Debugging of Distributed Computation via Memory-Efficient Enumeration of Global States

Artur Andrzejak, Komei Fukuda[1]
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2002-31
February 6th , 2002*

E-mail: artur_andrzejak@hp.com, fukuda@ifor.math.ethz.ch

distributed computations, debugging, global states

We develop a memory-efficient off-line algorithm for the enumeration of global states of a distributed computation. The algorithm allows the parameterization of its memory requirements against the running time. In the extreme case, only one global state of a distributed computation must be held in the memory of the enumerating system at a time. This is particularly useful for debugging of memory-intensive parallel computations, e.g. in image processing or data warehousing. We also show how to apply our technique to evaluate in a memory-efficient way the predicate *Definitely*(Φ) defined by Cooper and Marzullo. The basis for these algorithms is Reverse Search, a paradigm successfully applied for enumeration of a variety of geometric objects.

* Internal Accession Date Only Approved for External Publication
[1] Institute for Operations Research, ETH Zurich, ETH Zentrum, CH-8092 Zurich, Switzerland

Debugging of Distributed Computations via Memory-Efficient Enumeration of Global States

Artur Andrzejak
Hewlett-Packard Labolatories
1501 Page Mill Road
Palo Alto, CA 94304, USA
artur_andrzejak@hp.com

Komei Fukuda
Institute for Operations Research
ETH Zurich, ETH Zentrum
CH-8092 Zurich, Switzerland
fukuda@ifor.math.ethz.ch

Abstract

We develop a memory-efficient off-line algorithm for the enumeration of global states of a distributed computation. The algorithm allows the parameterization of its memory requirements against the running time. In the extreme case, only one global state of a distributed computation must be held in the memory of the enumerating system at a time. This is particularly useful for debugging of memory-intensive parallel computations, e.g. in image processing or data warehousing. We also show how to apply our technique to evaluate in a memory-efficient way the predicate *Definitely*(Φ) defined by Cooper and Marzullo [7]. The basis for these algorithms is Reverse Search [2], a paradigm successfully applied for enumeration of a variety of geometric objects.

1 Introduction

Detecting certain conditions of a distributed computation is fundamental to solving problems related to debugging and monitoring of parallel programs, especially of the critical ones. These problems include debugging, error reporting, process control, decentralized coordination or load balancing [11, 4].

The problem of detecting such conditions - or equivalently, of evaluating a *global predicate* of a distributed computation - is not trivial and has received a considerable amount of attention [3, 7, 8, 10]. Global predicates are evaluated over *global states*, which are essentially unions of local states of all processors.

There are several approaches for evaluating of global predicates. For stable global predicates, *i.e.,* predicates that do not become false once they are true, an algorithm by Chandy and Lamport [6] is used. Another research direction exploits the structure of a global predicate to identify the global states for which it might be true [9]. The major drawback of these methods is that they either apply to small classes of predicates or are predicate-specific.

If the structure of the predicate is not known a priori, which happens frequently during debugging, the most general approach must be taken - enumeration of all global states [7, 10]. Cooper and Marzullo propose in [7] off-line algorithms for predicates *Possibly*(Φ) and *Definitely*(Φ) based on the enumeration of all global states. The predicate *Possibly*(Φ) holds if the system could have passed through a global state satisfying the predicate Φ. *Definitely*(Φ)

holds if the system definitively passed through a global state with Φ being true.

Unfortunately, the known enumeration algorithms are not memory-efficient: it is possible that an exponential number of global states must be hold simultaneously in the memory of the machine evaluating the global predicates. If the distributed computation is memory intensive, e.g. in the case of image processing or database applications, already few global states might exceed the memory capacity of the enumerating system.

In this paper we propose a new algorithm for memory-efficient enumeration of global states in a distributed computation. Specifically, our contributions are following.

- We use the framework of Reverse Search by Avis and Fukuda [2] to set up an algorithm for enumeration of global states of a distributed computation. This off-line algorithm is memory-efficient and thus well suitable for debugging of memory-intensive distributed computations. In extreme case, only *one* global state of the distributed computation is hold in the memory of the evaluating machine at a time.

- We look at the problem of reducing the time overhead inherent to off-line enumeration algorithms and parameterize the enumeration time at a cost of the memory usage of our algorithm.

