On-the-Fly State Space Reductions for Weak Equivalences

Radu Mateescu
INRIA Rhône-Alpes / VASY
655, avenue de l'Europe, F-38330 Montbonnot St Martin, France
Radu.Mateescu@inria.fr

ABSTRACT

On-the-fly verification of concurrent finite-state systems consists in constructing and analysing their underlying state spaces in a demand-driven way. This technique is able to detect errors effectively in large systems; however, its performance can still be increased by reducing the state spaces incrementally in a way compatible with the verification problem. In this paper, we propose algorithms for three on-the-fly reductions of Labeled Transition Systems (LTSs), which preserve weak equivalence relations: τ-compression (collapsing of strongly connected components made of τ-transitions), τ-closure (transitive reflexive closure over τ-transitions), and τ-confluence (a form of partial order reduction). Each algorithm is described as a reductor module taking as input the successor function of an LTS and returning the successor function of the reduced LTS. The three reductors were implemented within the CADP toolbox using the generic OPEN/CÆSAR environment, which makes them directly available for any on-the-fly verification tool connected to OPEN/CÆSAR and compatible with the underlying reduction. Our experiments show that these reductors can improve significantly the performance of on-the-fly LTS generation, model checking, and equivalence checking.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal methods, Model checking; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical verification

General Terms

Algorithms, Design, Verification

Keywords

bisimulation, equivalence checking, graph exploration, labeled transition system, model checking, mu-calculus, partial order reduction

1. INTRODUCTION

The verification of concurrent finite-state systems is confronted in practice with the well-known *state explosion* problem (prohibitively large size of the underlying state spaces), which occurs for systems containing many parallel processes and complex data structures. The so-called *on-the-fly* verification technique attempts to combat state explosion by constructing the state space in a demand-driven manner: this allows to detect errors even when the state space of the system is too large to fit entirely in a computer memory. However, the performance of on-the-fly verification can be further improved by reducing the state space incrementally, provided that the reduction is compatible with the verification problem. A suitable notion of compatibility relies upon the rich theory of behavioural equivalences defined in the context of Labeled Transition Systems (LTSs): a reduction that preserves an equivalence relation between LTSs also preserves the set of temporal logic properties adequate w.r.t. that relation. In practice, a reduction preserving a weak equivalence relation, such as branching [26], observational [19], $\tau^*.a$ [6], or safety [4] equivalence, is likely to be effective for those LTSs containing many invisible transitions (labeled by the action τ), i.e., an important amount of internal behaviour, which is abstracted away by a weak equivalence.

In this paper, we consider three on-the-fly reductions compatible with weak equivalence relations on LTSs. The first reduction, called τ-compression, consists in replacing each strongly connected component containing only τ-transitions (τ-SCC) by one of its states, considered as *representative* for all the other states of the τ-SCC. It preserves branching equivalence and provides significant reductions for LTSs containing cycles of τ-transitions, which may be caused either by the cyclic interconnection of parallel processes (e.g., leader election protocols on ring networks), or by abstraction (hiding of actions in the LTS). The second reduction, called τ-closure, consists in computing the transitive reflexive closure over τ-transitions. It preserves $\tau^*.a$ equivalence and can provide important reductions for LTSs containing a high percentage of τ-transitions (e.g., when only a few actions are needed to be observable during verification). The third reduction, called τ-confluence, consists in keeping only the *confluent* τ-transitions (the execution of which does not change the observable behaviour of the system) going out of a state s, all the other transitions going out of s being deleted [12]. This form of partial order reduction preserves branching equivalence and can lead to significant reductions for LTSs containing loosely-coupled parallel processes.

For each of these reductions, we propose an algorithm working on-the-fly, i.e., using a forward traversal of the LTS to compute the successors of a state modulo the corresponding reduction. Each algorithm was implemented as a separate reductor module within the CADP toolbox [9] using the generic OPEN/CÆSAR environment [8] for on-the-fly manipulation of LTSs. Therefore, each reductor module is language-independent (it can be applied to every language equipped with a compiler generating LTSs compliant with the OPEN/CÆSAR interface) and application-independent (it can be inserted in front of every on-the-fly verification tool compatible with the corresponding reduction). We applied these reductors in conjunction with three on-the-fly verification tools of CADP: the LTS builder GENERATOR, the model checker EVALUATOR [18], and the equivalence checker BISIMULATOR [17, 2]. The experiments carried out on various LTSs taken from the CADP distribution have shown that the reductors can enhance the capabilities of these verification tools significantly.

Related work

The reduction of state spaces with the goal of improving the performance of verification has been investigated in various contexts. LTS reductions driven by the properties being checked were proposed in [1], modulo specific equivalence relations derived from the syntactic structure of properties; to facilitate the extraction of relevant information, properties are formulated in a *selective* variant of the modal μ-calculus, which makes explicit the set of LTS actions useful for the verification of a property. The reductions we consider in this paper are somewhat orthogonal to those proposed in [1]: they preserve standard equivalence relations between LTSs, and exploit the adequacy of certain classes of temporal logic properties w.r.t. these equivalences (e.g., the adequacy of observational μ-calculus with observational equivalence [22]). In this respect, our reductions are similar in spirit with the partial order reductions used in [16], which are compatible with observational equivalence and with a class of action-based, linear-time properties.

The on-the-fly detection of τ-SCCs using Tarjan's algorithm [23] was successfully used for verification [17] and test generation [14]. The algorithm we propose here for on-the-fly τ-compression clearly separates the detection and collapsing of τ-SCCs from the analysis tools further applied to the LTS. The computation of transitive reflexive closure in directed graphs is a long standing problem (see [13] for a survey on algorithms based upon graph traversals). As regards LTSs, the transitive reflexive closure over τ-transitions (τ-closure) was mostly used for equivalence checking [6] and test generation [14]. Existing algorithms for on-the-fly τ-closure usually make a time/space tradeoff, by choosing (as extreme cases) either to store all the information necessary for retrieving in constant time the visible successors of a state reached after τ-closure once the state was explored [15], or to recompute this information every time a state is encountered (e.g., the REDUCTOR tool of CADP). Our on-the-fly τ-closure algorithm makes a compromise between the amount of information being stored and that being recomputed, and identifies certain equivalent states (with the same visible transitions reached after τ-closure) in order to further reduce the LTS.

