DPOR Algorithm

Persistent Sets

A set T of transitions enabled in a state s is *persistent* in s iff for all nonempty sequence of transitions:

$$s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3, \cdots, \xrightarrow{t_n} s_{n+1}$$

in A_G and including only transitions $t_i \notin T$, $1 \leq i \leq n$, t_n is independent with all the transition in T.

DPOR Example

x, y, z = 0

T1	T2	T3
W(x)[1]	R(y)	R(z)
	l1 = R(x)[?]	l2 = R(x)[?]

What values can l1 and l2 take in an execution?

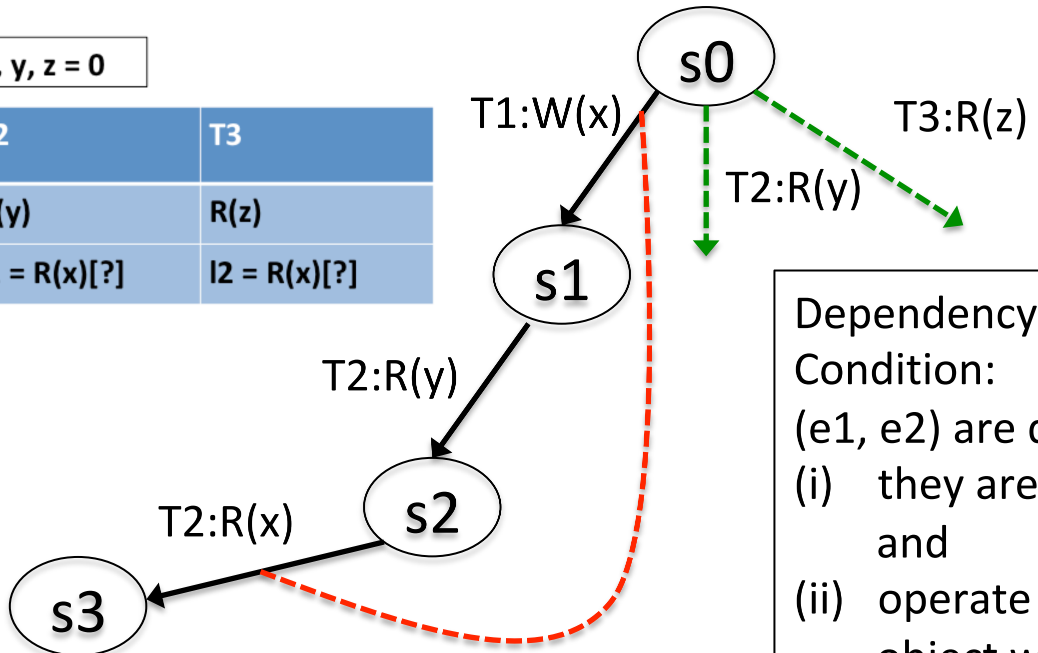
DPOR Strategy:

- Explore first maximal execution
- Discover actions that are mutually dependent and “reversible”
- Update the ample/persistent set suitably
- Re-run the program with the same choices until the point where “reversibility” is feasible

DPOR Example

x, y, z = 0

T1	T2	T3
W(x)[1]	R(y)	R(z)
	l1 = R(x)[?]	l2 = R(x)[?]



Dependency & Reversible Condition:

(e1, e2) are dependent if

(i) they are from different threads and

(ii) operate on the same shared object with at least one of them being a write

(iii) They are not HB ordered

HB ordering is the smallest relation s.t.:

- if $i \leq j$ and $(e_i, e_j) \in D$ then $I \to_E j$
- $\to_E$ is transitively closed.
- $i \to_E p$ if either (a) $proc(e_i) = p$ or (b) $\exists k \in \{I+1, \cdots, n\}$ s.t. $I \to_E k$ and $proc(e_k) = p$

DPOR Example

HB ordering is the smallest relation s.t.:

- if $i \leq j$ and $(e_i, e_j) \in D$ then $I \rightarrow_E j$
- $\rightarrow_E$ is transitively closed.
- $i \rightarrow_E p$ if either (a) $proc(e_i) = p$ or (b) $\exists k \in \{I+1, \cdots, n\}$ s.t. $I \rightarrow_E k$ and $proc(e_k) = p$

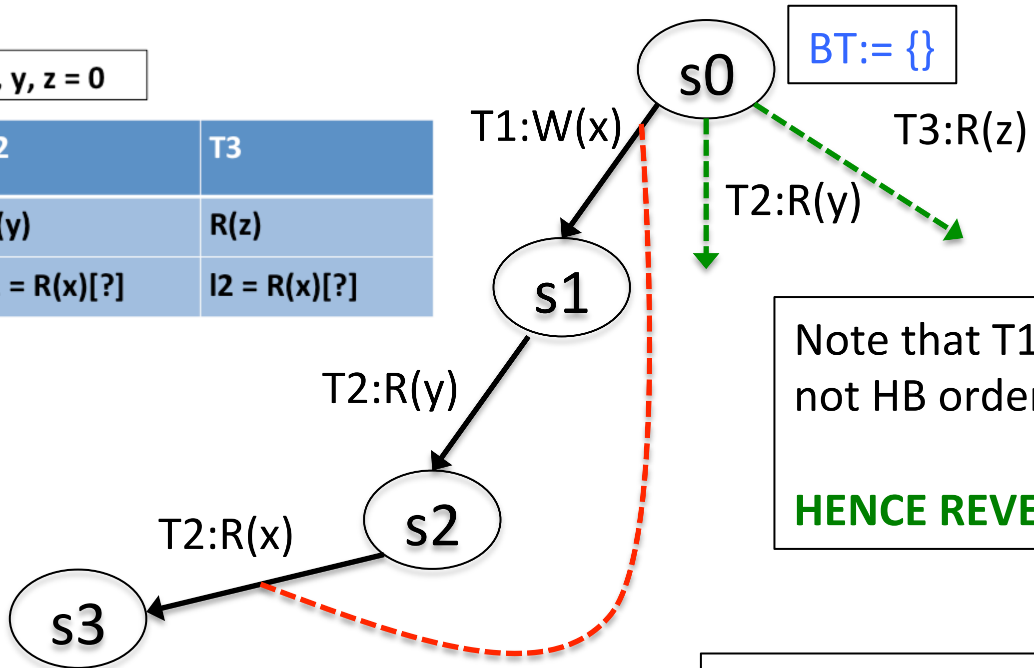
$i \nrightarrow p$ holds if:

- $proc(e_i) \neq p$ **and**
- $\nexists k \in \{i+1, \cdots, n\}$ s.t. $i \rightarrow k$ and $proc(e_k) = p$

DPOR Example

x, y, z = 0

T1	T2	T3
W(x)[1]	R(y)	R(z)
	l1 = R(x)[?]	l2 = R(x)[?]



Note that T1: W(x) and T2:R(x) are not HB ordered while still dependent!

HENCE REVERSIBLE DEPENDENCY!