- Further, we show how to evaluate in a memory-efficient way the predicate *Definitely*(Φ) using our technique (the evaluation of *Possibly*(Φ) in a memory-efficient way is trivial by our approach).

The above-mentioned framework of Reverse Search has been successfully used in both theoretical and practical sense for memory-efficient enumeration of a large variety of geometrical objects [5, 1].

2 Definitions

Informally, a distributed computation describes the execution of a distributed program by a collection of processes p_i, $i = 1, \ldots, n$. The activity of each sequential process p_i is modeled as executing a sequence of events $e_i^1, e_i^2, \ldots$. Each event e_i^k may be of the following type:

- an internal event, causing only a local state change of a process,

- sending of a message m to another process, denoted as $send(m)$,

- receiving of a message m from another process, denoted as $receive(m)$.

To cover the notion of "cause and effect" we introduce the binary relation $\rightarrow$ [3] defined over events, such that:

1. If e_i^j, e_i^k are events of a process p_i and $j < k$, then $e_i^j \rightarrow e_i^k$,

2. If $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$,

3. If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.

The only conclusion that can be drawn from $e \rightarrow e'$ is that the occurrence of e' *may* have been influenced by event e.

Formally, a distributed computation is a partially ordered set of events ordered by the relation $\rightarrow$. In an actual execution, all events for all processes occur in some total order (if events of two different processes occur at the same real-time, we may assume that the event of the process with the smaller index occurs before the event of the process with larger event). To be able to reason about executions in distributed systems, we introduce the notion of a *run*. A run R is a total ordering of all events of all processes that is consistent with the distributed computation (a linear extension of this partially ordered set).

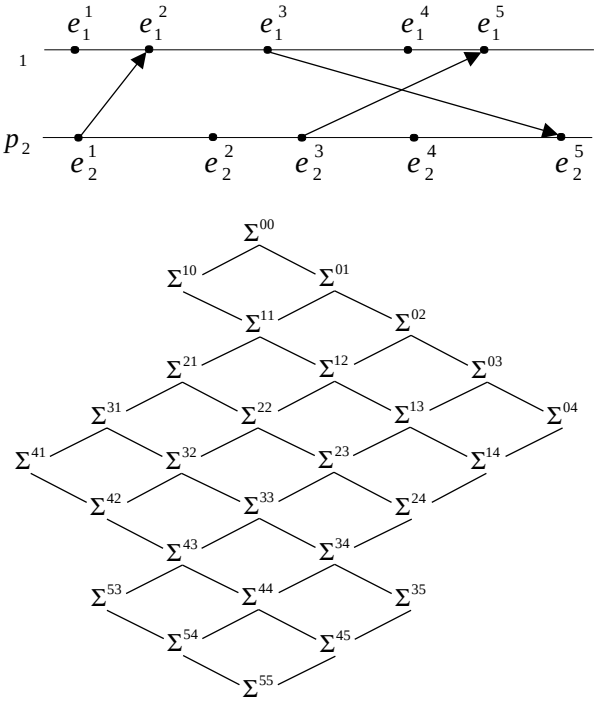

Figure 1: Example of a lattice

The *local history* h_i of a process p_i during the computation is a sequence of events $e_i^1 e_i^2 \ldots$. Let $h_i^k = e_i^1 e_i^2 \ldots e_i^k$ denote the *prefix* of h_i with the first k events belonging to process p_i. The *global history* H of a computation is a set $h_1 \cup \ldots \cup h_n$ containing all of its events. A tuple $(c_1, \ldots, c_n)$ of natural numbers specifies a *cut* $C = h_1^{c_1} \cup \ldots \cup h_n^{c_n}$ of the distributed computation. A cut C is *consistent* if for all events e and e' we have

$$(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C.$$

In other words, a consistent cut is left-closed under the causal precedence relation.

For process p_i, $1 \leq i \leq n$, and an integer $k \geq 0$ let σ_i^k denote the *local state* of a process immediately after having executed event e_i^k. The local state of a process may include information such as the values of local variables and the sequences of messages sent and received. The *global state* of a distributed computation is an n-tuple $\Sigma = (\sigma_1, \ldots, \sigma_n)$ of local states of all processes. Obviously there is a correspondence between the global states and cuts in the sense that a global state is a "frontier" of a cut. We also call a global state *consistent*, if it corresponds to a consistent cut. It is not hard to see that a run $R = e^1 e^2 \ldots$ results in a sequence of global states $\Sigma^0 \Sigma^1 \Sigma^2 \ldots$, where Σ^0 is the initial global state. Each global state Σ^i of the run R is obtained from the previous state Σ^{i-1} by some process executing the single event e^i. We say that Σ^{i-1} *leads to* Σ^i *in* R.