Partial order reductions of state spaces have been thoroughly studied in the literature (see [10] for a survey) and gave rise to efficient implementations [24, 21, 16]. However, most of the partial order reduction techniques rely upon the structure of the states and/or the interconnection topology between processes in order to detect particular kinds of LTS transitions (independent, inert, etc.) [21]; therefore, the corresponding reduction algorithms, although applicable to quite general classes of concurrent systems, remain language-dependent. Since we aim at constructing reductor modules which are language-independent, we adopt here an approach based upon τ-confluence, a form of partial order reduction preserving branching equivalence between LTSs [12]. The detection of τ-confluent transitions can be done by examining only the transitions of the LTS, and does not require any knowledge about the internal structure of states. The reduction proposed in [12], called τ-prioritisation, consists in keeping, for each state, only one of its outgoing τ-confluent transitions (if any), all the other transitions, which can still be executed after the τ-confluent one, being ignored. The resulting LTS can be further reduced by compressing the sequences of τ-confluent transitions produced by τ-prioritisation [12]. The first algorithms for on-the-fly τ-confluence reduction [3, 20] implemented τ-prioritisation by computing SCCs of τ-confluent transitions and by locally solving a boolean equation system, respectively. The on-the-fly τ-closure algorithm we propose here improves over [20] by combining τ-prioritisation and compression of τ-confluent transition sequences.

Paper outline

Section 2 describes in detail the algorithms proposed for the τ-compression, τ-closure, and τ-confluence reductors. Section 3 illustrates experimentally the usefulness of these reductors for increasing the performance of LTS generation, model checking, and equivalence checking. Section 4 concludes and indicates directions for future work.

2. ON-THE-FLY REDUCTIONS

The three reductions that we consider are defined on Labeled Transition Systems (LTSs), which are natural models for action-based specification languages, such as process algebras. An LTS is a tuple $M = (S, A, T, s_0)$, where S is the set of states, A is the set of actions (containing also the invisible action τ), $T \subseteq S \times A \times S$ is the transition relation, and $s_0 \in S$ is the initial state. A transition $(s_1, a, s_2) \in T$ (also written $s_1 \xrightarrow{a} s_2$) indicates that the system can move from state s_1 to state s_2 by performing action a. All states in S are reachable from the initial state s_0 via sequences of transitions in T. In the sequel, we consider LTSs represented *implicitly*, i.e., by their successor function, which gives for each state $s \in S$ the set of transitions $s \xrightarrow{a} s' \in T$ going out of s. This representation allows LTSs to be constructed incrementally starting at their initial state, and therefore is suitable for developing on-the-fly verification algorithms.

2.1 Tau-compression

The TAUCOMPRESSION algorithm (see Figure 1) that we propose for implementing the τ-compression reductor takes as input a state s of an LTS $M = (S, A, T, s_0)$ represented implicitly and produces as output the set of transitions (represented as couples (a, s')) going out of s after collapsing the τ-SCCs immediately reachable from s. This reductor, which preserves branching equivalence, uses a depth-first search (DFS) along the τ-transitions of T, performed recursively starting at s. A detection of τ-SCCs is performed during the DFS traversal, following Tarjan's algorithm [23]. For each τ-SCC detected, its *root* state (the state that was visited first among all states of the τ-SCC) is considered as *representative* for all the states of the τ-SCC. To each state u belonging to a τ-SCC is attached a pointer $rep(u)$ towards the representative w of the τ-SCC. All transitions going out from states of the τ-SCC and leading to states of other τ-SCCs are stored in a set $trans(w)$.

```
1.  V := ∅; stack := nil; c := 0;
2.  function TauCompression (s : S) : 2^(A×S) is
3.     if s ∈ V then
4.        return trans(rep(s))
5.     else
6.        n(s) := c; c := c + 1; low(s) := n(s);
7.        V := V ∪ {s}; stack := push(s, stack);
8.        r := ∅;
9.        forall (s, a, s′) ∈ T do
10.          if a ≠ τ then
11.             r := r ∪ {(a, s′)}
12.          else
13.             if s′ ∈ V then
14.                if s′ ∈ stack then
15.                   low(s) := min(low(s), n(s′))
16.                else
17.                   r := r ∪ {(a, rep(s′))}
18.                endif
19.             else
20.                r′ := TauCompression (s′);
21.                if s′ ∈ stack then
22.                   r := r ∪ r′
23.                else
24.                   r := r ∪ {(a, rep(s′))}
25.                endif;
26.                low(s) := min(low(s), low(s′))
27.             endif
28.          endif
29.       end;
30.       if low(s) = n(s) then
31.          while top(stack) ≠ s do
32.             rep(top(stack)) := s;
33.             stack := pop(stack)
34.          end;
35.          rep(s) := s;
36.          trans(s) := r
37.       endif;
38.       return r
39.    endif
40. end
```

Figure 1: Algorithm for on-the-fly τ-compression

TAUCOMPRESSION proceeds as follows. The set $V \subseteq S$ of the visited states and the *stack* used for storing the states contained in τ-SCCs are initially empty, and the counter c used for numbering the states according to the order in which they are visited by the DFS is initially 0 (line 1). If the state s has been already visited, then the set of transitions associated to its representative state is returned as result (lines 3–4); otherwise, the DFS traversal of τ-transitions is continued starting at s (lines 5–39). The DFS number $n(s)$ is the current value of the counter c, which is incremented; the *lowlink* number $low(s)$, equal to the smallest DFS number among those associated to the states belonging to the currently explored τ-SCC [23] is initialized to $n(s)$; the state s is inserted in the set V and also at the top of the τ-SCC *stack*; and the set of transitions r that accumulates the result is initialized to empty (lines 6–8).

Then, each transition $s \xrightarrow{a} s' \in T$ is traversed: if the transition is visible, then it is added to the set r (lines 10–11); otherwise, the successor state s' is examined. If s' has been already visited, two situations are possible: either it belongs to the τ-SCC of s (i.e., it is present on *stack*), in which case the number $low(s)$ is updated according to the value of $n(s')$; or it belongs to another already explored τ-SCC, in which case the pair $(a, rep(s'))$ is added to r, since the corresponding transition leads to (the representative state of) another τ-SCC (lines 13–18). If s' is a newly encountered state, then it is explored by a recursive call to TAUCOMPRESSION. Afterwards, if s' belongs to the τ-SCC of s, then the set r' of its outgoing transitions is added to r, since these transitions will be associated later to the root of the τ-SCC containing s and s'; otherwise, the pair $(a, rep(s'))$ is added to r (lines 20–25). In both cases, the number $low(s)$ is updated according to the value of $low(s')$ to take into account the other states identified as belonging to the τ-SCC of s during the exploration of s' (line 26).

After the exploration of the transitions going out of s is finished, if s is identified as the root of a τ-SCC, i.e., if its lowlink number is equal to its DFS number, then all the states of the current τ-SCC (which are placed above s on *stack*) are scanned and their representative is set to s (lines 30–34). Additionally, the set r of the transitions going out of all states of the current τ-SCC is associated to s (line 36). Finally, the set r is returned as result (line 38).

