A Harmony Search Based Algorithm for Detecting Distributed Predicates

Eslam Al Maghayreh
Computer Science Department
Faculty of Information Technology and Computer Science
Yarmouk University, Irbid 21163, Jordan

***Abstract*— Detection of distributed predicates (also referred to as runtime verification) can be used to verify that a particular run of a given distributed program satisfies certain properties (represented as predicates). Consequently, distributed predicates detection techniques can be used to effectively improve the dependability of a given distributed application. Due to concurrency, the detection of distributed predicates can incur significant overhead. Most of the effective techniques developed to solve this problem work efficiently for certain classes of predicates, like conjunctive predicates. In this paper, we have presented a technique based on harmony search to efficiently detect the satisfaction of a predicate under the possibly modality. We have implemented the proposed technique and we have conducted several experiments to demonstrate its effectiveness.**

Keywords- Distributed Systems; Detection of Distributed Predicates; Runtime Verification; Harmony Search; Testing; Debugging.

I. INTRODUCTION

The design and construction of dependable distributed applications is not an easy task. Several techniques have been used in the literature to improve the dependability of distributed applications. Detection of distributed predicates (runtime verification) is one of the techniques that have attracted a great deal of attention in this regard [1], [2], [3], [4], [5], [6].

Runtime verification techniques can be used to verify whether a given run of a distributed application satisfies certain properties or not. Figure 1 depicts the runtime verification environment [7].

Runtime verification can be used to verify a particular implementation rather than verifying a model of the application as is done in model checking. Moreover, in runtime verification, the properties to be verified can be formally specified, i.e. using temporal logic. Consequently, runtime verification is considered more powerful than traditional testing in this regard [1], [7].

In addition to runtime verification, detecting distributed predicates has several applications. The followings are some of these applications:

a) Detection of when a distributed computation enters a stable state. This involves the detection of a condition which once becomes true, remains true indefinitely. For example, termination detection and deadlock detection.

b) Testing and debugging of distributed programs. Any condition that must be true in a correct run of a distributed application can be specified and then its occurrence can be verified. For example, when debugging a distributed mutual exclusion algorithm, it is useful to monitor the system to detect concurrent accesses to the shared resources. Another example is detecting the presence of multiple leaders in distributed leader election.

c) Identifying bottlenecks. For example, detecting posi- tions during a run of a distributed application where more than n processes from some set are simultane- ously blocked.

Concurrency in distributed applications makes the detection of distributed predicates a very hard and expensive task. Consequently, several techniques have been introduced in the literature to reduce the cost of detecting distributed predicates [7], [1], [8]. A brief review of these techniques will be presented in Section III. However, most of these techniques work well for only certain classes of predicates. It has been proved that the problem of verifying whether a run of a distributed program satisfies certain predicate or not is, in general, an NP-complete problem [1].

In this paper, we exploit harmony search in developing a more powerful technique to detect distributed predicates. This technique can work efficiently for predicates under the possibly modality (a predicate under the possibly modality is evaluated to true if it is true in at least one global state [1]). Harmony search is a popular evolutionary algorithm that can be used effectively to solve problems with exponential size search space (as it is the case in distributed predicates' detection).

The remainder of this paper is organized as follows. In section II, we present a formal model of a run of a distributed program. We discuss other related works in section III. A brief introduction to harmony search is provided in section IV. We present the proposed algorithm in section V. Section VI presents the experimental results. Finally, we conclude our work in section VII.

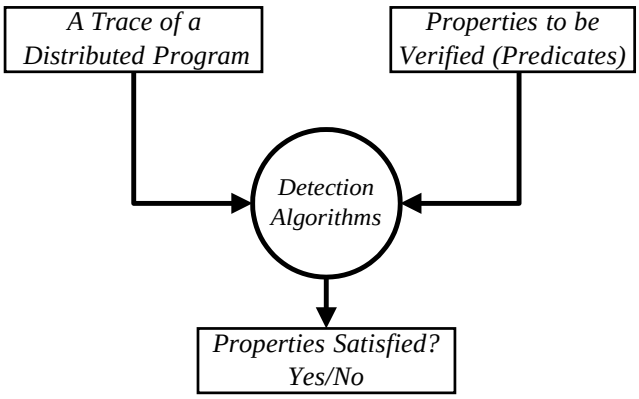


Figure1. Runtime verification environment.

