Runtime Verification of k-Mutual Exclusion for SoCs

Selma Ikiz
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712, USA
Email: ikiz@ece.utexas.edu

Alper Sen
Design Technology Organization,
Freescale Semiconductor Inc.
7700 W. Parmer Lane, Austin, TX 78729, USA
Email: alper.sen@freescale.com

***Abstract*— We present an efficient runtime verification environment for detecting mutual exclusion predicates. Such predicates are important for keeping the safe operation of concurrent systems. Our environment models execution traces as partial order traces to increase scalability in runtime verification. We compare two techniques implemented in POTA tool, namely k-exclusion and computation slicing. The k-exclusion problem is a generalization of the mutual exclusion problem in which up to k processes may be in their critical sections at the same time. Our k-exclusion algorithm exploits the fact that if there is a k-exclusion violation then it is impossible to partition events from critical sections into k queues. We earlier presented efficient computation slicing algorithms to detect predicates from a subset of temporal logic CTL. We performed experiments using POTA tool on scalable protocols. Our comparison shows that k-exclusion is substantially better than slicing both in terms of time and space. In all fairness, slicing handles general class of predicates from temporal logic CTL, whereas k-exclusion algorithm handles only a very specific, nonetheless useful, class of mutual exclusion predicates.**

I. Introduction

Modern day hardware systems often consist of individual IP blocks connected in meaningful arrangements carrying out independent as well as cooperative objectives. Such Systems-on-Chips (SoCs) are gradually becoming more and more diverse and complicated, thereby gaining popularity. Their enormous potential for re-using IP makes them economically viable solutions to many products out in the market today. The design of a SoC system can be very large and complex. Because SoC's have a large number of components, in general, it is not feasible to give a formal proof of correctness (manual or automatic). Therefore, extensive simulation is still the most common technique used in verifying industrial systems. We advocate the use of *Predicate Detection* (also called Runtime Verification, Assertion Based Verification) that combines formal methods and simulation. Specifically, predicate detection enables *efficient* verification of predicates (or properties) from a formal specification language on execution traces of actual scalable systems.

We also advocate the use of a predicate detection technique that works on *Partial Order Simulation Traces* instead of the traditional total order simulation traces. We use the partial order approach because it has several advantages over the total order approach. Traditional simulation methodologies are woefully inadequate in presence of concurrency and subtle synchronization. The bug in the system may appear only when the delay in synchronization is different from the delay in the simulation trace. A partial order trace model is a more faithful representation of concurrency [10], that is, only the events that have a causal dependency are ordered, e.g. the send of a message and the receive of that message. A partial order encodes possibly exponential number of total orders, therefore we can analyze exponentially more number of traditional traces. Hence, although our predicate detection technique is based on simulation, it becomes scalable in terms of bug detection thanks to partial order traces.

Programming, testing and debugging of concurrent programs is substantially more difficult than that of sequential programs both in terms of correctness and efficiency. Today, almost any large software application is an entire concurrent system made of varying number of processes. These processes often share resources which must be effectively protected against concurrent access. This is usually achieved by executing these actions in critical sections that are protected by semaphores or locks. Hence, an important predicate detection problem is to detect is whether the access to the critical sections are mutually exclusive. This problem is the well known *mutual exclusion* property. To ascertain this safety property, it is vital to detect whether any two processes are in their critical sections simultaneously, i.e., whether there is a mutual exclusion violation in the computation. In this paper, our goal is to detect mutual exclusion predicates based on two algorithms namely k-exclusion and computation slicing. This is the first study that shows some experimental results for k-exclusion algorithm on simulation traces rather than random partial order sets. Moreover, we define bounded sum form (BSF) of predicates and show that BSF can be applied more efficiently than disjunctive normal form (DNF) for some predicates.

The *k-exclusion* problem was posed by Fischer, et al. [5] as a generalization of the mutual exclusion problem in which up to k processes may be in their critical sections at the same time. The k-exclusion property is shown to be a special class of bounded sum predicates [3] and solved by using a max-flow algorithm. Our k-exclusion algorithm exploits the fact that if there is a k-exclusion violation then it is impossible to

partition events from critical sections into k queues [4]. We present an efficient, centralized and incremental algorithm.