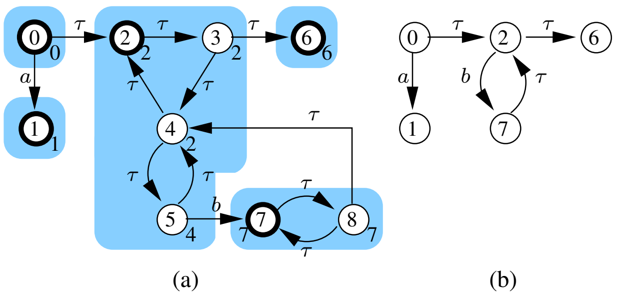


Figure 2: (a) An LTS and its τ-SCCs (grey boxes). States are identified by their DFS numbers and have attached their lowlink numbers. Roots of τ-SCCs are drawn as thick circles. (b) Reduced LTS obtained after calling TAUCOMPRESSION on states s_0 and s_7.

Figure 2 shows the result of executing TAUCOMPRESSION successively on two states of an LTS. Notice that the τ-transition $s_8 \xrightarrow{\tau} s_4$, which relates two states of different τ-SCCs, is replaced (when s_7 is popped from the τ-SCC *stack*) by the transition $s_7 \xrightarrow{\tau} s_2$ (stored in r as the pair $(\tau, rep(s_4))$ when s_8 was explored), which relates the roots of those τ-SCCs.

Each call TAUCOMPRESSION (s) has a time and space complexity linear in the size of the subgraph of τ-transitions reachable from s, because it performs a single DFS traversal of this subgraph. Moreover, since each visited state u is stored in V together with its representative $rep(u)$, a sequence of calls to TAUCOMPRESSION — for instance, a call on each state of the LTS — will have a cumulated complexity linear in the LTS size (each transition will be traversed only once). In practice, the insertion of a τ-compression reductor in front of an on-the-fly verification tool can significantly increase performance (see Sections 3.1 and 3.2).

2.2 Tau-closure

The implementation of the τ-closure reductor requires to compute the transitive reflexive closure over the τ-transitions of the LTS, which preserves $\tau^*.a$ equivalence. In order to simplify the underlying algorithm, we consider only LTSs without τ-cycles, which can be obtained by applying the τ-compression reductor described in Section 2.1.

The TAUCLOSURE algorithm (see Figures 3 and 4) that we propose for implementing the τ-closure reductor takes as input a state s of an LTS $M = (S, A, T, s_0)$ represented implicitly and produces as output the set of visible transitions (represented as couples (a, s')) reached from s after collapsing the sequences of τ-transitions going out of s. It uses two nested DFS traversals along the τ-transitions of T, performed recursively starting at s. The first DFS (line 2), implemented by the EXPLORE procedure, traverses a DFS *tree* [23] rooted at s, whose states (called *descendants* of s) are reachable from s via sequences of τ-transitions. The set of descendants of s having at least one visible outgoing transition is called the *frontier* of the DFS tree rooted at s. The DFS trees explored by successive invocations of TAUCLOSURE form a DFS *forest*, in which the DFS tree rooted at a state s will possibly be linked (via so-called *cross* τ-transitions [23]) to other DFS trees explored by previous invocations. The frontier of the DFS forest is the union of the frontiers of all DFS trees contained in the forest. Intuitively, TAUCLOSURE (s) computes all visible transitions originating from states in the portion of the DFS forest frontier reachable from s.

To speed up the τ-closure computation for every state u descendant of s (which may be required by later calls of the reductor), the call EXPLORE (s) computes three fields attached to u: a pointer $next(u)$ to the state following u on the frontier of the DFS tree rooted at s; a pointer $last(u)$ to the last state of the frontier portion reachable from u (i.e., the last state visited in the DFS subtree rooted at u); and a set $cross(u)$ containing the target states of all cross τ-transitions [23] originating from descendants of u and leading to other subtrees of the DFS forest. Additionally, $n(u)$ stores the DFS number associated to u and a boolean $has\_vt(u)$ indicates whether u has outgoing visible transitions. EXPLORE proceeds as follows. If s is a newly encountered state, after inserting it into the set V of visited states and initializing the various fields attached to it (lines 8–14), each transition $s \xrightarrow{a} s' \in T$ is traversed: if it is visible, the boolean $has\_vt(s)$ is updated accordingly; otherwise, the successor state s' is examined. If s' has been already visited and belongs to another DFS subtree, it is stored in the set $cross(s)$ (lines 19–22); otherwise, it is explored by a recursive call to EXPLORE. Afterwards, the frontier portion reachable from s is updated. If s' has some visible outgoing transitions, then it becomes the next state following the frontier portion already computed for s (line 26); otherwise, if the frontier portion reachable from s' is not empty, it is appended to the frontier portion of s (lines 28 and 30–32). Then, the set $cross(s)$ is updated with the target states (not contained in the DFS tree rooted at s) of the cross τ-transitions going out of the descendants of s' (line 33).

```
1.  function TauClosure (s : S) : 2^(A×S) is
2.     Explore (s);
3.     return VisibleSucc (s)
4.  end
5.
6.  V := ∅; c := 0;
7.  procedure Explore (s : S) is
8.     if s ∉ V then
9.        V := V ∪ {s};
10.       n(s) := c; c := c + 1;
11.       next(s) := nil;
12.       last(s) := s;
13.       cross(s) := ∅;
14.       has_vt(s) := false;
15.       forall (s, a, s') ∈ T do
16.          if a ≠ τ then
17.             has_vt(s) := true
18.          else
19.             if s' ∈ V then
20.                if n(s') < n(s) then
21.                   cross(s) := cross(s) ∪ {s'}
22.                endif
23.             else
24.                Explore (s');
25.                if has_vt(s') then
26.                   next(last(s)) := s'
27.                elsif next(s') ≠ nil then
28.                   next(last(s)) := next(s')
29.                endif;
30.                if has_vt(s') ∨ last(s') ≠ s' then
31.                   last(s) := last(s')
32.                endif;
33.                cross(s) := cross(s) ∪
                     {u ∈ cross(s') | n(u) < n(s)}
34.             endif
35.          endif
36.       end
37.    endif
38. end
```

Figure 3: Algorithm for on-the-fly τ-closure (I)

