# Design and Analysis of Communication Software

## Part 4:

## Inside VeriSoft

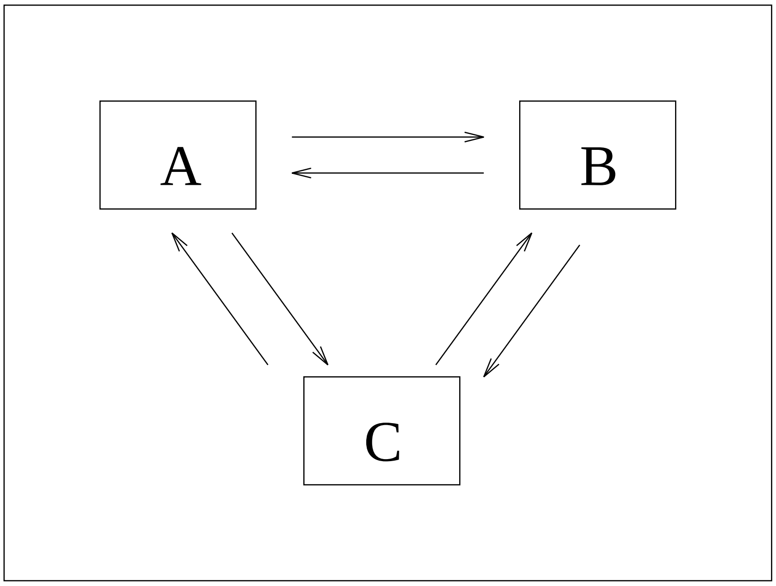
The Research Behind The Tool

# Overview

**Part 4.1:** (Oct 14)

- Introduction to Partial Order Methods
- Concurrent Systems and Dynamic Semantics
- Using Partial Orders to Tackle State Explosion
- Towards More Independence
- Persistent Sets
- How to Compute Persistent Sets
- Discussion

# Concurrent Reactive System

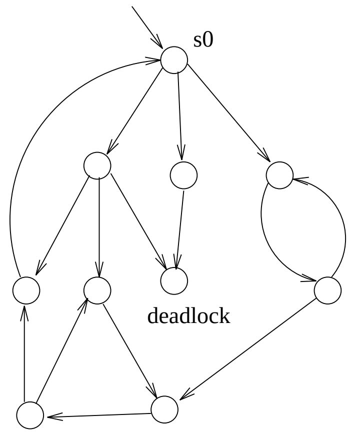



Each component is viewed as a “reactive” system, i.e., a system that continuously interacts with its environment.

**Examples:** network protocols, interconnected electronic circuits, telephone switches, plant controllers, flight control systems, etc.

**Developing, testing and debugging concurrent reactive systems is hard!**

# Systematic State-Space Exploration




The joint global behavior of all processes can be represented by a global transition system $A_G$ called *global state space*.

If $A_G$ is finite, verification of properties can be done by exploring $A_G$.

**Example of property:** absence of deadlocks

**Note:** verification means to check that *all* possible behaviors of the system satisfy a property.

# The State-Explosion Problem

## Example:

Initially: v1=v2= ... =vn=0

| Process 1 | Process 2 | | Process n |
|---|---|---|---|
| → s1: v1:=1; | → s2: v2:=1; | — — — | → sn: vn:=1; |
| s1': stop | s2': stop | | sn': stop |

→ n! interleavings

$2^n$ states

**→ State Explosion**

**→ Motivation of this work:**

to develop verification methods that avoid the part of the combinatorial explosion due to the modeling of concurrency by interleaving

**→ Partial-Order Methods**

# Partial-Order Methods: Basic Idea

Exploring *all* interleavings of concurrent transitions is not a priori necessary for verification.

Indeed, interleavings corresponding to the same concurrent execution contain related information:

- concurrent independent transitions should be left unordered, and
- concurrent executions should be viewed as *partial orders* of occurrences of transitions.

Hence, it should be sufficient to explore only some of these interleavings for the verification of most properties $\varphi$.

Which interleavings are required may depend on $\varphi$.

$\leadsto$ explore only a part $A_R$ of $A_G$ such that:

$$A_G \text{ sat } \varphi \text{ iff } A_R \text{ sat } \varphi$$

How to generate $A_R$?

# Concurrent Reactive System

We assume that a concurrent reactive system is composed of:

- finite set of *processes* executing aribitrary code (e.g., C, C++, Java, Tcl, ...)
- finite set of *communication objects* (e.g., message queues, semaphores, shared memory, TCP connections, UDP packets,...).

Each process $P \in \mathcal{P}$ executes a sequence of *operations* described in a sequential program.

Such programs are *deterministic*: every execution of the program on the same data performs the same sequence of operations.

Processes communicate by executing *operations* on communication objects.

# Concurrent Reactive System (Continued)

A communication object $O \in \mathcal{O}$ is defined by a pair $(V, OP)$, where $V$ is the set of all possible values for the object (its domain), and $OP$ is the set of *operations* that can be performed on the object.

Each operation $op_i \in OP$ is a (possibly partial) function $IN_i \times V \rightarrow OUT_i \times V$, where $IN_i$ and $OUT_i$ represent respectively the set of possible inputs and outputs of the operation.

**Example:** Consider an object "boolean variable" whose domain $V$ is the set $\{0, 1\}$. We define two operations on this object.

- A $Read$ operation for which the set $IN$ is $\{-\}$, and the set $OUT$ is $\{0, 1\}$. A $Read$ operation is always defined, and its effect is defined by $Read(-, v) \rightarrow (v, v)$, for all $v \in \{0, 1\}$.

- A $Write$ operation for which the set $IN$ is $\{0, 1\}$, and the set $OUT$ is $\{-\}$. A $Write$ operation is always defined, and its effect is defined by $Write(v', v) \rightarrow (-, v')$, for all $v, v' \in \{0, 1\}$.

