Distributed State Space Generator (preliminary)

Bert Lisser

October 11, 2006

Contents

1 Introduction **1**

2 Process Structure **2**

3 Database Structure **4**

4 Directory Structure (the `.dmp` format) **7**

5 Computing Transitions **9**

6 Selecting Unexplored States **10**

7 Resuming after Abort **11**

8 Manager **13**

9 Monitoring with tool `contact` **13**

10 Benchmarks **14**

1 Introduction

The described tool, called `instantiators`, performs a breadth first traversal through the state space defined by an mCRL specification. This tool consists of communicating Linux processes which run on a fixed set of distributed connected computing nodes, for instance a cluster. The main options are: which tell what must be done during traversal of the state space:

- Writing transitions
- Writing traces to actions satisfying a certain regular expression or to deadlock state

Extra features of the state space generator are:

- Abort and resume facility. An abort can vary from clean exit to an unexpected power down event.
- postpones `tick`-transitions to the moment at which no other transitions can be found.
- Possibility to monitor the remote running processes

The breadth first traversal will be done level after level. A level of the spanning tree of the state space will be simultaneously traversed by one or more computing nodes. Per level each computing node traverses a different part of the transitions departing from this level. After a computing node is finished with traversing its allocated part it have to wait for the completion of all other computing nodes. Meanwhile the waiting computing nodes send destination states to each other. After that the traversal of the next level will be started. Advantages of breadth first traversal are:

- The first trace found to a certain action or deadlock state is the shortest trace
- A simple termination criterium. Traversal is completed if a level contains 0 transitions.

There is only one output format for the transition system, called the `.dmp` format. The transition system written in `.dmp` format can be reduced or copied to other formats. For this the tools `ltsmin` and `ltscp` are available. This `.dmp` format will be also used, when the system is restarted, for entering into the state which the system has left after an abort. Advantages of having an output format different of the already existing ones are:

- Output format is chosen such that it is optimal for generating and restoring transition systems by this tool
- Output format is very simple. There is no meta information about the output, such as a file `info` which contains information about number of written states and transitions. Even if an abort occurs at a bad moment, the restart will succeed. Consistency of the written transition system will be checked by controlling the sizes of the dumped files. If there are consistency problems, the files will be truncated such that the written transition system is consistent.

2 Process Structure

The double lined nodes play the server role, the other nodes are clients. The messages mentioned left of the slash are send by the client; the messages right of the slash are send by the server.