Computation slicing was introduced in [1], [6], [13] as an abstraction technique for analyzing traces of distributed programs. Intuitively, a *slice* of a trace with respect to a predicate p is a sub-trace that contains all the states of the trace that satisfy p. Note that the set of states that satisfy p may be large, so one could not simply enumerate all the states efficiently either in space or time. A slice contains all the states that satisfy p such that it is computed efficiently (without traversing the state space) and represented concisely (without explicit representation of individual states). Slicing algorithms were developed for temporal logic predicates in [1]. Slicing based predicate detection approach was earlier shown to bring exponential reduction both in terms of time and space for verification of several safety and liveness predicates of scalable protocols [1], [13].

We implemented our k-exclusion algorithm in Partial Order Trace Analyzer (POTA) tool [13], which already implemented slicing based algorithms. We perform experiments on scalable protocols such as Cache Coherence, Distributed Dining Philosophers and PCI based System-on-Chip (SoC). All of these protocols have to satisfy the mutual exclusion property. Our comparison shows that k-exclusion is substantially better than slicing both in terms of time and space. In all fairness, slicing handles general class of predicates from temporal logic CTL, whereas k-exclusion algorithm handles only a very specific, nonetheless useful, class of mutual exclusion predicates.

Related Work: Predicate detection in partial order traces is a hard problem. Detecting even a 2-CNF predicate under EF modality has been shown to be NP-complete, in general [6]. The idea of using temporal logic for analyzing simulation traces has been attracting a lot of attention recently. We first presented a temporal logic framework for partially ordered execution traces in [12] and POTA tool for runtime verification in [13], [14]. Some other examples of using temporal logic for checking execution traces are the MaC tool [9], the Verisim tool [2], and the JMPaX tool [15].

Our work can be applied just as well to concurrent and distributed programs and to behavioral hardware descriptions in Verilog or VHDL. Several EDA companies provide Assertion Based Verification support. However, all those tools use total order simulation trace model, hence the coverage is limited.

II. Model

We assume a system consisting of processes denoted by $P_1, \ldots, P_n$. Examples of processes are a thread in a program, node sitting on a PCI bus or a cache in a cache coherence protocol. Processes execute events. Events on the same process are totally ordered. However, events on different processes are only partially ordered. In this paper, we relax the partial order restriction on the set of events and use directed graphs to handle traces and slices with the same model.

We model a *trace* (or a *computation*) as a directed graph, denoted by $\langle E, \rightarrow \rangle$, with vertices as the set of events E and edges as $\rightarrow$. We use event and vertex interchangeably. To limit our attention to only those consistent global states that can actually occur during an execution, we assume that the paths in $\langle E, \rightarrow \rangle$ contains at least the partial order relation.

A partial order relation known as Lamport's happened-before relation [10] has been used for modeling simulation traces. We use a mechanism known as *vector clocks* to represent the partial order relation. A vector clock assigns timestamps to events such that the partial order relation between events can be determined by using the timestamps.

III. k-exclusion Algorithm

The k-exclusion problem was posed by Fischer, et al. [5] as a generalization of the mutual exclusion problem in which up to k processes may be in their critical sections at the same time. The k-exclusion property is shown to be a special case of bounded sum predicates [3] and solved by using a max-flow algorithm. Here we define bounded sum form (BSF) of predicates to compare them in disjunctive normal form (DNF). Some examples of predicates where BSF can be applied more efficiently than DNF:

1) Two process mutual exclusion:
   (DNF) $\bigvee_{i,j \in 0 \ldots (n-1)} (\mathsf{EF}(CS_i \wedge CS_j))$
   (BSF) $(\mathsf{EF}(CS_0 + CS_1 + \ldots CS_{n-1} > 1))$
2) At least one server is available:
   (DNF) $(\mathsf{EF}(\neg avail_1 \wedge \neg avail_2 \wedge \ldots \neg avail_n))$
   (BSF) $(\mathsf{EF}(\neg avail_1 + \neg avail_2 + \ldots \neg avail_n > n-1))$
3) K process mutual exclusion:
   (DNF) $\bigvee_{i_0, i_1, \ldots, i_K \in 0 \ldots (n-1)} (\mathsf{EF}(CS_{i_0} \wedge CS_{i_1} \wedge \ldots CS_{i_K}))$
   (BSF) $(\mathsf{EF}(CS_0 + CS_1 + \ldots CS_{n-1} > K))$

The k-exclusion violation can be detected by checking whether more than k processes have executed their critical section simultaneously (concurrently). However, a detection algorithm that uses DNF form requires $C(n, k+1)$ combination checks on computation trace for each disjunct. On the other hand using a max-flow algorithm has $O(E^2.log(E))$ complexity. Our algorithm described below has a better complexity.