# Concurrent Reactive System (Continued)

We assume that operations on communication objects are executed atomically: namely, no process can execute an operation on a given communication object while another process is currently doing so.

Operations on communication objects are *visible*, other operations are *invisible*.

The execution of an operation is said to be *blocking* if it cannot be completed.

Only executions of visible operations may be *blocking*.

At any time, the concurrent system is said to be in a *state*.

The system is said to be in a *global state* when the next operation to be executed by *every* process is *visible*.

Initially, after the creation of all the processes of the system, we assume that all the processes eventually executes a visible operation, and hence that the system may reach a first and unique global state $s_0$, called the *initial global state* of the system.

We define a *process transition*, or *transition* for short, as one visible operation followed by a *finite* sequence of invisible operations performed by a single process.

The set of operations executed during a transition $t$ is denoted by $used(t)$.

Let $\mathcal{T}$ denote the set of all transitions of the system.

# Concurrent Reactive System (Continued)

A transition is said to be *disabled* in a global state $s$ when the execution of its visible operation is blocking in $s$.

Otherwise, the transition is said to be *enabled* in $s$.

A transition $t$ that is enabled in a global state $s$ can be *executed* from $s$.

Since the number of invisible operations in a transition is finite, the execution of an enabled transition always terminates.

When the execution of $t$ from $s$ is completed, the system reaches a unique global state $s'$, called the *successor* of $s$ by $t$.

We write $s \xrightarrow{t} s'$ ($s \stackrel{w}{\Rightarrow} s'$) to mean that the execution of the transition $t$ (sequence of transition $w$) leads from the global state $s$ to the global state $s'$.

If $s \stackrel{w}{\Rightarrow} s'$, $s'$ is said to be *reachable* from $s$.

# Dynamic Semantics

A concurrent system as defined here is a closed system: from its initial global state, it can evolve and change its state by executing enabled transitions.

Therefore, a very natural way to describe the possible *behaviors* of such a system is to consider its set of reachable global states and the transitions that are possible between these.

Formally, the joint *global* behavior of all processes $P_i$ in a concurrent system is represented by a transition system $A_G = (S, \Delta, s_0)$ such that

- $S$ is the set of global states of the system,
- $\Delta \subseteq S \times S$ is the *transition relation* defined as follows:

$$(s, s') \in \Delta \text{ iff } \exists t \in \mathcal{T} : s \xrightarrow{t} s',$$

- $s_0$ is the initial global state of the system.

An element of $\Delta$ corresponds to the execution of a single transition $t \in \mathcal{T}$ of the system.

The elements of $\Delta$ will be referred to as *global transitions*.

$A_G$ is called the *global state space* of the system.

# Computing $A_G$

Traditionally, when representing and storing states is possible, computing $A_G$ can be done as follows.

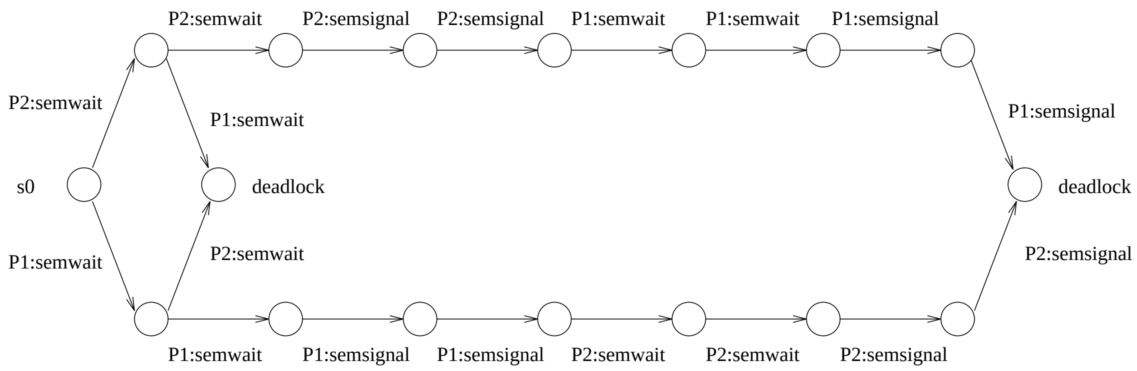
```
Initialize: Set is empty; H is empty;
            add s0 to Set;
Loop:  while Set ≠ ∅ do {
            take s out of Set;
            if s is NOT already in H then {
               enter s in H;
               T = enabled(s);
               for all t in T do {
                    s' = succ(s) after t;
                    add s' to Set;
                    }
               }
            }
```

When states cannot be explicitly represented and stored, a *state-less search* can be done. See later.

# Example

```
/* phil.c : dining philosophers (version without loops) */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define N 2

philosopher(i)
     int i;
{
  printf("philosopher %d thinks\n",i);
  semwait(i);           /* take left fork */
  semwait((i+1)%N);     /* take right fork */
  printf("philosopher %d eats\n",i);
  semsignal(i);         /* release left fork */
  semsignal((i+1)%N);   /* release right fork */
  exit(0);
}

main()
{
  int semid, i, pid;

  semid = semget(IPC_PRIVATE,N,0600);

  for(i=0;i<N;i++)
    semsetval(i,1);
  for(i=0;i<(N-1);i++) {
    if((pid=fork()) == 0)
      philosopher(i);
    };
  philosopher(i);
}
```

# Example (Continued)




The operation *exit* is a visible operation whose execution is always blocking.

Since all the processes are deterministic, nondeterminism in $A_G$ is caused only by concurrency.

**Note:** "VS_toss($n$)" returns any integer in $[0, n]$.

VS_toss is visible and *nondeterministic*: the execution of VS_toss($n$) may yield up to $n+1$ different successor states (define $n+1$ process transitions).

**Note:** "VS_assert(expression)" is visible and evaluates its argument. If evaluation returns "false", the assertion is *violated*.