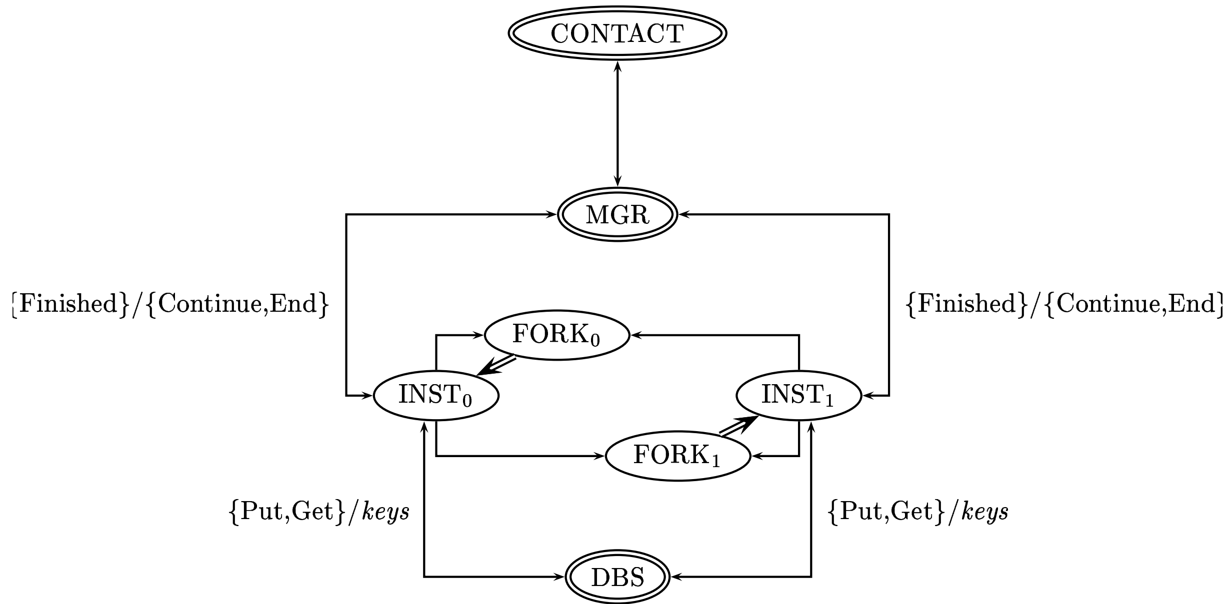

Figure 1: Distributed instantiator using two nodes

Manager (MGR)

Its main tasks is synchronization. It will take care of the breadth first character of the state space. An instantiator on a node will halt and send the message `finished` after a level of the breadth first tree is generated. After all instantiators are finished the manager sends `continue` messages to all clients if at least one client has produced an unexplored state. It sends `terminate` messages otherwise.

Database Server (DBS)

A state vector is a vector of closed terms, whose size is the number of process parameters. The state vector will be divided into two parts. The main task of the database server is to assign globally unique indexes to these parts. The databases used in the database server will be locally mirrorred to restrict network traffic.

Contact (CONTACT)

With this tool the user can monitor the distributed state space generating processes. This tool can be started on the local workstation if there is an internet connection with the cluster on which the distributed state space generator is running.

Instantiator (INST)

After having received a set of states, its two main tasks are:

- Writing for each unexplored state its outgoing transitions.
- Sending the destination states to the right data forks. A checksum over the state determines to which data fork a state will be sent.

Data Fork (FORK)

Selects the unexplored states out of the states send by the instantiators. Therefore it maintains a locally accessible database, in which each encountered state is stored. The newly added states will be written to an instantiator.

3 Database Structure

A state vector, which will be entered as a vector of closed terms, will be stored as a (balanced) binary tree. The main advantages of this approach above the natural vector representation are that less memory is needed, and there is less network traffic between client and database server. The nodes of the tree will be store into databases. There are three kind of data bases (nodes): top nodes, intermediate nodes, and leaf nodes. The databases are in fact indexed sets. The leaf nodes contain `ATerms`. The other nodes contain pairs of indexes. Each index refers to a data element of another node. The top node is the root of the tree. The core operations in the whole database tree are *fold* and *unfold*. The function *fold* returns the index, which is an unique pair of numbers, belonging to a state vector. The function *unfold* returns the state vector belonging to an index. The functions *fold* and *unfold* use the auxilary database functions *put* and *get*.

- $put(vdb_k, (x, y))$. Returns index of (x, y) in intermediate database vdb_k. If data element (x, y) is not present, then (x, y) will be added and the index $count(vdb_k) - 1$ will be returned.
- $get(vdb_k, i)$. Returns the ith data element (x, y) of intermediate database vdb_k
- $put(tdb, t)$. Returns index of t in leaf database database tdb. If data element t is not present, then t will be added and the index $count(tdb) - 1$ will be returned.

- $get(tdb, i)$. Returns ith data term of leaf database tdb.
- $fold([x_0, \ldots, x_{n-1}])$. Returns index belonging to the state vector which exists of n closed terms by performing the following algorithm.

  for $i \in [0, n)$
  $\quad v_{\sigma(i+n)} := put(tdb, x_i)$
  for $i := n - 1$ **downto** 2
  $\quad v_{\sigma i} := put(vdb_i, (v_{2i}, v_{2i+1}))$
  return $[v_{\sigma 2}, v_{\sigma 3}]$

  Fixed permutation σ can be omitted; its role is to keep the generated binary tree balanced. A permutation must be permitted in this algorithm. A permutation is permitted in this algorithm if

  $$(\forall i \in [2, n)\ \exists k_1, k_2 > i)\ \sigma(k_1) = 2i \wedge \sigma(k_2) = 2i + 1$$

- $unfold([z_0, z_1])$. Returns the state vector x which exists of n closed terms belonging to index (z_0, z_1) by performing the following algorithm.