Our k-exclusion algorithm exploits the fact that if there is a k-exclusion violation then it is impossible to partition critical section events into k queues (Dilworth's Chain Partition) [4]. The k-exclusion algorithm is centralized and incremental. We call the process *checker process* if it is executing the algorithm (central process). Each process, whenever changes its state, sends its local state along with its vector clock to the checker process. When a new event arrives, the checker process executes the algorithm if the new event is a critical section event, otherwise discards it. The k-exclusion algorithm rearranges the queues according to the new event and checks whether the number of queues in the resultant arrangement is more then k or not. The algorithm assumes that when a new event arrives, all events that happened-before it have already arrived and been processed. This is achieved by buffering the new event if it violates the assumption and processing it later. Whether to buffer an event can be determined efficiently by examining its vector clock.

Setup: The algorithm uses queues to store the events. Events are stored in an increasing order so that head is the smallest event and tail is the largest event in the queue. It keeps two type of queue compositions: work and history. We refer to the set of work and history queues as work and history space, respectively. W^j represents the work space when jth element arrives, while H^j represents the history space.

Execution: Whenever a new event arrives, k-exclusion tries to append it to one of the queue tails. However, this might not be always possible since new event may be concurrent to all the queue tails. In this case, it calls the $Merge$ function to merge s queues into $s-1$, if possible. The $Merge$ function takes s queues as input and returns three type of queue sets; input queues Q, output queues O, and history queues H'. K-exclusion updates its work space according to input queues of $Merge$, if there are s events that are mutually concurrent. Otherwise, it uses the output queues of $Merge$. The history space is updated by the history queues of $Merge$. Note that $s \leq k+1$ at all times.

Merge Function: The $Merge$ function begins with s queues called $input$ queues and $s-1$ empty queues called $output$ queues, and a spanning tree that is formed by the input and the output queues. The spanning tree has exactly k vertices and $k-1$ edges. An edge corresponds to an output queue and a vertex corresponds to an input queue. Therefore each edge has a label which identifies the output queue it corresponds with. No labels are duplicated in the tree, thus each output queue is represented once. Similarly, each input queue is represented exactly once. At each step, it removes some events from the input queues and append them to the output queues. This relocating procedure continues until $Merge$ finds s concurrent events or one of the input queues becomes empty. The spanning tree is used in the decision process of which output queue to append the events. The algorithm maintains the following invariant: If an edge between vertices q_i and q_j is labeled as o_k, then the heads of q_i and q_j are bigger than the tail of o_k. Intuitively,at a merge step, queue head a_i is selected for removal if it is less than one of the other input queue heads. $Merge$ updates its history and output queues if it witnesses s or $s-1$ concurrent events. It moves all the elements in output queues to history queues. Splitting the events into work and history queues reduces the number of comparisons for later steps. After each $Merge$ call, the heads of queues in the work space are mutually incomparable. Any event happened before them are placed into the history space. This splitting does not affect the future comparisons and we prove that it is sufficient to compare the new event to the events in work space [8].

Merge Example: An example for $Merge$ is given in Figure 1 to show how the state of the tree and of the queues are modified at each step. Figure 1a shows the initial setup. Q, O, and H' show the input, output and history queues, respectively, while dashed lines show the edges in the spanning tree. Remember that an edge labeled as an output queue between input queues states that the tail of the output queue is less than both of the input queue heads.

Step 1: [1,0,0] is less then [2,0,0],and [1,1,0] and [2,0,0] are concurrent. Hence, [1,0,0] is placed in history queues (see Figure 1b).

Step 2: [0,1,0] is less then [1,1,0],and [1,1,0] and [2,0,0] are concurrent. Hence, [0,1,0] is placed in history queues (see Figure 1c).

Step 3: [2,0,0] and [1,1,0] are both less then [2,2,0]. They are placed in output queues (see Figure 1d).

Step 4: [1,2,0] and [2,2,0] are both less then [2,3,0]. They are placed in output queues. Since there is an empty input queue, $Merge$ places the remaining events adhering to the spanning tree (see Figure 1d).