After the first DFS traversal is finished, the list of visible transitions reached after τ-closure is computed by a call to VISIBLESUCC (line 3). This function starts by invoking the NORMALIZE procedure (line 40), which performs the second DFS traversal of the LTS along the cross τ-transitions previously computed by the first DFS traversal. NORMALIZE (s) updates the set $cross(s)$ such that it contains all the states reachable from s by following cross τ-transitions and which dominate nonempty portions of frontier; the concatenation of these portions yields the frontier of the DFS forest reachable from s. NORMALIZE maintains a global set R, initially set to empty (line 65), containing the states visited during the second Dfs, and proceeds as follows. If s is a newly encountered state, it is inserted into R and the set *new_cross* is initialized to empty (lines 67–69). Then, each state s' in $cross(s)$ is normalized and its resulting set of reachable states dominating nonempty portions of frontier is accumulated in *new_cross* (lines 70–73). Finally, the set *new_cross* is stored into $cross(s)$ and, if s dominates a nonempty portion of frontier, it is added itself to $cross(s)$ (lines 74–77).

```
39. function VisibleSucc (s : S) : 2^(A×S) is
40.     Normalize (s);
41.     r := ∅;
42.     forall s' ∈ cross(s) do
43.         r := r ∪ VisibleTree (s')
44.     end;
45.     return r
46. end
47.
48. function VisibleTree (s : S) : 2^(A×S) is
49.     r := ∅;
50.     u := s;
51.     finish := false;
52.     repeat
53.         if has_vt(u) then
54.             r := r ∪ {(a, u') | (u, a, u') ∈ T ∧ a ≠ τ}
55.         endif;
56.         if u ≠ last(s) then
57.             u := next(u)
58.         else
59.             finish := true
60.         endif
61.     until finish;
62.     return r
63. end
64.
65. R := ∅;
66. procedure Normalize (s : S) is
67.     if s ∉ R then
67.         R := R ∪ {s};
69.         new_cross := ∅;
70.         forall s' ∈ cross(s) do
71.             Normalize (s');
72.             new_cross := new_cross ∪ cross(s')
73.         end;
74.         cross(s) := new_cross;
75.         if has_vt(s) ∨ last(s) ≠ s then
76.             cross(s) := cross(s) ∪ {s}
77.         endif
78.     endif
79. end
```

Figure 4: Algorithm for on-the-fly τ-closure (II)

After normalisation of s, VisibleSucc computes the set of visible transitions reachable from s after τ-closure by accumulating, for each state s' contained in $cross(s)$, the set of its visible transitions originating from the states in the frontier portion reachable from s' (lines 41–45). These sets are computed by calls to VisibleTree (s'), which scans the portion of frontier reachable from s', delimited by $next(s')$ and $last(s')$, and accumulates all visible transitions going out from states of that portion (lines 52–61).

Figure 5 shows the result of executing TauClosure successively on two states of an Lts. The sets $cross(s_3)$ and $cross(s_6)$ are obtained after normalisation: s_6 has a cross τ-transition to s_3, which in turn has a cross τ-transition to s_1, and both s_1 and s_3 dominate non empty portions of the Dfs forest frontier (they have outgoing visible transitions).

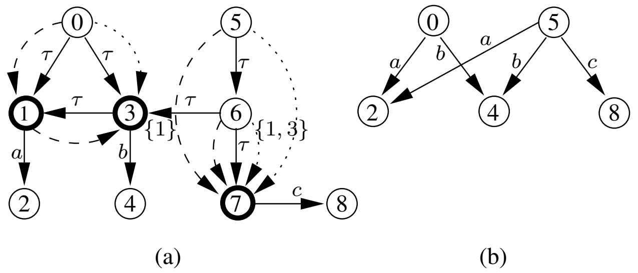


Figure 5: (a) An Lts with the fields *next* (dashed arrows), *last* (dotted arrows), and *cross* (marked when not empty). States in the frontier of the Dfs forest are drawn as thick circles. (b) Reduced Lts obtained after calling TauClosure on states s_0 and s_5.

The first call TauClosure (s) has a time and space complexity linear in the size of the subgraph of τ-transitions reachable from s, since only the first Dfs traversal of this subgraph is performed (the set $cross(s)$ is empty because there are no other trees in the Dfs forest). Each subsequent call TauClosure (u) such that the Dfs tree rooted at u has outgoing cross τ-transitions leading to other trees of the Dfs forest will trigger the normalisation of u and the computation of the set $cross(u)$; the cumulated size of these sets after a sequence of calls to TauClosure may be quadratic in the size of the Lts, since the same state can be present in several *cross* sets. It is worth noticing that, for an Lts in which all subgraphs of τ-transitions are trees, the cumulated complexity of a sequence of calls to TauClosure is linear, because the frontier information computed by the Explore procedure allows to retrieve in constant time the frontier of the Dfs subtree rooted at a given state. From this point of view, the TauClosure algorithm has a natural behaviour, its time and space complexity increasing (from linear towards quadratic) with the number of cross τ-transitions contained in the Lts.

In practice, a time/space tradeoff can be achieved by storing the *cross* sets produced by normalisation only for a subset of the Lts states, and recomputing the *cross* sets for the other states as needed. Another way to improve the performance of the τ-confluence reductor is to group the states having the same *cross* sets into equivalence classes, since they have the same set of visible transitions reachable after τ-closure. This scheme further reduces the implicit Lts produced as output by TauClosure, and can lead to good performance in practice (see Sections 3.1 and 3.2).

2.3 Tau-confluence

The form of partial-order reduction called τ-confluence was defined in [12], and shown to preserve branching equivalence between Ltss. It consists in identifying the τ-

transitions which are *confluent*, i.e., whose execution does not alter the observable behaviour of the system. Once a τ-confluent transition going out of a state was detected, one can safely delete all the other transitions going out of that state, without losing branching equivalence; this reduction is called τ-prioritisation [12].

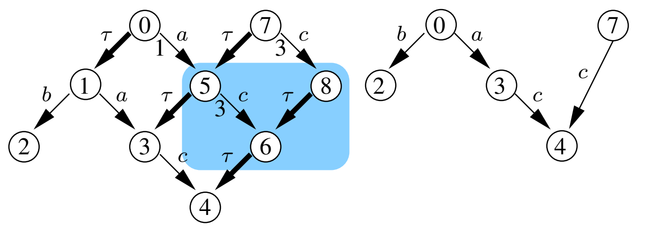
```
1.  V := ∅;
2.  function TauConfluence (s : S) : 2^(A×S) is
3.     if s ∉ V then
4.        V := V ∪ {s};
5.        seq := s; crt_s := s;
6.        end_of_seq := false;
7.        repeat
8.           if TauPrio (crt_s) = (τ, s′) then
9.              seq := seq . s′;
10.             crt_s := s′;
11.             if crt_s ∉ V then
12.                V := V ∪ {crt_s}
13.             else
14.                end_of_seq := true
15.             endif
16.          else
17.             rep(crt_s) := crt_s;
18.             end_of_seq := true
19.          endif
20.       until end_of_seq;
21.       l := seq;
22.       while l ≠ nil do
23.          s′ := head(l);
24.          rep(s′) := rep(crt_s);
25.          l := tail(l)
26.       end
27.    endif;
28.    return {(a, s′) | rep(s) -a-> s′ ∈ T}
29. end
30.
31. function TauPrio (s : S) : 2^(A×S) is
32.    found := false;
33.    forall (s, a, s′) ∈ T do
34.       if a = τ ∧ TauConfluent(s, a, s′) then
35.          found := true;
36.          break
37.       endif
38.    end;
39.    if found then
40.       return (a, s′)
41.    else
42.       return {(a, s′) | s -a-> s′ ∈ T}
43.    endif
44. end
```

Figure 6: Algorithm for on-the-fly τ-confluence

The detection of τ-confluent transitions can be encoded as a Boolean Equation System (Bes) [20] and performed during an on-the-fly construction of the Lts by solving the Bes locally. As pointed out in [12], the reduction by τ-prioritisation can be further improved by compressing the sequences of τ-confluent transitions, using the fact that the source and target states of such a transition are branching equivalent. The τ-confluence reductor described below explores an Lts on-the-fly and applies τ-prioritisation and compression of τ-sequences. In the sequel we consider only Ltss without τ-cycles, since this is required by the theory of τ-confluence [12]; these Ltss can be obtained by applying the τ-compression reductor described in Section 2.1.