  $v_{\sigma 2} := z_0$; $v_{\sigma 3} := z_1$
  for $i := 2$ **to** $n - 1$
  $\quad (v_{2i}, v_{2i+1}) := get(vdb_i, v_{\sigma^{-1}(i)})$
  for $i \in [0, n)$
  $\quad x_i := get(tdb, v_{\sigma^{-1}(i+n)})$
  return x

Top nodes

These databases, also called segments, are located on N data forks. Each segment is located on one data fork, and each data fork accesses one and not more than one segment. The keys which are stored are checkpointed, which means that for each key (v_0, v_1) on client k: $v_0 + v_1 \bmod N = k$. These data bases are internally numbered $0, -1, \ldots, N - 1$.

Intermediate nodes

These databases are maintained on the database server. These databases can be entered and updated via the network. Each client mirrors these databases to restrict network traffic. If the state vector of closed terms has fixed length n then these data bases are internally numbered $0, \ldots, n - 1$.

Behaviour of $get(vdb_k, i)$ invoked on a data fork

1. $i < count(vdb_k)$. The database manager doesn't have to be consulted. The pair (x, y) to which index i refers is available in the locally stored intermediate database vdb_k and will be returned.

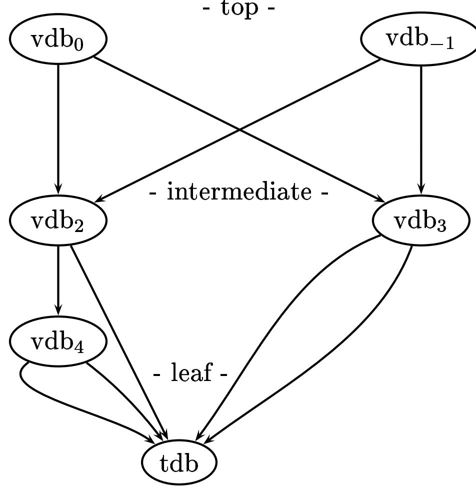

Figure 2: Database structure, 2 clients and 5 process parameters

2. $i \geq count(vdb_k)$. The pair (x, y) to which index i refers is not available in the locally stored intermediate database vdb_k. The pair $(count(vdb_k), i)$ will be sent to the database manager. The data base manager answers with ordered list of intermediate pairs which are present in vdb_k maintained by the database manager and not present locally. These pairs will be put one after one in the locally stored database vdb_k. The pair (x, y) to which index i refers is now available and will be returned.

Behaviour of $put(vdb_k, (x, y))$ invoked on a data fork

1. (x, y) is present locally in vdb_k. The database manager doesn't have to be consulted. The index of (x, y) in vdb_k will be returned.

2. (x, y) is not present locally in vdb_k. The pair $(count(vdb_k), (x, y))$ will be sent to the database manager. If his vdb_k does not contain (x, y), then this will be added and this will be appended to the file: nodes/vdb_k. The data base manager answers with ordered list of intermediate pairs which are present in vdb_k maintained by the database manager and not present locally. These pairs will be put one after one in the locally stored database vdb_k. The index of (x, y) in vdb_k will be returned.

Leaf nodes

The same for intermediate nodes holds for leaf nodes, except for the fact that the key is an `ATerm` (instead of a number pair). The keys in all leaf nodes don't refer to keys located in different databases, so it is possible to share the different leaves of the tree. All leaf nodes are shared by one leaf database. This database will be mirrored in each data fork.

Behaviour of $get(tdb, i)$ invoked on a data fork

1. $i < count(tdb)$. The database manager doesn't have to be consulted. The term t to which index i refers is available in the locally stored leaf database tdb and will be returned.

2. $i \geq count(tdb)$. The term t to which index i refers is not available in the locally stored leaf database tdb. The pair $(count(tdb), i)$ will be sent to the database manager. The data base manager answers with ordered list of intermediate pairs which are present in tdb maintained by the database manager and not present locally. These terms will be put one after one in the locally stored database tdb. The term tdb to which index i refers is now available and will be returned.

