

Discrete Partial Order Reduction for UPPAAL

Kenneth B. Holleufer, Jesper B. Rosenkilde, and Martin Toft
Department of Computer Science, Aalborg University, Denmark
{nokke, jbr, mt}@cs.aau.dk

May 30, 2006

Abstract

This article discusses the application of Partial Order Reduction (POR) to networks of timed automata in the UPPAAL verification engine. Timed automata and the networks of timed automata used in UPPAAL are defined. The POR method developed for untimed systems, and the problems of using it in networks of timed automata, are discussed. To circumvent these problems, a method is proposed that involves applying POR only to the discrete parts of a model. This is achieved by either adding all transitions involving time to the ample set or none. To evaluate POR, an unoptimised depth-first state-space exploration algorithm is implemented with and without POR. Properties for three widely employed protocols and algorithms are verified using the implementation, as well a property for a synthetic model to show that POR performs well on models with a mixture of discrete and timed parts. The experiments show that POR does not result in state-space reduction for highly time-dependent models, however, it creates only a small overhead in execution time. Additionally, the experiments confirm that POR results in large state-space reductions on models with a mixture of discrete and timed parts.

1 Introduction

This article discusses the implementation of Partial Order Reduction (POR) for UPPAAL [UPPa] in an effort to reduce the state-space [CGP01]. As society becomes more and more dependent on computer systems to control important tasks, the reliability of such systems becomes a key issue. To ensure the reliability, systems undergo extensive testing and verification to check their correctness. One of the methods used to verify software is called Model Checking. This involves exploring the different states of a system in order to check that one or more properties hold, e.g. the system never enters a deadlock, or the system will eventually enter a specific state or a series of states. As systems grow in size, the number of states they can en-

ter and the state graph both grow exponentially, resulting in a phenomenon called “state-space explosion” [CGP01]. To speed up the verification of systems, reduction of the explored state-space is a primary concern.

The idea proposed in this article is to apply POR to the networks of timed automata used by UPPAAL. This is not possible right away, since original POR, as described in [CGP01], only operates on networks of untimed automata. To solve this problem, POR is adapted to timed automata by applying it only in the discrete parts of the model. To test the effect of adding POR to the depth-first state-space exploration algorithm found in UPPAAL (henceforth abbreviated as DSE), a verification engine similar to UPPAAL’s is implemented.

The implementation is run with and without POR to verify properties for selected models. The source code for the implementation is available for review [HRT].

The following section explains standard timed automata [AD90, AD94] and how they are extended in UPPAAL [BDL04]. Section 2 introduces POR and clarifies how it is adapted for UPPAAL's timed automata. Section 3 presents and discusses results gained from running implementations of a model checking tool with and without POR. In Section 4 related work is discussed, and in Section 5 conclusions on the experimental results are drawn. Ideas to future work are given in Section 6.

1.1 Timed Automata

Today many computer systems depend on real-time constraints to accomplish their tasks. A way to model a system with such constraints is to use timed automata. A timed automaton is a non-deterministic finite automaton equipped with a finite number of real valued clocks. Edges in a timed automaton can be constrained on clock values and are also able to reset clocks. A timed automaton has the syntax of the non-deterministic finite automaton extended with clocks, clock constraints and clock resets. Clock constraints on edges are referred to as guards, whereas constraints on locations are referred to as invariants. The finite set $C = \{x_1, x_2, \dots\}$ representing the clock names used in the automaton, the syntax of the clock constraint is defined as follows.

Definition 1.1 ([AALS06]). The set $B(C)$ of clock constraints over the set of clocks C is defined by the abstract syntax:

$$g, g_1, g_2 ::= x \bowtie n \mid g_1 \wedge g_2$$

where $x \in C$ is a clock, $n \in \mathbb{N}$ and $\bowtie \in \{\leq, <, =, >, \geq\}$.

Each clock in C represents the amount of time elapsed from the last reset. This is expressed as a function $v : C \rightarrow R_{>0}$ called a clock valuation. The value of a clock is denoted by $v(x)$. There are two operations used to manipulate clock valuations, *delay* and *reset*. Let v be a clock valuation. The delay operation $v + d$ gives a clock valuation, where the value of every clock is increased by the amount given by the real number d .

Definition 1.2 ([AALS06]). For each $d \in R_{>0}$, the valuation $v + d$ is defined by

$$(v + d)(x) = v(x) + d, \forall x \in C$$

◆

For a subset r of clocks, the reset operation $v[r]$ gives a clock valuation where the values of clocks from r are set to zero and the other clocks remain unchanged.

Definition 1.3 ([AALS06]). For each $r \subseteq C$, the valuation $v[r]$ is defined by:

$$v[r](x) = \begin{cases} 0 & \text{if } x \in r \\ v(x) & \text{otherwise} \end{cases}$$

◆

With these two clock operations in place and the notion of clock constraints, the definition of when a clock constraint satisfies a given valuation is the following:

Definition 1.4 ([AALS06]). Let $g \in B(C)$ be a clock constraint for a given set of clocks C and let $v : C \rightarrow R_{>0}$ be a clock valuation. Evaluation of clock constraints ($v \models g$) is defined on the structure of g by:

$$\begin{aligned} v \models x \bowtie n & \Leftrightarrow v(x) \bowtie n \\ v \models g_1 \wedge g_2 & \Leftrightarrow v \models g_1 \text{ and } v \models g_2 \end{aligned}$$

where $x \in C$ is a clock, $n \in \mathbb{N}$, $g_1, g_2 \in B(C)$ and $\bowtie \in \{\leq, <, =, >, \geq\}$.

◆

With the notation from the previous definitions it is possible to make a formal definition of a timed automaton:

Definition 1.5 ([AALS06]). A timed automaton, over a finite set of clocks C and a finite set of actions Act , is the quadruple (L, l_0, E, I) where:

- L is a finite set of locations.
- $l_0 \in L$ is the initial location.
- $E \subseteq L \times B(C) \times Act \times 2^C \times L$ is a finite set of edges.
- $I : L \rightarrow B(C)$ assigns invariants to locations.

