# A Theory of Dynamic Evolution in Petri Net Models of Business Processes

*Presynopsis presentation*

by

Ahana Pradhan

under the guidance of

Prof. Rushikesh K. Joshi

Department of Computer Science and Engineering, IIT Bombay

August, 2016

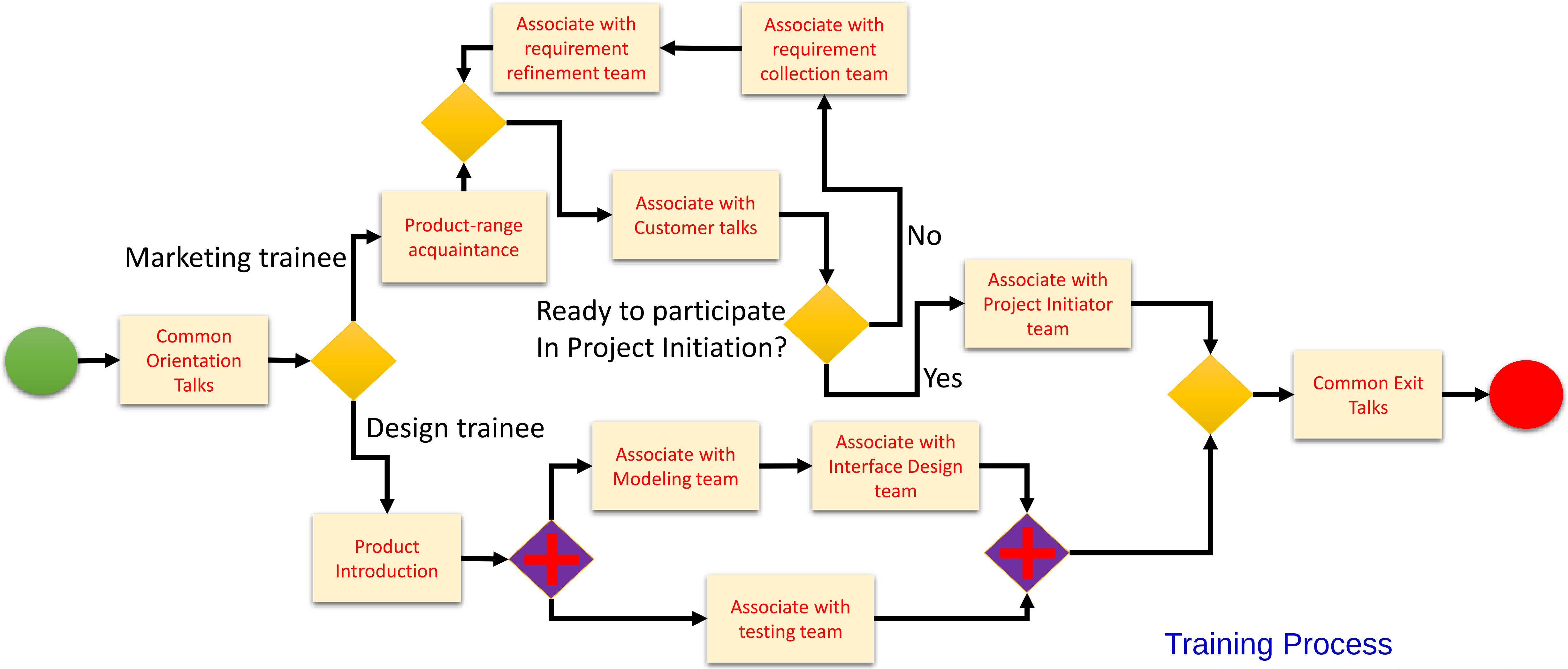
# Business Processes

- Complex Flow of activities to achieve a business goal of an Organization
- Examples:

| Domain | Business Process |
|---|---|
| Finance | Billing Process |
| Human Resource Management | Vacation Request/Approval Process |
| Banking | Account Opening Process |
| E-commerce | Product Delivery Process |
| Travel | Ticket Booking Process |
| Manufacturing | Product Assembly Process |
| Public Service | Passport Application Process |
| Academic | Admission Process |

- Activities in business process: manual tasks, user assisted automated tasks, web-services, other business processes etc.

# Business Processes Modeling

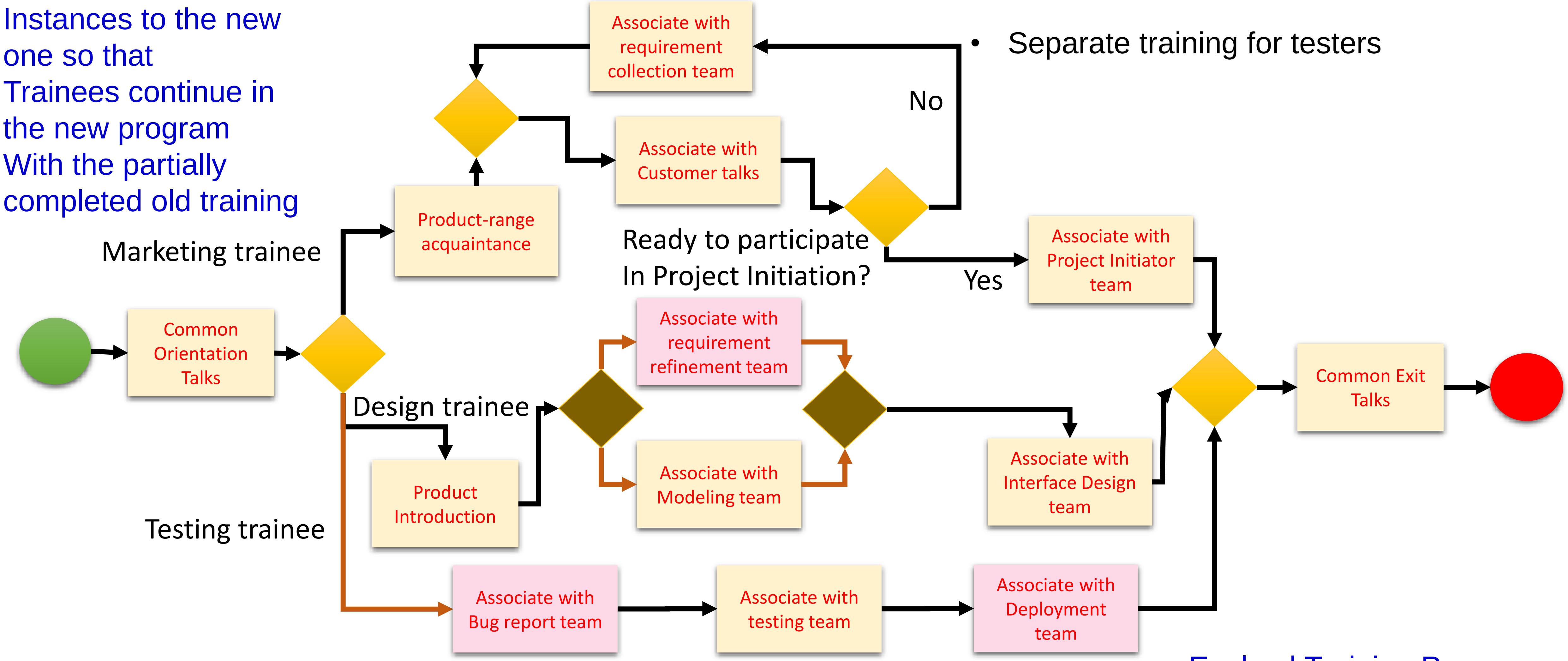



Training Process
For batch of Fresh Recruits

# Dynamic Evolution in Business Processes

Requires migration of old process Instances to the new one so that Trainees continue in the new program With the partially completed old training

- Design trainees need to know requirement
- Separate training for testers




Evolved Training Process

# Motivations

- Various approaches exist for model specific solutions
- Different situations require different notions of correctness
- Subtle interplays among notion of correctness, available algorithms and concepts not classified so far
- Requires consolidation of theoretical approaches to move forward in the research field
- Solutions not practice since theory has not developed enough
- Challenging problems in theory

# Problem Statement

- Given old and new schema, explore the problem of state-transfer under different notions of correctness
- Algorithmic solution to state-transfer that avoids state-space search
- Theoretical approach general enough to adapt in different modeling approaches
- Explore properties and proofs related to the problem

# Scope and Assumptions

- Control-flow structure of business processes
- Schema structures are correct
- Original and Evolved schema are provided
- Well-formed schema

# Not in Scope

Access control and user perspectives, Data-flow concerns, Methodology involved in evolution, Unstructured workflows, Deployment issues

# Contributions

## Algorithms

- YoYo algorithm for instance migration
- Algorithm for weak lookahead
- Accept/reject branching algorithm for strong lookahead
- PSCR computation algorithm
- Change region computation algorithm
- Distributed change region computation algorithm

## Properties

- YoYo compatibility, peer patterns
- Generator of Concurrent Submarking (GCS)
- Dysfunctional C-tree and Break-off Set
- Marking Preserving Embedding (MPE)
- Change properties
- Perfect Member and Overestimation
- Perfect Structural Change Region (PSCR)
- Fragmentation

## Taxonomy Framework for Consistency Models

- Structural equivalence
- Trail-based models
  - history equivalence
  - trace equivalence
  - purged-history equivalence
  - purged-trace equivalence
- Live model
- Lookahead models
  - strong
  - accommodative
  - weak

## Proofs

- Non-migratability lemma
- Perfect Member lemma
- Overestimation lemma
- SCR & PSCR lemma
- Proof of correctness for algorithms

## Representation Techniques for Analysis & Application

- C-tree, Derivation Tree
- Token transportation catalog
- Token transportation bridge

## Workflow Specification Languages

- CWS, ECWS

## New Consistency Models

- Strong lookahead
- Accommodative lookahead
- Weak lookahead

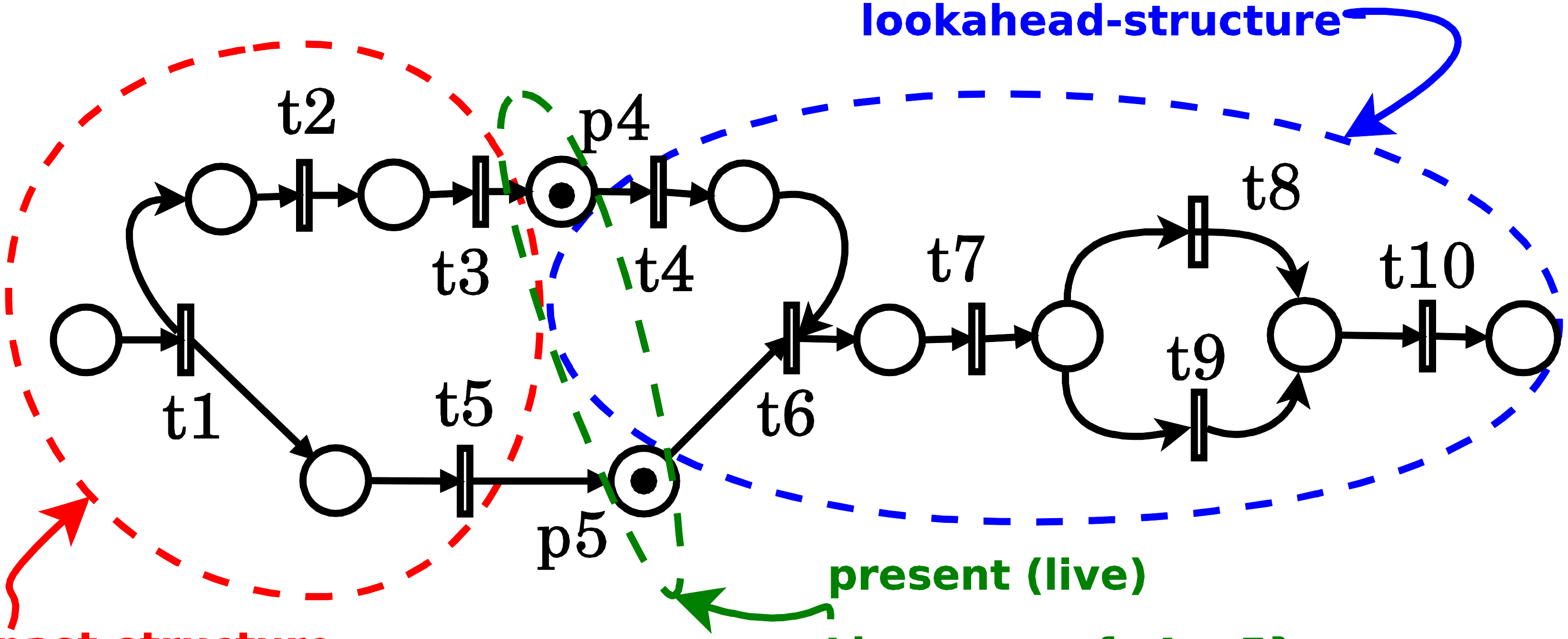
# Notions of Consistency

**Consistency:**
Formal criteria of correctness of **instance migration**
(state-transfer/token transportation)

State:
Marking in
Petri net models

**lookahead-traces: t4 t6 t7 t8 t10**
**or t4 t6 t7 t9 t10**

**lookahead-structure**




**present (live)**

**Live state: {p4, p5}**

**past-structure**

**past-trace: t1 t2 t3 t5**

Old and New markings
can be considered as
migration equivalent (consistent)
in various ways