The set of all consistent global states of a computation along with the leads-to relation defines a *lattice* L. Let $\Sigma^{k_1,\ldots,k_n}$ be a shorthand for the global state $(\sigma_1^{k_1}, \ldots, \sigma_n^{k_n})$ and let $\ell = k_1 + \ldots + k_n$ be its *level*. Figure 1 shows such a lattice together with the original distributed computation.

2.1 Nonstandard definitions

For a global state $(\sigma_1, \ldots, \sigma_n)$, its *signature* is the tuple $(k_1, \ldots, k_n)$ of subscripts of the recently executed events $e_1^{k_1}, \ldots, e_n^{k_n}$ leading to the local states $\sigma_1, \ldots, \sigma_n$.

3 The Reverse Search method

In this section we describe briefly the framework of the Reverse Search method by Avis and Fukuda [2]. Assume that we are given a graph G whose vertices are the objects to be evaluated. In our case the objects are global states and G is the lattice L. We could possibly enumerate the vertices of G by applying the depth-search first algorithm to G. Such an algorithm finds eventually a spanning tree of G by storing the encountered vertices. The critical problem of this approach is the requirement of storing all vertices of G at the same time in memory. On the other hand, for Reverse Search it is sufficient to store only one vertex of G in the computer memory at a time.

For the Reverse Search method to be applied, we need to define a rooted spanning tree of G. (We will assume, that G has only one connected component; if this is not the case, the algorithm can be repeated for each connected component.) Let v^* be a distinguished vertex of G (which later will be the root of the tree); in our case it is the initial global state. The spanning tree of G is implicitly defined via a *local search function* $f : V(G)\backslash\{v^*\} \longrightarrow V(G)$ (where $V(G)$ denotes the vertex set of G). For every $v \in V(G)\backslash\{v^*\}$ the pair $(v, f(v))$ must be an edge in G. Furthermore, for every $v \in V(G)\backslash\{v^*\}$ there is a finite folding $f(f(\ldots f(v)\ldots))$ which yields v^*. Intuitively, every application of the function f brings us closer to the distinguished vertex v^*. It is not hard to see that for each $v \in V(G)\backslash\{v^*\}$ the function f defines a unique path from v to v^*. The union of these paths (regarded as sets of directed edges) define the (directed) spanning tree of G we are looking for. This graph is called the *trace graph* or simply the *trace* of the local search function f. Note that the trace graph is completely determined by f.

Now an execution of an algorithm based on Reverse Search resembles an execution of the depth-first search on the trace graph of f. In order to present it formally, we still need to specify how the graph G is given. In particular, we assume the following:

- An integer δ which bounds from above the maximum degree of a vertex of G.
- An *adjacency oracle* Adj explained below.

For an integer $i \in \{1, \ldots, \delta\}$ and a vertex $v \in V(G)$ the adjacency oracle $\mathrm{Adj}(v, i)$ gives the i-th neighbor of v in G or *null*. The function Adj is exhaustive and injective, that is if i goes from 1 to δ, then we obtain all neighbors of v in G and each of them exactly once; see [2] for the complete description.

These two functions are used in the following procedure ReverseSearch2 taken from [2], which is the core of the Reverse Search method.

```
procedure ReverseSearch2(Adj,δ,v*,f);
  (* j: neighbor counter *)
  v := v*; j := 0;
  visit vertex v*;
  repeat
    while j < δ do
      j := j + 1;
      if f(Adj(v, j)) = v then
        (* reverse traverse *)
        v :=Adj(v, j); j := 0;
        visit vertex v;
      endif
    endwhile;
    if v ≠ v* then
      (* forward traverse *)
      w := v; v := f(v); j := 0;
      repeat j := j + 1
      until Adj(v, j) = w (* restore j *)
    endif
  until v = v* and j = δ.
```

The reader might test on some examples that this procedure indeed traverses G in a depth-first search manner.

