State Space Exploration

Wieger Wesselink

May 22, 2019

1 Untimed state space exploration

Consider the following untimed linear process specification P, with initial state d_0.

$$P(d) = \sum_{i \in I} \sum_{e_i} c_i(d, e_i) \to a_i(f_i(d, e_i)) \cdot P(g_i(d, e_i))$$

Below we define an algorithm for state space exploration of this process specification. The algorithm only reports events using callback functions. This is done to separate the exploration from its applications. The following events are distinguished:

discover_state	is invoked when a state is encountered for the first time
examine_transition	is invoked on every transition
start_state	is invoked on a state right before its outgoing transitions are being explored
finish_state	is invoked on a state after all of its outgoing transitions have been explored

Using these events, most applications of state space exploration can be implemented efficiently. Not that this idea has been adopted from the Boost Graph Library.

Let $rewr$ be a rewriter. An algorithm for state space exploration is

```
EXPLORELPS(P(d), d_0, rewr, discover_state, examine_transition, start_state, finish_state)
todo := {rewr(d_0, [])}
discovered := ∅
while todo ≠ ∅ do
    choose s ∈ todo
    todo := todo \ {s}
    discovered := discovered ∪ {s}
    start_state(s)
    for i ∈ I do
        condition := rewr(c_i(d, e_i), [d := s])
        if condition = false then continue
        E := {e | rewr(condition, [e_i := e]) = true}
        for e ∈ E do
            a := a_i(rewr(f_i(d, e_i), [d := s, e_i := e]))
            s' := rewr(g_i(d, e_i), [d := s, e_i := e])
            if s' ∉ discovered then
                todo := todo ∪ {s'}
                discovered := discovered ∪ {s'}
                discover_state(s')
            examine_transition(s, a, s')
    finish_state(s)
```

The set E is computed using the ENUMERATE algorithm. This computation may be expensive. Hence the condition $c(d, e_i)$ is first rewritten, since if it evaluates to $false$ the computation of E can be skipped.

2 Timed state space exploration

Consider the following timed linear process specification P, with initial state d_0.

$$P(d) = \sum_{i \in I} \sum_{e_i} c_i(d, e_i) \to a_i(f_i(d, e_i)) \, ◁ t_i(d, e_i) \, \cdot P(g_i(d, e_i)).$$

Note that the time tag $t_i(d, e_i)$ is optional. If it is omitted, the corresponding action may happen at an arbitrary time. In timed state space exploration, care is taken that on every trace the time tags are increasing. In order to achieve that, a time stamp is recorded for each state in the state space. We use the notation $t \ll s$ to denote the state s with associated time stamp t.

```
ExploreLPSTimed(P(d), d_0, rewr, discover_state, examine_transition, start_state, finish_state)
todo := { 0 ≪ rewr(d_0, [ ]) }
discovered := ∅
while todo ≠ ∅ do
    choose t ≪ s ∈ todo
    todo := todo \ { t ≪ s }
    discovered := discovered ∪ { t ≪ s }
    start_state(t ≪ s)
    for i ∈ I do
        condition := rewr(c_i(d, e_i), [d := s])
        if condition = false then continue
        E := {e | rewr(condition, [e_i := e]) = true}
        for e ∈ E do
            t' := rewr(t_i(d, e_i), [d := s, e_i := e])
            if t' ≤ t then continue
            a := a_i(rewr(f_i(d, e_i), [d := s, e_i := e]))
            s' := rewr(g_i(d, e_i), [d := s, e_i := e])
            if t' ≪ s' ∉ discovered then
                todo := todo ∪ { t' ≪ s' }
                discovered := discovered ∪ { t' ≪ s' }
                discover_state(t' ≪ s')
            examine_transition(t ≪ s, a◁t', t' ≪ s')
    finish_state(t ≪ s)
```

3 Search strategies

Three different search strategies have been implemented: breadth-first, depth-first and highway. They mainly differ in the order in which the elements of the todo set are processed. In breadth-first search nodes at the present depth are explored before nodes at a higher depth. In depth-first search the highest-depth nodes are explored first. Highway search is a variant that uses a breadth-first search, but it only explores a part of the state space.

In all three cases the *todo* list is stored in a double ended queue. We use the slicing operator to denote parts of a list. For example, $A[m:n]$ corresponds to the sublist $A[m,\ldots,n-1]$.

