 2007. IEEE International, pp.1-8, March 2007.

[13]. A. Laarman, R. Langerak, J.V.D. Pol, M. Weber, and A. Wijs, *Multi-core nested depth-first search*, ATVA'11, Springer-Verlag, Berlin, Heidelberg, pp. 321-335, 2011.

[14]. M. Mihail and C.H. Papadimitriou, *On the random walk method for protocol testing*, CAV'94, vol. 818 of LNCS, pp. 132-141, Springer-Verlag, 1994.

[15]. D. Owen and T. Menzies, *Lurch: a lightweight alternative to model checking*, SEKE'03, pp. 158-165, 2003.

[16]. N. Rungta and E. G. Mercer, *Generating Counter-Examples Through Randomized Guided Search*, SPIN'07, vol. 4595 of LNCS, pp. 39-57, Springer-Verlag, 2007.

[17]. H. Sivaraj and G. Gopalakrishnan, *Random walk based heuristic algorithms for distributed memory model checking*, PDMC'03, vol. 89(1) of ENTCS, pp. 51-67, Elsevier, 2003.