Past/Present/Future of a State

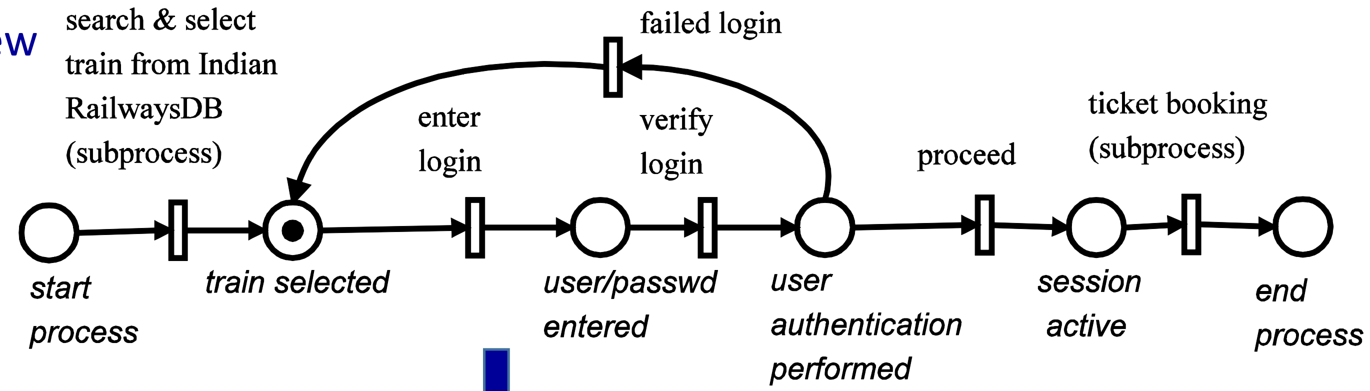
# Taxonomy and New Models of Consistency

| Consistency Models | Parameters | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Past based | Present based | Future based | | | Trace based | | | Structure based |
| | | | equal | subset | superset | Set based | Sequence based | purged | |
| Structural equivalence | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| Trace equivalence | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ |
| History equivalence | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| Purged-trace eq. | ✓ | ✗ | ✗ | ✗ | ✗ | | ✓ | ✓ | ✗ |
| Purged-history eq. | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ |
| Live | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| Strong lookahead | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ |
| Accommodative lookahead | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ |
| Weak lookahead | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ |

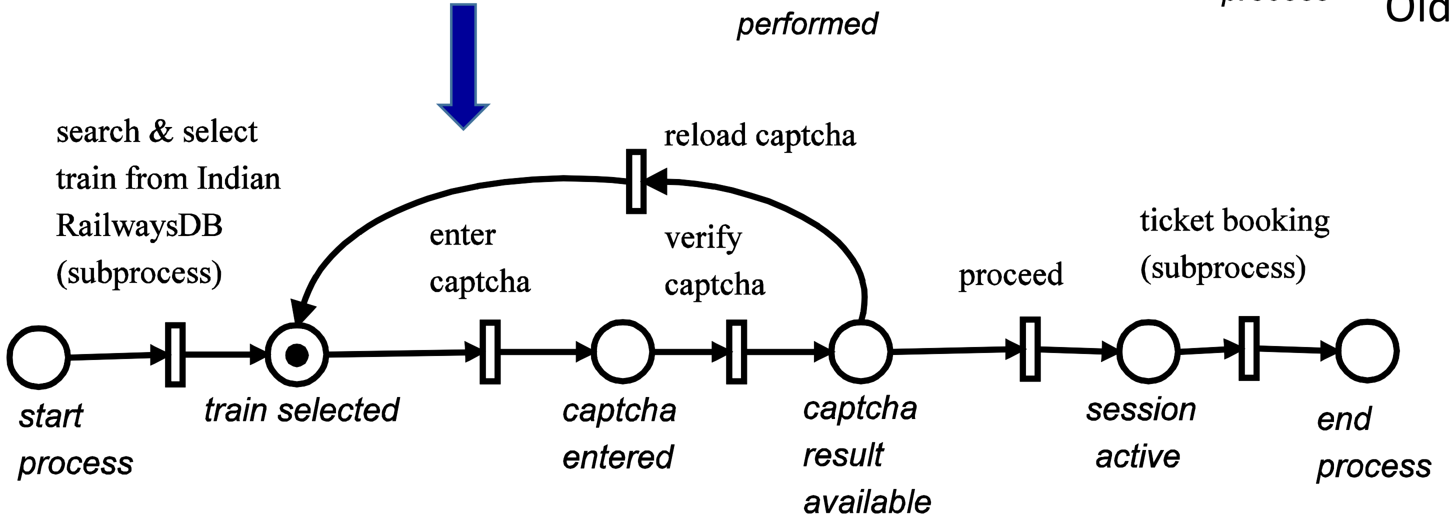
# Dynamic Instance Migration

## State-transfer Approach Vs. Change Region Approach

Marking
Transfers
From old to new
Net explicitly
as per the
Chosen
Consistency
model




Old ticket booking process




New ticket booking process

Marking
Transfers **as it is**
Outside change region,
No consistent migration
Inside change region
***(no explicit state-transfer)***

# Dynamic Instance Migration

| Citation to the Literature | Consistency | State-transfer | Change Region |
|---|---|---|---|
| Ellis et al. 1995, ACM COCS | Trace Equivalence | Some marking to Some marking | Inside: non-migratable/ migration to different marking<br>Outside: migration as it is<br>Construction: Not specified ( intuition given for SESE region) |
| Van der Aalst, 2001, Info. Sys. Frontiers (Springer) | Live Consistency | Some marking to **Same** marking | Inside: non-migratable/migration as it is<br>Outside: migration as it is<br>Construction: minimal SESE region covering structural changes<br>(modified SESE reg. black box, same state-space outside) |
| Sun et al., 2009, Info. Soft. Tech. (Elsevier) | Purged History Equivalence | Some marking to Some marking | Inside: non-migratable/ migration to different marking<br>Outside: migration as it is if every marking inside change region is migratable<br>Construction: minimal SESE region covering structural changes |
| Cicirelli et al., 2010, J. Sys. Soft. (Elsevier) | History Equivalence | Some marking to **Same** marking | Van der Aalst, 2001 |
| Zou et al., 2010, IEEE Advanced. Serv. Comp. | Trace Equivalence | Some marking to Some marking | Inside: non-migratable/ migration to different marking<br>Outside: migration as it is<br>Construction: Van der Aalst, 2001 |
| Hens et al.,2014, J. Sys. Soft. (Elsevier) | Live Consistency | Some marking to **Same** marking | Van der Aalst, 2001 |

Migration to same marking is Live Consistency

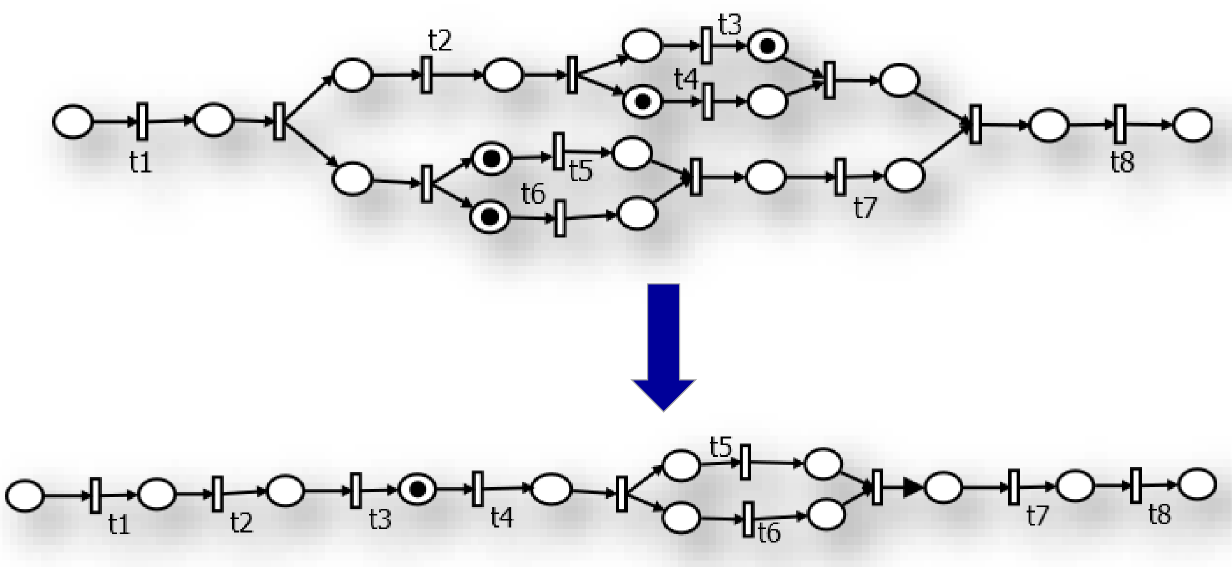
migration to different marking inside Change region violates Live consistency

Structural changes may retain state-reachability, e.g. loop to xor

For SESE change region, Migration as it is outside Change region may violate Trace/history based consistency e.g. downstream the change region