4 Re-executing a distributed computation

As it will be become clear in the next section, our application of Reverse Search requires to be able to "compute backwards in time", i.e. given a global state Σ, we should be able to compute a (certain) global state Σ' which leads to Σ. In this section we describe how to parameterize the trade-off between the time of such a computation and the required memory (or storage) size.

Assume that P is a path in a lattice L corresponding to a distributed computation, such

that each global state on P leads to the next one. Given a node (global state) Σ in P, we want to obtain its predecessor Σ'. Note that the only difference between the both global states is that in Σ' one of the processes p_j has not yet executed its next event $e_j^{k_j}$ for some j, $1 \leq j \leq n$. Thus, given Σ, our goal is to "set back in time" p_j to the state before its execution of $e_j^{k_j}$.

Obviously the method with the highest memory usage is to store all global states on the path P and then retrieve the required global state from the memory or other storage as necessary. Note that we can store the path in a "differential" way, *i.e.* for the above-mentioned global states Σ' and Σ we store for Σ only the local state of the processor p_j after the execution of the event $e_j^{k_j}$. The local states of all other processes are respectively identical in Σ' and Σ. Still, if Σ is the last global state in P and has the signature $(k_1, \ldots, k_n)$, then we must store k_i local states of the processor p_i, for each $i = 1, \ldots, n$, in total $\ell = k_1 + \ldots + k_n$.

To lessen the memory usage, we can store only some local states of each process. Let $q > 0$ be an integer which determines that for each process we store every q-th local state only (beginning with the initial one). For $q = 1$, the retrieval of Σ' from Σ is the same as above. However, for $q > 1$, we retrieve the stored local state of p_j with index $\lfloor \frac{k_j - 1}{q} \rfloor$ from the memory and "load" this state into p_j. Subsequently, we recompute the local states of this process just until the event $e_j^{k_j}$ by executing its program code. Since the messages received by p_j are considered as a part of its local state [3], we assume that we store for each process p_i, $1 \leq i \leq n$, every message sent to this process (together with the index of the sender and the event number causing this message). Thus, we can retrieve them from the local state of p_j in Σ, if they should be necessary for executing the code of this process.

For example in Figure 1 assume that the above-mentioned path is Σ^{00}, Σ^{10}, Σ^{11}, Σ^{21}, Σ^{22}. For $q = 2$, the saved local states of the processes are the two local states of $p_1$before the events e_1^1 and e_1^3 and the two local states of p_2 before the events e_2^1 and e_2^3.

Assume that R is an upper bound on the time to retrieve a local state (from memory or storage) and to "load" it into a process p_i, $1 \leq i \leq n$. Furthermore, we write E for an upper bound on the time which a process p_i, $1 \leq i \leq n$, needs to execute an event. For $q > 1$, in order to compute Σ' from Σ a process p_j must execute at most $q - 1$ events after retrieval of its last stored local state. Therefore, the time for computation of Σ' from Σ no bigger than:

$$R + E(q - 1). \tag{1}$$

Let S be the maximum storage size of a local state of a single process. It is not hard to see that if the level of the last global state Σ in the path P is $\ell = k_1 + \ldots + k_n$, then the total storage needed for P is at most

$$S\frac{\ell}{q}. \tag{2}$$

5 Enumeration of Global States

Our goal it to enumerate all (consistent) global states of a distributed computation by traversing the lattice L of global states. In this section we describe how to achieve this by applying the procedure ReverseSearch2 from Section 3. In order to do this, we need to specify several problem-dependent elements used by the Reverse Search method. The main challenge of this task is to find efficient implementations of these elements, especially of the local search function f.

In the following, we consider L as a (directed) graph, where the nodes are global states and (Σ^i, Σ^j) is an edge, if Σ^i leads to Σ^j. Observe that this graph is connected.

5.1 Problem-dependent elements of the Reverse Search

For the Reverse Search method to be applied, we need to specify the following problem-dependent elements:

- A local search function f and its implementation,
- An adjacency oracle Adj and its implementation,
- A distinguished vertex v^* , the root of the trace graph,
- The maximum out-degree of a global state in the lattice L.

For technical reasons, we also provide the following element:

- An implementation of the test $f(\text{Adj}(v, j)) = v$ in the reverse traverse step which does not use f nor Adj.

All three the local search function, the adjacency oracle and the test $f(\text{Adj}(v, j)) = v$ are discussed below. As a distinguished vertex v^* we set the initial global state Σ^0 of the computation. Finally, the maximum degree in the lattice is the number of processes n, as a global state can lead to another global state only by executing an event on one of them.