Instead of $(l, g, a, r, l') \in E$, edges are written as $l \xrightarrow{g, a, r} l'$, where l is the source location, g is the guard, a is the action, r is the set of clocks to be reset and l' is the target location. \blacklozenge

Timed automata behaves as following. The current state of a timed automaton consists of a pair (l, v) , where l is the current location that the automaton is in, and v is the valuation determined by the current clock values. The state is legal only if the valuation v satisfies the invariant of the location l . An edge is enabled, and thus can be taken, if its source location matches the current location l , and its guard is satisfied by the current valuation v . The target location of the edge then becomes the current, and all the clocks on the edge are reset. It is also possible to delay in the current location by increasing the value of all the clocks by the amount of time d . This does not change the location and is only possible if the invariant of the current location is satisfied by the valuation $v+d$. The timed transition system generated by a given timed automaton is defined as:

Definition 1.6 ([AALS06]). Let $A = (L, l_0, E, I)$ be a timed automaton over a finite set of clocks C and a set of actions Act . The timed transition system $T(A)$ generated by A is defined as $T(A) = (S, Lab, \xrightarrow{a} | a \in Lab)$ where:

- $S = \{(l, v) \mid (l, v) \in L \times (C \rightarrow R_{>0}) \text{ and } v \models I(l)\}$ is the set of states.
- $Lab = Act \cup R_{>0}$ is the set of labels.
- The transition relation is defined as:

- $(l, v) \xrightarrow{a} (l', v')$
if there is an edge $(l \xrightarrow{g, a, r} l') \in E$ s.t.
 $v \models g$, $v' = v[r]$, and $v' \models I(l')$.
- $(l, v) \xrightarrow{d} (l, v+d)$
for all $d \in R_{>0}$ s.t. $v \models I(l)$ and
 $v+d \models I(l)$.

Let v_0 denote the valuation such that $v_0(x) = 0$ for all $x \in C$. If v_0 satisfies the invariant of the initial location l_0 , then (l_0, v_0) is called the initial state of $T(A)$. \blacklozenge

Because each state in a timed automaton contains a particular valuation of clocks, a timed automata can, even if it only contains one clock,

generate infinitely many reachable states. Therefore a way to model infinitely many clock valuations finitely is needed. This can be done with region graphs that partition the collection of infinitely many valuations into finitely many equivalence classes. The partitioning must be done in such way that valuations from the same class does not create any significant difference in the behaviour of the system. That is, if a set of states, which share their discrete part, and their associated clock valuations are located in the same region, then these states can reach the same regions. To do this, an equivalence relation \equiv over clock valuations that implies when two states are un-timed bisimilar is defined. This relation must have finitely many equivalence classes. Before defining the equivalence relation \equiv , there is a need for some notation.

Definition 1.7 ([AALS06]). Let $d \in R_{>0}$ be a real number. By $\lfloor d \rfloor$ the integer part is denoted d , and $frac(d)$ stands for the fractional part of d . \blacklozenge

The following definition defines when two clock valuations are equivalent with respect to the requirements.

Definition 1.8 ([AALS06]). Let A be a timed automaton and let c_x denote the largest constant, which the clock $x \in C$ is ever compared with either in the guards or invariants in A . We say that two clock valuations v and v' are called equivalent, and write $v \equiv v'$, iff

1. For each $x \in C$, it holds that either both $v(x)$ and $v'(x)$ are greater than c_x or

$$\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor \quad (1)$$

2. For each $x \in C$ such that $v(x) \leq c_x$ we have

$$frac(v(x)) = 0 \Leftrightarrow frac(v'(x)) = 0 \quad (2)$$

3. For all $x, y \in C$ such that $v(x) \leq c_x$ and $v(y) \leq c_y$ we have

$$frac(v(x)) \leq frac(v(y)) \Leftrightarrow frac(v'(x)) \leq frac(v'(y)). \quad (3)$$

\blacklozenge

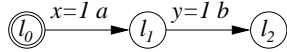


Figure 1: A timed automaton with two clocks.

In part 1 of Definition 1.8, the valuations $v(x)$ and $v'(x)$ are compared to the greatest constant clock x is ever compared to. If they both are greater, it means that they can only leave their region if they are reset. This implies that they have the exact same behaviour and they can be put in the same equivalence class. Continuing in part 1, since clocks are only compared to values in \mathbb{N} on guards and invariants, the fractional part of a clock valuation can be ignored. So if two valuations share the same integer part, they can be put in the same equivalence class. As an example suppose, in a location l , we have an edge with the guard $x \leq 1$ and another edge with $x \leq 2$. From the two states $(l, [x = 1.2])$ and $(l, [x = 1.4])$ it is only possible to take the second edge, so the valuations are in the same equivalence class. From the states $(l, [x = 0.4])$ and $(l, [x = 0.9])$ it is possible to take both edges, so these valuations are together in another equivalence class.

But there is a special case when the fractional part equals zero. For example the valuations $[x=1]$ and $[x=1.4]$ are not equivalent, because the first one could take both edges. This is taken care of in part 2. Two valuations, where one has a zero fractional part, can only be equivalent if the other also has a zero fractional part.

Part 3 of the definition deals with the ordering of the fractional parts between two different clocks. For the timed automaton seen in Figure 1 imagine the valuations $v_1 = [x = 0.8, y = 0.3]$ and $v_2 = [x = 0.5, y = 0.9]$. Both requirements from 1 and 2 are met, but the states (l_0, v_1) and (l_0, v_2) can lead to different behaviours. State (l_0, v_1) can delay 0.2 time units, take edge a , delay 0.5 units again, and take b . The state (l_0, v_2) cannot match this, because it would first have to delay 0.5 to take a resulting in y growing to 1.4 and disabling b .

Each clock valuation v can be represented by an equivalence class, denoted by $[v]_{\equiv}$, also called a region. Each region consists of a finite collection of clock constraints that it satisfies. It is finally possible to define a region graph, where every state (l, v) is replaced by a symbolic state $(l, [v]_{\equiv})$, and

$[v]_{\equiv}$ is the region represented by v .

Definition 1.9 ([AILS06]). The region graph of a timed automaton A over a set of clocks C and actions Act is a labelled transition system $T_r(A) = (S, Act \cup \{\epsilon\}, \{\xrightarrow{a} \mid a \in Act \cup \{\epsilon\}\})$ where:

- $S = \{(l, [v]_{\equiv}) \mid l \in L, v : C \rightarrow R_{>0}\}$ are symbolic states
- \Rightarrow on symbolic states is defined as follows:
 - For each label $a \in Act$, we have $(l, [v]_{\equiv}) \xrightarrow{a} (l', [v']_{\equiv}) \Leftrightarrow (l, v) \xrightarrow{a} (l', v')$
 - $(l, [v]_{\equiv}) \xrightarrow{\epsilon} (l, [v']_{\equiv}) \Leftrightarrow (l, v) \xrightarrow{d} (l, v')$, for some $d \in R_{>0}$.