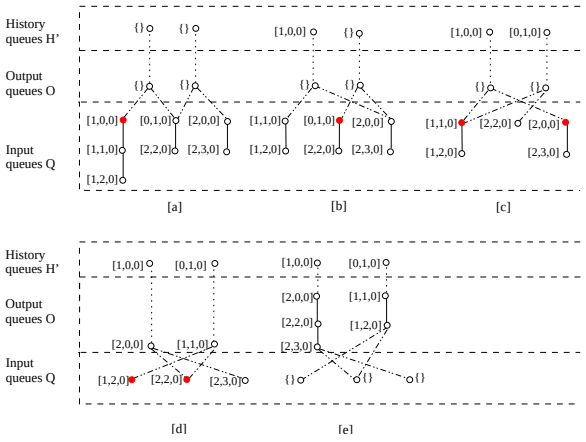

Fig. 1. An example for Merge

k-exclusion example: Consider the computation in Figure 2. Assume that the order of events arrived to checker process is x_2, x_1, x_3, x_4, x_5, x_6, x_7, x_8, x_9,x_{10}.We demonstrate two steps where splitting the queues happens since other steps are trivial.

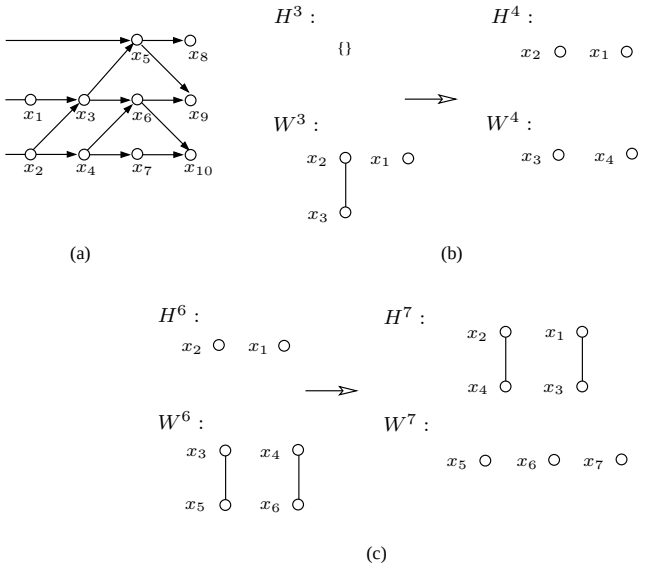

Fig. 2. (a) A computation (b) transition in steps 3 & 4 (c) transition in steps 6 & 7

1) $Merge$ is successful: At time 4, the checker has seen $X^3 = \{x_2, x_1, x_3\}$ and the new event is x_4. There are two queues in the work space and its content is as follows: q_1 contains x_2 and x_3, and q_2 contains x_1. There is no queue to append x_4 since both x_1 and x_3 are incomparable with x_4, hence k-exclusion creates a new queue q_3 and appends x_4 to q_3, and invokes $Merge$. Observe that s is three. $Merge$ places x_2 in o_1 and x_1 in o_2, and observes x_3 and x_4 as concurrent. Since $s-1$ is two, it appends the contents of output queues to history queues. It places x_3 in o_1, x_4 in o_2, and returns queues back, input queue being empty. Then, k-exclusion assigns the output queues as work queues, and appends the history queues of $Merge$ to its history space. Figure 2b shows the transition between time 3 and 4.

2) There are s mutually concurrent events: At time 6, the checker has seen $X^6 = \{x_2, x_1, x_3, x_4, x_5, x_6\}$ and the new event is x_7. There are two queues in the work space and their contents are as follows: q_1 contains x_3 and x_5, and q_2 contains x_4 and x_6. There is no queue to append x_7 since both x_5 and x_6 are incomparable with x_7, hence k-exclusion creates a new queue q_3 and appends x_7 to q_3, and makes a $Merge$ call. $Merge$ places x_3 in o_1 and x_4 in o_2, and computes x_5, x_6, and x_7 as concurrent. Since s is three and there are three mutually concurrent events, no reduction is possible. $Merge$ appends the contents of output queues to history queues, and returns back. Then, k-exclusion assigns the input queues as work queues, and appends the history queues of $Merge$ to its history space. Figure 2c shows the transition between time 6 and 7.