5.2 The Adjacency Oracle Adj

For a given global state Σ and an integer j, $1 \leq j \leq n$, we define $\text{Adj}(\Sigma, j)$ as follows. If there exists a global state Σ' such that Σ leads to Σ' by executing the next event on the process j, then $\text{Adj}(\Sigma, j)$ is Σ'; otherwise $\text{Adj}(\Sigma, j)$ is *null*. It is not hard to see that Adj indeed determines the lattice L of the distributed computation.

As for the implementation, we assume that the current global state Σ is "loaded" on the n processes $p_1, \ldots, p_n$. The value of $\text{Adj}(\Sigma, j)$ is computed as follows. First we test, whether the process p_j is waiting for a message and cannot execute the next event (see also Section 7.2). If this is the case, then $\text{Adj}(\Sigma, j)$ returns *null*. Otherwise, we let p_j execute the next event. As a consequence the new global state on the process p_i is exactly $\text{Adj}(\Sigma, j)$.

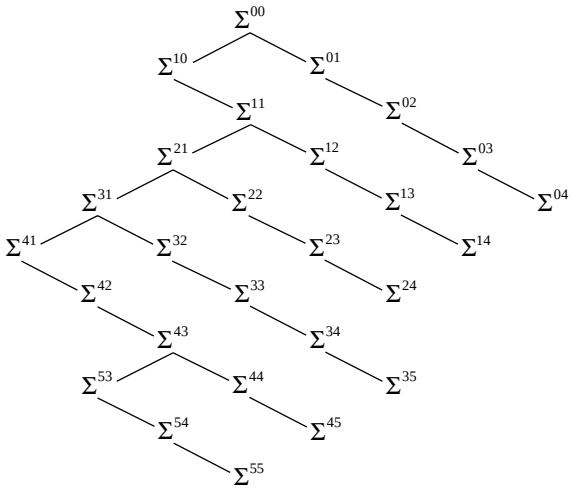

Figure 2: A trace of the lattice from Figure 1 induced by the local search f

Note that our definition of Adj implies that its values are only the "outgoing" global states in respect to the first argument of Adj (i.e. global states with level one higher than the argument). However, it is not hard to see that the correctness of the Reverse Search method is not influenced by this fact.

5.3 The local search function f

A possible and quite natural realization of a local search function is the following one. For a global state Σ of level ℓ we define $f(\Sigma)$ to be the global state of level $\ell - 1$ which leads to Σ and has lexicographically smallest signature. For example, in Figure 1 we have $f(\Sigma^{43}) = \Sigma^{42}$ and $f(\Sigma^{53}) = \Sigma^{43}$. Figure 2 shows the trace of the local search for the lattice in Figure 1.

The question is how to implement f, since the computation of $f(\Sigma)$ from Σ is a "time-

reversed" execution of an event of one of the processes. We propose two approaches. The first one is very memory-efficient and can be used for both the execution of f in the test $f(\mathrm{Adj}(v,j)) = v$ and the execution of f in the forward traverse step. However, it has a large computational overhead. The second approach works only for the forward traverse case, but it allows us to trade the computational time at a cost of additional memory usage.

Both approaches assume the case that the current global state - the argument of f - is "loaded" on the parallel machine which executes the distributed computation.

5.3.1 Universal implementation of f

The following implementation has a very small memory usage - in the order of memory requirement of one global state. It can be applied both for the reverse traverse and the forward traverse cases. On the other hand, it has a large computational overhead.

Let $(k_1, \ldots, k_n)$ be the signature of the current global state Σ. Starting with $j = 1$, we try to compute for the consecutive values of $j \leq n$ the local state $\sigma_j^{k_j-1}$ of the process p_j after the event $e_j^{k_j-1}$ as follows. First we save the current local state of the process p_j. Then we initialize p_j with its initial local state and let the process p_j "replay" all events $e_j^1 \ldots, e_j^{k_j-1}$. The constraint is that p_j must not use the messages sent "after" the current global state, i.e. if p_j uses a message for "replaying" an event, this message must be sent by an event e_i^k with $k \leq k_i$ for an $1 \leq i \leq n$, $i \neq j$. If p_j cannot reach the local state $\sigma_j^{k_j-1}$, then there is no global state with signature $(k_1, \ldots, k_j - 1, \ldots)$ (or $(k_j - 1, \ldots)$ for $j = 1$). In this case we restore the local state of the process p_j and repeat this procedure for the consecutive values of j etc. until success. Because we are testing for the existence of the appropriate global state in the order of lexicographically increasing signatures, the first such a global state found will be the correct $f(\Sigma)$. Note that we have simultaneously computed the global state $f(\Sigma)$, not only its signature.