◆

UPPAAL and similar programs use a more compact representation of regions with so called zones. These zones can be stored in memory in a data structure called a Difference Bound Matrix (DBM) [AILS06].

1.2 Timed Automata in UPPAAL

UPPAAL models a system as a network of several timed automata in parallel, and it extends the model with bounded discrete variables that are part of the state. These variables can be read, written and used in common arithmetic operations. A state in UPPAAL is defined by the locations of all automata, clock constraints, and the values of the discrete variables. An edge can be taken separately or synchronise with another automaton through channels, and results in a new state. Timed Automata in UPPAAL are often composed into a network over a common set of clocks and actions, consisting of n timed automata $A_i = (L_i, l_i^0, E_i, I_i)$, $1 \leq i \leq n$.

Definition 1.10 ([AILS06, BDL04]). Let $A_i = (L_i, l_i^0, E_i, I_i)$ be a network of n timed automata over a set of clocks C and actions $Act = \{c! \mid c \in Chan\} \cup \{c? \mid c \in Chan\} \cup N$, where N is a finite set of ordinary actions, and $Chan$ is a finite set of channel names. The timed transition system is defined as $\langle S, s_0, \rightarrow \rangle$, where $S = L_1 \times \dots \times L_n \times (C \rightarrow R_{>0})$ is the set of states, $s_0 = (\bar{l}_0, u_0)$ is the initial state, and $\rightarrow \subseteq S \times S$ is the transition relation defined by:

- $(\bar{l}, v) \rightarrow (\bar{l}, v + d)$ for all $d \in R_{>0}$ such that $v + d' \models I(\bar{l})$ for each real number d' in interval $[0, d]$.
- $(\bar{l}, v) \rightarrow (\bar{l}[l'_i/l_i], v')$ if there exists $(l_i \xrightarrow{g, \tau, r} l'_i)$ such that $v \models g$, $v' = v[r]$, $v' \models I(\bar{l}[l'_i/l_i])$, and $\tau \in N$.
- $(\bar{l}, v) \rightarrow (\bar{l}[l'_j/l_j, l'_i/l_i], v')$ if there exist $(l_i \xrightarrow{g_i, c^?, r_i} l'_i)$ and $(l_j \xrightarrow{g_j, c!, r_j} l'_j)$ such that $v \models g_i \wedge g_j$, $v' = v[r_i \cup r_j]$ and $v' \models I(\bar{l}[l'_j/l_j, l'_i/l_i])$.

Where $\bar{l} = (l_1, \dots, l_n)$ is a location vector. $\bar{l}[l'_i/l_i]$ is used to denote the vector where the i th element l_i of \bar{l} is replaced by l'_i and

$$I(\bar{l}) = \bigwedge_{i \in \{1, \dots, n\}} I_i(l_i)$$

is the invariant function over location vectors. \blacklozenge

Additionally, the timed automata in UPPAAL have a collection of features that distinguish them from normal timed automata. Features include template automata that are defined with a set of parameters that can be of any type. These parameters include constants with integer values that can never be modified. The guards, invariants and assignments may contain expressions that range over bounded integer variables. The bounds are checked during verification, and if a state violates a bound it is discarded. Arrays can also be declared, and are allowed for clocks, channels, constants and integer variables. Integer variables and arrays of integer variables can be initialised.

Edges can synchronise with each other in different ways. In a binary synchronisation a channel c can be declared, which enables an edge labelled with $c!$ to synchronise with another labelled $c?$. If several combinations are available, a pair is non-deterministically chosen. It is also possible to synchronise through broadcast channels, where one sender $c!$ can synchronise with an arbitrary number of receivers $c?$. Any receiver that can synchronise in the current state must do so, but if there are no receivers, the sender can still execute the $c!$ action. A channel can also be declared urgent, which disables any delays before the edge is taken. Because of this, edges using urgent channels for synchronisation cannot have time constraints.

Locations also have some additional features. A location can be declared urgent, which means that time is not allowed to pass when the system is in the location. This is equivalent to adding an extra clock x that is reset on all incoming edges, and having an invariant $x \leq 0$ on the location. A location can also be committed. If any location in a state is committed, then the state itself is committed and is not able to delay. Furthermore the next transition must involve an outgoing edge of at least one of the committed locations.

Expressions in UPPAAL range over clocks and integer variables, and they are used with guard, invariant, synchronisation and assignment labels. Out of these, only assignment may have a side effect, and it allows assignment of integer values to clocks.

The reachability algorithm for a timed automaton, see Algorithm 1, is constructed based on the symbolic timed automata semantics in Definition 1.9. In the algorithm, P is the passed-list, which contains all explored states, W is the waiting-list, which contains encountered symbolic states waiting to be explored, and $testProperty$ is a function, which checks whether the property is satisfied. [BBD⁺02]

Algorithm 1 Timed Automaton Reachability [BBD⁺02]

```

1  $W = \{(l_0, Z_0 \wedge I(l_0))\}$ 
2  $P = \emptyset$ 
3 while  $W \neq \emptyset$  do
4    $(l, Z) = W.popstate()$ 
5   if  $testProperty(l, Z)$  then return true
6   if  $\forall (l, Y) \in P : Z \not\subseteq Y$  then
7      $P = P \cup \{(l, Z)\}$ 
8     forall  $(l', Z') : (l, Z) \Rightarrow (l', Z')$  do
9       if  $\forall (l', Y') \in W : Z' \not\subseteq Y'$  then
10         $W = W \cup \{(l', Z')\}$ 
11       end if
12     end do
13   end if
14 end while
15 return false

```

Extending Algorithm 1 to networks of timed automata and to timed automata with data variables is straight-forward by replacing the single location with a location vector and by adding a variable

vector. UPPAAL implements the reachability algorithm in Algorithm 1 with a change of the underlying data structure for P and W due to optimisation. The optimisation reduces the algorithm’s memory requirements and is not important for testing state-space reduction methods such as POR. [BBD⁺02]

2 Partial Order Reduction

POR is a method for reducing the state-space graph explored by model checking tools. It removes some of the interleavings that must be considered on asynchronous systems, thus decreasing the number of possible computation paths. POR addresses the commutativity exhibited by concurrently executed transitions leading to the same state for different interleavings. Most of the definitions, rules and conditions for POR in the following originate from [CGP01]. Deviations from this source are marked clearly.