The worst case complexity of k-exclusion algorithm is $O(k|E|^2)$, where E is the set of critical section events, and k is the allowed number of concurrent events. When k is one, the complexity reduces to $O(|p|.|E|)$, where $|p|$ is the number of temporal operators in a bounded sum predicate p in BSF. The correctness discussion of the algorithm can be found in [8].

IV. Computation Slice

Informally speaking, a computation slice (or a slice) is a concise representation of all those global states of the computation that satisfy a global predicate (or simply predicate or property).

In the next two sections, we will illustrate the basic slicing algorithm on a simple example. A formal treatment of slicing algorithms can be found in [1].

A. Non-temporal Predicate Slicing

Figure 3c shows the slice of the computation in Figure 3a with respect to the non-temporal predicate $CS_1 \wedge CS_2$, which is a conjunction of two local predicates from the processes P_1 and P_2, respectively. We say that a predicate is *local* if it only depends on variables of a single process. In the figure, if a local predicate is true for an event of a process, we represent it using an empty circle, otherwise with a filled circle. Intuitively, to obtain the slice with respect to a local predicate, we add an edge from the successor of a filled circle back to it, thereby increasing the number of incoming

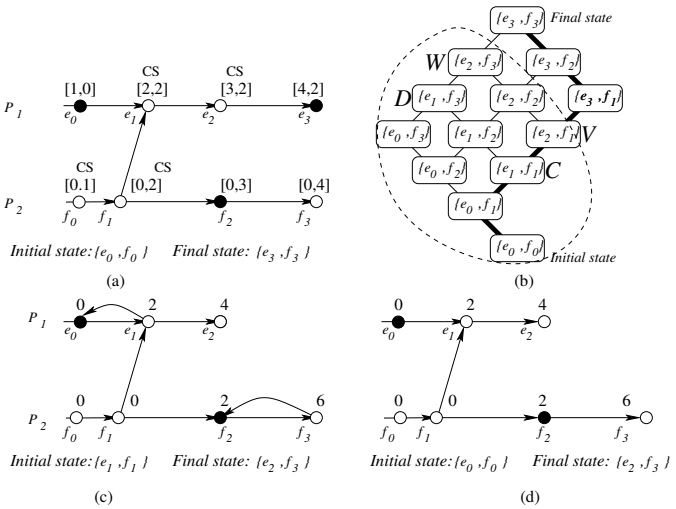

Fig. 3. (a) A computation (b) its set of all reachable global states (c) its slice with respect to $CS_1 \wedge CS_2$ (d) its slice with respect to $\mathsf{EF}(CS_1 \wedge CS_2)$

neighbors and disabling the local event that does not satisfy the predicate from being a consistent global state of the newly obtained trace. Now, we will further demonstrate why these additional edges give us the slice by showing that addition of edges results in removing consistent global states of the computation that do not satisfy the predicate. Observe that, the following global states, $\{e_0, f_0\}$,$\{e_0, f_1\}$, $\{e_0, f_2\}$, and $\{e_0, f_3\}$ of the computation in Figure 3c depicted in Figure 3b, do not satisfy the local predicate CS_1. This is because in all of these four states P_1 has executed only its initial event, where local predicate CS_1 is false. We then add the edge (e_1, e_0) in Figure 3c. From the definition of a consistent global state, any consistent global state of the slice, upon including e_0 must include e_1, which is an incoming neighbor. Since all of the global states $\{e_0, e_1\}$, $\{e_0, f_1\}$, $\{e_0, f_2\}$, $\{e_0, f_3\}$ include e_0, they must have included e_1 as well, which they do not. Hence, they cannot belong to the consistent global states of the slice. Also, note that if we added an edge from (e_2, e_0) instead of (e_1, e_0), we would have eliminated states such as $\{e_1, f_1\}$, which satisfy the predicate, from the slice. This is because any consistent global state upon including e_1 must include e_2, due to transitivity of the incoming neighbors. Similarly, we can reason about the additional edge (f_3, f_2) in the slice. As a result, from the thirteen consistent global states in Figure 3b, by adding two edges and not traversing the state space, this exercise eliminates nine – retaining states $\{e_1, f_1\}$, $\{e_1, f_3\}$, $\{e_2, f_1\}$, and $\{e_2, f_3\}$, which are the only states of the computation that satisfy $CS_1 \wedge CS_2$. □

It has been shown that the slice exists for all predicates, however it is, in general, intractable to compute the slice for an arbitrary predicate [6]. Our approach to computation slicing is based on exploiting the structure of the predicate itself.