II. Model And Problem Definition

A distributed program consists of n processes denoted by P_0 , P_1 , ..., P_{n-1} and a set of unidirectional channels. An event is the result of executing a statement in a distributed program. An event can be computational event or message event (send/receive). Events are related by their execution order in a process and/or message send/receive relations across processes. The happened-before relation (→) defined by Lamport in [9] applies to all events executed.

***Definition* 1**: A **run** of a distributed program is an event structure <E, →>, where E is the set of events executed, and (→) is the happened-before relation among the events in E.

The space-time diagram shown in Figure 2 depicts a run of a distributed program involving three processes. Time is represented in the horizontal direction and space in the vertical direction. Events are shown as circles in the space-time diagram. Event e_{ij} is the j^{th} event of process Pi. A directed edge is used to link every send event with the corresponding receive event.

The happened-before relation → is a partial order relation on the set of events of any run of a distributed application. $e_{ij} \rightarrow e_{kl}$ if and only if there is a directed path in the corresponding space-time diagram from event e_{ij} to event e_{kl}. If two events e_{ij} and e_{kl} are not related by the happened-before relation, we say that they are **concurrent events** (denoted by $e_{ij} \parallel e_{kl}$). For example, in Figure 2, $e_{01} \parallel e_{21}$ because $\neg(e_{01} \rightarrow e_{21})$ and $\neg(e_{21} \rightarrow e_{01})$.

A consistent cut C of a run $\langle E, \rightarrow \rangle$ is a finite subset of E ($C \subseteq E$) such that if $e_{ij} \in C$ and $e_{kl} \rightarrow e_{ij}$ then $e_{kl} \in C$. The dotted line shown in Figure 2 represents a consistent cut $C1 = \{ e_{01} , e_{11}\}$. Every consistent cut corresponds to a global state of the distributed application represented by the values of the program variables and channels states attained upon the completion of the execution of the events in that consistent cut. The set of global states of a given run endowed with set union and set intersection operations forms a distributive lattice, referred to as the state lattice [10].

Figure 3 depicts the state lattice associated with the run depicted in Figure 2. Each global state can be labeled by the most recent event executed in each process upon reaching it. For example, $(e_{01} , e_{13} , e_{22})$ is the state reached after executing event e_{01} in P_0 , event e_{13} in P_1 and event e_{22} in P_2 . Using the state lattice, we can verify whether a run of a given distributed program satisfies the necessary properties or not (Distributed Predicates Detection).

A predicate is called distributed predicate if the variables involved in expressing the predicate belongs to more than one process. Consequently, the evaluation of a distributed predicate requires the collection of information from several processes. A predicate that involves variables of a single process is called local predicate.

possibly: φ is true if the predicate φ is evaluate to true in at least one global state in the state lattice. definitely: φ is true if, for all paths from the initial global state to the final global state, φ is true in at least one global state along that path [11], [12]. In this paper, we will consider predicates under the possibly modality.

The difficulty of the detection of a distributed predicate is due to the following characteristics of a distributed application [1]:

1) There is no global clock in a distributed application. Consequently, the events of a given run of a distributed application can only be partially ordered.

2) The processes of a distributed application do not have shared memory. As a result, collecting the information necessary to detect a predicate will incur significant message overhead.

3) In any distributed application, there will be several processes running concurrently. As a result, the number of global tates that must be considered to detect a predicate will be exponential in number of processes.

In fact it has been proved that it is, in general, NP-complete to detect a distributed predicate in a run of a distributed application [1]. The next section is dedicated to explore the techniques presented in the literature to detect a predicate in a run of a given distributed application.