When applying POR to model checking, the explored number of transitions from a state is reduced by choosing a smaller but sufficient set of transitions from the set of enabled transitions. This subset is called the ample set. For a state s , the ample set $ample(s) \subseteq enabled(s)$ must be computed. The transitions in $ample(s)$ are then further explored instead of the transitions in $enabled(s)$, which reduces the size of the exploration. Of course the transitions in $ample(s)$ must be equivalent to the transitions in $enabled(s)$ with respect to the property being checked. The conditions needed for this to hold are examined in Section 2.2. First the needed concepts of independence and invisibility are introduced in Section 2.1.

The version of POR presented in the following two sections only covers untimed automata and cannot be utilised directly on UPPAAL’s timed automata, see Section 2.3. This problem is addressed in Section 2.4.

2.1 Independence and Invisibility

Independence and invisibility are two of the main concepts involved in POR. They are needed to determine whether $ample(s)$ is equivalent with $enabled(s)$. The independence concept establishes an independence relation between transitions in the state graph based on *enabledness* and *commutativ-*

ity. Moreover, the independence of a transition pair is determined by the transition’s update and guard properties, and their target location’s invariants. The invisibility concept builds a set of transitions in the state graph, which are invisible with regard to the checked property.

The independence relation $I \subseteq T \times T$, where T is the set of all transitions, is a symmetric and anti-reflexive relation, which for each state s in the state graph and for each pair of transitions $(\alpha, \beta) \in I$ satisfies enabledness and commutativity, as defined in Definition 2.1 and Definition 2.2, respectively.

Definition 2.1 ([CGP01]). Enabledness:
If $\alpha, \beta \in enabled(s)$ then $\alpha \in enabled(\beta(s))$. \blacklozenge

Definition 2.2 ([CGP01]). Commutativity:
If $\alpha, \beta \in enabled(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$. \blacklozenge

Enabledness says that two transitions cannot disable each other, however, one is allowed to enable the other. The commutativity condition ensures that independent transitions, executed in any order, result in the same state. Whether an ample set is equivalent to the enabled set is determined using the dependency relation $D = (T \times T) \setminus I$, which is the complement of I . The following two rules are examples of what can determine dependency between transitions.

- Two transitions that update the same variable are dependent.
- If one transition updates a variable and another transition reads that variable, then the transitions are dependent.

Section 2.4 clarifies what defines the dependency relation for models in UPPAAL.

By labelling each state in the graph with a set of atomic propositions, the set of invisible transitions with regard to the checked property is built. The labelling function L is defined as $L : S \rightarrow 2^{AP}$, where S is the set of states in the state graph and AP is an atomic proposition.

A transition $\alpha \in T$ between two states, $s, s' \in S$, is invisible with regard to a set of atomic propositions $AP' \subseteq AP$ if $L(s) \cap AP' = L(s') \cap AP'$. Put differently, a transition is invisible with regard to the propositional variables in AP' if it does not change the value of any of them. Visible transitions are transitions, which are not invisible. The set of

visible transitions are used to ensure that atomic propositions, which affects the checked property, are not cut from the ample set.

2.2 Conditions

An ample set for a state s is valid if it is equivalent with the enabled set of s , with respect to the checked property. Four conditions, C0-C3, are used to ensure this. When a state s is explored, the verification process searches for a valid ample set. Small ample sets are preferred, as this gives the smallest local state-space. If no valid ample set smaller than $enabled(s)$ is found, $enabled(s)$ is used.

Condition C0 ensures that if there exists at least one enabled transition, it is taken. Put differently, $ample(s) = \emptyset$ iff $enabled(s) = \emptyset$.

Condition C1 states that if a transition α in a state s depends on a transition in $ample(s)$, then α may not be carried out until some transition in the ample set has been carried out. Checking this potentially takes at least as long as doing unoptimised model checking, i.e. exponential time, since the subtree under s must be explored. This makes POR unusable in practical applications.

Instead of checking according to the original C1, a conservative heuristic is used, which works in linear time proportional to the number of transitions in s . The heuristic only looks one state ahead and discards $ample(s)$, if the transition α , which depends on a transition in $ample(s)$, is not carried out in the states that follow from $ample(s)$. This eliminates some ample sets that might be correct, but it is efficient and ensures the original condition.

Condition C2 states that if a state s is not fully expanded, i.e. $ample(s) \subset enabled(s)$, then every transition in $ample(s)$ must be invisible.

Condition C3 restricts the structure of cycles in the state graph. If a cycle contains a state in which some transition α is enabled, and α is not included in $ample(s)$ for any state s on the cycle, then C3 is not satisfied. The complexity of checking C3 is too high, since the verifier potentially would have to take the entire reduced state graph into

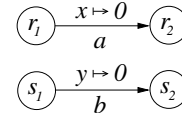


Figure 2: Effects of transition interleaving.

consideration. Instead a stronger condition is used, which can be checked without looking at the entire graph. Stating that no transition in $ample(s)$ may reach a passed state if s is not fully expanded. If a state is passed it is on the search stack because the graph search algorithm is depth-first.

2.3 Partial Order Reduction and Timed Automata

When exploring an untimed automata, POR would normally select a representative from each set of equivalent transition interleavings from the full state space. Thereby visiting only a reduced portion of the state space. However, in networks of timed automata clocks advance simultaneously in all automata, which causes dependencies between transitions in individual components [Min99]. Different interleavings may therefore produce different clock valuations.

Consider the system of two automata in Figure 2. The initial state is $(\langle r_1, s_1 \rangle, [x = y])$. If transition a is executed first, the state $(\langle r_2, s_1 \rangle, [x \leq y])$ is reached. This holds because clock x is reset and therefore is not greater than the value of y . Hereafter b is executed and clock y is reset, which results in the state $(\langle r_2, s_2 \rangle, [x \geq y])$. However, if b is executed first, the state $(\langle r_1, s_2 \rangle, [x \geq y])$ is reached followed by the state $(\langle r_2, s_2 \rangle, [x \leq y])$ after having executed a .