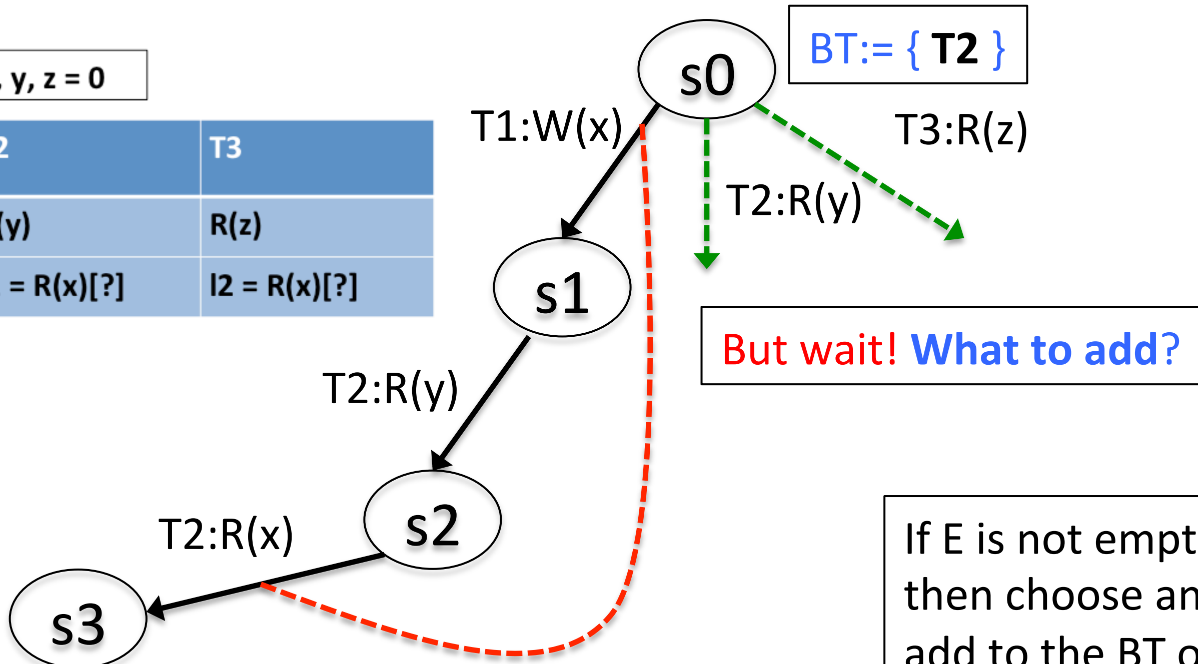
But wait! What to add and **where to add?**

$i \not\rightarrow p$ holds if:

- $proc(e_i) \neq p$ **and**
- $\nexists k \in \{i+1, \cdots, n\}$ s.t. $i \rightarrow k$ and $proc(e_k) = p$

DPOR Example

x, y, z = 0		
T1	**T2**	**T3**
W(x)[1]	**R(y)**	**R(z)**
	l1 = R(x)[?]	**l2 = R(x)[?]**



$$E = \{q \in En(pre(e_i)) \mid$$
$$q = p \vee$$
$$(\exists j \in Dom(E) : j > i \wedge q = proc(e_j) \wedge j \rightarrow_E p)\}$$