# State-transfer Approach

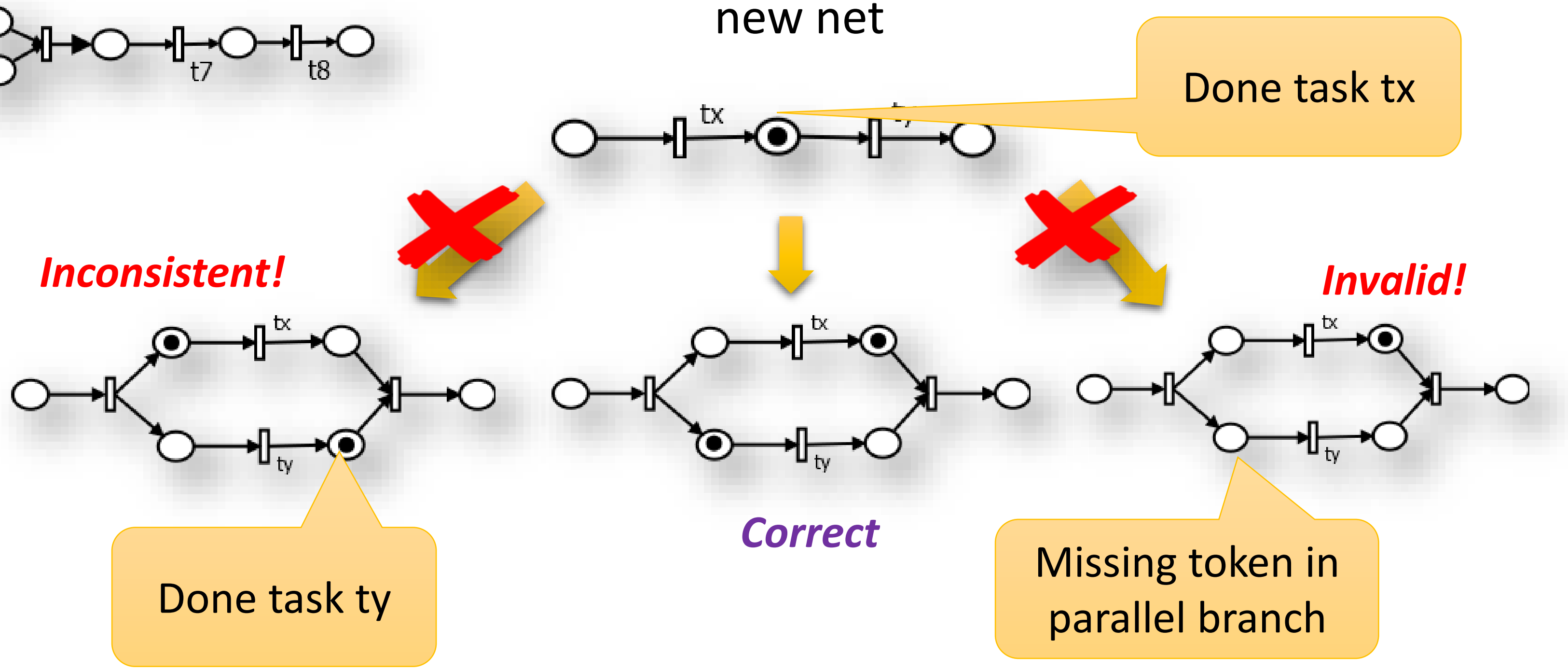
# History Equivalence Consistency Model




History: t1, t2, t3

***Consistency***
preservation of history (done tasks in old ↔ done tasks in new)

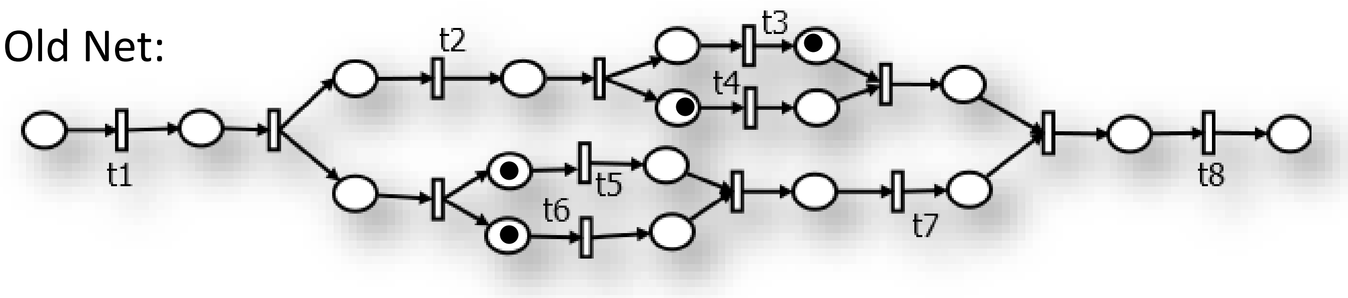
***Validity***
reachability of marking in the new net



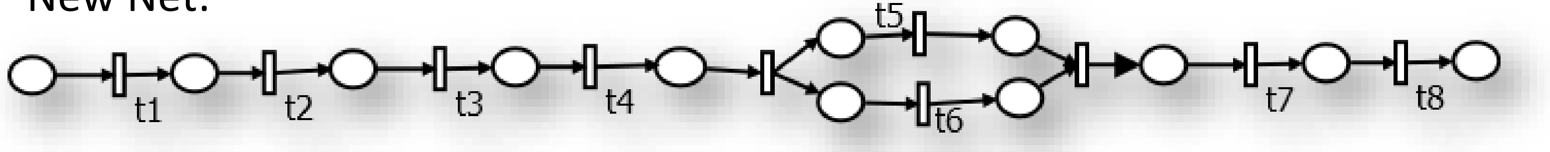
# YoYo Approach

*Token transportation by: Folding, transport, Unfolding*

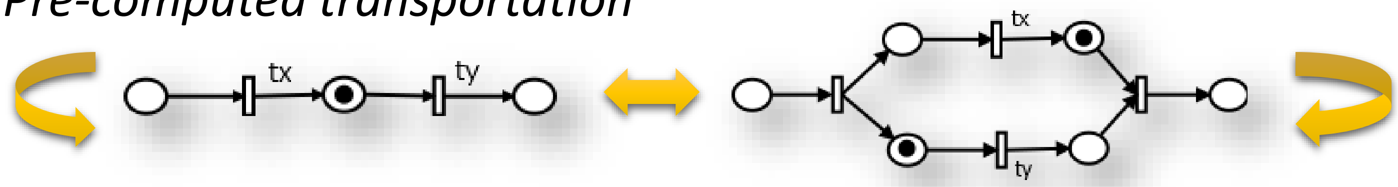
Old Net:




New Net:

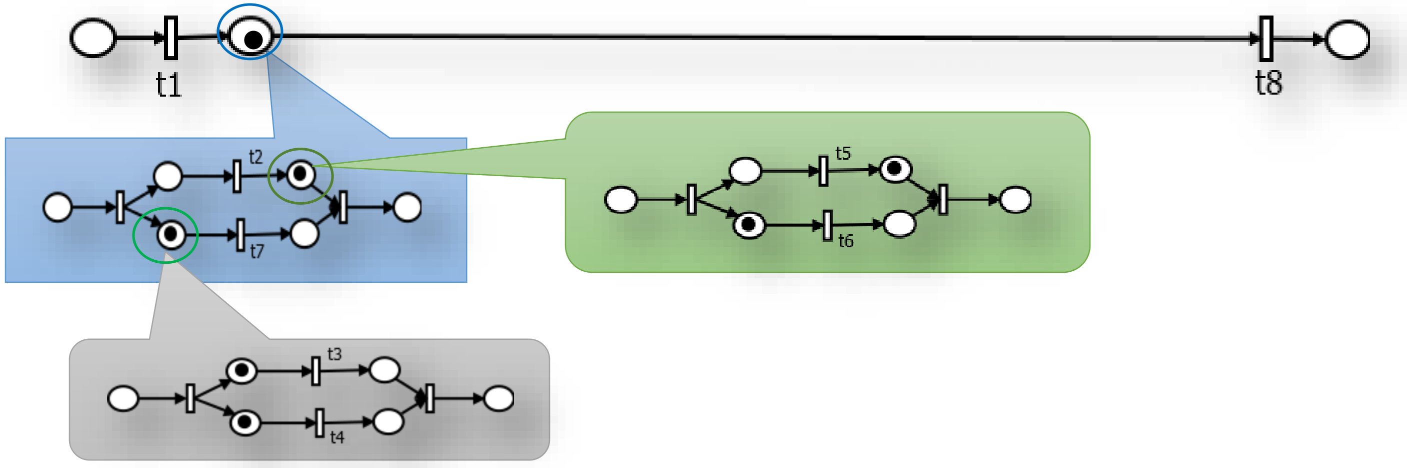



*Pre-computed transportation*

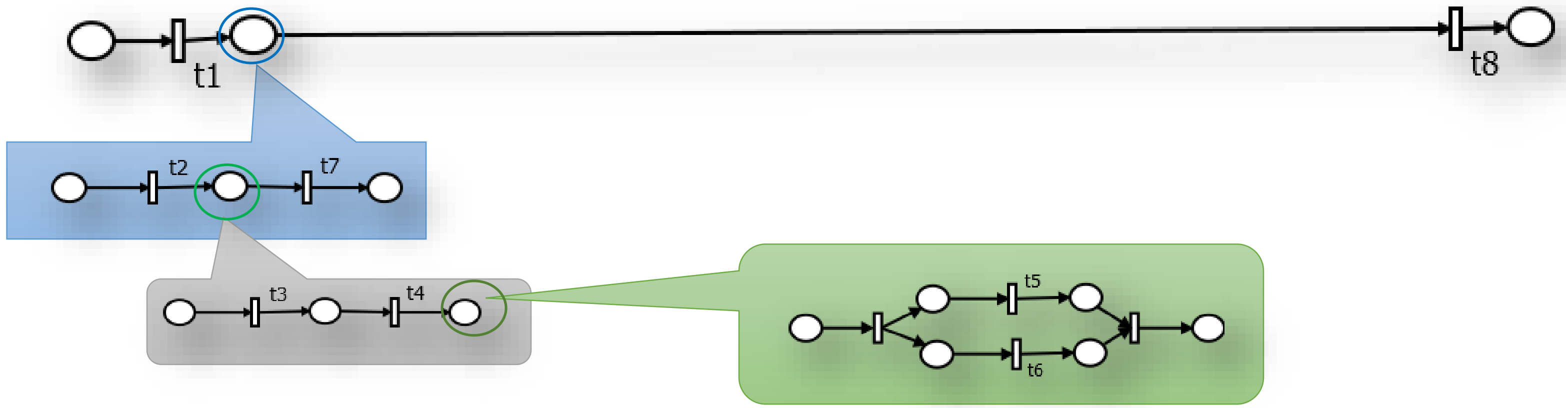


# Folding

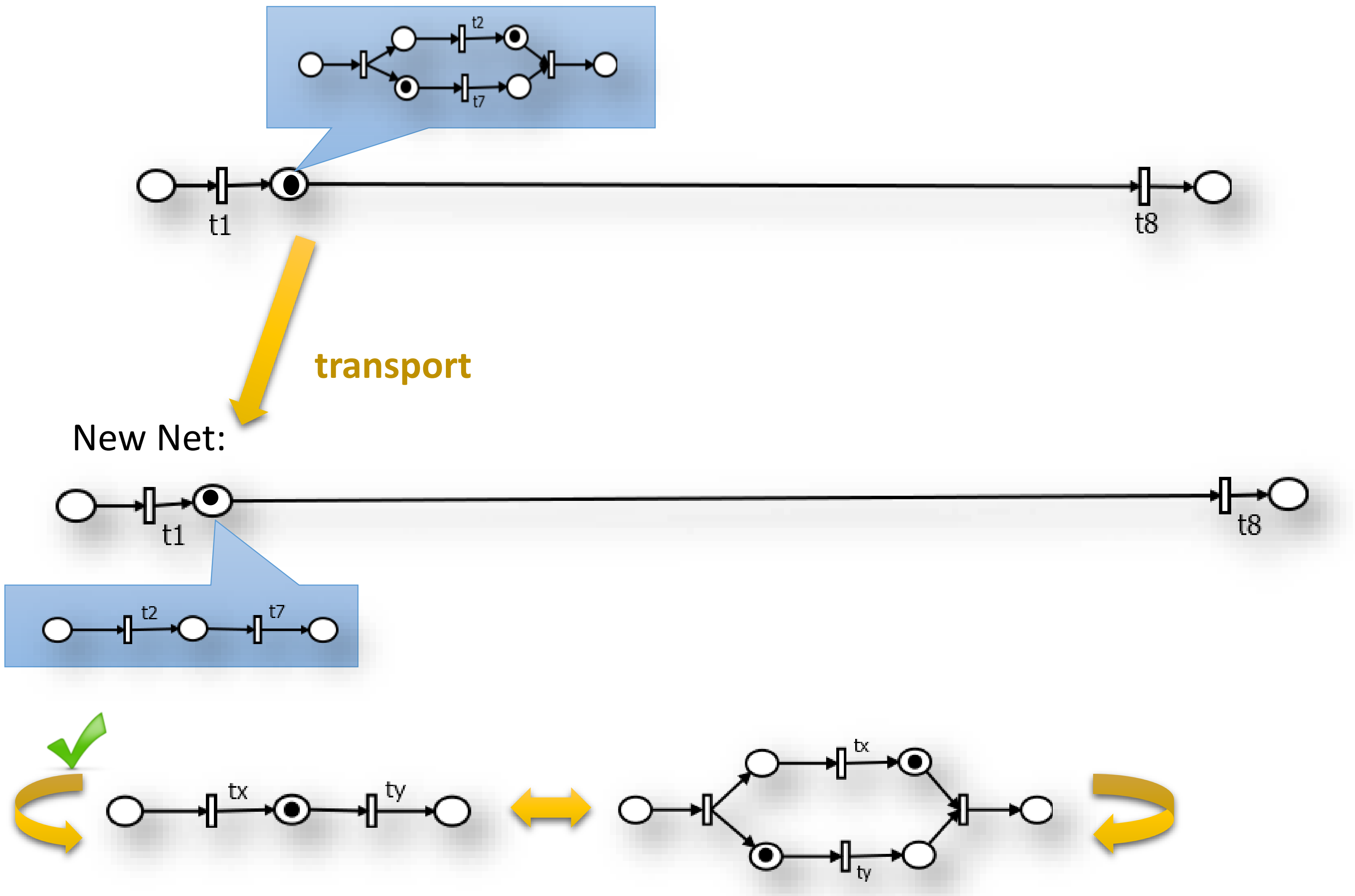
Old Net:



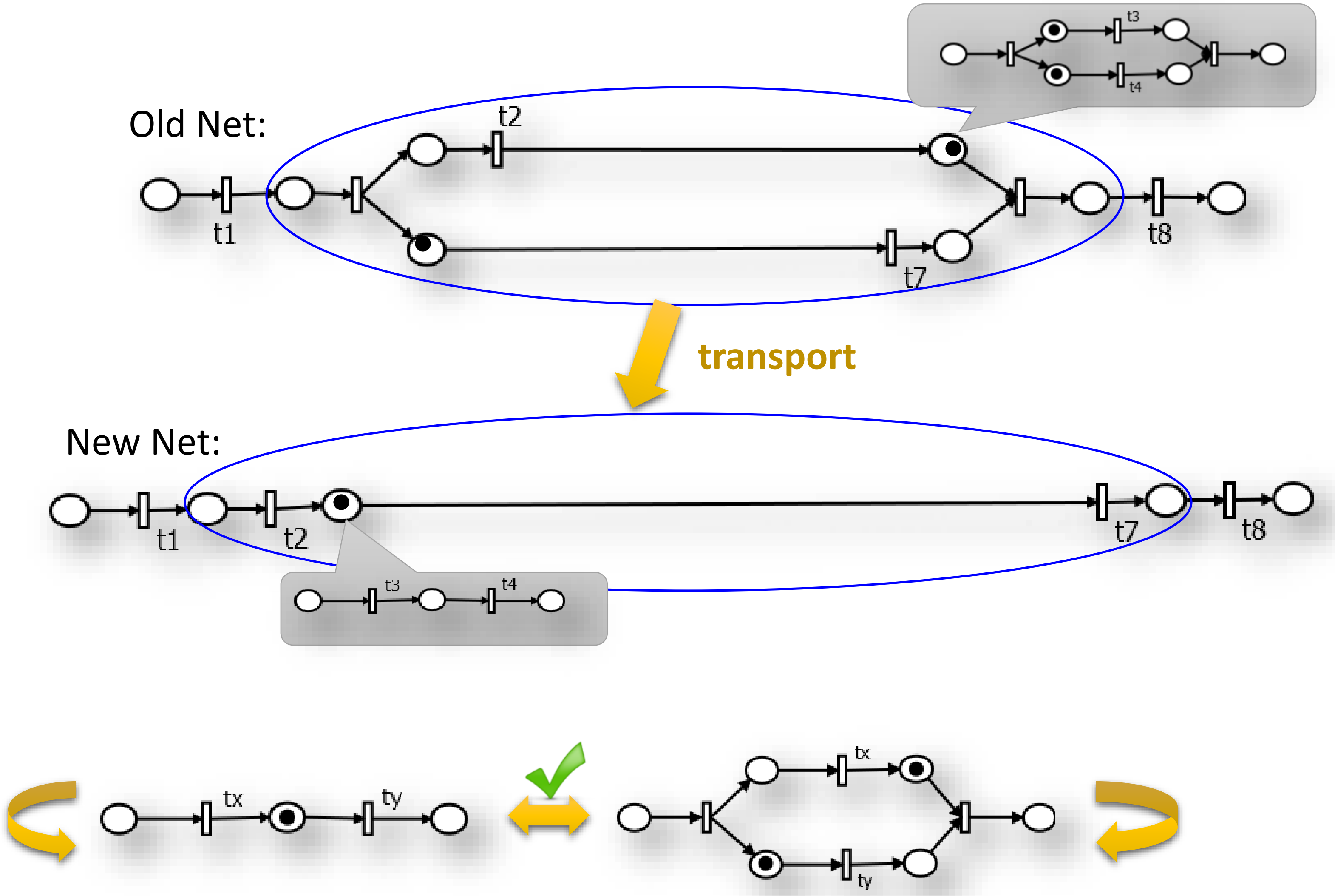

New Net:



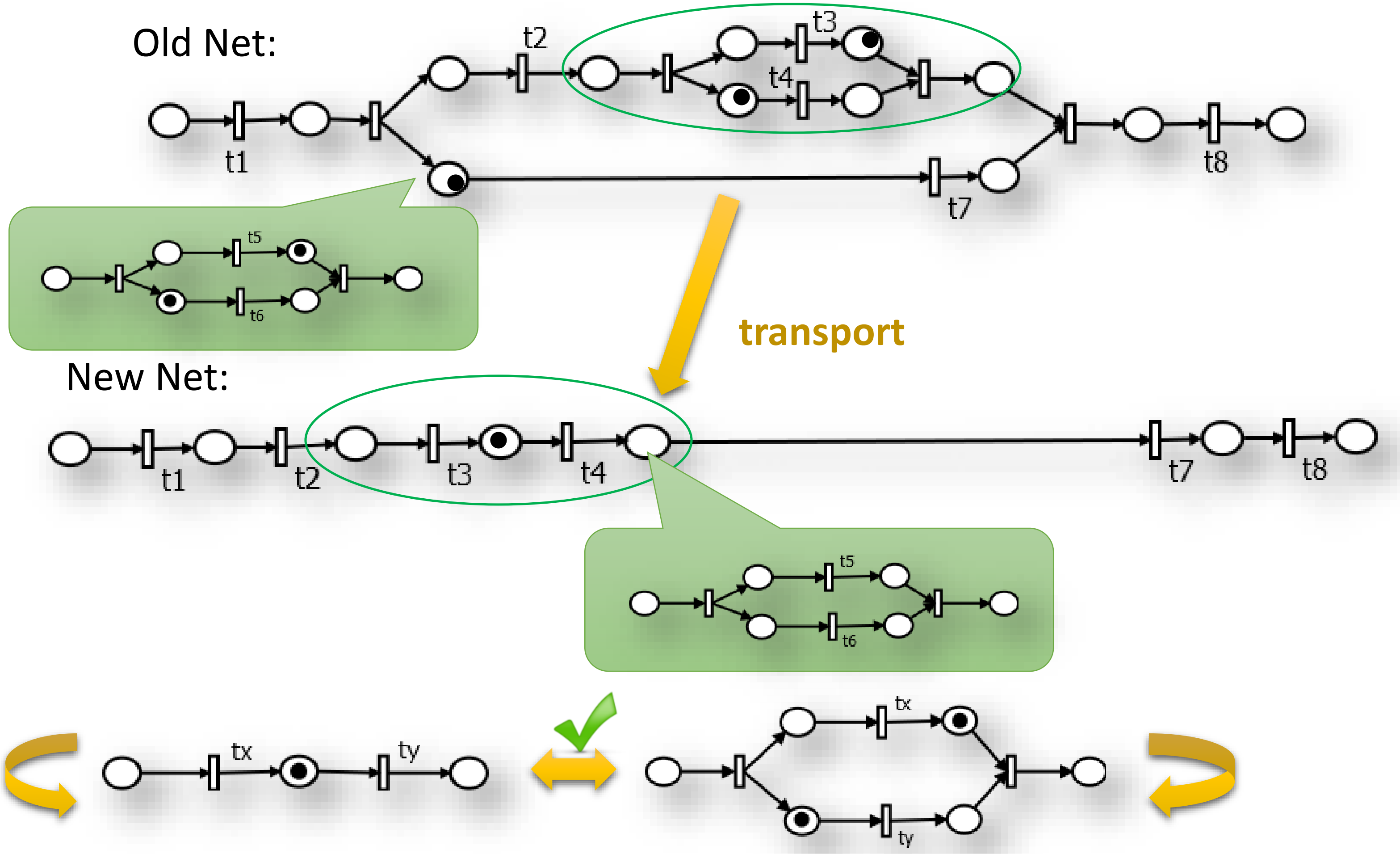
# Transport & Unfolding



# Transport & Unfolding

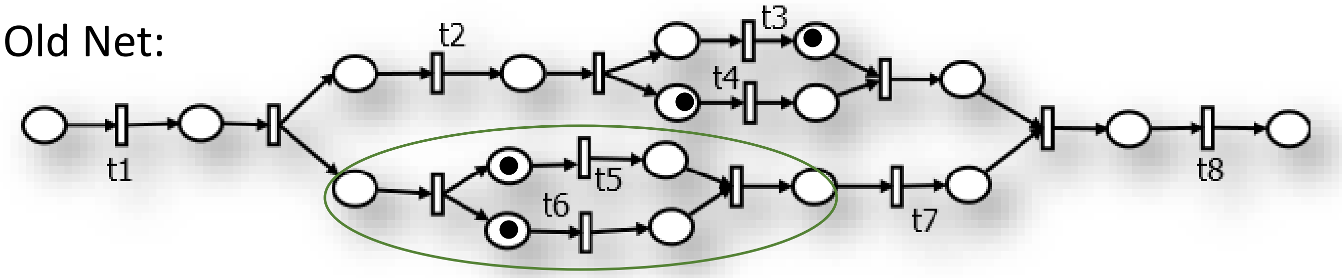


# Transport & Unfolding



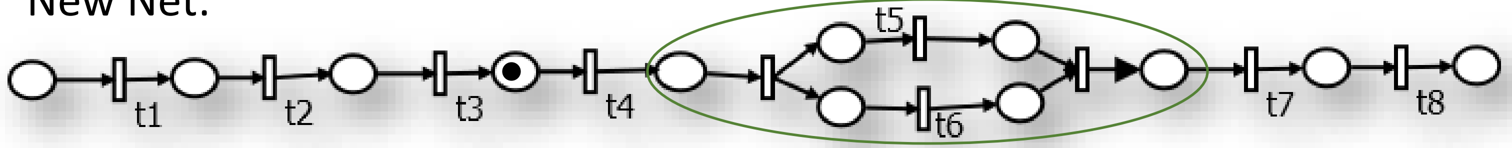
# Transport & Unfolding

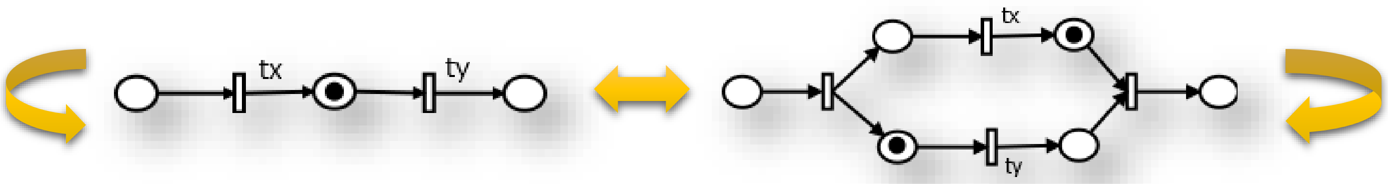
Old Net:




No transport required

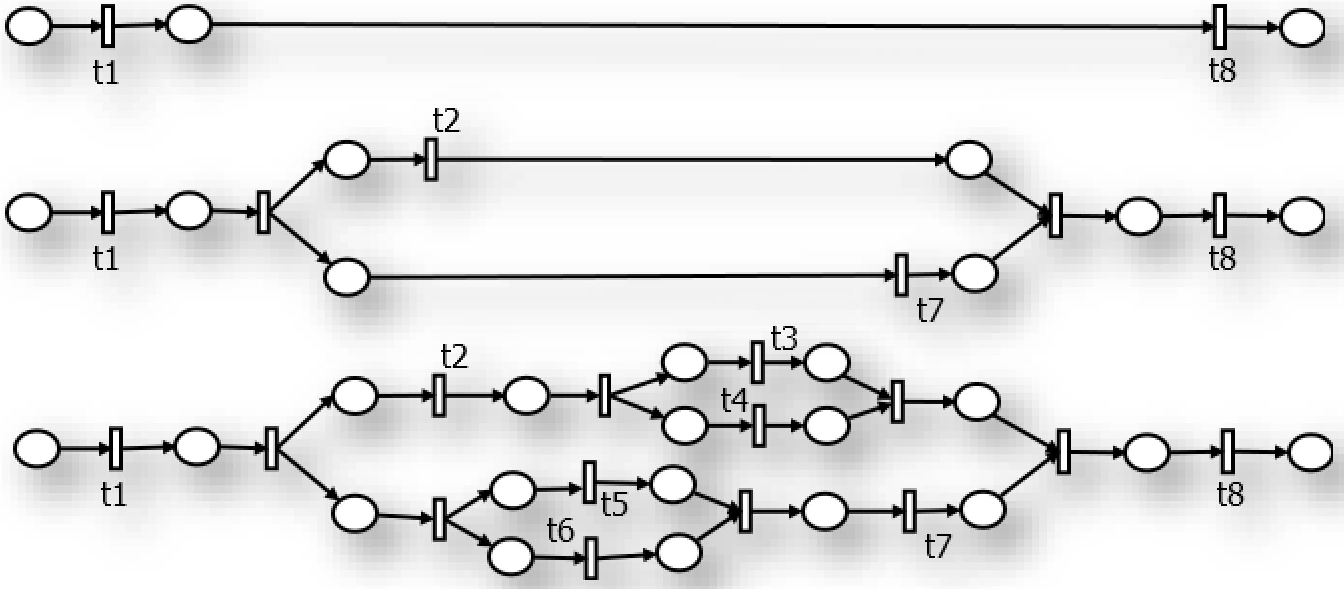
New Net:



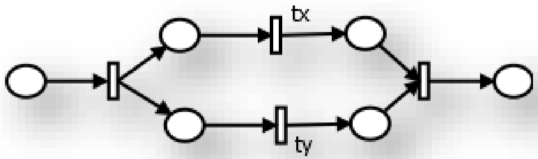
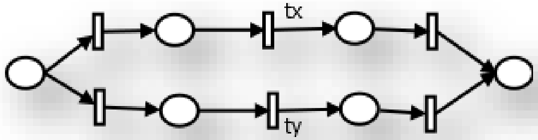



# CWS Grammar

Example:




**Folding steps follow such order of derivation..**

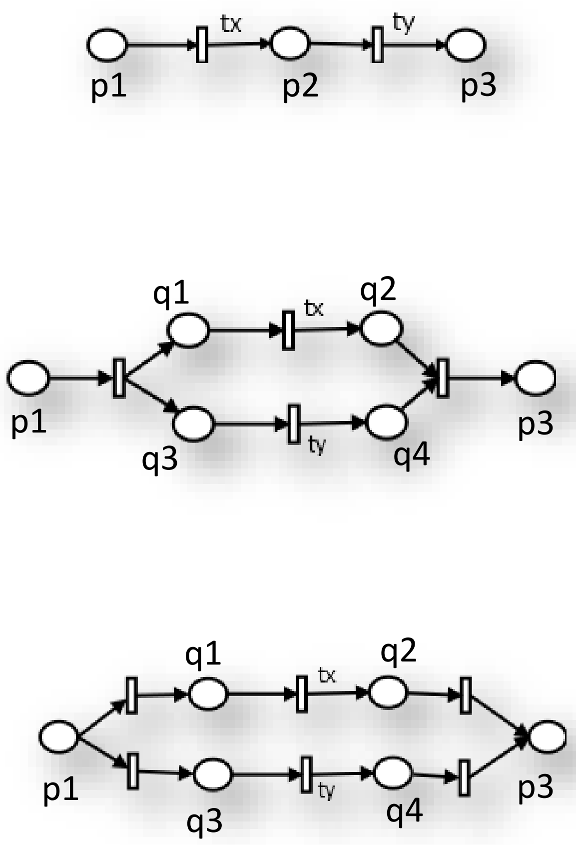
| Pattern | Specification | Net Model |
|---|---|---|
| SEQ: | tx ty |   |
| AND: | (tx) (ty) |   |
| XOR: | [tx] [ty] |   |

Start → SEQ;
SEQ → SEQ t SEQ t SEQ
| SEQ AND SEQ
| SEQ XOR SEQ | e
AND → ( SEQ t SEQ ) ( SEQ t SEQ )
XOR → [ SEQ t SEQ ] [ SEQ t SEQ ]

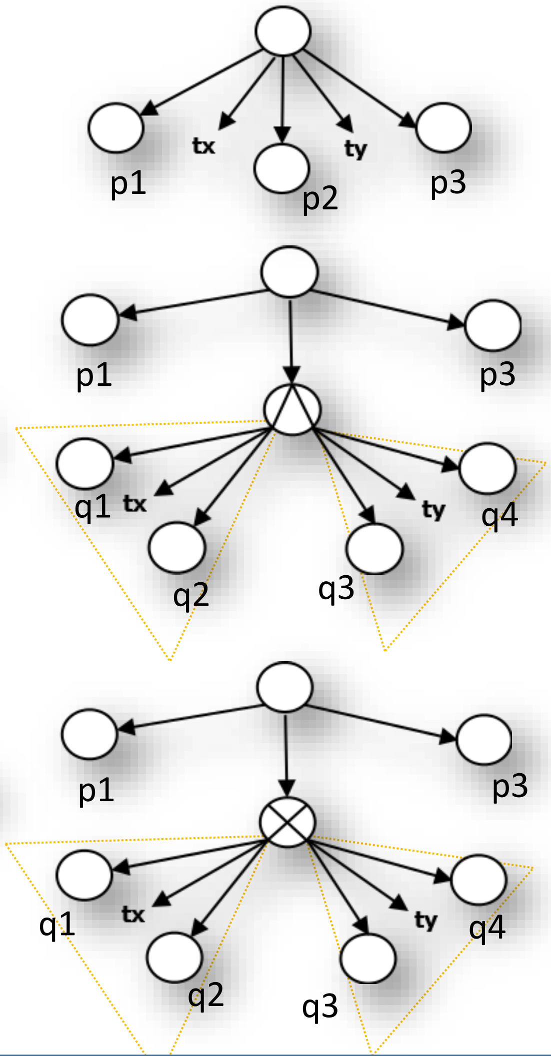
Acyclic Workflow nets
Even no. of transitions