Behaviour of $put(tdb, t)$ invoked on a client

1. t is present locally in tdb. The database manager doesn't have to be consulted. The index of t in tdb will be returned.

2. t is not present locally in tdb. The pair $(count(tdb), t)$ will be sent to the database manager. If his tdb does not contain term t, then this will be added and this will be appended to the file: nodes/tdb. The data base manager answers with ordered list of intermediate pairs which are present in tdb maintained by the database manager and not present locally. These pairs will be put one after one in the locally stored database tdb. The index of t in tdb will be returned.

4 Directory Structure (the `.dmp` format)

Steppers part

A record of a file in the stepper part consists of a pair of integers. A record can be

Database reference. A pair of integers greater equal 0. Each integer refers to a database element which is stored in the NODES part.

End of level marker. This marks the end of level, it is the pair $(-1, n)$, in which n is the number of transitions in that level. The end of level marker

is needed during reading the files in the case of resuming state space generation after an abort.

Artificial unreachable state after deadlock. It is the pair $[0, -11]$.

An ith transition from segment j to k, $trans_{jki}$, is a triple consisting of the ith records of the files: $\text{steppers}_j/\text{src}_k$, $\text{steppers}_k/\text{act}_j$, and $\text{steppers}_k/\text{dest}_j$. The sizes of these three files are equal and are a multiple of 8 bytes. The pairs in $\text{steppers}_j/\text{src}_k$ are stored in data fork j, and the pairs in $\text{steppers}_k/\text{dest}_j$ will be stored in data fork k.

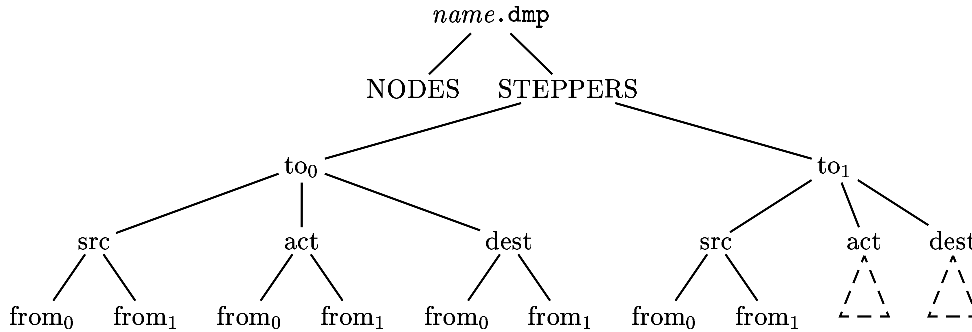

Figure 3: steppers part - number of clients 2

Deadlocks

At initialisation each top database contains a deadlock state marker. A marker located in database segment s will be notated as $\texttt{deadlock}_s$. If a deadlock is encountered at segment s then a transition from the source state, with label `deadlock`, to $\texttt{deadlock}_s$ will be written.

Nodes part

`adb` contains contains strings which represent actions

`tdb` contains strings which represent the shared leaf nodes

$\texttt{vdb}_k$ contains pairs of integers in which each left (right) integer is an index belonging to a data record stored in its left (right) child node.

5 Computing Transitions

Let N be the number of computation nodes. Suppose each computation node is handling an unique database segment, which is numbered from 0 until $N-1$. Let s be the number of the database segment which is handled by a computation node.

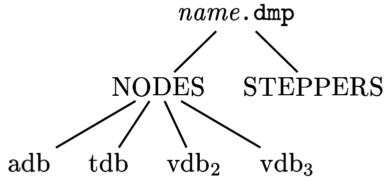

Figure 4: nodes part - number of process parameters is 5

Writing transitions

This will be done by $\mathit{WriteTransition}_l(x, id_x, a, y)$ in which x is a source state, id_x is the index of x stored in data fork s, a is an action, and y is a destination state. Let l be a checksum of y. Then $\mathit{WriteTransition}_l(x, id_x, a, y)$ writes respectively x, $[a, id_x]$, and y in the files: $\mathrm{steppers}_s.\mathrm{src}_l$, $\mathrm{steppers}_l.\mathrm{act}_s$, and $\mathrm{steppers}_l.\mathrm{dest}_s$. Action a and index id_x are represented by integers, and the states x and y are represented by a pair of integers.