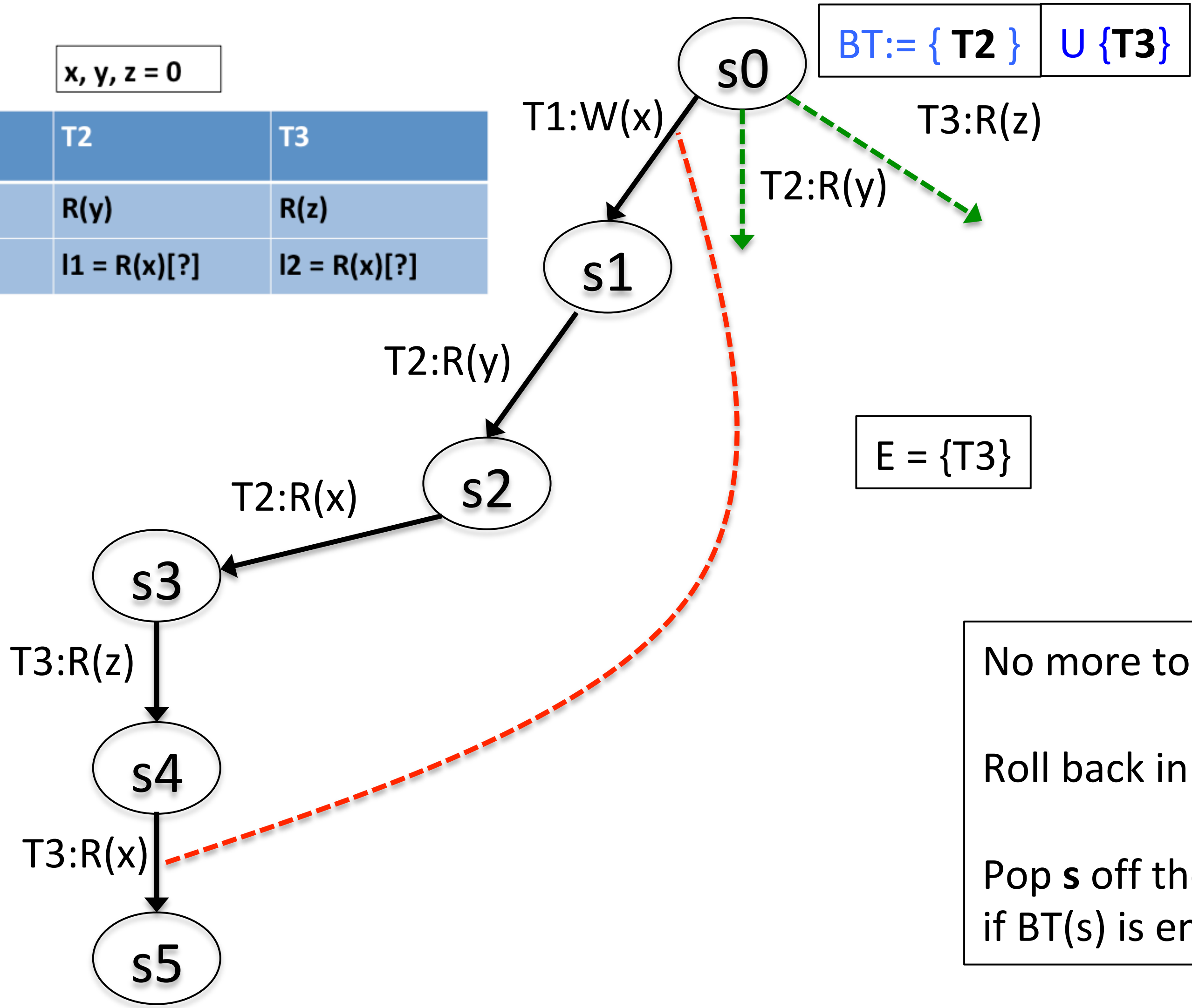
If E is not empty,
then choose any q and
add to the BT of pre(e_i)

Else add En(pre(e_i)) to BT of pre(e_i)

DPOR Example

x, y, z = 0

T1	T2	T3
W(x)[1]	R(y)	R(z)
	l1 = R(x)[?]	l2 = R(x)[?]



No more to explore.