The TauConfluence algorithm (see Figure 6) that we propose for implementing the τ-confluence reductor takes as input a state s of an Lts $M = (S, A, T, s_0)$ represented implicitly and produces as output the set of transitions (represented as couples (a, s')) reachable from s after collapsing the sequences of τ-confluent transitions going out of s. It uses a forward traversal of the Lts along the τ-transitions of T, performed iteratively starting at s. To each visited state u is attached a pointer $rep(u)$ towards the representative of u, which is the last state reachable from u via a sequence of τ-confluent transitions. TauConfluence proceeds as follows. If s is a newly encountered state, then it is added to the set V of visited states (initially empty), and the exploration of a (potential) sequence of τ-confluent transitions going out of s is initialized (lines 3–6). The states of this sequence are stored in *seq* and the last state of the sequence is stored in *crt_s*. At each step, the transitions going out of *crt_s* after τ-prioritisation are computed by calls to TauPrio (*crt_s*) (line 8). If this function returns a single (confluent) τ-transition, this transition is appended to *seq* (lines 9–10); the sequence terminates either when its last state was already explored (line 14), or the state *crt_s* does not have any τ-confluent outgoing transition (lines 17–18). Afterwards, the sequence is scanned and the representative of its last state becomes the representative of all states of the sequence (lines 21–26).

The call TauPrio (s) examines each transition $s \xrightarrow{a} s' \in T$ and uses the predicate TauConfluent (implemented by a local resolution of a Bes [20]) to detect whether it is τ-confluent (lines 32–38): if this is the case, then (a, s') is returned as result, all the remaining transitions being ignored by τ-prioritisation; otherwise, the set of all transitions is returned (lines 39–43).



(a) (b)

Figure 7: (a) An Lts with τ-confluent transitions (thick arrows) and state representatives computed by compressing τ-sequences. The grey box shows states deleted by τ-prioritisation. (b) Reduced Lts obtained after calling TauConfluence on states s_0 and s_7.

Figure 7 shows the result of executing TauConfluence successively on two states of an Lts. The compression of τ-sequences takes place only for $s_0 \xrightarrow{\tau} s_1$ and $s_7 \xrightarrow{\tau} s_5 \xrightarrow{\tau} s_3$, which are explored by TauConfluence; the other τ-

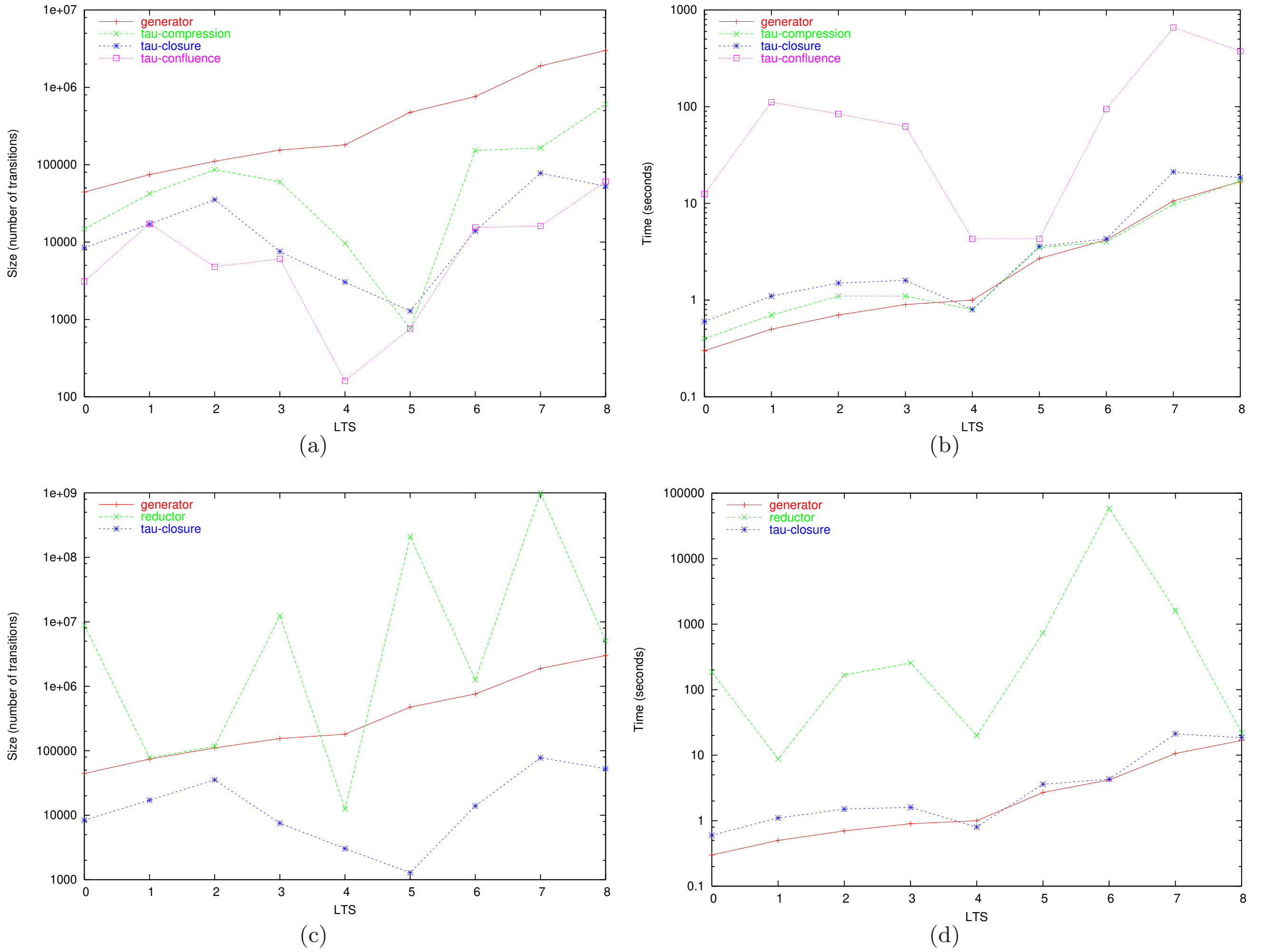


Figure 8: (a) LTS reduction and (b) execution time of GENERATOR with reductions w.r.t. GENERATOR alone. (c) LTS reduction and (d) execution time of GENERATOR with τ-closure w.r.t. REDUCTOR.

confluent transitions $s_8 \xrightarrow{\tau} s_6$ and $s_6 \xrightarrow{\tau} s_4$ are explored during the resolution of the BES when the confluence of $s_7 \xrightarrow{\tau} s_5$ and $s_5 \xrightarrow{\tau} s_3$ is detected.