# Theorem

**Thm: [G97]** Consider a concurrent system as defined above, and let $A_G$ denote its state space. Then, all the deadlocks that are reachable after the initialization of the system are global states, and are therefore in $A_G$. Moreover, if there exists a state reachable after the initialization of the system where an assertion is violated, then there exists a global state in $A_G$ where the same assertion is violated.

**Proof:** Since only visible operations may be blocking, deadlocks are global states.

Let $s$ be a reachable state where assertion $a$ is violated.

Let $P_i$ be the process about to execute $a$ in $s$.

Then, $a$ is also violated in the global state reachable from $s$ where all the processes $P_j$, $j \neq i$, are about to execute their next visible operation reachable from $s$.

Indeed, the execution of invisible operations may not change the value of any variable local to another process.

## Remark

More general representation of concurrent systems are also possible.

- Each process could have more than one transition enabled.
- Processes could synchronize on joint transitions (multi-way rendez-vous).
- Process transitions could execute more than one operation on communication objects.

See [G96].

# Using Partial orders to Tackle State Explosion

Even if $A_G$ is finite, it can be huge!

All interleavings of all concurrent transitions of the system are represented in $A_G$: we will show this is not necessary.

The intuition behind the methods presented here is that concurrent executions are really partial orders where concurrent "independent" transitions should be left unordered.

Intuitively, transitions are independent when the order of their occurrence is irrelevant.

# The notion of Independent Transitions

Formally, we have (adapted from [KP92]):

**Definition.**
Let $\mathcal{T}$ be the set of process transitions and $D \subseteq \mathcal{T} \times \mathcal{T}$ be a binary, reflexive, and symmetric relation. The relation $D$ is a *valid dependency relation* iff for all $t_1, t_2 \in \mathcal{T}$, $(t_1, t_2) \notin D$ ($t_1$ and $t_2$ are independent) implies that the two following properties hold for all global states $s \in S$ of $A_G$:

1. if $t_1$ is enabled in $s$ and $s \xrightarrow{t_1} s'$, then $t_2$ is enabled in $s$ iff $t_2$ is enabled in $s'$ (independent transitions can neither disable nor enable each other); and

2. if $t_1$ and $t_2$ are enabled in $s$, then there is a unique state $s'$ such that $s \overset{t_1 t_2}{\Rightarrow} s'$ and $s \overset{t_2 t_1}{\Rightarrow} s'$ (commutativity of enabled independent transitions).

This definition characterizes the properties of possible "valid" dependency relations for the transitions of a given system.

It is not practical to check these two properties for all pairs of transitions in all states $s$ in $A_G$.

In practice, it is possible to give easily checkable *syntactic* conditions that are *sufficient* for transitions to be *independent*.

For instance, in our context,
a *sufficient* syntactic condition for transitions $t_1$ and $t_2$ in $\mathcal{T}$ to be independent is that:

1. the process executing $t_1$ is different from the process executing $t_2$, *and*

2. the set of objects that are accessed by $t_1$ is disjoint from the set of objects that are accessed by $t_2$.

# Using Independence for Defining Equivalence of Transition Sequences

- Two sequences of transitions are *equivalent* if they can be obtained from each other by successively permuting adjacent independent transitions.
- *Equivalence classes* are called Mazurkiewicz's traces. The trace containing a sequence of transitions $w$ will be denoted $[w]$.
- The set of transition sequences in $[w]$ correspond to the set of linearizations (interleavings) of a partial order of transition occurrences.

**Example.**
Consider $\mathcal{T} = \{t_1, t_2, t_3\}$ , and $D =$ $\{(t_1,t_1),(t_2,t_2),(t_3,t_3),(t_1,t_2),(t_2,t_1),(t_1,t_3),(t_3,t_1)\}$.
Then, the sequence $w = t_1t_2t_3t_1$ defines the trace $[w] = \{t_1t_2t_3t_1, t_1t_3t_2t_1\}$.

$$\begin{tikzcd} & t_2 \arrow[dr] & \\ t_1 \arrow[ur] \arrow[dr] & & t_1 \\ & t_3 \arrow[ur] & \end{tikzcd}$$

**Background:** A relation $R \subseteq A \times A$ on a set $A$ that is reflexive, antisymmetric, and transitive is called a *partial order*.

A partial order $R \subseteq A \times A$ is also a *total order* if, for all $a_1, a_2 \in A$, either $(a_1, a_2) \in R$ or $(a_2, a_1) \in R$.

# Basic Property of Traces

**Theorem.**
Let $s$ be a state in $A_G$. If $s \overset{w_1}{\Rightarrow} s_1$ and $s \overset{w_2}{\Rightarrow} s_2$ in $A_G$, and if $[w_1] = [w_2]$, then $s_1 = s_2$.

**Proof.**
By definition, all $w' \in [w]$ can be obtained from $w$ by successively permuting pairs of *adjacent* independent transitions. It is thus sufficient to prove that, for any two words $w_1$ and $w_2$ that differ only by the order of *two* adjacent independent transitions, if $s \overset{w_1}{\Rightarrow} s'$ then $s \overset{w_2}{\Rightarrow} s'$.

Let us thus assume that $w = t_1 \ldots ab \ldots t_n$ and $w' = t_1 \ldots ba \ldots t_n$. We have

$$s \overset{t_1}{\rightarrow} s_1 \overset{t_2}{\rightarrow} s_2 \ldots \overset{t_i}{\rightarrow} s_i \overset{a}{\rightarrow} s_{i+1} \overset{b}{\rightarrow} s_{i+2} \ldots \overset{t_n}{\rightarrow} s_n$$

and

$$s \overset{t_1}{\rightarrow} s_1 \overset{t_2}{\rightarrow} s_2 \ldots \overset{t_i}{\rightarrow} s_i \overset{b}{\rightarrow} s'_{i+1} \overset{a}{\rightarrow} s'_{i+2} \ldots \overset{t_n}{\rightarrow} s'_n.$$

Since $a$ and $b$ are independent, it follows that $s_{i+2} = s'_{i+2}$. Since the transitions in $w_1$ from $s_{i+2}$ and the transitions in $w_2$ from $s'_{i+2}$ are identical, we have $s_n = s'_n$.