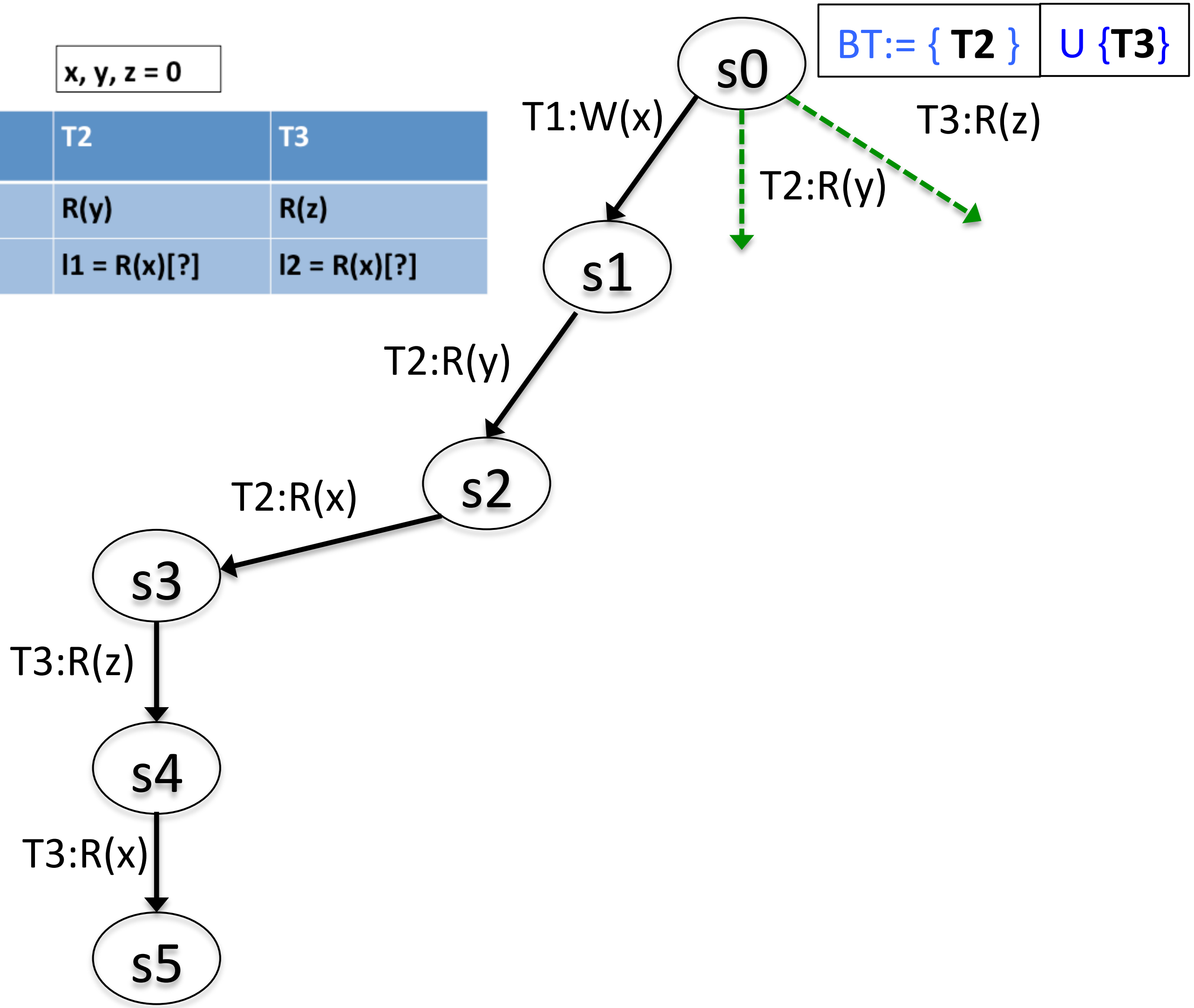
Roll back in DFS-style.

Pop **s** off the stack
if BT(s) is empty!

DPOR Example

x, y, z = 0

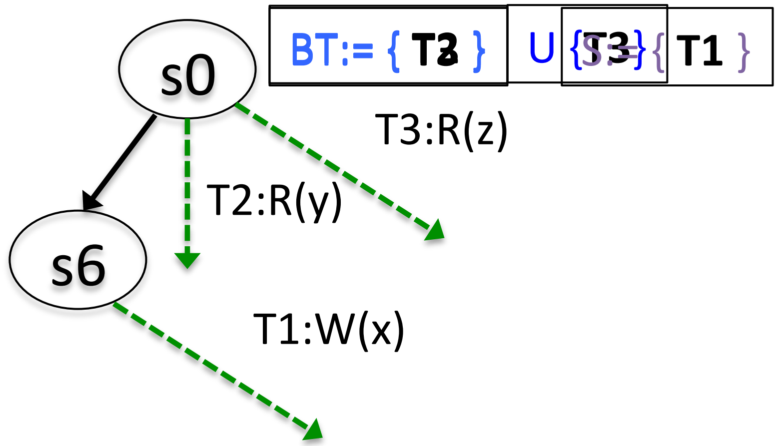
T1	T2	T3
W(x)[1]	R(y)	R(z)
	l1 = R(x)[?]	l2 = R(x)[?]