B. Temporal Predicate Slicing

To illustrate predicate detection using computation slicing, consider the computation in Figure 3a again. Let $p = CS_1 \wedge CS_2$, and suppose we want to detect $\mathsf{EF}(p)$, that is, whether there exists a global state of the computation that satisfies p. Without computation slicing, we are forced to examine all global states of the computation, thirteen in total, to decide whether the computation satisfies the predicate. Alternatively, we can compute the slice of the computation with respect to the predicate $\mathsf{EF}(p)$ and use this slice for predicate detection. For this purpose, first, we compute the slice with respect to p as explained above. The slice is shown in Figure 3c. The slice contains only four states C, D, V and W (see Figure 3b) and has much fewer states than the computation itself – exponentially smaller in many cases – resulting in substantial savings. Next, using the slice in Figure 3c, we can obtain the largest reachable state that satisfies p in the computation, which is denoted by W. We also know from the definition of $\mathsf{EF}(p)$ that every global state of the computation that can reach W satisfies $\mathsf{EF}(p)$, e.g., states enclosed in the dashed ellipse in Figure 3b. Therefore, applying this observation we can compute the slice with respect to $\mathsf{EF}(p)$ as shown in Figure 3d. In the slice, there are less number of vertices than the computation and the slice and the computation have the same set of global states up-to W. Finally, we check whether the initial state of the computation is the same as the initial state of the slice, since predicate detection is concerned whether the initial state of a computation satisfies a given predicate. If the answer is yes, then the predicate is satisfied, otherwise not, and a counter-example is returned. In this case the predicate p is satisfied, that is mutual exclusion is violated.

Note that the above temporal slicing algorithm returns all consistent global states of the computation that satisfies the given predicate not just one.

V. Predicate Detection using Slicing

We present efficient computation slicing algorithms for a subset of temporal logic CTL in [1]. The slices are computed recursively starting from the deepest nesting of predicates and applying the appropriate slicing algorithm, either temporal or boolean, while proceeding outwards.

Predicate detection is concerned about checking whether the initial consistent cut of a computation satisfies a given predicate. Since the slice contains all consistent cuts of the computation that satisfies the predicate, upon computing the slice, we simply check whether the initial consistent cut of the computation belongs to the slice. This can be done by ascertaining whether both initial cuts of the computation and the slice are the same.

Our predicate detection algorithm for a k-exclusion predicate in DNF is as follows. First, we compute slices wrt each conjunctive predicate p in every disjunct. Second, we obtain the slices wrt EF for every disjunct. Third, we check whether the initial cut of the computation is the same as the initial cut of the slice wrt $\mathsf{EF}(p)$ for any disjunct.

The complexity of our trace based predicate detection technique is polynomial in n, that is, $O(|p| \cdot n^2 |E|)$, where $|p|$ is the number of boolean and temporal operators in a predicate p, E is the set of vertices and n is the number of processes. However, this complexity is for a general class of temporal logic predicates. When p is a 1-exclusion predicate, the complexity can be reduced to $O(n|E|)$, since the complexity of slicing wrt conjunctive predicates is $O(|E|/n)$ and instead of computing the slices wrt $\mathsf{EF}(p)$, we check if the slice for p is empty or not. Similarly, the complexity is $C(n, k+1).(k+1).|E|/n$ for k-exclusion.

Note that slicing leads to efficient algorithms even for predicates such as $p_1 \wedge p_2$, where p_1 has efficient slicing algorithms but p_2 does not such as $p_2 = x_1 + x_2 * x_3 < 7$. This is because it is better to detect p_2 on the slice for p_1 instead of the computation, using say an exhaustive search algorithm, since the the state space of the slice is much smaller than the state space of the original computation.

VI. Experiments

We ran experiments with our tool Partial Order Trace Analyzer (POTA) with general slicing algorithms and the k-exclusion algorithm for monitoring execution traces of programs for mutual exclusion violations. The tool contains three modules; analyzer, translator, and instrumentor. POTA instruments the given program by inserting code at the appropriate places in the description to be monitored. The instrumented description is such that it outputs the values of variables relevant to the predicate in question and keeps a vector clock. POTA also implements computation slicing and k-exclusion algorithms. All experiments were performed on a 3.06 Ghz Xeon processor machine running Linux. We restricted the memory usage to 512MB, but did not set a time limit. The two performance metrics we measured are running time and memory usage. In the case of slicing both metrics also include the overhead of computing the slice. Further experimental results can be obtained from the website [11]. As experimental testbeds, we chose distributed dining philosophers, cache coherence and PCI protocols. We could stretch our verification algorithms by incrementing both the complexity and the number of processes in the systems, which also demonstrates the effectiveness of the algorithms for very large state spaces and complex designs.