3.1 Breadth-first search

ExploreLPSBreadthFirst($P(d), d_0, rewr$, discover_state, examine_transition, start_state, finish_state)
$todo := [rewr(d_0, [\,])]$
$discovered := \emptyset$
while $|todo| > 0$ **do**
  $s := todo[0]$
  $todo := todo[1 : |todo|]$
  $discovered := discovered \cup \{s\}$
  start_state(s)
  for $i \in I$ **do**
    $condition := rewr(c_i(d, e_i), [d := s])$
    if $condition = false$ **then continue**
    $E := \{e \mid rewr(condition, [e_i := e]) = true\}$
    for $e \in E$ **do**
      $a := a_i(rewr(f_i(d, e_i), [d := s, e_i := e]))$
      $s' := rewr(g_i(d, e_i), [d := s, e_i := e])$
      if $s' \notin discovered$ **then**
        $todo := todo +\!\!+ [s']$
        $discovered := discovered \cup \{s'\}$
        discover_state(s')
      examine_transition(s, a, s')
  finish_state(s)

3.2 Depth-first search

ExploreLPSDepthFirst($P(d), d_0, rewr$, discover_state, examine_transition, start_state, finish_state)
$todo := [rewr(d_0, [\,])]$
$discovered := \emptyset$
while $|todo| > 0$ **do**
  $s := todo[|todo| - 1]$
  $todo := todo[0 : |todo| - 1]$
  $discovered := discovered \cup \{s\}$
  start_state(s)
  for $i \in I$ **do**
    $condition := rewr(c_i(d, e_i), [d := s])$
    if $condition = false$ **then continue**
    $E := \{e \mid rewr(condition, [e_i := e]) = true\}$
    for $e \in E$ **do**
      $a := a_i(rewr(f_i(d, e_i), [d := s, e_i := e]))$
      $s' := rewr(g_i(d, e_i), [d := s, e_i := e])$
      if $s' \notin discovered$ **then**
        $todo := todo +\!\!+ [s']$
        $discovered := discovered \cup \{s'\}$
        discover_state(s')
      examine_transition(s, a, s')
  finish_state(s)

3.3 Highway search

In highway search (see [3]) a breadth first search is done, with the restriction that at most N states are put in the todo list for each level. The variable L maintains the number of states in the todo list corresponding to the current level, and the variable c counts how many elements have been added corresponding to the next level. Once c reaches the maximum value N, elements are being overwritten randomly.

```
ExploreLPSHighway(P(d), d_0, rewr, N, discover_state, examine_transition, start_state, finish_state)
todo := [rewr(d_0, [ ])]
discovered := ∅
L := |todo|
c := 0
while |todo| > 0 do
    s := todo[0]
    todo := todo[1 : |todo|]
    discovered := discovered ∪ {s}
    start_state(s)
    for i ∈ I do
        condition := rewr(c_i(d, e_i), [d := s])
        if condition = false then continue
        E := {e | rewr(condition, [e_i := e]) = true}
        for e ∈ E do
            a := a_i(rewr(f_i(d, e_i), [d := s, e_i := e]))
            s' := rewr(g_i(d, e_i), [d := s, e_i := e])
            if s' ∉ discovered then
                todo := todo ++ [s']
                discovered := discovered ∪ {s'}
                discover_state(s')
                c := c + 1
                if c ≤ N then
                    todo := todo ++ [s']
                else
                    k := random({1, ..., c})
                    if k ≤ N then
                        discovered := discovered \ {todo[|todo| − k]}
                        todo[|todo| − k] := s'
            examine_transition(s, a, s')
    finish_state(s)
    L := L − 1
    if L = 0 then
        L := |todo|
        c := 0
```

In Algorithm 1 of [3], the set Q_d stores todo elements corresponding to the current level, and the set Q_{d+1} stores todo elements corresponding to the next level. The algorithm above uses only one list *todo* that stores both of them. At each iteration of the while loop the first L elements of *todo* list belong to the current level, and the remaining elements belong to the next level. Furthermore, the algorithm above contains only one application of a random generator, compared to two applications in the original version. The element k is chosen randomly in the range $[1, \ldots, c]$. There is an N/c probability that this value is in the range $[1, \ldots, N]$.