DPOR Example

x, y, z = 0

T1	T2	T3
W(x)[1]	R(y)	R(z)
	l1 = R(x)[?]	l2 = R(x)[?]



SEQUENCES EXPLORED

T1:W(x) T2:R(y) T2:R(x) T3:R(z) T3:R(x)	≡	T2:R(y) T1:W(x) .... T2:R(x) ...... T3:R(x)

If we explore from s6 T1:W(x) then we get the same sequence as explored in the previous run!

IDEA: Make the thread already explored from that state go to **sleep** **[Sleep Sets]**

We don't want to explore this redundant sequence!

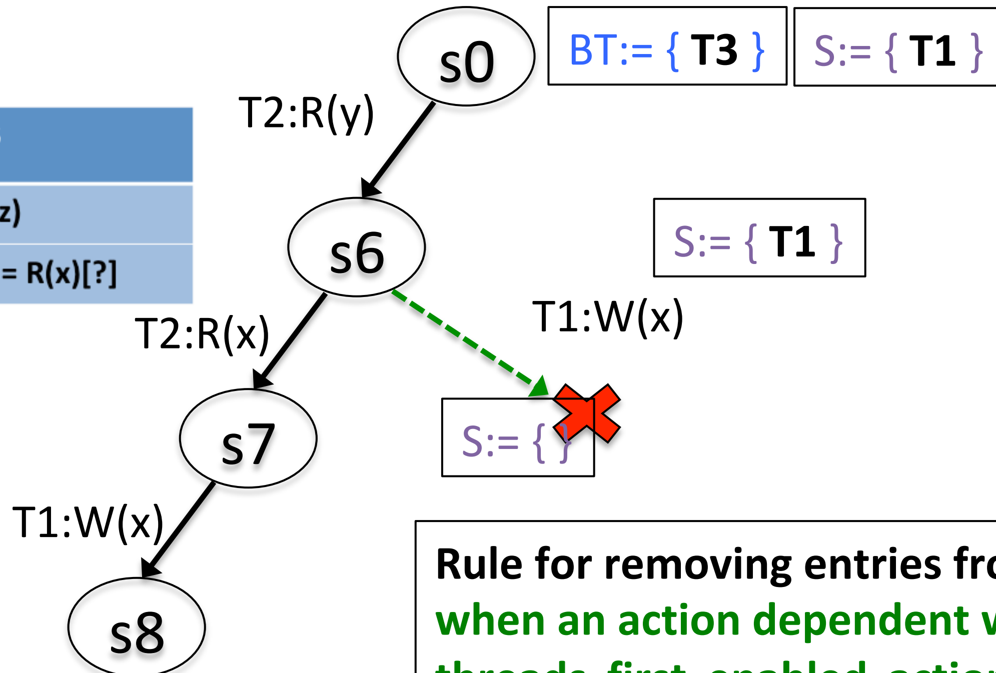
DPOR Example

x, y, z = 0

T1	T2	T3
W(x)[1]	R(y)	R(z)
	l1 = R(x)[?]	l2 = R(x)[?]

SEQUENCES EXPLORED

T1:W(x)
T2:R(y)
T2:R(x)
T3:R(z)
T3:R(x)



Rule for removing entries from sleep sets: **when an action dependent with a sleeping threads first enabled action is fired, then that sleeping thread is woken up!**