5.3.2 Implementation of f with parameterized running time

For the current global state Σ let $P(\Sigma)$ be the (inverted) unique path induced by the local search function f on the lattice L of the distributed computation. We maintain an ordered list of signatures of the global states on $P(\Sigma)$ in the following way. Each time when the reverse traverse step of the algorithm takes place (i.e. the test $f(\mathrm{Adj}(v,j)) = v$ is successful), we store the signature of the newly determined global state in the list. Since the signatures have very small memory usage compared to global states, the additional memory requirement for the list can be neglected.

Furthermore, we apply the technique from Section 4 and store a subset of the local states of each process on the path $P(\Sigma)$. The size of the subset is controlled by the parameter q defined there.

Now it is not hard to see that if Σ is the currently visited global state, then $f(\Sigma)$ is the before-last element of $P(\Sigma)$. Thus, given the information in our signature list and the stored local states, we can compute $f(\Sigma)$ as described in Section 4. The maximum computation time for f is then given by (1), and the maximum memory usage for $P(\Sigma)$ by (2).

Note that this approach cannot be used for the test $f(\mathrm{Adj}(v,j)) = v$ in the reverse traverse step, since $\mathrm{Adj}(\Sigma, j)$ is not in $P(\Sigma)$. Thus, the implementation of f presented above can be only used in conjunction with the "atomic" implementation of the test $f(\mathrm{Adj}(v,j)) = v$ from Section 5.4.

5.4 Implementation of the test $f(\mathbf{Adj}(v,j)) = v$

For technical reasons we assume that the following approach is used in conjunction with the implementation of f from Section 5.3.2. We first describe the conditions under which the test $f(\mathrm{Adj}(v,j)) = v$ is true, and turn then our attention to its implementation.

Assume that the current global state Σ has the signature $(k_1, \ldots, k_n)$. For an integer j, $1 \leq j \leq n$, the global state $\mathrm{Adj}(\Sigma, j)$ (if exists) has the signature $(k_1, \ldots, k_j + 1, k_{j+1}, \ldots)$ (or $(k_1 + 1, k_2, \ldots)$ for $j = 1$). By the definition of f either $j = 1$ and so $f(\mathrm{Adj}(\Sigma, j)) = \Sigma$, or for $j > 1$ we find $f(\mathrm{Adj}(\Sigma, j))$ by successively testing the existence of the global states with the signatures $(k_1 - 1, k_2, \ldots, k_j + 1, \ldots)$, $(k_1, k_2 - 1, k_3, \ldots, k_j + 1, \ldots)$ till $(k_1, \ldots, k_j, \ldots)$, respectively. Thus, the test returns true if $\mathrm{Adj}(\Sigma, j)$ exists and $j = 1$ or $f(\mathrm{Adj}(\Sigma, j))$ has the signature $(k_1, \ldots, k_j, \ldots)$ (the last inspected).

In the actual implementation, we first check whether $\mathrm{Adj}(\Sigma, j)$ exists as described in Section 5.2; if not, the test fails. Otherwise the test is immediately successful in case $j = 1$. For $j > 1$, we have to check successively the existence of the global states with signatures $(k_1 - 1, k_2, \ldots, k_j + 1, \ldots)$ until $(k_1, \ldots, k_{j-1} - 1, k_j, \ldots)$ by attempting to recompute these global states. If one of them exists, the test fails. Otherwise it is true since then $f(\mathrm{Adj}(\Sigma, j))$ has the signature $(k_1, \ldots, k_j, \ldots)$. We recompute each of these global states by the technique described in Section 4. This approach can be applied since only one of the processes p_i, $1 \leq i < j$, needs to re-execute its computation. We use for this aim the data of the local states stored for the computation of f from Section 5.3.2.