A sequence of calls to TAUCONFLUENCE has a cumulated time and space complexity linear in the size of the LTS, each τ-transition being traversed only once. However, the on-the-fly detection of τ-confluent transitions induces an additional cost, which is locally quadratic [20], because each τ-transition going out of a state s must be checked to be confluent w.r.t. all other transitions going out of s. In practice, this cost is sometimes compensated by compressing the sequences of τ-confluent transitions, and can improve performance for LTSs containing a sufficient degree of concurrency (see Sections 3.1 and 3.3).

3. IMPLEMENTATION AND MEASURES

The three on-the-fly reduction algorithms described in Section 2 were implemented as separate modules (8, 300 lines of C code in total) within CADP [9] using the generic OPEN/CÆSAR environment [8], which provides versatile primitives for developing LTS exploration algorithms (lists of transitions, stacks, hash tables, etc.). As a consequence, each reductor module can be inserted as an "accelerator" in front of any on-the-fly verification tool developed using OPEN/CÆSAR, which will transparently use the successor function produced as output by the reductor instead of the successor function provided by OPEN/CÆSAR (representing the original LTS of the system being analysed). In the sequel, we illustrate the benefits of applying the reductor modules as accelerators for three on-the-fly verification tools of CADP, dedicated to LTS generation, model checking, and equivalence checking. All experiments were performed on LTSs corresponding to various examples taken from the CADP distribution (communication protocols, hardware devices, etc.), using a Linux PC with a Pentium III at 730 MHz and 1 GB of memory. LTS sizes range from 18 Kstates and 44 Ktransitions to 935 Kstates and 3 Mtransitions.

3.1 State space generation

We consider first the GENERATOR and REDUCTOR tools of CADP, which convert LTSs from the implicit OPEN/CÆSAR representation (successor function implemented in C) to the explicit BCG (*Binary Coded Graphs*) representation (list of transitions encoded in a binary file). GENERATOR does not modify the transition relation of the LTS, whereas REDUCTOR performs an on-the-fly reduction by τ-closure, using an algorithm which recomputes the τ-closure every time a state is encountered. We use our three reductor modules in conjunction with GENERATOR (note that GENERATOR with τ-closure achieves the same functionality as REDUCTOR) and compare their behaviour.

Figure 8(a) shows the reductions in LTS size (number of transitions) obtained by each of the three GENERATORs with reduction w.r.t. GENERATOR alone. The highest reductions (up to three orders of magnitude on some LTSs) are achieved by GENERATOR with τ-confluence, but at the price of an increased execution time (up to two orders of magnitude). The lowest reductions (which can still reach one or two orders of magnitude), but also the fastest ones, are achieved by GENERATOR with τ-compression. Note that, depending upon the structure of the transition relation, the τ-closure may increase the number of transitions in the LTS; however, if the LTS contains τ-transitions, the number of states is always decreased. Figure 8(b) indicates the execution times for the same set of examples. As expected, all reductors are more time consuming than GENERATOR alone; however, in many cases the loss in speed is compensated by the reduction achieved, making the reductors useful as accelerators for on-the-fly verification tools.

Figures 8(c) and 8(d) compare the reductions and execution times achieved by GENERATOR with τ-closure and by REDUCTOR. On all examples, we observe that the former tool reduces LTSs much stronger than the latter; this is due to the identification of $\tau^*.a$ equivalent states by our τ-closure algorithm. Note that the topmost point (LTS number 7) on Figures 8(b) and 8(d) denotes a disk space shortage that occurred when executing REDUCTOR, due to a prohibitive number of transitions in the resulting LTS. Moreover, GENERATOR with τ-closure is much faster (up to two orders of magnitude) than the algorithm used by REDUCTOR.

3.2 Model checking

We focus now on the on-the-fly model checker EVALUATOR [18], which verifies LTSs against temporal properties written in regular alternation-free μ-calculus. To study the effect of our reductors on EVALUATOR's performance, we consider the two properties below (a, b, c are visible LTS actions).

P_1 : $[\ true^* \ . \ a \ . \ (not\ b)^* \ . \ c\ ]\ false$

P_2 : $[\ true^* \ . \ a\ ]\ \langle\ true^* \ . \ b\ \rangle\ true$

P_1 is a safety property similar to mutual exclusion (after an a action occurs, c cannot happen before b) and is compatible with $\tau^*.a$ equivalence, therefore being preserved by the τ-closure reductor. P_2 is a liveness property expressing the response to a stimulus (every a action is potentially followed by b) and is compatible with branching equivalence, therefore being preserved by the τ-compression reductor. In order to observe the effect of reductors on the worst-case executions of EVALUATOR, the actions a, b, c have been chosen such that P_1 and P_2 are satisfied by each LTS considered, which forces the model checker to explore the entire LTS. Moreover, to make reductions more substantial, all actions other than a, b, c were hidden (i.e., renamed into τ) during LTS exploration.

Figure 9 shows the performance gains brought by τ-closure (resp. τ-compression) to EVALUATOR for checking property P_1 (resp. P_2). We observe reductions of memory consumption and execution time up to a factor 3. It is worth noticing that, on the examples considered, the reductor by τ-confluence (which is more time consuming than the other two), does not bring any improvement to EVALUATOR, whose model checking algorithm, based on local BES resolution [17], has a linear time complexity in the LTS size.

3.3 Equivalence checking

Finally, we examine the effect of our reductors in conjunction with the on-the-fly equivalence checker BISIMULATOR [17, 2]. In fact, the τ-compression and τ-closure reductors are already used by BISIMULATOR for encoding weak equivalences in terms of boolean equation systems [17]; therefore, we study here only the effect of adding the τ-confluence reductor. For $\tau^*.a$ and safety equivalences, which are based only upon τ-closure, the corresponding reductor proves to be sufficiently fast to prevent τ-confluence of bringing any improvement; however, the situation is different for observational and branching equivalences, which require more complex computations.

Figures 10(a) and (b) show the performance impact of extending BISIMULATOR with τ-confluence for checking observational equivalence. We observe strong reductions of memory consumption (up to two orders of magnitude), which are sometimes obtained at the price of an increase of execution time (up to one order of magnitude).