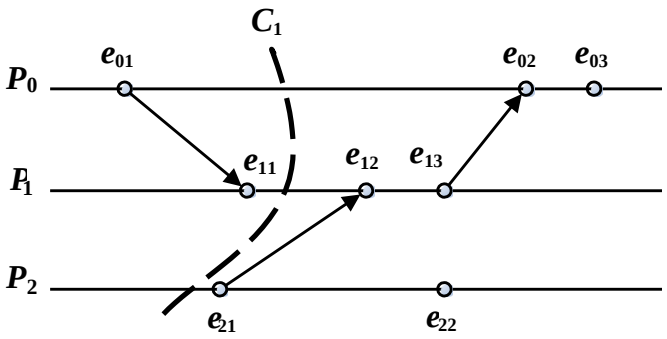


Figure2. A space-time diagram corresponding to a run of a distributed program.

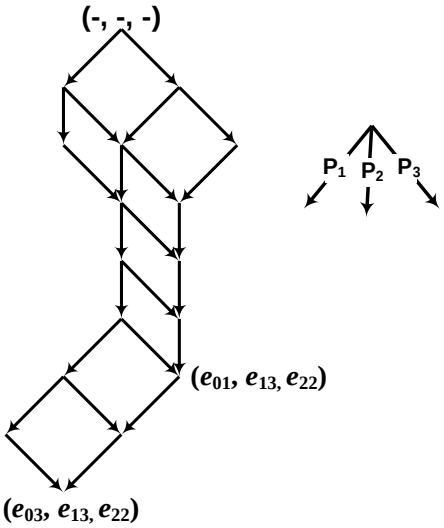


Figure 3. The state lattice corresponding to the run shown in Figure 2.

III. Related Works

Three main approaches have been presented in the literature to detect a predicate in a run of a distributed application. The first approach exploits the global snapshot algorithm proposed by Chandy and Lamport [13], [14], [15]. In this approach, a global state of a run is captured and the predicate to be detected is evaluated on it, if the predicate is evaluated to false, another global state will be captured. This process will continue until we found a global state satisfying the desired predicate. This approach is applicable for stable predicates (once they become true they will not turn false).

The second distributed predicates detection approach was proposed by Cooper and Marzullo [11]. This approach can be used to detect both stable and unstable predicates. It can be used to detect possibly: φ and definitely: φ. However, the detection is very expensive because it requires the construction of the entire state lattice and exploring m^n global states in the worst case, where n is the number of processes and m is the number of local states in each process.

To avoid the construction of the state lattice, the third detection approach exploits the structure of the predicate to identify a subset of the global states, such that if the predicate is true, it must be true in one of the states in this subset. This approach is not as general as the second approach, but it can be used to develop more efficient algorithms for certain classes of predicates. Garg and Waldecker [5], [6] have proposed algorithms of complexity $O(n^2 m)$ to detect possibly : φ and definitely : φ when φ is a conjunction of local predicates.

Several techniques have been introduced in the literature to reduce the cost of detecting a predicate. Computation slicing is one of these techniques. It was introduced in [16], [17], [18], [19] as an abstraction technique for analyzing distributed computations (finite execution traces). A computation slice, defined with respect to a distributed predicate, is the computation with the minimum number of global states that contains all global states satisfying the predicate.

Computation slicing can be used to eliminate the irrelevant global states of the original computation, and keep only the states that are relevant for our purpose. In [19], Mittal and Garg proved that a slice exists for all global predicates. However, it is, in general, NP-complete to compute the slice. They developed efficient algorithms to compute slices for special classes of predicates.

Lamport has presented a theorem on atomicity to simplify verification of distributed and parallel systems [20]. According to this theorem, a sequence of statements in a distributed program can be grouped together to form an atomic action under some stated conditions. An atomic action can receive information from other processes, followed by at most

one externally visible event (for example, changing the value of a variable involved in one of the properties to be verified) before sending information to other processes. Based on this theorem, a distributed program can be abstracted and hence the cost of verifying it can also be reduced.

In [21], [22], [23], the authors have formally defined the notion of atomic actions in message-passing distributed programs. They have exploited the atomicity concept in reducing the state space to be considered in runtime ver- ification of message-passing distributed programs.

In this paper, we exploit harmony search in developing a general and at the same time efficient detection algorithm of distributed predicates under the possibly modality. In the following two sections we will present a brief introduction to harmony search and we will present the details of our proposed detection algorithm.