The two interleavings lead to the same control state, but two different clock regions and therefore different symbolic states. Because of this the two transitions are not independent and POR for untimed systems cannot be applied directly for networks of timed automata. A solution to this problem has been discussed in [Min99], see Section 4. The solution involves using a local-time semantic for every automaton in the network. This allows the automata to execute independently thus avoiding the problem.

2.4 Applying Partial Order Reduction to UPPAAL

In order to use POR with UPPAAL’s models some adaption must be done, the inclusion of time is the largest. Instead of trying to adopt POR to work with time, POR is only be applied to the discrete parts of the model. First, independence and invisibility must be clearly defined with respect to UPPAAL’s models.

Instead of defining what is independent, dependency is defined. All transitions are dependent on themselves. Two transitions are dependent if one of the transitions reads a variable x , and the other transition writes to x . If two transitions write to the same variable they are also dependent. This means that the two transitions cannot be taken independently as their execution order potentially yields different results. Guards, updates as well as the invariant of the location following the transition must be checked for variable reads and writes. Two transitions are also dependent, if they belong to two different processes and they synchronise over the same channel. All transitions in the same process are interdependent [CGP01], since they change the current location that determines which transitions are enabled. In order to ensure that all transitions concerning time are always executed, every transition is dependent on all transitions which change or reads clocks. This make the model highly interdependent, however, it makes sure that only discrete parts of a model are reduced.

As with independence, invisibility is easier defined by stating what is visible. A transition is visible if its guard, update or the following location’s invariant writes or reads a variable, which is part of the property being checked. Also, if a location is part of the checked property, the transitions going into this location are marked as visible. This is to make sure that the reduced model reaches these locations.

Conditions C0 and C2 does not need any special adaption to work with UPPAAL’s models. The heuristic for C1 is applied and should make sure that if a transition $\alpha \in enabled(s) \setminus ample(s)$ is dependent on an transition $\beta \in ample(s)$ then a state following α must have the same values as the state following β for the variables written and read by the β transition. Also, if β has a synchronisation then a state following α must have a corresponding

synchronisation. So checking for C3 simply means to search the passed list for a given state. C3 is easily fulfilled by using depth-first search, because all the transitions on the passed list lies higher in the search tree.

Since all edges are dependent on all other edges, which reads or writes a clock, the overhead introduced when applying POR to models with large non-discrete parts is high and yields no state space reduction. In this case, it would be reasonable to use the enabled set, instead of the ample set, for exploration of these parts of the model. This is because the cardinality of the two sets would be nearly equal, and calculating and validating the ample set is computational heavy.

A method, where the enabled set is analysed for transitions involving clocks, is therefore added. If a timed transition exists in the enabled set, the tedious job of calculating and validating the ample set is simply skipped.

3 Experimentation

To see if there is a performance gain in using POR that justifies the additional overhead, an implementation is constructed. In order to evaluate the proposed method, some experiments are conducted with this implementation. Several models, which are already available for UPPAAL, as well as a synthetic model, have been selected. Both the DSE algorithm and DSE optimised with POR are used to verify the models. Two things are tested; the state-space reduction when using POR in comparison to the DSE algorithm, and what the difference in computational time is. We expect to see some good results on models with relative many discrete transitions, where POR will be able to reduce the state-space significantly. However, in models with a very high ratio of time related transitions, the reduction of the state-space may be non-existent.

3.1 Outline

The tests are conducted in pairs. First, the DSE algorithm is applied to a model. The time it takes to verify the property and number of states stored and explored during the verification is noted. Second, POR is applied to the same model and property. Time and number of states are again noted.

For every run of a specific model, the number of processes is increased, which results in a growing state-space. It starts at two processes and ends with eight. The tested models are the CSMA/CD protocol, Fischer’s mutual exclusion algorithm and the Token Ring FDDI protocol, all of which are widely used, but also a model made especially for good results with POR is tested.

3.2 The CSMA/CD Protocol

The Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol [Tan03] makes it possible for multiple stations to share a medium for communication. The protocol is widely employed, as it is used in the Medium Access Control (MAC) sublayer of Ethernet. The “carrier sense” part of the protocol reflects that a station, when it is about to transmit, first listens on the medium to see if another station is transmitting at the moment. If the medium is in use, the station waits until it becomes idle. Compared to CSMA, the collision detection in CSMA/CD saves time and bandwidth by stopping a transmission immediately, when a collision is detected, rather than finishing the current data block. We generate models and properties using the CSMA/CD script at UPPAAL’s benchmark homepage [UPPb].

The property to be checked for CSMA/CD ensures that a situation where two stations transmits simultaneously never occurs for more than a specified amount of time (52 time units). We do not expect POR to reduce the checked state-space of CSMA/CD, since all edges in the medium process and in a station process either reads or writes clocks. Due to the fact that the time zones in the CSMA/CD model keeps increasing for every new state, the zone is extrapolated with regard to a list of maximum clock constants; every clock has its own maximum constant. A constant is found by searching the model for the maximum constant that a clock is compared to.

3.3 Fischer’s Mutual Exclusion Algorithm

Fischer’s mutual exclusion algorithm is used to guarantee mutual exclusion in a concurrent system consisting of several processes, using clock constraints and a shared variable [Fis85]. The model of

Fischer’s algorithm differs fundamentally from the models of the CSMA/CD and Token Ring FDDI protocols, since Fischer’s algorithm does not have a medium process for the other processes to communicate through. Again, the models and properties are generated using a script from UPPAAL’s benchmark homepage [UPPc].

The property to be checked for Fischer’s algorithm specifies that no more than one process enters the critical section. In the model, every edge reads or writes a clock, which means that there should be a high degree of dependency among the transitions. Thus we do not expect particular good results with POR on this model.

3.4 The Token Ring FDDI Protocol

The Fiber Distributed Data Interface (FDDI) utilises a token ring network topology whose MAC sublayer is controlled by an algorithm widely known as the “Token Ring FDDI Protocol” [MK93]. As in Ethernet, this sublayer specifies the access mechanism used by stations connected to the network. A special packet, called a token, is passed around from station to station in the circular network and presents a station with the opportunity to transmit data. A station must remember to release the token when its transmission of data is completed. As with the CSMA/CD protocol and Fischer’s algorithm, the models and properties for the Token Ring FDDI protocol are generated using a script from UPPAAL’s benchmark homepage [UPPd].