A. Distributed Dining Philosophers

We use the Java protocol from [7] for this exercise. The protocol consists of multiple philosophers who sit around a table and spend their time thinking and eating. However, a philosopher requires shared resources, such as forks, to eat. The protocol coordinates access to the shared resources. Each philosopher has 3 local states namely $think$, $hungry$, and eat. The philosophers do not have a central server that they can query for fork availability. Instead each philosopher has a servant who communicates with the two neighboring servants to negotiate the use of the forks. The servants send "need left fork", "need right fork", "pass left fork", and "pass right

fork" messages back and forth. Each fork is always in the possession of some philosopher, one of the two on either side of the fork. When a philosopher finishes eating, it labels its two forks as dirty. A hungry philosopher's servant is required to give up a dirty fork in its possession, if asked for by its hungry neighbor's servant. This prevents starvation. We check the following property. We require mutually exclusive use of forks, that is, a shared resource should not be used by more than one philosopher at a time. This can be ascertained by checking whether two neighbor philosophers are eating at the same time. This safety property can be stated as $\bigwedge_{i,j \in 0 \ldots (n-1)} (\mathsf{AG}(\neg eat_i \vee \neg eat_j))$, where eat_i denotes that philosopher i is in eating state and j denotes the neighbor of philosopher i. We can check whether this property is violated by checking the complement of the safety property, which is $\bigvee_{i,j \in 0 \ldots (n-1)} (\mathsf{EF}(eat_i \wedge eat_j))$.

Figure 4 displays our results for this property.

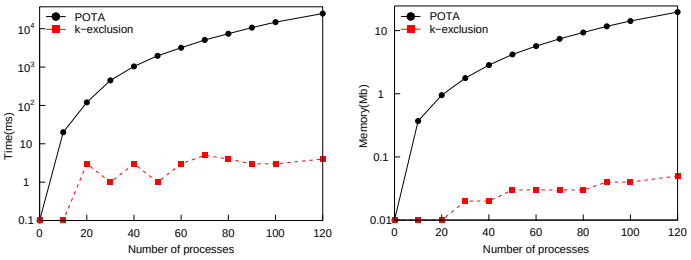

Fig. 4. Dining philosophers verification results for mutual exclusion property

B. Cache Coherence Protocol

The MSI (Modified Shared Invalid) cache coherence protocol is a protocol to maintain data consistency among a number of caches connected to a central directory structure in a multiprocessor system. The protocol is a directory based scheme in which individual processes snoop on all other processors' activities over a shared directory. The details of the protocol can be found in [11].

The property we checked on the MSI protocol is the safety property, "two caches cannot be in the modified state simultaneously". The complement of the property is $\mathsf{EF} smodified_i \wedge modified_j$, where i and j are cache identifiers.

Figure 5 displays our results for this property. k-exclusion took two millisecond and 1 MB to complete for 120 processes, whereas, POTA took 390 seconds and 220 MB.Due to the overhead associated in generating traces, we stopped generating traces for more than 120 processes.

C. PCI

The PCI Local Bus is a high performance bus with multiplexed address and data lines. It is used as an interconnect mechanism between peripheral controller components or add-in boards and processor/memory systems. Our example contains an arbiter, and a parameterizable number of nodes that can act either as a master or a target machine.

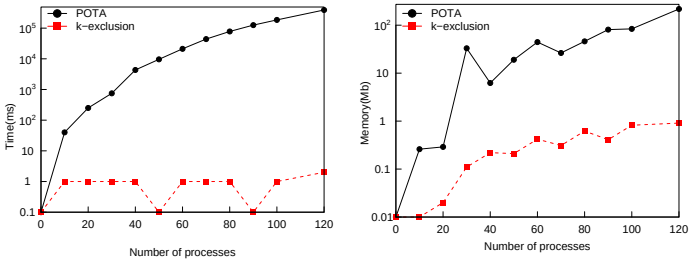

Fig. 5. MSI verification results for for mutual exclusion property

In the SoC based on the PCI backbone, we have a parameterizable number of devices that are non-deterministically sending requests to the arbiter for accessing the bus as a master and interacting with each other. The safety property that we verified is "there cannot be two masters of the bus". Symbolically, $\mathsf{EF}(master_i \wedge master_j)$, where i and j are device identifiers.