IV. Introduction To Harmony Search

Harmony Search (HS) is metaheuristic algorithm (also known as an evolutionary algorithm) that emulates the improvisation behavior of musicians [24]. In the music improvisation process, each musician (decision variable) plays (generates) a note (value) for finding a best harmony (global optimum). HS can handle discrete variables [25] as well as contin- uous variables [26], [27]. It has been successfully applied to a wide variety of both discrete and continuous practical optimization problems such as game scheduling problems [28], [29], clustering problems [30], timetabling problems [31], and structural design [32].

The HS algorithm consists of the following main steps [24]:

Step 1. Initialization of the optimization problem and algo- rithm parameters:

In this step, the optimization problem is characterized as a function $\mathbf{f}$ to be optimized (minimize or maximize). Additionally, the control parameters of the Harmony Search are specified in this step including: the Harmony Memory Size (HMS); Harmony Memory Consideration Rate (HMCR);

Pitch Adjusting Rate (PAR); and the stop criterion (i.e. Number of Improvisations (Iterations)).

Step 2. Harmony Memory (HM) initialization:

As shown in (1), the HM composed of HMS (Harmony Memory Size) candidate solutions with N decision variables

$\mathbf{x}_i = [x_i^1, \ldots, x_i^N], \; i \in \{1, \ldots, \mathrm{HMS}\}$. The objective function $\boldsymbol{f}$ in (1) measures the solution quality.

$$HM = \begin{pmatrix} x_1^1 & x_1^2 & \ldots & x_1^N & | & f(\mathbf{x}_1) \\ x_2^1 & x_2^2 & \ldots & x_2^N & | & f(\mathbf{x}_2) \\ \vdots & \vdots & \ldots & \vdots & & \vdots \\ x_{HMS}^1 & x_{HMS}^2 & \ldots & x_{HMS}^N & & |f(\mathbf{x}_{HMS}) \end{pmatrix} \quad (1)$$

In this step, the HM is randomly initialized within the solution space.

Step 3. New Harmony improvisation:

In this step, a new harmony vector $\mathbf{x}_{new} = (x^1_{new}, x^2_{new}, \ldots, x^N_{new})$ is generated based on three operators: (1) memory consideration, (2) pitch adjustment, and (3) random selection.

Step 4. Harmony memory update:

In this step, the objective function value is evaluated for the vector x_{new} to determine if the new harmony should be included in the harmony memory. If the new harmony vector is better than the worst harmony in the HM, then the worst harmony is replaced with the new harmony.

Step 5. Repetition of Steps 3 and 4 until the termination criterion is satisfied:

Steps 3 and 4 will be repeated until the stop criterion is satisfied (i.e. maximum number of improvisations).

V. The Use Of Hs For Detecting Distributed Predicates

In this section, we will present an approach based on HS that can be used to detect Possibly: P for any predicate P As we have shown earlier, there are some approaches that can efficiently detect Possibly: P for certain classes of predicates P , like conjunctive predicates. However, there is no efficient approach that can be used to detect Possibly: P for any predicate P. This is due to the fact that the number of global states to be considered in detection is exponential (m^n global states in the worst case, where n is the number of processes and m is the number of local states in each process). Harmony search can be used in this case to provide a powerful general solution in such a case where the search space size to be considered is exponential.

The algorithms developed to detect distributed predicates can be online or offline. Online detection works during the execution of the application, and hence it may change the behavior of the application in unexpected manner. However, it has the advantage of avoiding the need to keep very large trace files as it is the case in offline detection. Offline detection collects the necessary information at runtime and later analyzes it to decide whether a given predicate has been satisfied during the execution or not. Offline detection does not have a strong impact on the behavior of the application under consideration. However, it requires the collection of very large trace files.

In this paper we are adopting the offline approach. In fact the use of harmony search fits more appropriately with the offline approach due to the fact that harmony search operators investigate global states in the whole search space randomly and does not investigate global states in the order in which they may appear during the execution of a given distributed application.