Computing set of outgoing transitions

This set will be returned by `enabled(x)`, in which x is a state represented as a pair of integers greater equal 0. The number of process parameters n must be greater equal 2. The following steps occur:

- The state x, which is a pair of integers greater equal 0, must be unfolded to a vector of closed terms of size n.
- The enabled set of outgoing transitions is computed by the internal function `step`, which also is used by the single processor instantiator of the mCRL toolset. Each outgoing transition consists of a pair $(\texttt{aterm}, [\overbrace{\texttt{aterm}, \ldots, \texttt{aterm}}^{n}])$, which is a pair existing of an action and a state vector.
- The state vector of closed terms will be fold into a pair of indexes. The action term will be put into a database. All generated indexes are globally unique.

Algorithm - INST

Definition *main*

```
ticks := ∅ ; transitions := 0
repeat
  tag = RecvTagFromMgr()
  if tag = process_tick
    repeat
      {x, y, id_x} := pop (ticks)
      l = Checksum(y)
      WriteTransition_l(x, id_x, tick, y)
      transitions := transitions + 1
      write(fork_l, {y, tick, id_x})
    until isEmpty (ticks)
  else
  if tag = continue
    repeat
      {x, id_x} := read(fork_s)
      if x_0 ≠ endlevelmarker
        if (enabled(x) = ∅)
          enabled(x) := [{deadlock, deadlock_s}]
      for (a, y) ∈ enabled(x)
        if a = tick
          append (ticks, {x, y, id_x})
        else
          l = Checksum(y)
          WriteTransition_l(x, id_x, a, y)
          transitions := transitions + 1
          write(fork_l, {y, a, id_x})
    until x_0 = endlevelmarker
    transitions := transitions + x_1
  for l ∈ [0, N)
    write(fork_l, {{endlevelmarker, 0}, undef, undef})
    WriteEndLevelMarker_l
  SendToMgr(finished, transitions)
```

6 Selecting Unexplored States

This will be done by process Fork, which is a database handler. It consists of a database and a data stream handler. The function *put* and *new* are interfaces to the database. The states are the keys and the values, wich will be stored, are references to parent states. Each reference is an identifier of the parent state database followed by its index. The stored values are needed for (shortest) trace generation.

Algorithm - Fork

Definition *main*

$Initialstate(output)$
repeat
  repeat
    $\{y, id_x\} := pop\ (output)$
    $write(inst_s, \{y, id_x\})$
  until $y_0 =$ `endlevelmarker`
  for $k \in [0, N)$
    repeat
      $\{y, a, id_x\} := read(inst_k)$
      if $y_0 \neq$ `endlevelmarker`
        if $new\ (y)$
          $id_y := put(y, \{k, id_x, a\})$
          $append\ (output, \{y, id_x\})$
    until $y_0 =$ `endlevelmarker`
  $append\ (output, \{[\texttt{endlevelmarker}, 0], undef\})$

Definition $initialstate(output)$

$x := fold(initialstate)\ ;\ l = Checksum(x)$
if $l = s$
  $id_x := put(x, \{undef, undef, undef\})$
  $append\ (output, \{x, id_x\})$
$append\ (output, \{[\texttt{endlevelmarker}, 0], undef\})$

7 Resuming after Abort

The processes must continue from the state which was left when the processes are aborted. The cause of the abort can be anything; from clean exit to power failure. The abort and resume must be invisible in the output files. Before starting the state space generating algorithm the following tasks will be done:

- Database Manager loads all consistent data elements into its intermediate and leaf databases.
- Each client truncates the dumped files of states so that they are consistent and contain the same number of level markers.
- Each client s loads all explored states into their top databases *explored* from the file: $\texttt{steppers/vdb}_s$, eventually with their belonging references to their parent transitions.
- Truncate each quadruple of transition files such that all files in each quadruple have the same number of records and that the source states in $\text{steppers}_s/\text{src}_j$ are all present in database *explored* of segment s.