The property to be checked for the Token Ring FDDI protocol specifies that two stations are not allowed to be in a transmission state simultaneously. Since most edges in this model synchronises with other edges, the transitions are highly dependent and cannot be reduced. Therefore we do not expect good results from POR.

3.5 DSS Coffee-Room Mutual Exclusion Model

A synthetic model has been devised, which describes the behaviour of a coffee machine in the Distributed Systems and Semantics (DSS) unit, with a limited number of coffee fills and a number of computer scientists. The scientists are able to get coffee from the machine by synchronising with it

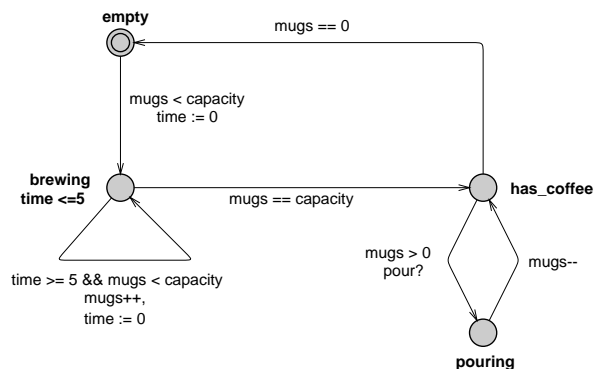


Figure 3: Coffee machine model.

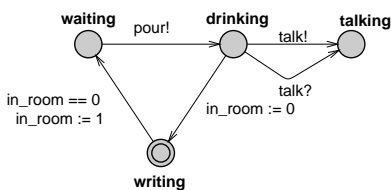


Figure 4: Computer scientist model.

over the *pour* channel. If a scientist meets a co-worker in the coffee-room, they begin talking and waste valuable time in the eyes of the department administration. When the machine becomes empty, it spends a period of time brewing new coffee before being able to serve the scientists again. The property to be checked is that no scientist enter the talk state. We expect POR to perform very well on the model due to few clocks and a small number of synchronisations. See Figure 3 for the model of the coffee machine and Figure 4 for the model of the computer scientist.

3.6 Results

Tables 1, 2 and 3 show the results of verifying properties for CSMA/CD, Fischer’s algorithm and Token Ring FDDI using the unoptimised DSE algorithm and the DSE algorithm optimised by POR. Table 4 shows the results for the DSS coffee-room model. All the models satisfy their property, which forces the verifier to continue until no new states are discovered. The four tested models have separate tables, each of which contains rows for the

particular test cases. A row contains the number of processes (medium counted out in CSMA/CD and Token Ring FDDI), and the number of stored and explored states in a tuple. The DSE columns indicate unoptimised DSE, whereas the POR columns indicate DSE optimised by POR.

The results in all tables confirm the expectations mentioned in Sections 3.2, 3.3, 3.4 and 3.5; POR does not reduce the state-spaces explored for CSMA/CD, Fischer’s algorithm and Token Ring FDDI, and it performs well on the DSS coffee-room model. The small variations seen in Table 1 for CSMA/CD with seven and eight processes probably result from the POR implementation using another search order through the state-space than DSE. The variations are relatively small and fluctuate to each side, making it impossible to draw conclusions from them.

The experiments’ execution time relative to the number of involved processes are plotted semi-logarithmic in Figures 5 and 6. The graphs for CSMA/CD, Fischer’s algorithm and Token Ring FDDI emphasise a valuable property of POR; the method does not create a large computational overhead for models that are not reduced well. However, as seen in the graph for the DSS coffee-room model, the execution time for POR starts diverging from the execution time of DSE on models that are reduced well. In models that are highly time dependent, POR saves time by frequently choosing to proceed with the enabled set instead of trying to validate a number of ample sets, while in highly discrete models it spends time on frequently validating ample sets.

The small, initial overhead seen for CSMA/CD, Fischer’s algorithm and Token Ring FDDI in their plots, originates from generating the visibility set and the dependency relation before starting the exploration. For these three models, the exploration times for DSE and POR converge toward each other, and indicate no increase in execution time for POR. The DSS coffee-room model has lower start-up constant because of the lower number of transitions resulting from a smaller state-space. However, the added complexity of calculating the ample set results in the execution time increasing, under the number of processes, relative to DSE.

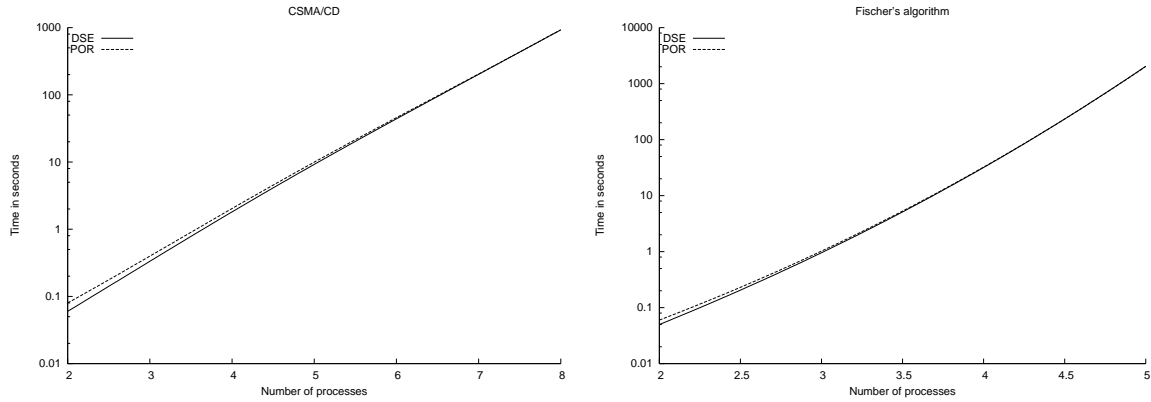


Figure 5: Results plotted for CSMA/CD (left) and Fischer's algorithm (right).