Now we will describe the details of the harmony search algorithm used in our solution. We will start with the representation of a run of a distributed program that the harmony search algorithm can manipulate. We assume that the algorithm to be developed wants to detect the predicate Possibly: P where P is the predicate $x_0 + x_1 + \cdots + x_{n-1} = c$ where x_i is a variable of process P_i and c is a constant.

There is no algorithm presented in the literature to efficiently detect this predicate [1].

We will assume that each process is instrumented to collect at runtime the local states that may affect the desired predicate along with their vector clock timestamp (The vector clock is a very well known technique to assign timestamps to the events of a run of a given distributed program [10], [33]). Consequently, each process P_i will have a trace file of the form

Val_1 , $timestamp_1$

Val_2 , $timestamp_2$

.

.

.

Val_m, $Timestamp_m$

Where Val_j is the value of variable x_i of process P_i in the local state number j. $Timestamp_j$ is the vector clock time stamp of local state j. For example, if we have the run shown in Figure 4 (a) and we want to detect the predicate $x + y + z = 2$, then the trace files of these processes will be as shown in Figure 4 (b). Each trace file contains a list of all local states that may be part of a global state that satisfies the desired predicate in the given run. The local state of a process P_i involves the value of the variables involved in the predicate of interest and the vector clock time stamp of the local state. For example, the first local state of process P_0 is 0, (1, 0, 0). This means that the value of variable x (which is one of the variables involved in the predicate of interest) at this local state is zero and the vector clock time stamp of this local state is (1, 0, 0).

Assuming that we have n processes we will have n trace files. The size of each of them is linear with respect to the number of events executed by each corresponding process. These files will represent the input to the HS algorithm that will be used to detect the above mentioned predicate. The contents of these files can be changed to include additional information if we want to detect other types of predicates. For example, if the predicate of interest involves two or more variables of process P_i , then the values of these variables has to be added to each local state in the trace file of P_i.

Now we will move to the details of the harmony search algorithm itself starting with the representation of the harmony memory (HM) to be used. The HM is a two dimensional array where each row represents a solution of the problem under consideration along with its fitness (quality). In our problem (Distributed predicates detection), each row in the HM represents a global state of the distributed application under consideration. Consequently, each row will have n local states (one from each process) such that all of the local states form a global state of the application. Each global state in the HM is a candidate solution for our problem which is mainly finding a global state that satisfies the predicate to be detected. Moreover, the last element of each row contains the fitness of the solution encoded in that row.

In our example, where we have the run shown in Figure4 (a) and we want to detect the predicate $x + y + z = 2$, the HM can have the form shown in Figure 4 (c) where the size of it is HMS and each row represents a global state along with its fitness. Each element in column i is a local state of process P_i taken from its trace file. For example, the element HM[0][0] shown in Figure 4 (c) is one of the local states of process P_0 taken from its trace file shown in Figure 4 (b).

HS operators may result in solutions that do not represent global states due to the fact that the cuts represented by the resulting chromosomes are not consistent. This problem can be solved by increasing the fitness of such solutions (solutions with small fitness value are better than others with larger fitness value). In our algorithm we will assign the value of 9999999 as the fitness value for any chromosome that does not represent a global state.

Now we will move to the most important step in our algorithm, namely, the design of the fitness function. In fact all of the above steps in the algorithm will be identical for any predicate under consideration. The only difference between the algorithms to detect two different predicates is in the fitness function. This is considered as a strong side in HS. More precisely, HS algorithms are more general than other algorithms in the sense that they can be used to detect any predicate with small modifications on the fitness function. There is no need to develop a tailored algorithm to efficiently detect each type of predicates.

The fitness function has to evaluate each possible solution in the HM and assign to it a fitness value indicating whether the solution is close to the optimal solution or not. In our example, the value assigned by the fitness function to each solution will indicate whether the global state represented by the solution satisfies the predicate of interest or not, and if it does not satisfy the predicate, how close it is to the values that can satisfy the predicate.