In worst case, we need to compute $n-1$ global states, and each computation needs at most the time specified by (1). Together with the upper bound E on the time required to execute Adj, the whole test needs no longer than

$$E + (n-1)(R + E(q-1)). \qquad (3)$$

5.5 Optimizing the forward traverse part of Reverse Search

By taking a closer look at the Reverse Search method we notice that for restoring the value of j (in the forward traverse part) the adjacency oracle Adj does not need to compute a new global state $\mathrm{Adj}(\Sigma, j)$ since only the signature of $\mathrm{Adj}(\Sigma, j)$ is needed. The computation of such a signature is trivial, since if a global state Σ has signature $(k_1, \ldots, k_n)$, then $\mathrm{Adj}(\Sigma, j)$ has signature $(k_1, \ldots, k_j + 1, \ldots, k_n)$. We can then rewrite the forward traverse part of the procedure ReverseSearch2 from Section 3 as follows. Let $Adj_{sig}(\Sigma, j)$ be a function which computes only the signature of $\mathrm{Adj}(\Sigma, j)$.

if $v \neq v^*$ **then**
  (* forward traverse *)
  $w :=$ signature of v;
  $v := f(v)$; $j := 0$;
  repeat $j := j + 1$
  until $\mathrm{Adj}_{sig}(v, j) = w$ (* restore j *)
endif

By this change, the time for restoring the value of j becomes constant.

5.6 Analysis of running time

In this section we bound from above the running time of the evaluation algorithm for the case of the solution with parameterized running time, *i.e.* if the implementations from Section 5.3.2 and Section 5.4 are used. If the implementation from Section 5.3.1 is used, no good bound of the running time can be given, since then the execution time of f depends strongly on its argument.

Basically, we only need to apply the Theorem 2.4 from [2]. First we translate the symbols to

our application (using the notation from Section 4) and supply their values:

- $t(\text{Adj})$, the execution time for Adj is at most E (Section 5.2),
- δ, the maximum (out-)degree in the lattice L is n,
- $t^R(\text{Adj}, f)$, the worst-case time for the test $f(\text{Adj}(v, j)) = v$ is given by (3),
- $t(f)$, the worst-case time for f is given by (1),
- $t^F(\text{Adj}, f)$, the time needed to restore j is constant (Section 5.5),
- $|L|$ is the total number of global states in the lattice.

Theorem [2, Theorem 2.4]. *The time complexity of ReverseSearch2 is*

$$O((t(Adj)+\delta t^R(Adj,f)+t(f)+t^F(Adj,f))|L|).$$

Using notation from Section 4 and assuming that E, R and q are non-constant, we have then:

Corollary 1. *Suppose that the implementations from Section 5.3.2 and from Section 5.4 is used. Then the running time of the enumeration of all global states is*

$$O(n^2(Eq+R)|L|),$$

and the storage needed by the algorithm is at most

$$S(n+\frac{\ell_{max}}{q}),$$

where ℓ_{max} is the maximum level in the lattice.

6 Detecting *Definitely*(Φ) in a memory-efficient way

As noted in the introduction, the algorithms given in [7] for detecting of *Possibly*(Φ) and *Definitely*(Φ) are not memory-efficient. In worst case, each algorithm holds in memory all global states of a single level of L. This number is exponential in n.

We can apply our memory-efficient algorithm for detecting of *Possibly*(Φ) in a straightforward way: just enumerate all global states until Φ applies or the enumeration terminates.

For the detection of *Definitely*(Φ) some more effort is needed. Assume that we remove from the lattice L all global states for which Φ applies, obtaining a graph L'. The idea is to run our enumeration algorithm on L' instead of L. Note that *Definitely*(Φ) does *not* hold exactly if there is a path from the initial state to a terminal global state in L'. Thus, our enumeration will reach a terminal state exactly in this case. Since we traverse L' in a depth-first search manner, the case that *Definitely*(Φ) is not true is detected relatively fast. On the other hand, we might have to evaluate almost all global states to reach the conclusion that *Definitely*(Φ) is true.

We need only slight changes of f and Adj to enumerate the global states of L' instead of L. At each computation of Adj we evaluate Φ for the new created global state; if Φ applies, we simply return *null*. The local search f is modified in an analogous way.

7 Practical implementation issues

7.1 The enumeration system

For the practical implementation of the enumeration we suggest two models of the enumeration system:

Model A The real system which runs the distributed computation is actively used in the enumeration process. This is a suitable option e.g. if the target system is heterogeneous. Furthermore, this approach simpli-

fies the programming of the enumeration environment.