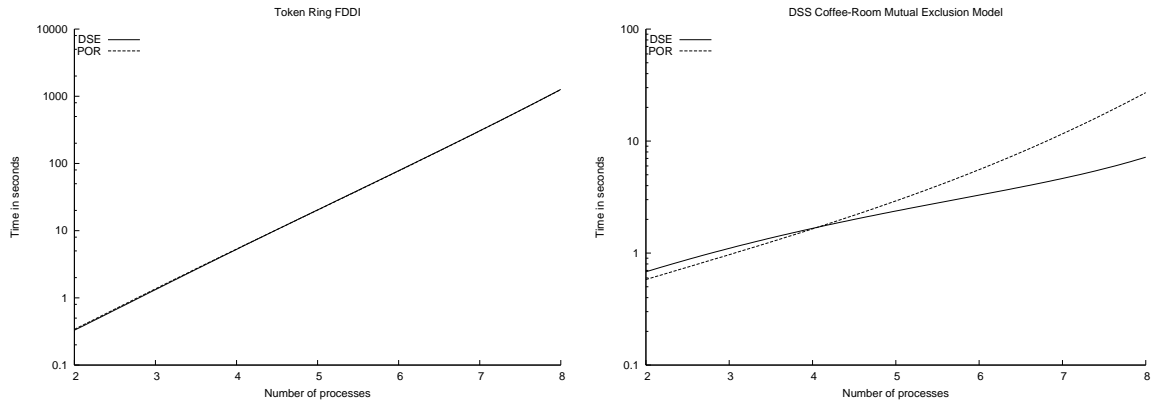


Figure 6: Results plotted for Token Ring FDDI (left) and DSS coffee-room model (right).

P	DSE	POR
2	(11, 14)	(11, 14)
3	(62, 93)	(62, 93)
4	(273, 443)	(273, 443)
5	(1067, 1828)	(1075, 1841)
6	(3538, 6411)	(3538, 6411)
7	(11772, 22659)	(11861, 22810)
8	(36583, 75936)	(35508, 74106)

Table 1: Results for CSMA/CD.

P	DSE	POR
2	(25, 37)	(25, 37)
3	(252, 463)	(252, 463)
4	(4544, 10190)	(4544, 10190)
5	(155415, 412411)	(155415, 412411)
6	(-, -)	(-, -)
7	(-, -)	(-, -)
8	(-, -)	(-, -)

Table 2: Results for Fischer's algorithm. For more than five processes the test computer ran out of memory.

P	DSE	POR
2	(100, 124)	(100, 124)
3	(310, 384)	(310, 384)
4	(990, 1245)	(990, 1245)
5	(2743, 3442)	(2743, 3442)
6	(7932, 9945)	(7932, 9945)
7	(22554, 28232)	(22554, 28232)
8	(63587, 79499)	(63587, 79499)

Table 3: Results for Token Ring FDDI.

P	DSE	POR
2	(255, 412)	(124, 161)
3	(357, 600)	(146, 202)
4	(459, 788)	(168, 243)
5	(561, 976)	(190, 284)
6	(663, 1164)	(212, 325)
7	(765, 1352)	(234, 366)
8	(867, 1540)	(256, 407)

Table 4: Results for the DSS coffee-room model.

4 Related Work

POR is not the only method for dealing with the state-space explosion problem, other methods exist as well. In [GY04] two approaches using static analysis are presented, called *path reduction* and *dead variable reduction*. Static analysis is the act of analysing computer software without executing it, which corresponds to not constructing a state graph for a process when applying the reduction methods. Both methods utilise static analysis of the model to reduce the model before the verification process. This is different from POR, which influences the verification process, and makes the methods in [GY04] easier to incorporate into existing model checking software. However, this difference also makes it easy to combine the methods with one another and POR.

Path reduction examines computation paths in the control flow of a process. The idea is to identify paths consisting of consecutive operations that do not affect the property being checked, and exchange the paths with single transitions that accomplish the same operations. The method produces reduced state graphs with shorter computation paths, whereas POR produces fewer computation paths. Promising results on verification of implementations of known protocols are achieved

in [GY04], where the reduced transition systems are between 8 to 37% of the original.

Dead variable reduction examines use of variables everywhere within a process and detects areas, where one or more variables are insignificant, i.e. they are not used later in the possible computation paths from that point. States in the process’s automaton that only differ with regard to insignificant variables are then pruned. The experiments in [GY04] show less impressive results for this method – it generated reduced transition systems, which were between 55 to 100% of the original. Dead variable reduction is refined to *partially dead variable reduction*, which introduces a more fine-grained examination for dead variables. The experimental results for this version vary and are infrequently significant. In both cases the experiments were conducted on models of known protocols, as was the case with the path reduction method.

Another method, described in [Min99], makes the application of partial order reduction possible in timed automata. It involves a partial order reduction algorithm for networks of timed automata that can be used for model checking of properties described in a timed extension of Linear Temporal Logic (LTL). The algorithm is based on a modified local-time semantic, which allows the individual automaton to execute independently except when dealing with synchronisation transitions.

The utilised local-time semantic takes into account that when a single automaton in a network enables a transition, then the resulting change of state for that automaton does not change the state of the other automata in the network, unless it is a synchronisation. Thus action transitions involving two disjoint sets of automata are independent. A delay transition will, however, change the state of all automata and become dependent on any action transition that also changes clock values. This entails that a delay can be viewed as a set of simultaneous transitions with equal delay in all the automata. This means that time-induced dependencies can be removed by separating the delay transition in all the automata into individual transitions.

The experiments in [Min99] show a reduction in the state-space, which comes from two sources. First, the partial order reduction algorithm itself reduces the state-space and, second, the local-time model reduces the number of generated time regions.

5 Conclusion

POR is a well tested method for reducing the state space in discrete models. Earlier attempts to adapt POR for timed models have introduced complex semantics and additional computational overhead. This article has presented a method for applying POR only to the discrete parts of a timed model, and thus avoiding defining new time semantics.

The developed method was applied to UPPAAL timed automata, and tested on three real world timed models, as well as a synthetic model. The real world models are all highly dependent on time or synchronisations, and as expected POR performed very poorly, state-space reduction wise, on these. The additional computational overhead was kept to a minimum with POR, so the method did not have any adverse effects. This suggests that there is no penalty in applying the method to timed automata, however, the models' states-spaces remained the same size.

On the synthetic model the stored state-space was reduced by about 65% and the explored states by about 70%. This shows that POR reduced the state-space on the discrete part of the model significantly. Applying POR to a large portion of the model, combined with a large number of processes, generates significant computational overhead. This was expected as POR possibly has to combine 2^P ample sets, where P is the number of processes in the model.