The fitness function will first check whether the solution represents a global state or not (consistent cut), if the chromosome does not represent a global state then the fitness value assigned to it will be 9999999 indicating that this solution cannot represent a global state that satisfies the predicate. Otherwise the fitness function will assign a fitness value to the solution according to the following formula:

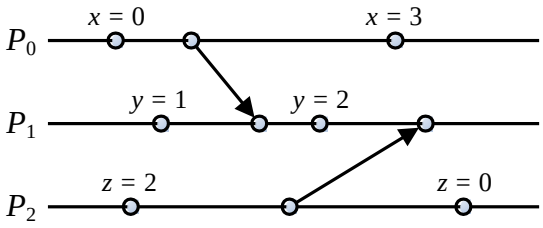
$$Fitness = x_0 + \cdots + x_{n-1} - c$$

For example the fitness of the solution presented in HM[0]

in Figure 4 (c) is $0 + 2 + 2 - 2 = 2$

Consequently, when the solution satisfies the predicate its fitness value will be 0. When the algorithm finds a global state that satisfies the predicate, it will terminate directly since we are looking for the predicates under the possibly modality.

To avoid keep running the algorithm forever in cases there is no global state satisfying the predicate in the run under consideration, we can also fix the maximum number of iterations the HS algorithm can go through. However, it is better to choose a large enough bound on the number of iterations depending on the complexity of the predicate, the number of processes, and the length of the trace files. The larger the number of processes and trace files the larger the number of iterations that need to be considered.



(a)

Trace file of P_0	*Trace file of P_1*	*Trace file of P_2*
0, (1, 0, 0) 3, (3, 0, 0)	1, (0, 1, 0) 2, (2, 3, 0)	2, (0, 0, 1) 0, (0, 0, 3)

(b)

	Harmony Memeory (HM)			*Fitness*
HM[0]	**0, (1, 0, 0)**	**2, (2, 3, 0)**	**2, (0, 0, 1)**	**2**
	∘ ∘ ∘	∘ ∘ ∘	∘ ∘	∘ ∘ ∘
HM[HMS-1]	**3, (3, 0, 0)**	**2, (2, 3, 0)**	**0, (0, 0, 3)**	**3**

(c)

Figure 4. An example to demonstrate the use of HS in detecting distributed predicates.

VI. Implementation And Experimental Results

In this section we will give more details about the implementation of the proposed HS-based distributed predicates detection algorithm. Moreover, we will present some experimental results. We have implemented our algorithm using Java pro- gramming language. We assume that the algorithm to be developed wants to detect the predicate Possibly : P where P is the predicate ($x_0 + \cdots + x_{n-1} = c$) where x_i is a variable of process P_i and c is a constant. We assume that we have n processes. In the previous section we have described the fitness function used in our example. Other parts of the HS algorithm can be implemented in a general manner and can be used in detecting any other predicate. The only thing that has to be changed if we want to detect other predicates is the fitness function.

We have executed our algorithm on a computer with Intel Core 2 Duo CPU, 2.4GHz with 2GB of RAM. We have executed the algorithm on a trace of a distributed application that involves 25, 50, and 75, processes where each process has executed 1000, 2000, 3000 events in each run. We have set the parameters of the HS as follows (HMS = 10, HMCR = 0.9 and PAR = 0.4). The results are summarized in Table 1. For example, given a run that involves 75 processes where each process has executed 3000 events, the algorithm was able to detect the predicate $x_0 + x_1 + \cdots + x_{74} = 67$ after 88457221 iterations and the total time required to detect the predicate was 1687.125 seconds.

The other general approach that can be used to detect any predicate under the possibly modality is to construct the entire state lattice and to test the global states in it one by one until we reach a global state where the predicate of interest is satisfied. This approach requires exploring m^n global states in the worst case where n is the number of processes and m in the number of local states in each process. Obviously, exploring such a large number of states will require much more time than the time required by the HS-based algorithm. For example, a run that involves 20 processes where each process have executed 5 events will have (5^{20}) global states. If we want to examine all the global states in this small run, and assuming that we can examine 10^9 global states per second, then we need 26.49 hours to finish. Consequently, it is clear (See Table 1) that the detection algorithm based on HS is much more powerful.