If k is inside the range, the element in the *todo* list with index k (counting from the end) is overwritten. This behaviour matches with the original version. A subtle aspect of highway search is the states in the next level Q_{d+1} that are overwritten are not added to the set of discovered states V. In the algorithm above this is achieved by removing the overwritten states from the set *discovered*.

4 Caching

The computation of the set of solutions E in the ExploreLPS is expensive. Therefore it may be a good idea to cache these solutions. Caching can be done locally (i.e. using a separate cache for each summand), or globally. This leads to the following variants of the algorithm. We assume that FV is a function that computes free variables of an expression. Let $\mathcal{D}$ be the set of process parameters (i.e. the elements of d).

4.1 Local caching

In the local caching algorithm for each summand i a mapping C_i is maintained. The cache key is comprised of the actual values of the process parameters that appear in the condition $c_i(d, e_i)$.

ExploreLPSLocallyCached($P(d), d_0, rewr$, discover_state, examine_transition, start_state, finish_state)
$todo := \{d_0\}$
$discovered := \emptyset$
for $i \in I$ **do**
    $C_i := \{:\}$
    $\gamma_i := FV(c_i(d, e_i)) \cap \mathcal{D}$
while $todo \neq \emptyset$ **do**
    choose $s \in todo$
    $todo := todo \setminus \{s\}$
    $discovered := discovered \cup \{s\}$
    start_state(s)
    for $i \in I$**do**
        $key := \gamma_i[d := s]$
        if $key \in keys(C_i)$ **then**
            $E := C_i[key]$
        else
            $E := \{e \mid rewr(c_i(d, e_i), [d := s, e_i := e]) = true\}$
            $C_i := C_i \cup \{(key, E)\}$
        for $e \in E$ **do**
            $a := a_i(rewr(f_i(d, e_i), [d := s, e_i := e]))$
            $s' := rewr(g_i(d, e_i), [d := s, e_i := e])$
            if $s' \notin discovered$ **then**
                $todo := todo \cup \{s'\}$
                $discovered := discovered \cup \{s'\}$
                discover_state(s')
            examine_transition(s, a, s')
    finish_state(s)

4.2 Global caching

In the global caching algorithm one mapping C is maintained. To achieve this, the condition of the summands is added to the cache key. If many summands share the same condition, global caching may be beneficial.

In practice this doesn't seem to happen much.

```
ExploreLPSGloballyCached(P(d), d_0, rewr, discover_state, examine_transition, start_state, finish_state)
todo := {d_0}
discovered := ∅
C := ∅
for i ∈ I do
    γ_i := FV(c_i(d, e_i)) ∩ 𝒟
while todo ≠ ∅ do
    choose s ∈ todo
    todo := todo \ {s}
    discovered := discovered ∪ {s}
    start_state(s)
    for i ∈ I do
        key := c_i(d, e_i) ++ γ_i[d := s]
        if key ∈ keys(C) then
            T := C[key]
        else
            T := {t | rewr(c_i(d, e_i), [d := s, e_i := t]) = true}
            C := C ∪ {(key, T)}
        for e ∈ E do
            a := a_i(rewr(f_i(d, e_i), [d := s, e_i := e]))
            s' := rewr(g_i(d, e_i), [d := s, e_i := e])
            if s' ∉ discovered then
                todo := todo ∪ {s'}
                discovered := discovered ∪ {s'}
                discover_state(s')
            examine_transition(s, a, s')
    finish_state(s)
```

In this algorithm C is a mapping, with $keys(C) = \{k \mid \exists_v : (k, v) \in C\}$. We use the notation $C[k]$ to denote the unique element v such that $(k, v) \in C$.

5 Confluence Reduction

Confluence reduction (see [4], [1] and [2]) is an on-the-fly state space exploration method that produces a reduced state space. For confluence reduction we assume that the set of summands I is partitioned into a set $I_{regular}$ of 'regular' summands, and a set $I_{confluent}$ of confluent *tau*-summands. The confluent *tau*-summands are used to determine a unique representative state that is reachable via confluent τ steps. This is done using the graph algorithm FINDREPRESENTATIVE. This leads to the following variant of the algorithm:

EXPLORELTSCONFLUENCE($P(d), d_0, rewr$, discover_state, examine_transition, start_state, finish_state)
$todo := \{$ FINDREPRESENTATIVE($rewr(d_0, [\,])$) $\}$
$discovered := \emptyset$
while $todo \neq \emptyset$ **do**
  choose $s \in todo$
  $todo := todo \setminus \{s\}$
  $discovered := discovered \cup \{s\}$
  start_state(s)
  for $i \in I_{regular}$ **do**
    $condition := rewr(c_i(d, e_i), [d := s])$
    if $condition = false$ **then continue**
    $E := \{e \mid rewr(condition, [e_i := e]) = true\}$
    for $e \in E$ **do**
      $a := a_i(rewr(f_i(d, e_i), [d := s, e_i := t]))$
      $s' :=$ FINDREPRESENTATIVE($rewr(g_i(d, e_i), [d := s, e_i := t])$)
      if $s' \notin discovered$ **then**
        $todo := todo \cup \{s'\}$
        $discovered := discovered \cup \{s'\}$
        discover_state(s')
      examine_transition(s, a, s')
  finish_state(s)

As suggested in [2] Tarjan's strongly connected component (SCC) algorithm (see [5]) can be used to compute a unique representative.

5.1 Tarjan's SCC algorithm

A recursive implementation of Tarjan uses four global variables *stack*, *low*, *disc* and *result*. The function TARJANRECURSIVE computes the connected component reachable from node u. In this function it is assumed that the function call $successors(u)$ returns the successor states of u in a deterministic order.

```
stack := [ ]
low := {:}
disc := {:}
result := [ ]

TarjanRecursive(u)
k := |disc|
disc[u] := k
low[u] := k
stack := stack ++ [u]
for v ∈ successors(u) do
    if v ∉ low then
        TarjanRecursive(u)
        low[u] := min(low[u], low[v])
    else if v ∈ stack then
        low[u] := min(low[u], disc[v])
    if low[u] = disc[u] then
        component := [ ]
        while true do
            v := stack[|stack| − 1]
            stack := stack[0 : |stack| − 1]
            component := component ++ [v]
            if v == u then break
        result := result ++ [component]
```

A side effect of a call TarjanRecursive(u) is that *result* contains the connected components that have been found.

5.2 FindRepresentative

Due to properties of confluent *tau*-summands, there is always only one terminal strongly connected component, i.e. a strongly connected component without outgoing edges. Furthermore, the first strongly connected component reported by Tarjan's algorithm is always terminating. For our implementation of FindRepresentative we prefer to use an iterative version of Tarjan's SCC algorithm. The reason for this is that an iterative version can be more easily interrupted once the first SCC has been found. The algorithm description in [6] has been used as a model for our solution.

```
FindRepresentative(u)
stack := [ ]
low := {:}
disc := {:}
work := [(u, 0)]
while work ≠ [ ] do
    (u, i) := work[|work| − 1]
    work := work[0 : |work| − 1]
    if i = 0 then
        k := |disc|
        disc[u] := k
        low[u] := k
        stack := stack ++ [u]
    recurse := false
    for j ∈ [i, . . . , |successors(u)|] do
        v := successors(u)[j]
        if v ∉ disc then
            work := work ++ [(u, j + 1)]
            work := work ++ [(v, 0)]
            recurse := true
            break
        else if v ∈ stack then
            low[u] := min(low[u], disc[v])
    if recurse then continue
    if low[u] = disc[u] then
        result := u
        while true do
            v := stack[|stack| − 1]
            stack := stack[0 : |stack| − 1]
            if v == u then break
            if v < result then result := v
        return result
    if work ≠ [ ] then
        v := u
        (u, z) := work[|work| − 1]
        low[u] := min(low[u], low[v])
```

References

[1] Stefan Blom. Partial t-confluence for efficient state space generation, 2001.

[2] Stefan Blom and Jaco van de Pol. State space reduction by proving confluence. In *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 596–609. Springer, 2002.

[3] Tom A. N. Engels, Jan Friso Groote, Muck van Weerdenburg, and Tim A. C. Willemse. Search algorithms for automated validation. *J. Log. Algebr. Program.*, 78(4):274–287, 2009.

[4] Jan Friso Groote and Jaco van de Pol. State space reduction using partial tau-confluence. In *MFCS*, volume 1893 of *Lecture Notes in Computer Science*, pages 383–393. Springer, 2000.

[5] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM JOURNAL ON COMPUTING*, 1(2), 1972.

[6] Jesper Öqvist. Iterative tarjan strongly connected components in python. https://llbit.se/?p=3379. Accessed: 2019-03-26.