After that the same algorithm as described in the previous chapter will be performed, except for that *Initialstate* is replaced by the below defined function *RestoreStates*.

Definition of function *RestoreStates*

repeat
  $transitions := 0$
  for $j \in [0, N)$
    repeat
      $\{x, [id_x, a], y\} := ReadTransition_j$
      if $x_0 \neq$ `endlevelmarker`
        if $new\ (y)$
          $id_y := put(y, \{j, id_x, a\})$
          $append\ (nextlevel, \{y, id_x\})$
        $transitions := transitions + 1$
    until $x_0 =$ `endlevelmarker` **or** `eof`
  $append\ (output, \{$`endlevelmarker`$, transitions\}, undef)$
  while $pop(nextlevel, \{y, id_x\})$
    $append\ (output, \{y, id_x\})$
until `eof`

Reading Transitions

This will be done by $ReadTransition_j$ which reads $\{x, [id_x, a], y\}$ from respectively the files: $\mathrm{steppers}_j.\mathrm{src}_s$, $\mathrm{steppers}_s.\mathrm{act}_j$, and $\mathrm{steppers}_s.\mathrm{dest}_j$.

8 Manager

The following algorithm defines the core task of the manager which can handle postponed tick transitions.

Algorithm

```
repeat
  repeat
    transitions := 0
    for k ∈ [0, N)
      SendTo(inst_k, continue)
    for k ∈ [0, N)
      tag  := RecvFinished(inst_k)
      transitions := transitions + ReadFrom(inst_k)
  until transitions = 0
  for k ∈ [0, N)
    SendTo(inst_k, process_tick)
    for k ∈ [0, N)
      tag := RecvFinished(inst_k)
      transitions := transitions + ReadFrom(inst_k)
until transitions = 0
for k ∈ [0, N)
  SendTo(inst_k, terminate)
```

Other tasks are:

- Sending command line options to the clients
- Extracting report information from the clients and displaying this
- Synchronizing the process of exchanging destination states between the clients
- Computing a trace out of the parent information located on each client
- Communicating with `contact`, the GUI running on the work station of the user.

9 Monitoring with tool `contact`

The whole state space generation system can be monitored by the tool `contact`. The way contact connects to `instantiators` depends on two cases:

- `instantiators` directly launched.
  - After invocation `contact`. It is assumed that `instantiators` is launched from the same node. `instantiators` initiates the connection with `contact`.

  - After invocation `contact` *hostname*. `contact` initiates connection with `instantiators` running on *hostname*.

- `instantiators` launched by PBS.

  - After invocation `contact`. It is assumed that `instantiators` is put in queue by the command `qsub` entered on the same node. `instantiators` initiates the connection with `contact`.
  - After invocation `contact` *jobid hostname*. `contact` initiates connection with `instantiators`, which has obtained job identifier *jobid* from PBS, starting via the command `qsub` entered on *hostname*.

A window pops up. The window is divided into four parts. From above to below:

- Global information, such as computing time and total size of the written output in files.
- Channel LEDS. Each row and each column belongs to a segment. The rows indicate the write side and the columns the read side of the channels. The colors in each cell can be red, yellow, green, or white. Red means a node at one of the ends of the channel is busy computing, yellow means there is a data flow from the segment denoted by its row to the segment denoted by its column, green means both nodes at the ends of the channel are waiting, and white otherwise.
- Computing node information, such as number of generated transitions and output of `top` running on that node.
- Database information, such as the tree structure of the database and the encountered actions.

On this moment the only possible action given by the user which influence the behaviour of `instantiators` is pushing the button `abort`. This action aborts `instantiators`.

10 Benchmarks

Here follows the computing times of a distributed state space traversal without writing the transition system

Lift 6

The state space contains 33949609 states and 165318222 transitions. Each computation node uses a AMD Athlon 64 2200 MHz processor. The size of the produced output 4.3G.

nodes	cpu time
3	$76m58s$
6	$41m19s$
12	$24m17s$
16	$16m23s$
29	$16m54s$

Cache Coherence Protocol

spec	levels	states	transitions	cpu 31 nodes	output size
ccp33	298	97451014	1061619779	$2h38m26s$	54G