**Note.** To determine if a state is reachable by a trace, it is sufficient to explore *only one* transition sequence in this trace!

Thus, for deadlock detection, it is sufficient to explore *only one* interleaving per trace:

- for all deadlocks $d$ : $\exists w : s_0 \stackrel{w}{\Rightarrow} d$
- $\forall w' \in [w] : s_0 \stackrel{w'}{\Rightarrow} d$

## How to explore one interleaving per trace?
## Selective Search

- **Idea:**
  Perform a classical search,
  but instead of executing systematically *all* enabled transitions in each state encountered during the search, execute *only some* of them.

- Only a subset of the global state space is explored.

- Which transitions should be selected?
  **Persistent sets** and **Sleep sets**

# Towards More Independency

**Definition.** Let $\mathcal{T}$ be the set of transitions in a system. Two transitions $t_1, t_2 \in \mathcal{T}$ are *independent* if:

1. the process executing $t_1$ is different from the process executing $t_2$, and
2. $\forall op_1 \in used(t_1)$ and $\forall op_2 \in used(t_2)$, if $op_1$ and $op_2$ are two operations on a same object, then $op_1$ and $op_2$ are independent.

**Definition.** Let $O = (V, OP)$ be an object, and $D_O \subseteq OP \times OP$ be a binary and symmetric relation. The relation $D_O$ is a *valid dependency relation* for $O$ iff for all $op_1, op_2 \in OP$, $(op_1, op_2) \notin D_O$ ($op_1$ and $op_2$ are independent) implies that the two following properties hold for all values $v \in V$, and for all inputs $in_1$ and $in_2$:

1. if $op_1(in_1, v)$ is defined, with $op_1(in_1, v) \rightarrow (out_1, v'_1)$, then $op_2(in_2, v)$ is defined iff $op_2(in_2, v'_1)$ is defined; and
2. if $op_1(in_1, v)$ and $op_2(in_2, v)$ are defined, then $\exists out_1, out_2, v'_1, v'_2, v''$ such that:
   - $op_1(in_1, v) \rightarrow (out_1, v'_1)$ and $op_2(in_2, v'_1) \rightarrow (out_2, v'')$; and
   - $op_2(in_2, v) \rightarrow (out_2, v'_2)$ and $op_1(in_1, v'_2) \rightarrow (out_1, v'')$

   (commutativity of operations, together with preservation of the outputs).

**Example:**
Consider an object "boolean variable" whose domain $V$ is the set $\{0,1\}$. We define two operations on this object.

- A $Read$ operation for which the set $IN$ is $\{-\}$, and the set $OUT$ is $\{0,1\}$. A $Read$ operation is always defined, and its effect is defined by $Read(-,v) \rightarrow (v,v)$, for all $v \in \{0,1\}$.

- A $Write$ operation for which the set $IN$ is $\{0,1\}$, and the set $OUT$ is $\{-\}$. A $Write$ operation is always defined, and its effect is defined by $Write(v',v) \rightarrow (-,v')$, for all $v,v' \in \{0,1\}$.

A valid dependency relation between the operations on this object is:

| DEP. | $Write$ | $Read$ |
|---|---|---|
| $Write$ | $+$ | $+$ |
| $Read$ | $+$ | $-$ |

# Refining Dependencies between Operations

**1. By refining the operations themselves.**

**Example:** object representing a boolean value.

Add the operation $Compl$ such that $Compl(-,0) \rightarrow (-,1)$ and $Compl(-,1) \rightarrow (-,0)$ (always defined).

| DEP. | $Write$ | $Compl$ | $Read$ |
|---|---|---|---|
| $Write$ | $+$ | $+$ | $+$ |
| $Compl$ | $+$ | $-$ | $+$ |
| $Read$ | $+$ | $+$ | $-$ |

The new dependency relation may yield less dependencies between the transitions of the system. It is thus preferable to use $Compl$ rather than $Write$ whenever possible.

# Refining Dependencies between Operations

## 2. By using conditional dependency.

**Definition.**
Let $\mathcal{T}$ be the set of transitions in a systen and $D \subseteq \mathcal{T} \times \mathcal{T} \times S$. The relation $D$ is a *valid conditional dependency relation* for the system iff for all $t_1, t_2 \in \mathcal{T}, s \in S$, $(t_1, t_2, s) \notin D$ ($t_1$ and $t_2$ are independent in $s$) implies that $(t_2, t_1, s) \notin D$ and that the two following properties hold in state $s$:

1. if $t_1$ is enabled in $s$ and $s \overset{t_1}{\rightarrow} s'$, then $t_2$ is enabled in $s$ iff $t_2$ is enabled in $s'$ (independent transitions can neither disable nor enable each other); and
2. if $t_1$ and $t_2$ are enabled in $s$, then there is a unique state $s'$ such that $s \overset{t_1 t_2}{\Rightarrow} s'$ and $s \overset{t_2 t_1}{\Rightarrow} s'$ (commutativity of enabled independent transitions).

Idem for conditional dependency between operations.

In what follows, two operations on an object $O_j$ will be said to be independent in state $s$ iff they are independent for the value $v \in V_j$ of the object $O_j$ in state $s$.