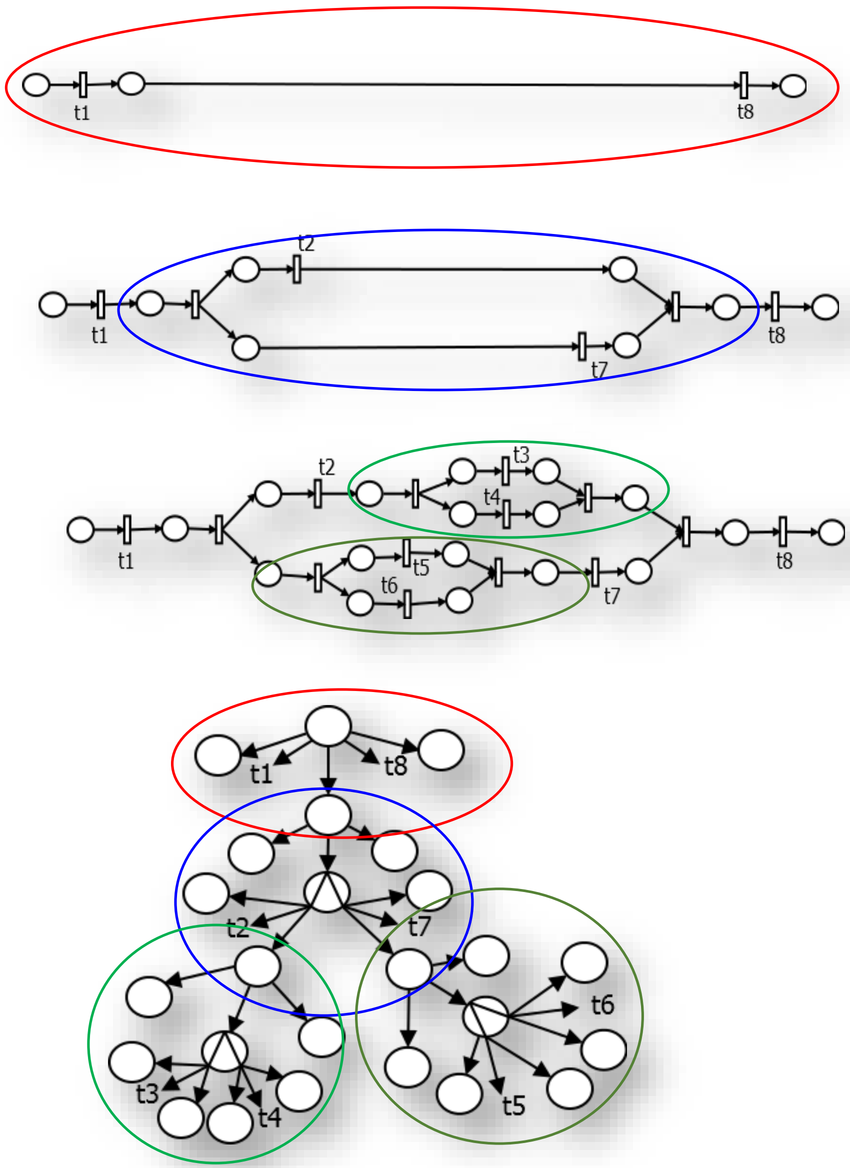
# Derivation Tree

### Primitive Block




### Derivation Tree



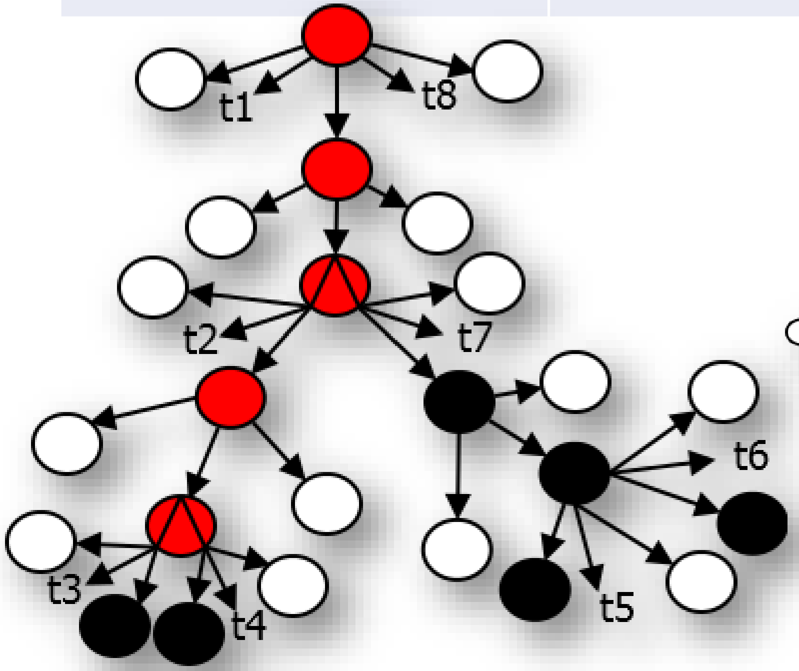



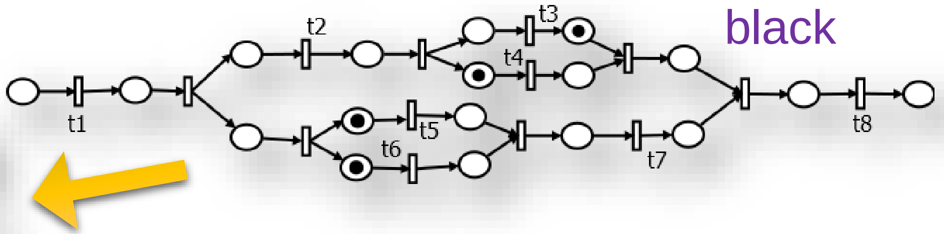
# Colored Derivation Tree

| Node Type | | Description |
|---|---|---|
| Leaf/Non-leaf | ○ | Unmarked folded/unfolded place |
| Leaf | ● | marked place in net |
| Non-leaf | ● | abstraction of null-executed subnet |
| Non-leaf | 🔴 | abstraction of subnets where at least one labeled transition has been fired |

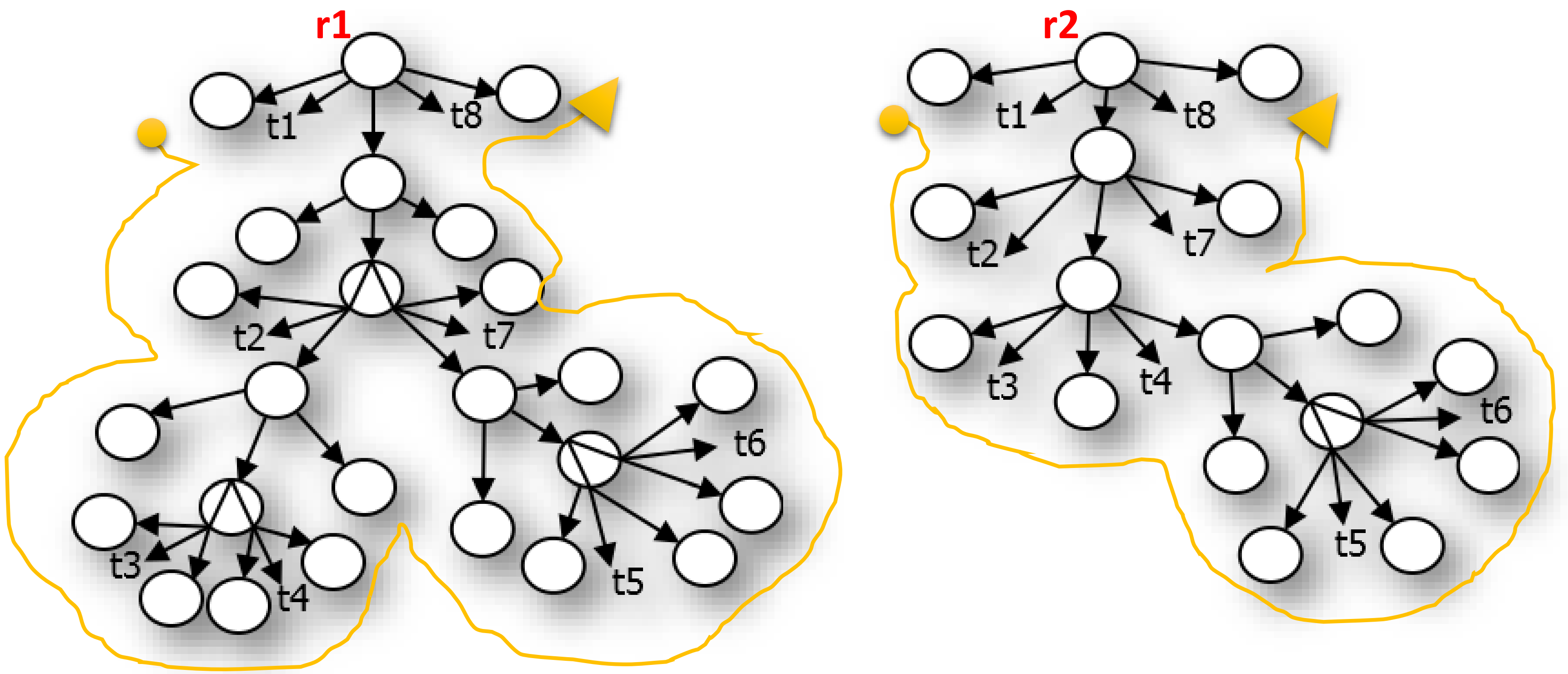
**Red node:**
Color parent red

**Black node:**
Check if any transition Sibling has color at right,
If yes, color parent red; Else color parent black



# YoYo Compatibility




Yield of r1 =

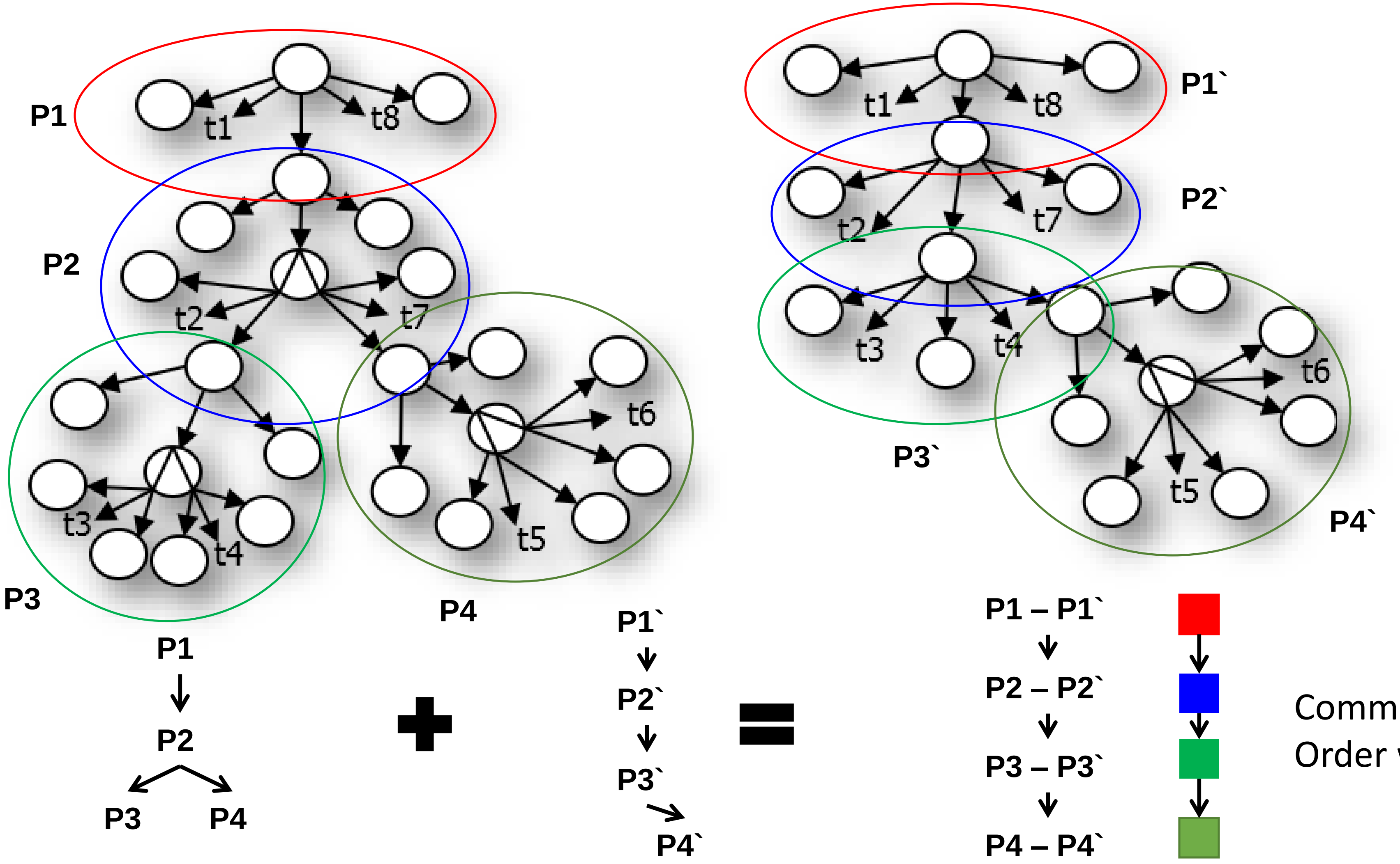
**t1 { t2 { t3, t4 } , { t5, t6 } t7 } t8**