VII. Conclusions And Future Work

In this paper, we have developed an efficient distributed predicates detection algorithm based on harmony search. The proposed algorithm can be used to detect distributed predicates under the possibly modality. Distributed predicates detection (also referred to as runtime verification) is an effective technique to reason about a particular implementation of a given distributed application. Consequently, the results presented in this paper can be exploited in developing more dependable distributed applications.

Genetic algorithms (GAs) can be used effectively to solve problems with exponential size search space as it is the case in distributed predicates' detection. In one of our current research efforts we are trying to investigate the effects of using the mutation operator of GAs in HS to solve certain optimization problems.

Table I
THE RESULTS OF SEVERAL EXPERIMENTS.

Number of Processes	Number of events executed by each process	Time spent to detect the predicate/ seconds	Number of Iterations
25	1000	0.031	72
	2000	0.047	282
	3000	0.062	587
50	1000	1.234	92580
	2000	1.5	117323
	3000	2.032	159467
75	1000	830.454	44732576
	2000	1504.641	80262355
	3000	1687.125	88457221

One possible avenue for the continuation of the work presented in this paper is to consider the effects of using the mutation operators of GAs on the performance of the HS algorithm developed to detect distributed predicates. Moreover, we can develop a detecting approached based completely on GAs and compare it with the approach developed based on HS. Another important avenue for future work is improving the fitness function. In fact, if we can find a more powerful fitness function, then the performance of the proposed algorithm will certainly be improved.

REFERENCES

[1] Vijay K. Garg, Elements of distributed computing, John Wiley & Sons, Inc., New York, NY, USA, 2002.

[2] G. Dumais and H.F. Li, "Distributed predicate detection in series-parallel systems," IEEE Transactions on Parallel and Distributed Systems, vol. 13, no. 4, pp. 373 –387, apr 2002.

[3] Felix C. Freiling and Arshad Jhumka, "Global predicate detection in distributed systems with small faults," in Pro- ceedings of the 9th international conference on Stabilization, safety, and security of distributed systems, Berlin, Heidelberg,2007, SSS'07, pp. 296–310, Springer-Verlag.

[4] Chunbo Chu and M. Brockmeyer, "Predicate detection modality and semantics in three partially synchronous mod- els," in the Seventh IEEE/ACIS International Conference on Computer and Information Science, may 2008, pp.444–450.

[5] Vijay K. Garg and Brian Waldecker, "Detection of weak unstable predicates in distributed programs," IEEE Trans. Parallel Distrib. Syst., vol. 5, no. 3, pp. 299–307, 1994.

[6] Vijay Garg and Brian Waldecker, "Detection of strong unstable predicates in distributed programs," IEEE Trans. Parallel Distrib. Syst., vol. 7, no. 12, pp. 1323–1333, 1996.

[7] Eslam Al Maghayreh, Simplifying Runtime Verification of Distributed Programs: Ameliorating the State Space Explosion Problem, VDM Verlag, 2010.

[8] Craig M. Chase and Vijay K. Garg, "Detection of global predicates: Techniques and their limitations," Distributed Computing, vol. 11, no. 4, pp. 191–201, 1998.

[9] Leslie Lamport, "Time, clocks, and the ordering of events in a distributed system," Commun. ACM, vol. 21, no. 7, pp. 558–565, 1978.

[10] Friedemann Mattern, "Virtual Time and Global States of Distributed Systems," in Proceedings of the International Workshop on Parallel and Distributed Algorithms, Château de Bonas, France, October 1989, pp. 215–226.

[11] Robert Cooper and Keith Marzullo, "Consistent detection of global predicates," SIGPLAN Not., vol. 26, no. 12, pp. 167– 174, 1991.

[12] Roland Jegou, Raoul Medina, and Lhouari Nourine, "Linear space algorithm for on-line detection of global predicates.," in Proceedings of the International Workshop on Structures in Concurrency Theory (STRICT), Berlin, 1995, pp. 175–189.

[13] K. Mani Chandy and Leslie Lamport, "Distributed snapshots: Determining global states of distributed systems,"ACM Trans. Comput. Syst., vol. 3, no. 1, pp. 63–75, 1985.