4. CONCLUSION AND FUTURE WORK

Tools for on-the-fly verification are complex software artifacts, the development, testing, and optimization of which are costly activities. The architecture of such tools should be as modular as possible in order to reduce these costs by enhancing reusability and reliability. The three reductor modules we presented (τ-compression, τ-closure, and τ-confluence) aim at improving the performance of on-the-fly verification tools, by reducing the size of an LTS incrementally during verification. Building these reductor modules using the generic OPEN/CÆSAR environment [8] of CADP [9] made them immediately available as "accelerators" for every on-the-fly verification tool developed on top of OPEN/CÆSAR. The experiments carried out so far have shown the benefits these reductors can bring to LTS generation, model checking, and equivalence checking.

Reduction	Temporal logic properties	Equivalence relation
τ-compression τ-confluence	ACTL $\setminus$ X obs. μ-calculus	branching, obs., $\tau^*.a$, safety
τ-closure	PDL (safety prop. of the form $[R]$ *false*)	$\tau^*.a$, safety

Table 1: Adequacy of reductions with temporal logics and equivalence relations

Table 1 summarizes the compatibility of the three reductions considered with various classes of temporal logic properties and weak equivalence relations. τ-compression and τ-confluence are compatible with branching and observational equivalences, and therefore preserve properties expressed in ACTL $\setminus$ X [5] and in observational μ-calculus [22], which are adequate w.r.t. these two relations, respectively. τ-closure is compatible with $\tau^*.a$ and safety equivalences, and thus preserves safety properties expressed as PDL [7] formulas of the form $[R]$ *false*, where the regular expression R may not contain explicit occurrences of the τ action except in τ^* subexpressions.

We plan to continue this work along several directions. Firstly, we will continue experimenting our reductor modules using larger LTSs, such as those provided by the VLTS benchmark suite [25], which contains realistic examples of state spaces for the assessment of verification and graph manipulation tools. Secondly, reductor modules implementing other reductions on LTSs (e.g., τ-inertness [11], weak τ-confluence [12], etc.) can be developed and experimented. Thirdly, it would be interesting to study the effect of using these reductors in conjunction with other tools for on-the-fly verification present in CADP (e.g., the EXHIBITOR tool for searching regular sequences in LTSs, the OCIS interactive simulator, etc.).

5. ACKNOWLEDGMENTS

We are grateful to Alban Catry for implementing the first version of the τ-compression reductor.

6. REFERENCES

[1] R. Barbuti, N. de Francesco, A. Santone, and G. Vaglini. Selective mu-calculus and formula-based equivalence of transition systems. *Journal of Computer and System Sciences*, 59(3):537–556, 1999.

[2] D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu. Bisimulator: a modular tool for on-the-fly equivalence checking. In N. Halbwachs and L. Zuck, editors, *Proc. of TACAS'2005*, LNCS vol. 3440, pp. 581–585. Springer Verlag, April 2005.

[3] S. Blom and J. van de Pol. State space reduction by proving confluence. In E. Brinksma and K. G. Larsen, editors, *Proc. of CAV'02*, LNCS vol. 2404, pp. 596–609. Springer Verlag, July 2002.

[4] A. Bouajjani, J.-C. Fernandez, S. Graf, C. Rodríguez, and J. Sifakis. Safety for branching time semantics. In *Proc. of ICALP'91*, LNCS vol. 510, pp. 76–92. Springer Verlag, 1991.

[5] R. De Nicola and F. W. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes*, LNCS vol. 469, pp. 407–419. Springer Verlag, April 1990.

[6] J.-C. Fernandez and L. Mounier. "On the Fly" verification of behavioural equivalences and preorders. In K. G. Larsen and A. Skou, editors, *Proc. of CAV'91 (Aalborg, Denmark)*, LNCS vol. 575. Springer Verlag, 1991.

[7] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences* 18(2):194–211, 1979.

[8] H. Garavel. Open/Cæsar: an open software architecture for verification, simulation, and testing. In B. Steffen, editor, *Proc. of TACAS'98*, LNCS vol. 1384, pp. 68–84. Springer Verlag, March 1998.

[9] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4, 2002.

[10] P. Godefroid. *Partial-order methods for the verification of concurrent systems – an approach to the state-explosion problem*, LNCS vol. 1032. Springer Verlag, January 1996.

[11] J. F. Groote and M. P. A. Sellink. Confluence for process verification. *Theoretical Computer Science*, 170(1–2):47–81, 1996.

[12] J. F. Groote and J. van de Pol. State space reduction using partial τ-confluence. In M. Nielsen and B. Rovan, editors, *Proc. of MFCS'00*, LNCS vol. 1893, pp. 383–393. Springer Verlag, August 2000.

[13] Y. Ioannidis, R. Ramakrishnan, and L. Winger. Transitive closure algorithms based on graph traversal. *ACM Tr. on Database Systems*, 18(3):512–576, 1993.

[14] T. Jéron and P. Morel. Test generation derived from model-checking. In N. Halbwachs and D. Peled, editors, *Proc. CAV'99*, LNCS vol. 1633, pp. 108–122. Springer Verlag, July 1999.

[15] T. Jéron and P. Morel. Abstraction, τ-réduction et déterminisation à la volée: application à la génération de test. In G. Leduc, editor, *Congrès Francophone sur l'Ingéniérie des Protocoles CFIP'97*. Hermès, 1997.

[16] J. Magee and J. Kramer. *Concurrency — state models and Java programs*. John Wiley and Sons, 1999.

[17] R. Mateescu. A generic on-the-fly solver for alternation-free boolean equation systems. In H. Garavel and J. Hatcliff, editors, *Proc. of TACAS'2003*, LNCS vol. 2619, pp. 81–96. Springer Verlag, April 2003.

[18] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46(3):255–281, March 2003.

[19] R. Milner. *Communication and concurrency*. Prentice-Hall, 1989.

[20] G. Pace, F. Lang, and R. Mateescu. Calculating τ-confluence compositionally. In Jr W. A. Hunt and F. Somenzi, editors, *Proc. of CAV'2003*, LNCS vol. 2725, pp. 446–459. Springer Verlag, July 2003.

[21] Y. S. Ramakrishna and S. A. Smolka. Partial-order reduction in the weak modal mu-calculus. In A. Mazurkiewicz and J. Winkowski, editors, *Proc. of CONCUR'97*, LNCS vol. 1243, pp. 5–24. Springer Verlag, July 1997.

[22] C. Stirling. *Modal and temporal properties of processes*. Springer Verlag, 2001.

[23] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[24] A. Valmari, J. Kemppainen, M. Clegg, and M. Levanto. Putting advanced reachability analysis techniques together: the "ARA" tool. In J. Woodcock and P. G. Larsen, editors, *Proc. of FME'93*, LNCS vol. 670, pp. 597–616. Springer Verlag, April 1993.