**Example.**
Consider an object representing a bounded FIFO channel (buffer) of size $N$. The domain $V$ of this object is $\{\emptyset\} \cup M \cup M^2 \cup \ldots \cup M^N$, where $M$ is the set of messages that can be transmitted via the channel. We define three operations $Send$, $Receive$ and $Length$ on this object such that:

- $Send(v, v_1 v_2 \ldots v_n) \rightarrow (-, v_1 v_2 \ldots v_n v)$ defined if $n < N$ and $v \in M$,
- $Receive(-, v_1 v_2 \ldots v_n) \rightarrow (v_1, v_2 \ldots v_n)$ defined if $n > 0$,
- $Length(-, v_1 v_2 \ldots v_n) \rightarrow (n, v_1 v_2 \ldots v_n)$ always defined.

The following tables give respectively a constant and a conditional dependency relation between these operations.

| DEP. | $Send$ | $Receive$ | $Length$ |
|---|---|---|---|
| $Send$ | $+$ | $+$ | $+$ |
| $Receive$ | $+$ | $+$ | $+$ |
| $Length$ | $+$ | $+$ | $-$ |

| DEP. | $Send$ | $Receive$ | $Length$ |
|---|---|---|---|
| $Send$ | $n < N$ | $n = 0$ or $n = N$ | $n < N$ |
| $Receive$ | $n = 0$ or $n = N$ | $n > 0$ | $n > 0$ |
| $Length$ | $n < N$ | $n > 0$ | $-$ |

Thanks to conditional dependency, operations that are dependent for some but not all values $v \in V$ are no more considered as being dependent for all values.

We can still reduce dependencies between operations by simultaneously refining the operations and by using conditional dependency.

**Example.**
Consider the previous example. In real protocol models, the operation $Length$ is often used to test if a channel is empty or full [GP93]. Let us introduce two new operations $Empty$ and $Full$ defined as follows:

- $Empty(-, v_1 v_2 \ldots v_n) \rightarrow$ (if $(n = 0)$ then $true$ else $false$, $v_1 v_2 \ldots v_n$) always defined.
- $Full(-, v_1 v_2 \ldots v_n) \rightarrow$ ( if $(n = N)$ then $true$ else $false$, $v_1 v_2 \ldots v_n$) always defined.

A new dependency relation can then be defined:

| DEP. | $Send$ | $Receive$ | $Length$ | $Empty$ | $Full$ |
|---|---|---|---|---|---|
| $Send$ | $n < N$ | $n = 0$ or $n = N$ | $n < N$ | $n = 0$ | $n = N - 1$ |
| $Receive$ | $n = 0$ or $n = N$ | $n > 0$ | $n > 0$ | $n = 1$ | $n = N$ |
| $Length$ | $n < N$ | $n > 0$ | – | – | – |
| $Empty$ | $n = 0$ | $n = 1$ | – | – | – |
| $Full$ | $n = N - 1$ | $n = N$ | – | – | – |

# Summary

In practice, for each type of (communication) objects, a valid conditional dependency relation between all possible operations on the object is given.

Then, a valid conditional dependency relation between the transitions of a system can be defined from valid conditional dependency relations between operations on objects as follows.

**Definition.**
Let $\mathcal{T}$ be the set of transitions in a system. Two transitions $t_1, t_2 \in \mathcal{T}$ are *independent* in state $s \in S$ if:

1. the process executing $t_1$ is different from the process executing $t_2$, and

2. $\forall op_1 \in used(t_1)$ and $\forall op_2 \in used(t_2)$, if $op_1$ and $op_2$ are two operations on the same object, then $op_1$ and $op_2$ are independent in $s$.

This valid conditional dependency relation determines the dependencies between all the transitions of the system.

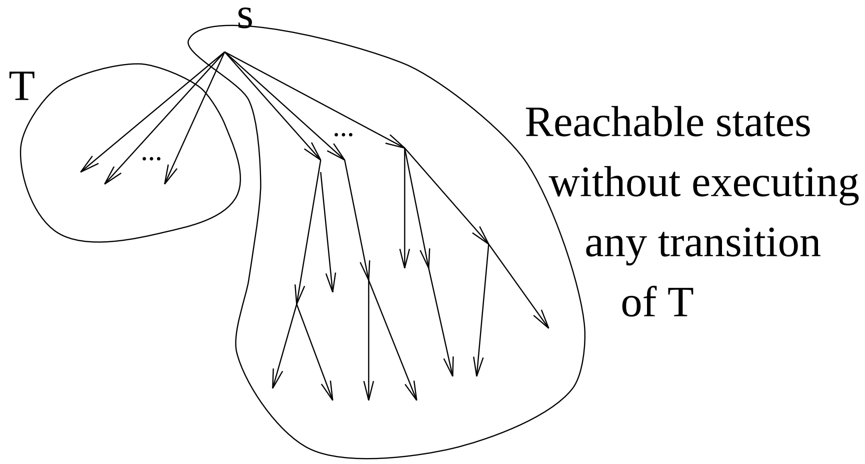
# Persistent Sets [GP93]

Intuitively, a set $T$ of enabled transitions are *persistent in* $s$ if whatever one does from $s$, while remaining outside of $T$, does not interact with or affect $T$.

**Definition:**
A set $T$ of transitions enabled in a state $s$ is *persistent in* $s$ iff, for all nonempty sequences of transitions

$$s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$$

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



# Persistent-Set Selective Search

```
Initialize: Set is empty; H is empty;
            add s0 to Set;
Loop:  while Set ≠ ∅ do {
            take s out of Set;
            if s is NOT already in H then {
               enter s in H;
               T = Persistent_Set(s);
               for all t in T do {
                    s' = succ(s) after t;
                    add s' to Set;
                    }
               }
            }
```

# Persistent Sets Preserve Deadlocks

**Theorem.**
Let $A_R$ be the reduced state-space explored by a persistent-set selective search. Let $s$ be a state in $A_R$, and let $d$ be a deadlock reachable from $s$ in $A_G$ by a sequence $w$ of transitions. Then, $d$ is also reachable from $s$ in $A_R$.