For this experiment, we have varied the number of nodes from 10 to 120 for the safety property The reason for the less number of nodes in the safety property experiments is that the property, given in DNF, size is larger in this case (that is, every possible combination of nodes i and j exists in the property). The related time and memory usage is displayed in Figure 6 for safety property. k-exclusion took two millisecond and 3 MB to complete for the worst case , whereas, POTA took 640 seconds and 390 MB.

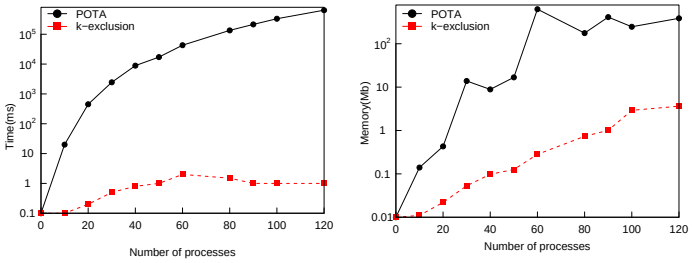

Fig. 6. SoC verification results for mutual exclusion property

VII. Conclusion

We presented a comparison of two techniques for detecting mutual exclusion violations. Our experimental results using POTA tool show that the the specialized k-exclusion algorithm performs substantially better than the slicing based algorithm. In all fairness, the slicing based algorithms are meant for general classes of predicates, hence do not exploit the properties of mutual exclusion predicates as k-exclusion algorithm does. Also, although the complexity of the slicing based algorithm specifically for 1-exclusion is similar to that of k-exclusion algorithm, we believe that due to the implementation overhead we obtain poor slicing performance. Nonetheless, for mutual exclusion predicates we have a very efficient and scalable algorithm implemented in POTA. For future work, we would like to investigate whether the ideas used in k-exclusion

algorithm can be used for other types of predicate detection as well.

REFERENCES

[1] A. Sen and V. K. Garg. Detecting Temporal Logic Predicates in Distributed Programs Using Computation Slicing. In *Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS)*, La Martinique, France, December 2003.

[2] K. Bhargavan, C. A. Gunter, I. Lee, O. Sokolsky, M. Kim, D. Obradovic, and M. Viswanathan. Verisim: Formal Analysis of Network Simulations. *IEEE Transactions on Software Engineering*, 28(2):129–145, 2002.

[3] C. Chase and V. K. Garg. Efficient Detection of Restricted Classes of Global Predicates. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 303–317, France, September 1995.

[4] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[5] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Resource Allocation with Immunity to Process Failure. In *Proceedings of the 20th IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 234–254. IEEE Computer Society Press, 1979.

[6] V. K. Garg. *Elements of Distributed Computing*. John Wiley & Sons, 2002.

[7] S. Hartley. *Concurrent Programming: The Java Programming Language*. Oxford University Press, 1998.

[8] S. Ikiz and V. K. Garg. Efficient Incremental Optimal Chain Partition of Distributed Program Traces. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, Lisbon,Portugal, July 2006.

[9] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a Run-time Assurance Tool for Java Programs. In *Proceedings of the 1st International Workshop on Runtime Verification (RV)*, volume 55 of *ENTCS*. Elsevier Science Publishers, 2001.

[10] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[11] Partial Order Trace Analyzer (POTA) web site, 2003. http://maple.ece.utexas.edu/~sen/POTA.html.

[12] A. Sen and V. K. Garg. Detecting Temporal Logic Predicates on the Happened-Before Model. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*, Fort Lauderdale, Florida, April 2002.

[13] A. Sen and V. K. Garg. Partial Order Trace Analyzer (POTA) for Distributed Programs. In *Proceedings of the 3rd International Workshop on Runtime Verification (RV)*, volume 89 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2003.

[14] A. Sen and V. K. Garg. Formal Verification of Simulation Traces Using Computation Slicing. *IEEE Transactions on Computers*, 56(4), 2007.

[15] K. Sen, G. Rosu, and G. Agha. Runtime Safety Analysis of Multithreaded Programs. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, 2003.