Yield of r2 =

**t1 t2 t3 t4 { t5, t6 } t7 t8**

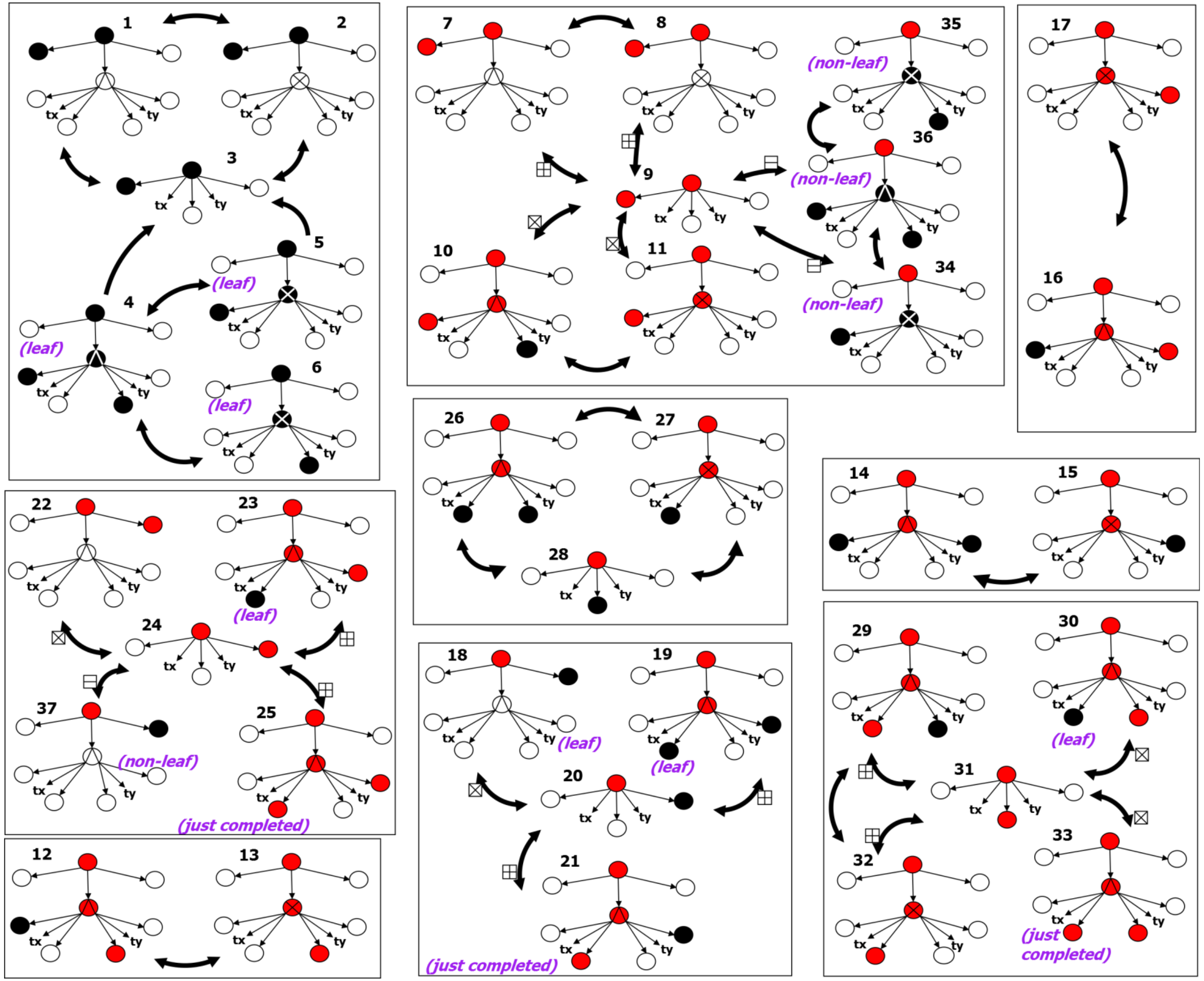
Both can generate the same sequence **t1 t2 t3 t4 t5 t6 t7 t8** → ***Folding order* exists**

# Folding Order



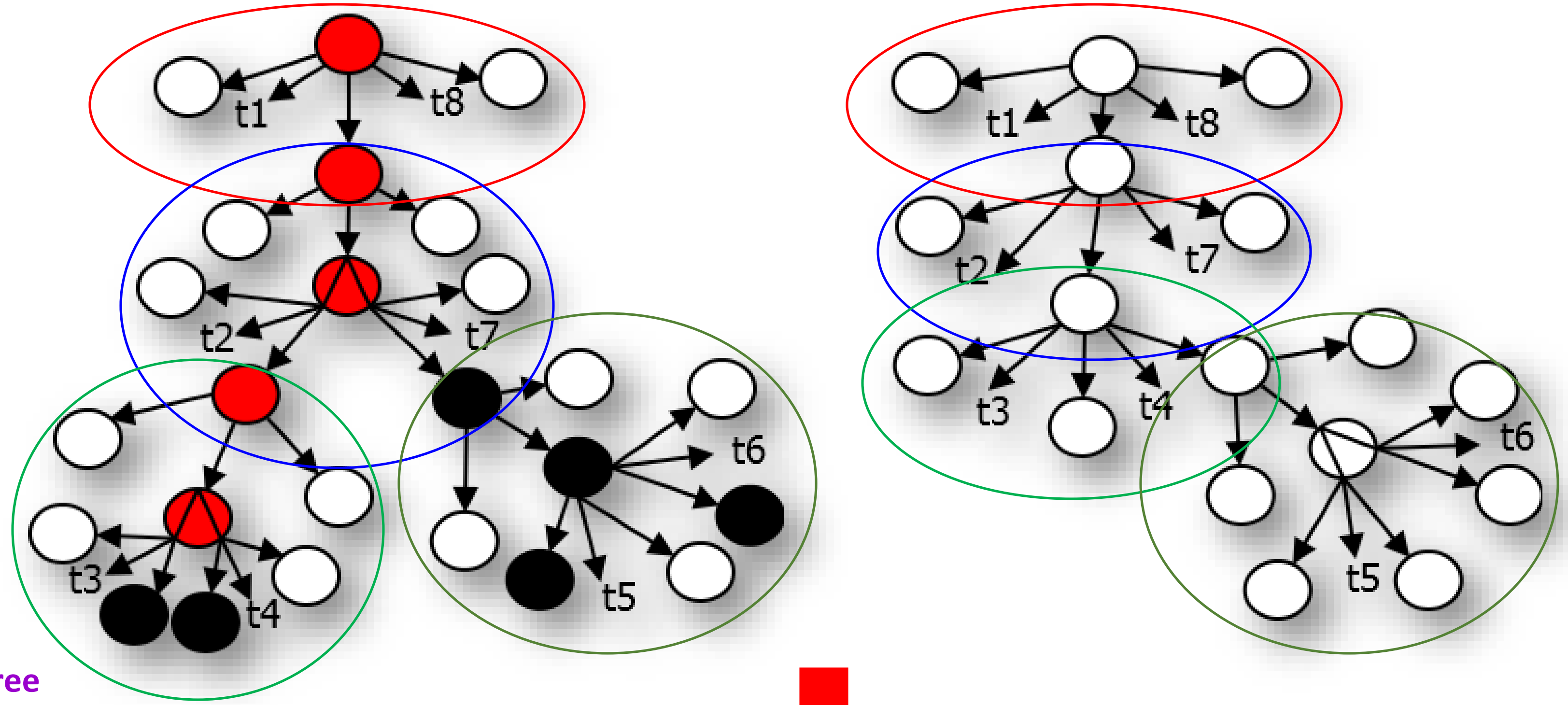

Common derivation
Order w.r.t. peer patterns

# Token Transportation Catalog



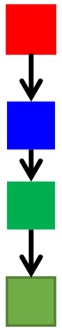

Every possible
Color-mapping
Between peer patterns

# YoYo Algorithm



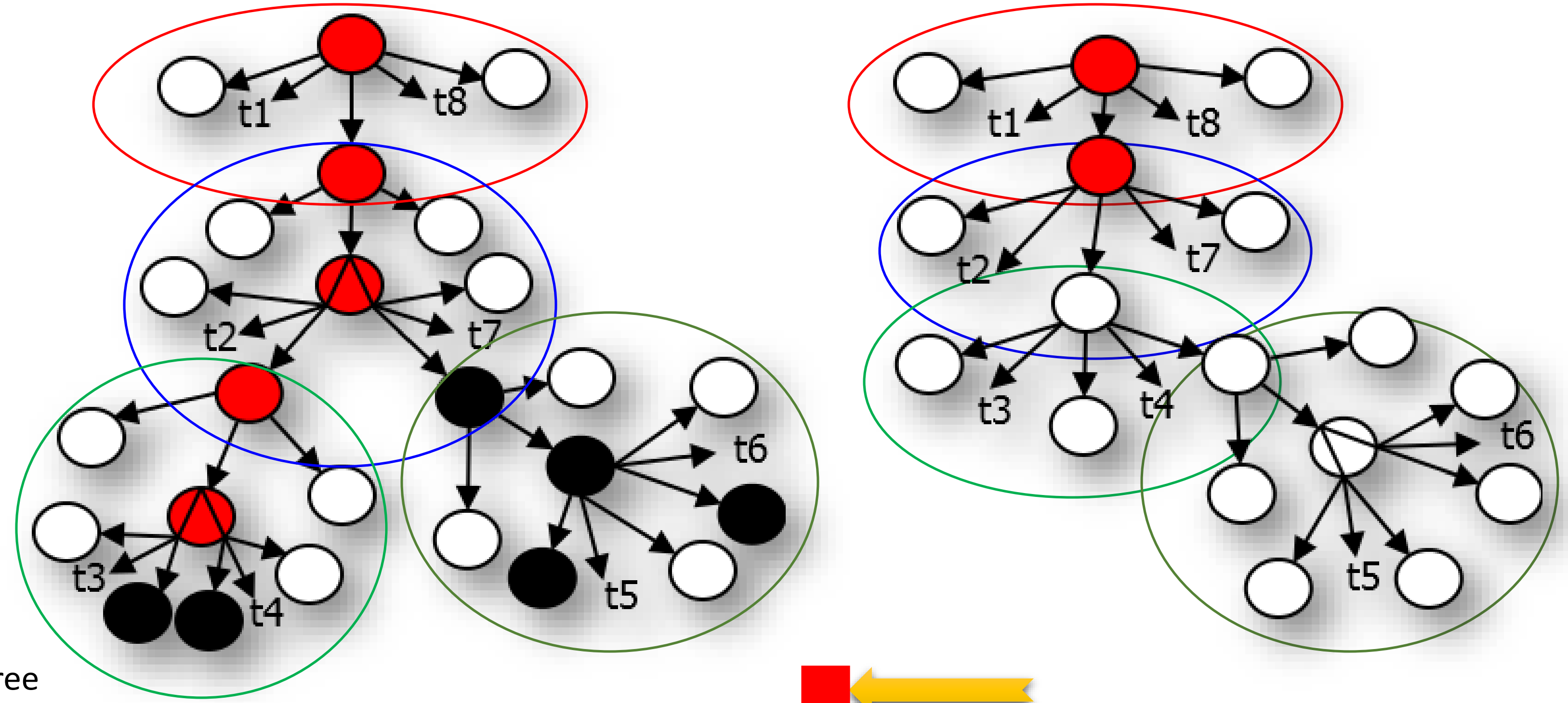

**1. Color old tree**

2. <**p-q**> be **1st peer patterns** to appear in **folding order *F***
3. Color transfer between p, q
4. **for** each next <**p-q**> in **F**,
   **if q** has colored root,
      **if p** is colored,
         color transfer between p, q
      **else**
         localPropagation(q)