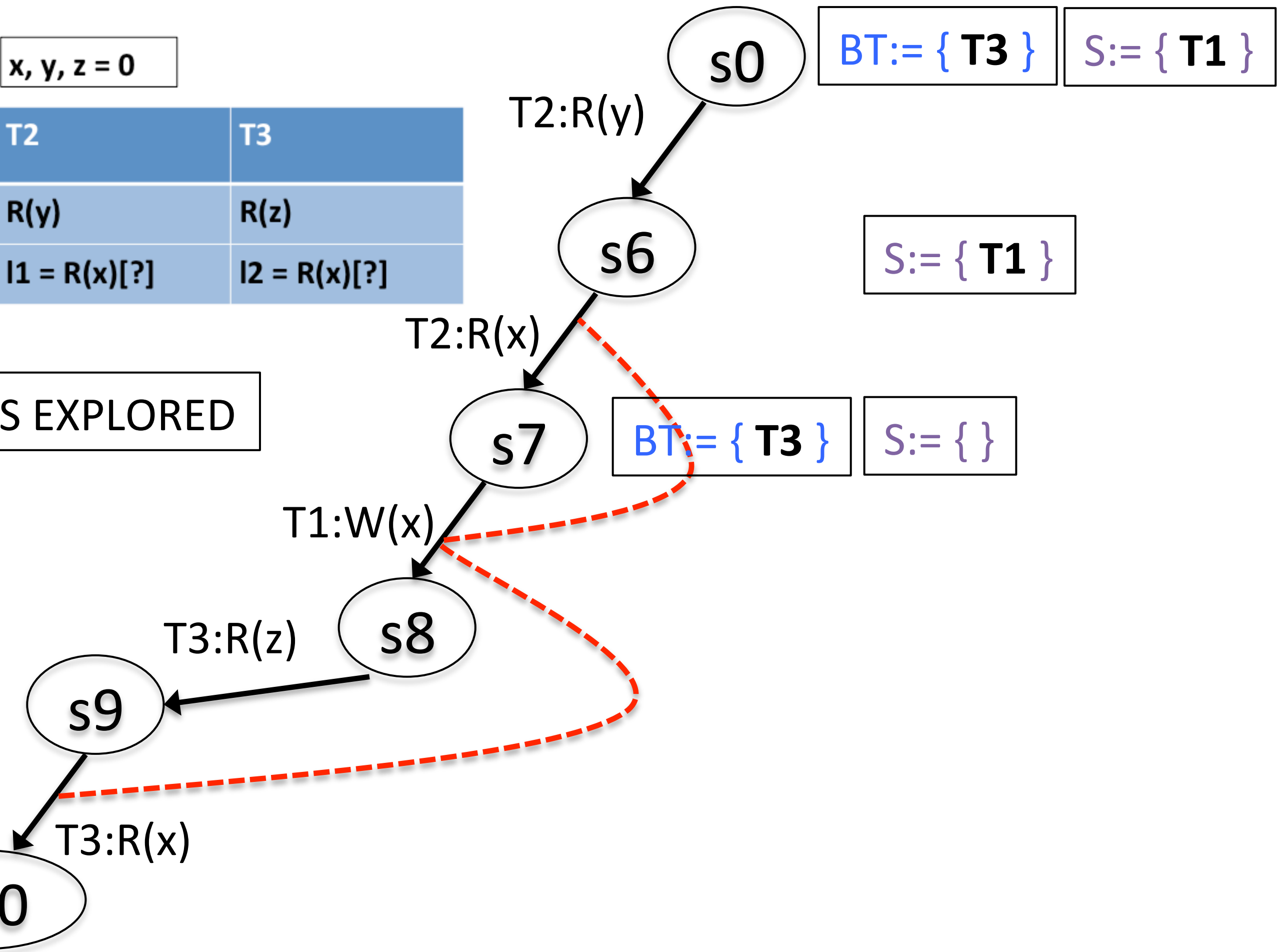
DPOR Example

x, y, z = 0

T1	T2	T3
W(x)[1]	R(y)	R(z)
	l1 = R(x)[?]	l2 = R(x)[?]