**Proof.** Induction on the length of $w$.
$|w| = 0$: Immediate.

$|w| = n + 1$: thus $w = tw'$ with $|w'| = n$ and $s \xrightarrow{t} s' \xRightarrow{w'} d$.

- $t$ is selected: immediate.
- $t$ not selected: use the fact that unselected transitions are independent w. r. t. selected transitions.
  - Some selected transition appears in $w'$ (if not, no deadlock). Let $t'$ be the first of these: $w' = w_1 t' w_2$.
  - $t$ and all transitions in $w_1$ are independent with respect to $t'$ since none of these is in the selected set.
    One has: $[tw_1t'w_2] = [t'tw_1w_2]$.
  - Thus $s \xrightarrow{t'} s'' \xRightarrow{tw_1w_2} d$ and the result follows by the inductive hypothesis ($|tw_1w_2| = n$).

# Computing Persistent Sets

Several algorithms have been proposed (somewhat independently):

- "Conflicting Transitions" [GW91]
- "Overman's Algorithm" [O81]
- "Stubborn Sets" [V88-91]
- "Conditional Stubborn Sets" [GP93]

In [G96], it is shown that they all compute persistent sets.

- All these algorithms infer the persistent sets from the structure of the system being verified.
- Algorithms can be more or less complex/expensive depending on how much information about the structure of the system they use.
- The heuristic goal is to obtain a (nonempty) persistent set that is as small as possible.
- The set of all enabled transitions in a state $s$ is persistent in $s$: it is not possible to move from $s$ without executing a transition in the set.

# Computing Persistent Sets
# General Strategy

- $T = \{t\}$ start with a set that contains a single enabled transition.
- **Repeat**
  - Add all transitions that "can interfere" with some transition in $T$.
- **Until** no more transition need be added.

Worst case: return all enabled transitions.

# Can Interfere?

"Can interfere" (on operations or transitions) can be defined using:

**1. can-be-dependent:**([GW91])

Two operations $op_1$ and $op_2$ on a same object *can-be-dependent* if there exists a state $s$ in $S$ such that $op_1$ and $op_2$ are dependent in $s$.

**Example:** bounded FIFO channel of size $N$

| can-be-dependent | $Send$ | $Receive$ | $Full$ |
|---:|---|---|---|
| $Send$ | $+$ | $+$ | $+$ |
| $Receive$ | $+$ | $+$ | $+$ |
| $Full$ | $+$ | $+$ | $-$ |

**2. do-not-accord:** ([V88-91])

Two operations $op_1$ and $op_2$ on the same object *do-not-accord* with each other if there exists a state $s$ in $S$ such that $op_1$ and $op_2$ are defined in $s$ and are dependent in $s$.

**Example:** bounded FIFO channel of size $N$

| do-not-accord | $Send$ | $Receive$ | $Full$ |
|---:|---|---|---|
| $Send$ | $+$ | $-$ | $+$ |
| $Receive$ | $-$ | $+$ | $+$ |
| $Full$ | $+$ | $+$ | $-$ |

# Can Interfere?

### 3. Definition of $\not\triangleright_s$: ([GP93])

The relation $op_1 \not\triangleright_s op_2$ holds if there exists a sequence $s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$ of transitions from $s$ in $A_G$ such that $\forall 1 \leq i < n : \forall op$ on $O$ used by $t_i$: $op_1$ and $op$ are independent in state $s_i$, $t_n$ uses $op_2$, and $op_1$ and $op_2$ are dependent in $s_n$.

**Example:** bounded FIFO channel of size $N$

| $\not\triangleright_s$ | $Send$ | $Receive$ | $Full$ |
|---|---|---|---|
| $Send$ | $n < N$ | $n = N$ | $n = N - 1$ |
| $Receive$ | $n = 0$ | $n > 0$ | $n > 0$ |
| $Full$ | $n < N$ | $n = N$ | $-$ |

**Comparison:**

in states $s$ where both $op_1$ and $op$ are enabled, $\not\triangleright_s \subseteq$ do-not-accord $\subseteq$ can-be-dependent.

Hence $\not\triangleright_s$ models more finely the possible interactions between operations on a given object (see [G96]).

**Note:**

$\not\triangleright_s$ tells also what to select when $op_1$ and $op$ are not both enabled.

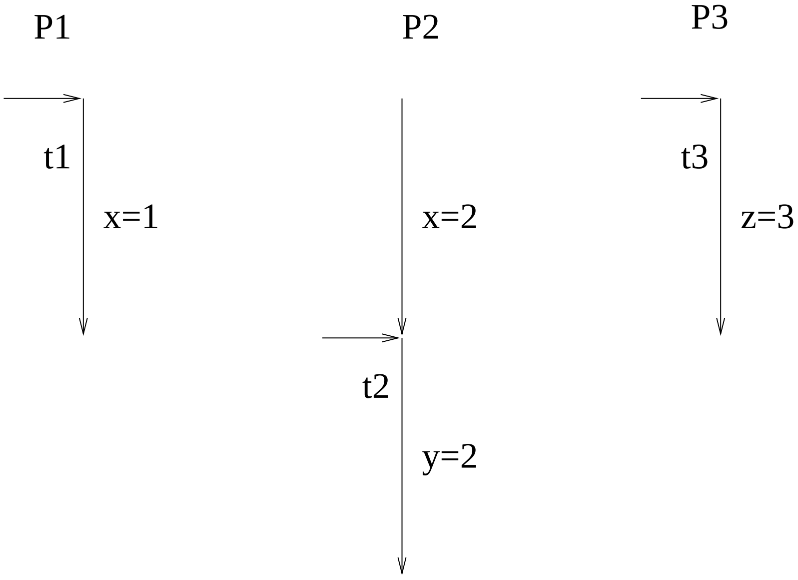
Example: if $Send$ and $n = N$ (thus $Send$ is disabled), select $Receive$.