Model B The target system and its processes are simulated on different machine (which might be itself a parallel computer). This requires more programming overhead for the enumeration environment. However, the enumeration can be then parallelized efficiently, and also more powerful machines than the target system can be used. This option is also useful, if the data redistribution overhead on the target system is large.

In case of Model A we assume that we use the target system with n processes which carry out the distributed computation and an external process *observer*, which controls the program execution of every other process, similar in capacities to a distributed debugger. In particular, for every process p_j, $1 \leq j \leq n$, the observer is able to:

- Check, whether p_j is waiting for a message and cannot execute the next event,
- If possible, start the execution of the next event on the process p_j which stops after this event is processed,
- Start an execution of a series of events on p_j, and provide p_j with the (previously recorded) messages necessary to carry out this execution,
- Record the current local state of p_j and store it,
- Restore a previously stored local state on p_j.

In case of Model B the role of the observer will be taken by the simulation environment. The abilities listed above are also necessary, yet they easily integrated into the simulation framework.

7.2 Detection of *receive*-events for time-sensitive computations

A particular attention must be paid to the detection of the *receive*-events. The occurrences of these events might depend on the timing relations between the processes and the message transmission times. Since our enumeration is an off-line algorithm (as the algorithms by Cooper and Marzullo [7]), the timing relations of the processes are in general distorted, and so the occurrences of the *receive*-events might be shifted in time or not detected at all.

Consider the situation that a process p_i is executing a job. At some moment in time it receives a message m from another process, stops the current processing and starts another action. This gives rise to a new event e_i^j. The point is that if in such a distributed computations p_i does not know *a priori*, when (and at all) the message m will arrive, and so the event e_i^{j-1} proceeding the new event is "dynamically" changed by the arrival of m. As a consequence, the lattice L of such a distributed computation depends on the execution times of the processes and communication times between them.

There are several remedies for this problem. We assume in the following, that the internal clock of each process is increased only when the process is executed or simulated, i.e. it measures the time as in a real run.

1. One solution is to make a real run of the distributed computation on the target system, and store for each process the arrival time of every message which gives rise to a new event of this process (or the state of the program counter of the process at the arrival time etc.) During the enumeration, the end of an event can be easily reconstructed from this data.
2. Another solution is to artificially limit the length of the execution of a single event. This has the disadvantage that the total number of global states might increase.

8 Acknowledgments

The first author would like to thank Professor Friedemann Mattern, ETH Zurich, for valuable suggestions.

References

[1] A. Andrzejak and K. Fukuda. Optimization over k-set polytopes and efficient k-set enumeration. In *Proc. 6th International Workshop on Algorithms and Data Structures (WADS'99)*, LNCS 1663, pages 1-12, Vancouver, August 1999.

[2] D. Avis and K. Fukuda. Reverse search for enumeration. *Disc. Applied Math.*, 65:21–46, 1996.

[3] O. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: fundamental concepts and mechanisms. In S. J. Mullender, editor, *Distributed Systems*, Chapter 4, ACM Press, 1993.

[4] O. Babaoğlu and M. Raynal. Specification and Verification of Behavioral Patterns in Distributed Computations. In *Proc. Fourth IFIP Working Conference on Dependable Computing for Critical Applications,* San Diego, 1994.

[5] A. Brüngger, A. Marzetta, K. Fukuda, and J. Nievergelt. The parallel search bench zram and its applications. *Annals of Operations Research*, 90:45-63, 1999.

[6] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63-75, February 1985.

[7] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163-173, Santa Cruz, California, May 1991.

[8] C. Diehl, C. Jard, and J. X. Rampon. Reachability analysis on distributed executions. In J.-P. Jouannaud M.-C. Gaudel, editor, *Proceedings of TAP-SOFT*, LNCS 668, pages 629-643, Orsay, Paris, France, April 1993.

[9] V. K. Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Trans. on Parallel and Distributed Systems*, 7(12):1323–1333, 1996.

[10] K. Marzullo and G. Neiger. Detection of global state predicates. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WDAG-91)*, pages 254-272, Delphi, Greece, October 1991.

[11] N. Mittal and V. K. Garg. Debugging distributed programs using controlled reexecution. *Symposium on Principles of Distributed Computing*, 239-248, 2000.