In conclusion, when applying POR to models where it has no effect, it does not slow down the state-space exploration. But when applied to a model with a discrete part, POR yield a state-space reduction in return for an increase in time for the state-space exploration. Based on this we can conclude that if real world models exist with large discrete parts, POR will be able to reduce their state-space. Since there is no significant computational overhead when applying POR to non-discrete models there is nothing that speaks against

implementing POR in a real-time systems model checker. This would allow a such a model checker to be used as a general purpose model checker. This has the advantage of a linear growing state-space for discrete models, which would otherwise grow exponentially.

6 Future Work

As path reduction aims at shorter computation paths, and POR aims at fewer computation paths, it would be interesting to combine the two methods. As mentioned in Section 4, path reduction is a static analysis method, and therefore does not interfere with the state-space exploration. This makes it easy to combine path reduction with POR and benefit from both methods.

References

- [AD90] Rajeev Alur and David L. Dill. Automata for Modeling Real-Time Systems. *Lecture Notes in Computer Science*, 443:322–335 [1990].
- [AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Journal of Theoretical Computer Science*, 126(2):183–235 [1994].
- [AILS06] Luca Aceto, Anna Ingólfssdóttir, Kim G. Larsen, and Jiri Srba. Reactive Systems: Modelling, Specification and Verification, (DRAFT) [2006].
- [BBD⁺02] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL Implementation Secrets. *Lecture Notes in Computer Science*, 2469:3–22 [2002].
- [BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal [2004].
URL: <http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf>
- [CGP01] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. Model Checking. The MIT Press [2001]. ISBN: 0-262-03270-8.
- [Fis85] M. Fischer. Re: Where are you? Arpanet message number 8506252257.AA07636@YALE-BULLDOG.YALE/ARPA [1985]. E-mail message to Leslie Lamport.
- [GY04] Orna Grumberg and Karen Yorav. Static Analysis for State-Space Reductions Preserving Temporal Logics. *Formal Methods in System Design*, 25:67–96 [2004].
- [HRT] Kenneth B. Holleufer, Jesper B. Rosenkilde, and Martin Toft. Implementation of the UPPAAL Verification Engine with and without Partial Order Reduction.
URL: <http://www.cs.aau.dk/~jbr/dpor/>
- [Min99] Marius Minea. Partial Order Reduction for Verification of Timed Systems. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburg, PA 15213 [1999].
- [MK93] Sonu Mirchandani and Raman Khanna. FDDI - Technology and Applications. John Wiley & Sons, Inc. Pp. 10–11 [1993]. ISBN: 0-471-55896-6.
- [Tan03] Andrew S. Tanenbaum. Computer Networks. Prentice Hall Professional Technical Reference. Pp. 255–258, fourth edition [2003]. ISBN: 0-13-038488-7.
- [UPPa] UPPAAL Official Homepage, updated the 4th of October, 2005.
URL: <http://www.uppaal.com>
- [UPPb] UPPAAL Benchmark Homepage, CSMA/CD script, version from the 19th of September, 2001.
URL: http://www.it.uu.se/research/group/darts/uppaal/benchmarks/genCSMA_CD.awk
- [UPPc] UPPAAL Benchmark Homepage, Fischer script, version from the 14th of September, 2001.
URL: <http://www.it.uu.se/research/group/darts/uppaal/benchmarks/genFischer.awk>
- [UPPd] UPPAAL Benchmark Homepage, Token Ring FDDI script, version from the 19th of September, 2001.
URL: <http://www.it.uu.se/research/group/darts/uppaal/benchmarks/genHDDI.awk>

Resume

The article presents definitions of timed automata, the networks of timed automata in UPPAAL, and Partial Order Reduction (POR). The purpose is to test whether POR applied in the discrete parts of the timed automata in UPPAAL yields viable results. To assist the experiment, an implementation of an unoptimised and a POR optimised depth-first state-space exploration algorithm has been developed.

A timed automaton is a finite state automaton extended with the notion of time. This allows modelling of real-time systems using a well-established formalism. The disadvantage of applying the continuous notion of time, to finite state automata, is the introduction of an infinite number of states. Since it is impossible to traverse an infinite state-space, the concept of clock regions is introduced in order to model infinite many states in a finite way.

UPPAAL is a system for modelling, validating and verifying real-time systems modelled as networks of timed automata. The UPPAAL automata are networks of timed automata extended with data types. UPPAAL uses a compact representation of clock regions called Difference Bound Matrices. Due to the fact that time is such an integrated part of UPPAAL, even discrete models yields an exponential growing state-space.

POR is a well-known state-space reduction technique for networks of untimed automata, among other modelling formalisms. The method utilises the commutativity found in asynchronous systems to omit superfluous interleavings. This reduces the number of computation paths needed to be explored to verify some property for a system.

Because different computation paths through a timed system may yield different symbolic states, POR cannot be applied without adaption. One way to adapt POR to deal with time is to introduce local-time semantics. As this has already been investigated by several people, we have chosen not to dwell on the subject. Instead we have focused on detecting discrete parts of timed models and applying POR to these. This avoids the whole issue of time, and it yields good results on models with discrete parts, without a time penalty to highly time-dependent models.

To arrive at the above mentioned results, we experiment with applying POR to three widely employed time-dependent protocols and algorithms, as well as a synthetic model created specially for good results with POR. The experiment is conducted with an implementation of an unoptimised and a POR optimised depth-first state-space exploration algorithm. The models are verified with properties, which guarantee full state-space exploration, in order to compare POR with a standard state-space exploration.

As expected, POR does not reduce the state-space explored on highly time-dependent models. However, there is nearly no computational overhead for these models, as POR is never run on the timed parts. When applied to a model with a large discrete part, POR produces substantially smaller state-spaces at the price of increased execution time. The addition of POR in UPPAAL may help UPPAAL become a more general purpose model checker, since discrete models will then result in greatly reduced state-spaces.

In our study on the subject of state-space reduction, we found a method called path reduction. This method seems ideal in combination with POR, since POR creates fewer computation paths, whereas path reduction reduces the length of the paths. This gives a narrower as well as a shallower state-space. As path reduction is done statically, and POR dynamically, they do not interfere with each other's domain and are therefore painlessly combined.