[14] Luc Bougé, "Repeated snapshots in distributed systems with synchronous communications and their implementation in CSP," Theor. Comput. Sci., vol. 49, pp. 145–169, 1987.

[15] Madalene Spezialetti and Phil Kearns, "Efficient distributed snapshots," in ICDCS, 1986, pp. 382–388.

[16] H. F. Li, Juergen Rilling, and Dhrubajyoti Goswami, "Granularity-driven dynamic predicate slicing algorithms for message passing systems," Automated Software Engg., vol. 11, no. 1, pp. 63–89, 2004.

[17] Alper Sen and Vijay K. Garg, "Detecting temporal logic pred- icates in distributed programs using computation slicing." in OPODIS, 2003, pp. 171–183.

[18] Vijay K. Garg and Neeraj Mittal, "On slicing a distributed computation," in ICDCS '01: Proceedings of the The 21st nternational Conference on Distributed Computing Systems, 2001, p. 322.

[19] Neeraj Mittal and Vijay K. Garg, "Computation slicing: Techniques and theory," in DISC '01: Proceedings of the 15th International Conference on Distributed Computing, London, UK, 2001, pp. 78–92, Springer-Verlag.

[20] L. Lamport, "A theorem on atomicity in distributed algo- rithms," Distributed Computing, vol. 4, no. 2, pp. 59–68, 1990.

[21] Eslam Al Maghayreh, "Block-based atomicity to simplify the verification of distributed applications," in 24th Cana- dian Conference on Electrical and Computer Engineering (CCECE), may 2011, pp. 887–891.

[22] H. F. Li, Eslam Al Maghayreh, and D. Goswami, "Detecting atomicity errors in message passing programs," in PDCAT'07: Proceedings of the Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies. 2007, pp. 193–200, IEEE Computer Society.

[23] H. F. Li, Eslam Al Maghayreh, and D. Goswami, "Using atoms to simplify distributed programs checking," in DASC'07: Proceedings of the Third IEEE International Symposium on Dependable, Autonomic and Secure Computing, 2007, pp.75–83.

[24] Zong Woo Geem, Joong-Hoon Kim, and G. V. Loganathan, "A new heuristic optimization algorithm: Harmony search," Simulation, vol. 76, no. 2, pp. 60–68, 2001.

[25] Zong Geem, "Novel derivative of harmony search algorithm for discrete design variables," Applied Mathematics and Computation, vol. 199, no. 1, pp. 223–230, 2008.

[26] Kang S. Lee and Zong W. Geem, "A new meta-heuristic algorithm for continuous engineering optimization: harmony search theory and practice," Computer Methods in Applied Mechanics and Engineering, vol. 194, no. 36-38, pp. 3902–3933, Sept. 2005.

[27] Zong Woo Geem, "Global optimization using harmony search: Theoretical foundations and applications," in Foun- dations of Computational Intelligence (3), pp. 57–73. 2009.

[28] Zong Geem, "Harmony search algorithm for solving sudoku," in Knowledge-Based Intelligent Information and Engineering Systems (KES), 2007, pp. 371–378.

[29] Zong Geem, "Harmony search for multiple dam scheduling," in Encyclopedia of Artificial Intelligence, pp. 803–807. 2009.

[30] Osama Moh'd Alia, Mohammed Azmi Al-Betar, Mandava Rajeswari, and Ahamad Tajudin Khader, "Data clustering using harmony search

algorithm," in Proceedings of Second International Conference Swarm, Evolutionary, and Memetic Computing (SEMCCO 2), 2011, pp. 79–88.

[31] Mohammed Azmi Al-Betar and Ahamad Tajudin Khader, "A harmony search algorithm for university course timetabling," Annals OR, vol. 194, no. 1, pp. 3–31, 2012.

[32] Zong Woo Geem, Kang Seok Lee, and Chung-Li Tseng, "Harmony search for structural design," in Proceedings of Genetic and Evolutionary Computation Conference (GECCO 2005), 2005, pp. 651–652.

[33] Colin Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering," in Proceedings of the 11th Australian Computer Science Conference, 1988, pp. 56–66.