[25] Vasy team. VLTS benchmark suite. `http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html`, March 2003.

[26] R. J. van Glabbeek and W. P. Weijland. Branching-time and abstraction in bisimulation semantics. CS R8911, Amsterdam, 1989. Also in *Proc. of 11th IFIP World Computer Congress*, 1989.

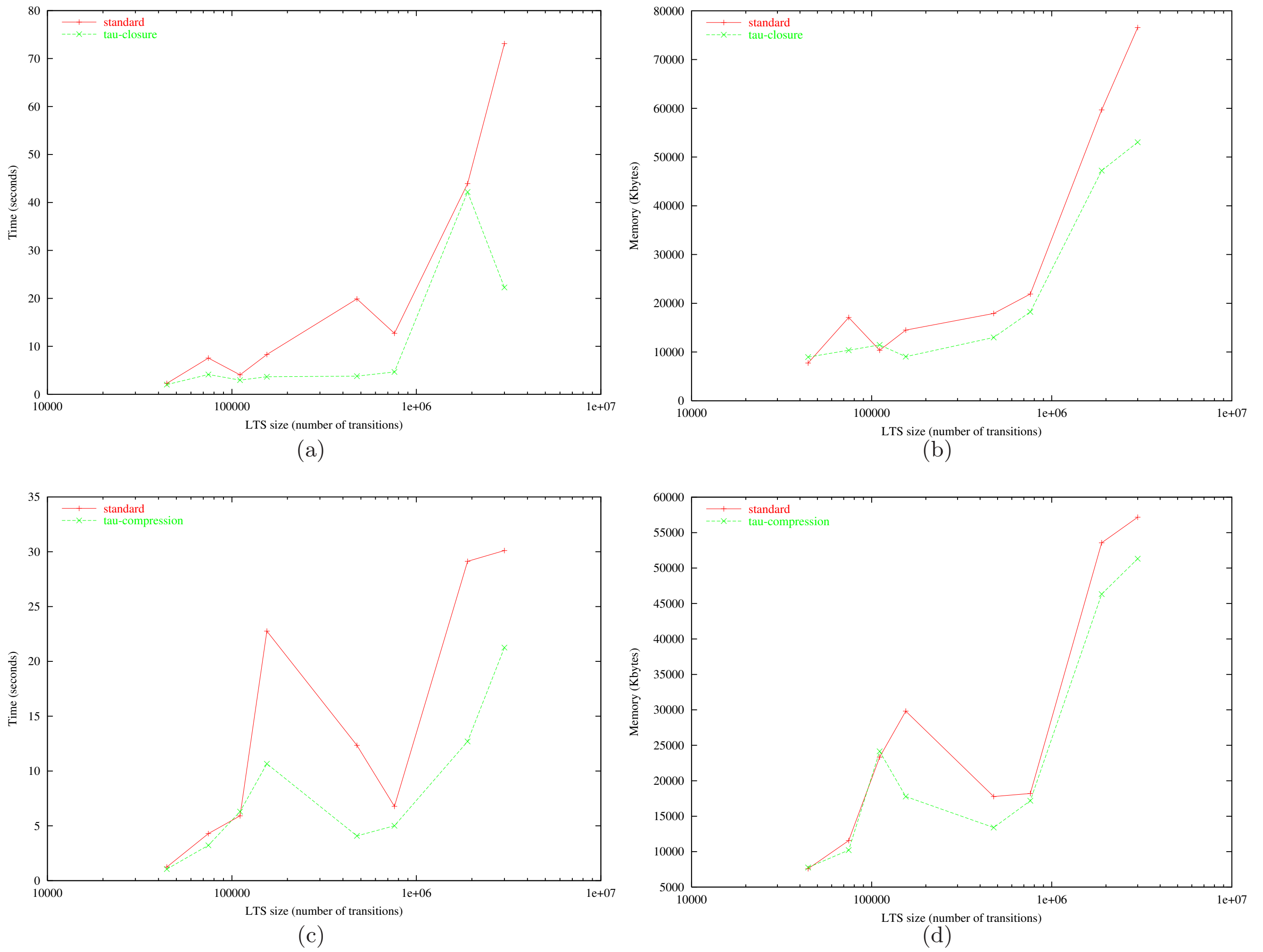


Figure 9: (a) Time and (b) memory of Evaluator **with τ-closure for checking P_1. (c) Time and (d) memory of** Evaluator **with τ-compression for checking P_2.**

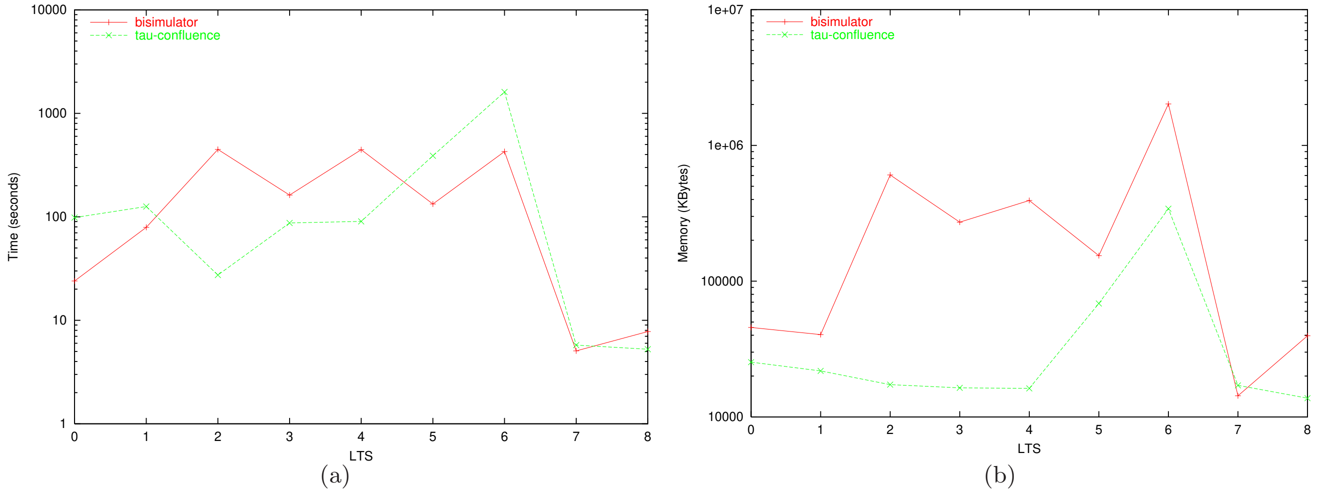


Figure 10: (a) Time and (b) memory of Bisimulator **with τ-confluence w.r.t.** Bisimulator **alone for checking observational equivalence**