**note:** $\not\triangleright_s$ is not necessarily symmetric!

# Computing Persistent Sets
# First Strategy: Only Consider Enabled Transitions

- $T = \{t\}$ start with a set that contains a single enabled transition.
- **Repeat**
  - For each transition $t \in T$, for each transition $t'$ such that $t \mid\!\triangleright_s t'$:
    * if $t'$ is enabled, add it to $T$;
    * otherwise, stop and select all transitions.
- **Until** no more transition need be added.

# Example




| $\triangleright_s$ | $Write$ | $Read$ |
|---|---|---|
| $Write$ | $+$ | $+$ |
| $Read$ | $+$ | $-$ |

$$Algo_1(t_1) = \{t_1, t_2, t_3\}$$

# Complexity

For a given state $s$, let $Algo_1(t)$ denote the persistent set that is returned by Algorithm 1 when $t$ is the enabled transition chosen in step 1 of the algorithm.

Assume that, from any transition $t$, it takes $O(1)$ time to obtain a transition $t'$ satisfying $t \triangleright_s t'$.

Then, the worst-case time complexity of $Algo_1(t)$ is $O(|enabled(s)|^2)$.

Let $PS_1(s)$ denote the set of persistent sets in a state $s$ that can be computed by Algorithm 1:
$PS_1(s) = \{Algo_1(t) | t \in enabled(s)\}$.

It is easy to see that

$$\forall t' \in Algo_1(t) : Algo_1(t') \subseteq Algo_1(t).$$

Therefore, it may be useful to rerun Algorithm 1 with transitions $t'$ taken from a persistent set already obtained by a previous run.

The worst-case time complexity to compute the smallest persistent set in $PS_1(s)$, let us denote it by $min(PS_1(s))$, is also $O(|enabled(s)|^2)$.

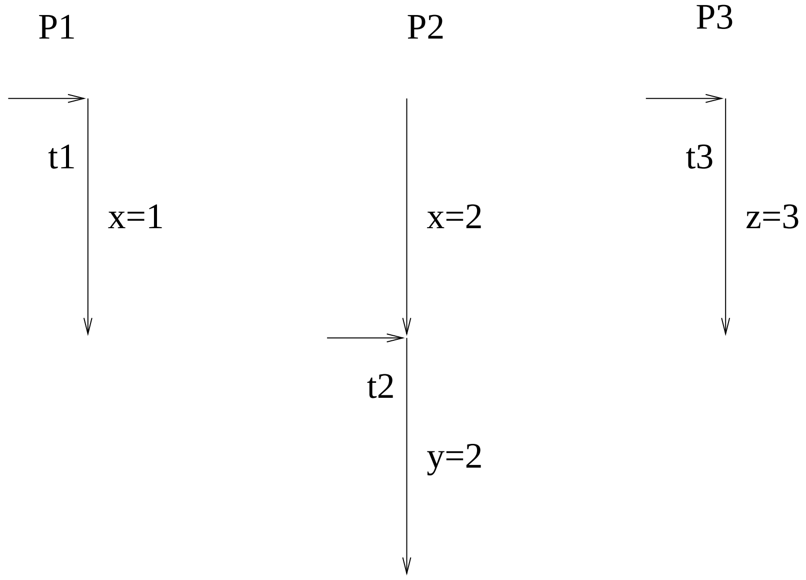
**Note:** when $\triangleright_s$ is symmetric and $Algo_1(t)$ did not encounter any disabled transitions, we have

$$\forall t' \in Algo_1(t) : Algo_1(t') = Algo_1(t).$$

## Computing Persistent Sets
## Second Strategy: Consider Disabled Transitions, but Ignore Process Structure

- $T = \{t\}$ start with a set that contains a single enabled transition.
- **Repeat**
  - For each transition $t \in T$, for each transition $t'$ such that $t \not\triangleright_s t'$:
    * if $t'$ is enabled, add it to $T$;
    * otherwise, add the next transition $t''$ (enabled or disabled) of the process $P$ to which $t'$ belongs (executing $t''$ could lead to enabling $t'$).
- **Until** no more transition need be added.
- Restrict $T$ to enabled transitions.

# Example




| $\not\triangleright_s$ | $Write$ | $Read$ |
|---|---|---|
| $Write$ | $+$ | $+$ |
| $Read$ | $+$ | $-$ |

$$Algo_1(t_1) = \{t_1, t_2, t_3\}$$

$$Algo_2(t_1) = \{t_1, t_2\}$$

# Complexity

For a given state $s$, let $Algo_2(t)$ denote the persistent set that is returned by Algorithm 2 when $t$ is the enabled transition chosen in step 1 of the algorithm.

Assume that, from any process $P_i$, it takes $O(1)$ time to obtain a process $P_j$ satisfying $t \ntriangleright_s t'$.

Then, the worst-case time complexity of $Algo_2(t)$ is also $O(|\mathcal{P}|^2)$.

Let $PS_2(s)$ denote the set of persistent sets in a state $s$ that can be computed by Algorithm 2:
$PS_2(s) = \{Algo_2(t) | t \in enabled(s)\}$.