SEQUENCES EXPLORED

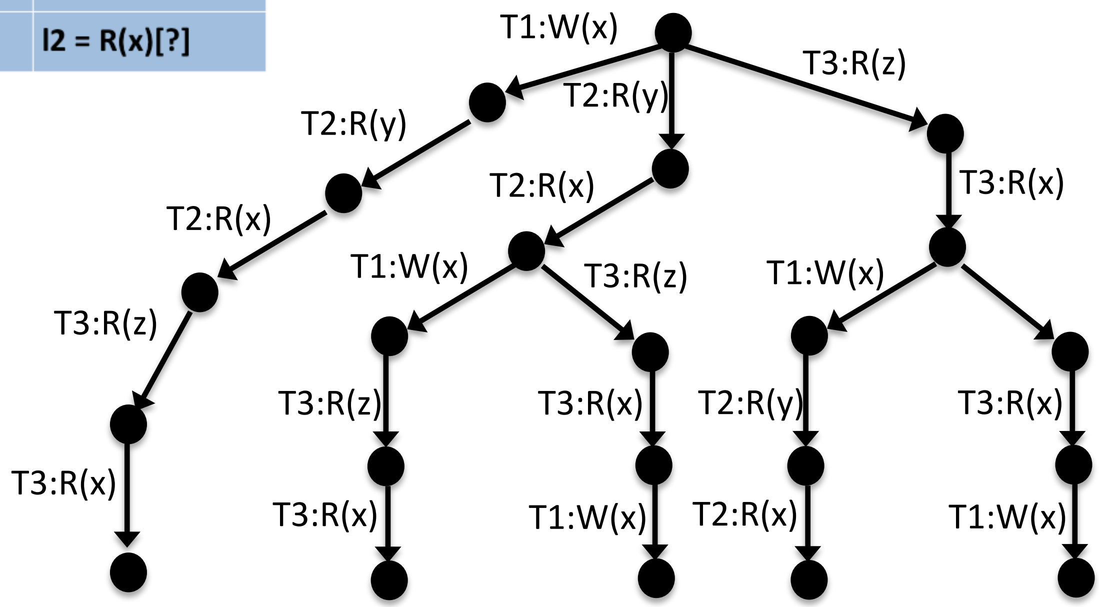
T1:W(x)
T2:R(y)
T2:R(x)
T3:R(z)
T3:R(x)



DPOR Example

x, y, z = 0

T1	T2	T3
W(x)[1]	R(y)	R(z)
	l1 = R(x)[?]	l2 = R(x)[?]



DPOR Algorithm

Initially: Explore($\emptyset$);

Explore(S) {
    let $s = last(S)$;
    for all processes p {
        if $\exists i = max(\{i \in dom(S) \mid S_i$ is dependent and may be co-enabled with $next(s,p)$ and $i \not\to_S p\})$ {
            let $E = \{q \in enabled(pre(S,i)) \mid q = p$ or $\exists j \in dom(S) : j > i$ and $q = proc(S_j)$ and $j \to_S p\}$;
            if $(E \neq \emptyset)$ then add any $q \in E$ to $backtrack(pre(S,i))$;
                else add all $q \in enabled(pre(S,i))$ to $backtrack(pre(S,i))$;
        }
    }
    if $(\exists p \in enabled(s))$ {
        $backtrack(s) := \{p\}$;
        let $done = \emptyset$;
        while $(\exists\, p \in (backtrack(s) \setminus done))$ {
            add p to $done$;
            Explore($S.next(s,p)$);
        }
    }
}

Correctness of DPOR Alg.

Thm1: Whenever a state s reached after a transition sequence E is backtracked, during the search performed by the Alg. in an acyclic state space, the post-condition of Explore(E) is satisfied, then the set of transitions explored from s is a persistent set in s.

How is HB Computed?

Using **clock vectors**.

- $CV : \mathcal{P} \to \mathcal{N}$
- For $p_i, C(p_i) \in CV = \langle c_1, c_2, \cdots, c_m \rangle$ where c_j is the index of the last transition of p_j that p_i knows.
- More generally, $i \to p$ iff $i < C(p)(proc(e_i))$

Subsequent DPOR Work

- Stateful DPOR [SPIN 2007]
- Distributed DPOR [SPIN 2008, RV 2012]