It is easy to see that

$$\forall t' \in Algo_2(t) : Algo_2(t') \subseteq Algo_2(t).$$

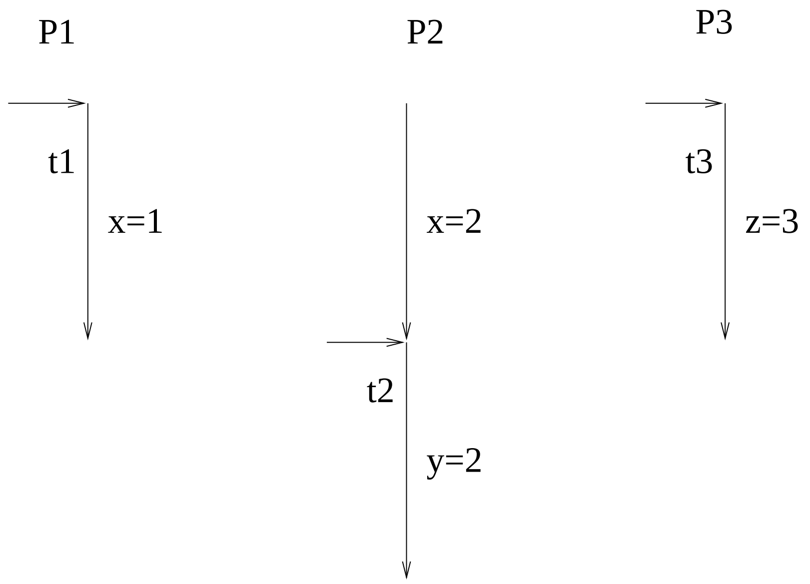
Therefore, it may be useful to rerun Algorithm 2 with transitions $t'$ taken from a persistent set already obtained by a previous run.

The worst-case time complexity to compute the smallest persistent set in $PS_2(s)$, let us denote it by $min(PS_2(s))$, is also $O(|\mathcal{P}|^2)$.

# Computing Persistent Sets Third Strategy: Consider Disabled Transitions and Process Structure

- $T = \{t\}$ start with a set that contains a single enabled transition.
- **Repeat**
  - For each transition $t \in T$:
    - if $t$ is enabled, add to $T$ every transition $t'$ such that $t \mid\!\!\triangleright_s t'$;
    - otherwise, add at least one transition $t'$ that needs to be executed for $t$ to become enabled.
- **Until** no more transition need be added.
- Restrict $T$ to enabled transitions.

# Example




| $\triangleright_s$ | $Write$ | $Read$ |
|---|---|---|
| $Write$ | $+$ | $+$ |
| $Read$ | $+$ | $-$ |

$$Algo_1(t_1) = \{t_1, t_2, t_3\}$$

$$Algo_2(t_1) = \{t_1, t_2\}$$

$$Algo_3(t_1) = \{t_1\}$$

# Complexity

For a given state $s$, let $Algo_3(t)$ denote one of the persistent sets that may be returned by Algorithm 3 when $t$ is the enabled transition chosen in step 1 of the algorithm.

Assume that, from any transition $t$, it takes $O(1)$ time to obtain a transition $t'$ satisfying either condition 2.a or 2.b.

Then, the worst-case time complexity of $Algo_3(t)$ is $O(|\mathcal{T}|^2)$.

To avoid redundant work during successive executions of Algorithm 3 when searching for a minimal persistent set, a systematic approach, investigated in [Val88a,Val88b], consists in viewing each transition in $\mathcal{T}$ as a vertex of a directed graph, and each relation of the form "if $t$ is in $T_s$, then add $t'$ to $T_s$" according to step 2.a or 2.b as an edge from vertex $t$ to vertex $t'$.

The problem of finding the smallest persistent set in $PS_3(s)$ can be solved in $O(|\mathcal{T}|^3|)$.

If the nondeterminism of step 2.b of Algorithm 3 is resolved in a unique way for each disabled transition, then the time complexity becomes linear in the number of edges in the graph, i.e., $O(|\mathcal{T}|^2|)$.

# Comparison

**Theorem.**
For all transitions $t$ that are enabled in a state $s$, we have $Algo_2(t) \subseteq Algo_1(t)$.

**Theorem.**
For all transitions $t$ that are enabled in a state $s$, there exists an execution of Algorithm 3 that returns a persistent set $Algo_3(t)$ such that $Algo_3(t) \subseteq Algo_2(t)$.

The smallest persistent set that can be computed by Algorithm $i$ can also be computed by Algorithm $j$ with $i < j$, while the converse is not true, as seen with the previous examples.

The worst-case time complexity to compute $Algo_1(t)$, $Algo_2(t)$, and $Algo_3(t)$ are, respectively, $O(|enabled(s)|^2)$, $O(|\mathcal{P}|^2)$, and $O(|\mathcal{T}|^2)$.

Clearly, the more information about the system description the algorithm uses and can exploit, the more sophisticated the algorithm is, the smaller the persistent set that it returns can be, but the larger the run-time is.

Note: persistent sets are really *what* we want to compute, while the algorithms that we have presented (including the notion of stubborn sets) rather tell us *how* to compute persistent sets.

# Discussion

Which algorithm among Algorithms 1, 2, and 3 should be used in conjunction with a relation $\triangleright_s$?

On one hand, if a persistent set $T$ in a state $s$ is a subset of another persistent set $T'$ in $s$, then the reduced state-space $A_R$ obtained by choosing $T$ in state $s$ is smaller than the reduced state-space $A'_R$ obtained by choosing $T'$ in state $s$.

On the other hand, if a persistent set $T$ in a state $s$ contains less transitions than another persistent set $T'$ in $s$, but is not a subset of $T'$, then choosing $T$ instead of $T'$ is just a heuristics.

This implies that there is no "best" algorithm for computing persistent sets since $min(PS_j)$, the smallest persistent set that can be computed by Algorithm $j$, is not necessarily included in $min(PS_i)$, $i < j$.

Therefore, in practice, the choice of a persistent-set algorithm is a trade-off between the complexity of the algorithm, its additional run-time expense, and the reduction it can yield.

This choice also depends on the type of representation of concurrent systems (some information is hard to obtain), and on the type of systems that have to be analyzed (some optimizations are useless for some classes of examples).

In VeriSoft, Algorithm 2 is used.

## Main References

On VeriSoft:

[G97] "Model Checking for Programming Languages using VeriSoft", P. Godefroid, POPL'97.

On Partial-Order Methods:

[G96] "Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem", P. Godefroid, LNCS 1032.

See `http://www.bell-labs.com/~god`