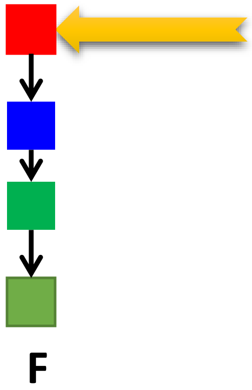


**F**

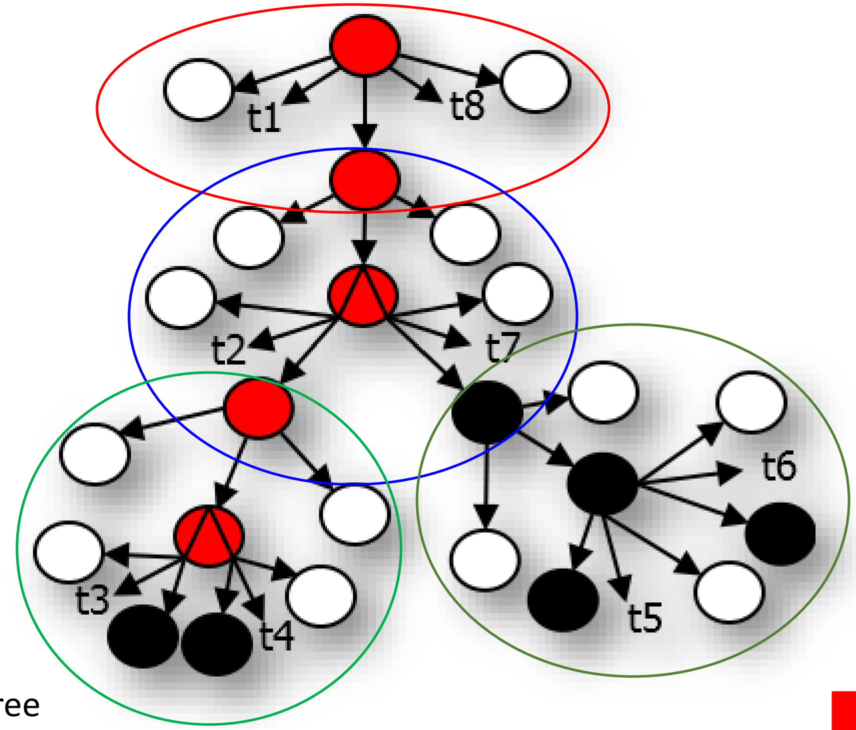
# YoYo Algorithm

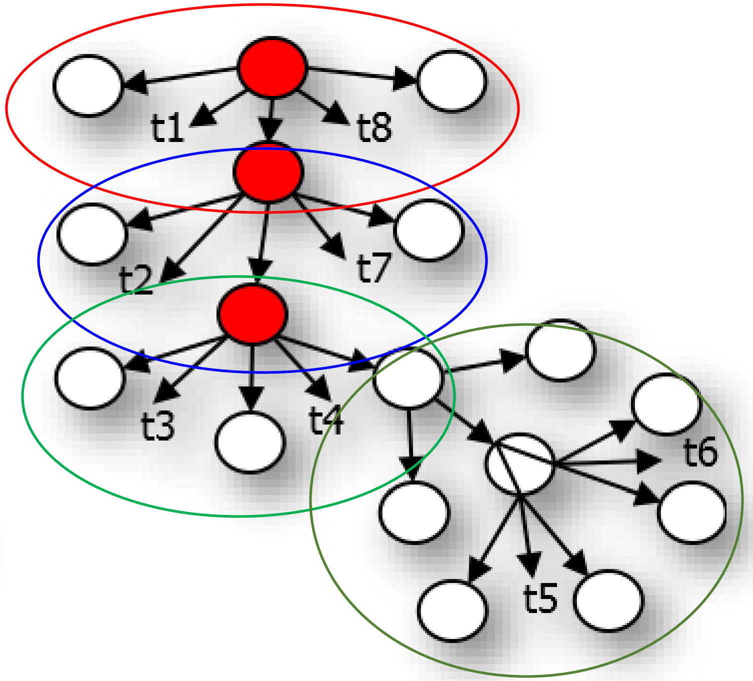



1. Color old tree
2. **<p-q> be 1st peer patterns to appear in folding order *F***
3. **Color transfer between p, q**
4. **for** each next <**p-q**> in **F**,
   **if q** has colored root,
      **if p** is colored,
         color transfer between p, q
      **else**
         localPropagation(q)

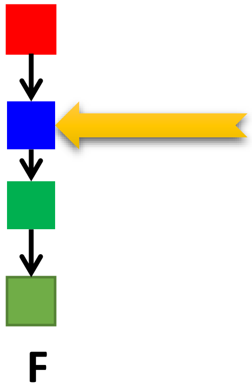


# YoYo Algorithm

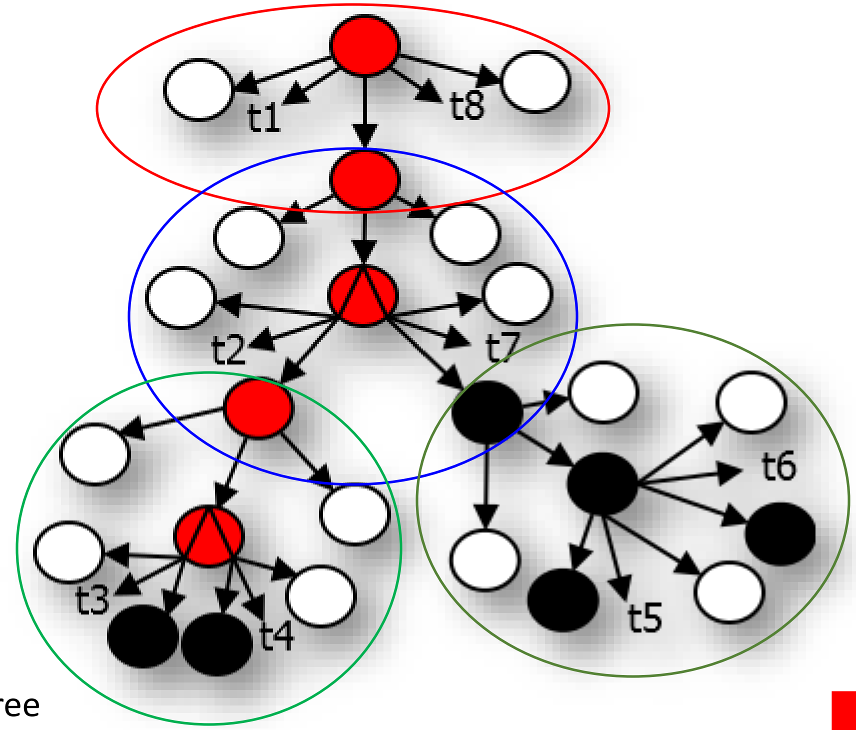


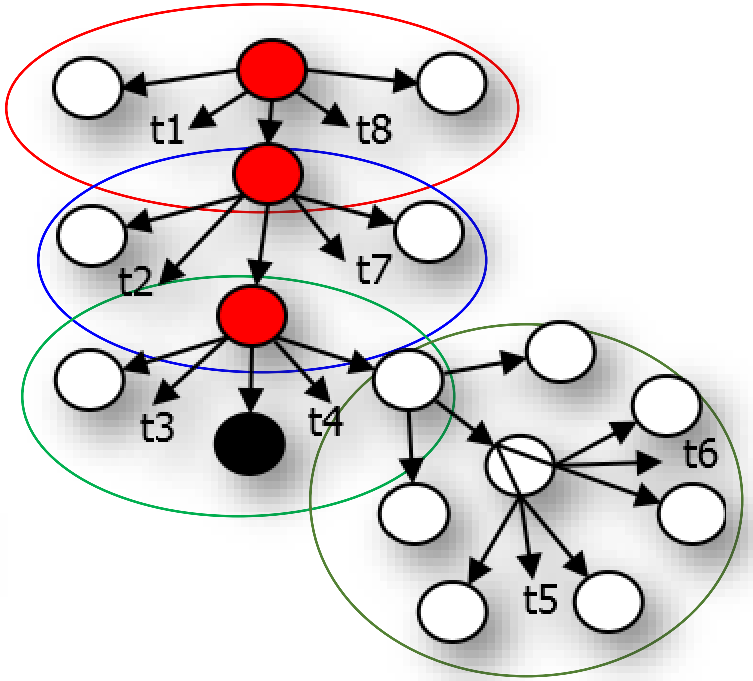




1. Color old tree
2. <p-q> be **1st peer patterns** to appear in **folding order *F***
3. Color transfer between p, q
4. **for** each **next <p-q> in F,**
   **if q has colored root,**
   **if p is colored,**
   **color transfer between p, q**
   **else**
   localPropagation(q)

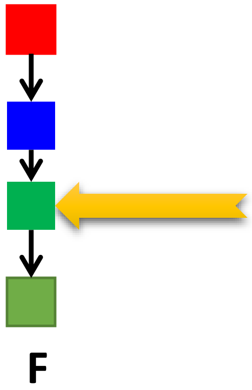


# YoYo Algorithm

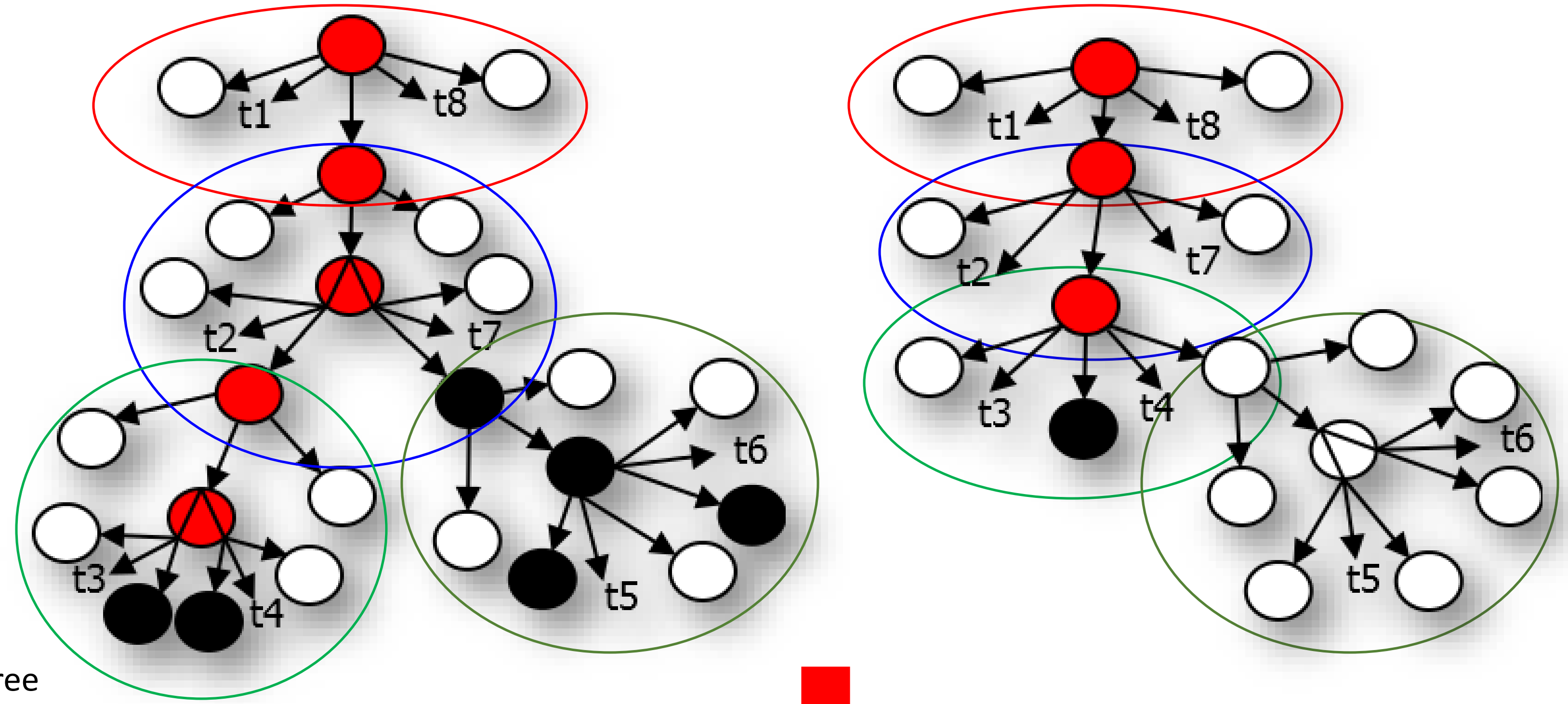



1. Color old tree
2. <p-q> be **1st peer patterns** to appear in **folding order *F***
3. Color transfer between p, q
4. **for** each **next <p-q> in F,**
   **if q has colored root,**
   **if p is colored,**
   **color transfer between p, q**
   **else**
   localPropagation(q)



# YoYo Algorithm




1. Color old tree
2. <p-q> be **$1^{st}$ peer patterns** to appear in **folding order *F***
3. Color transfer between p, q
4. **for** each **next <p-q> in F,**

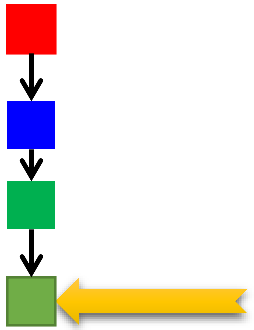
   **if q has colored root,** **false**

   **if p** is colored,

   color transfer between p, q

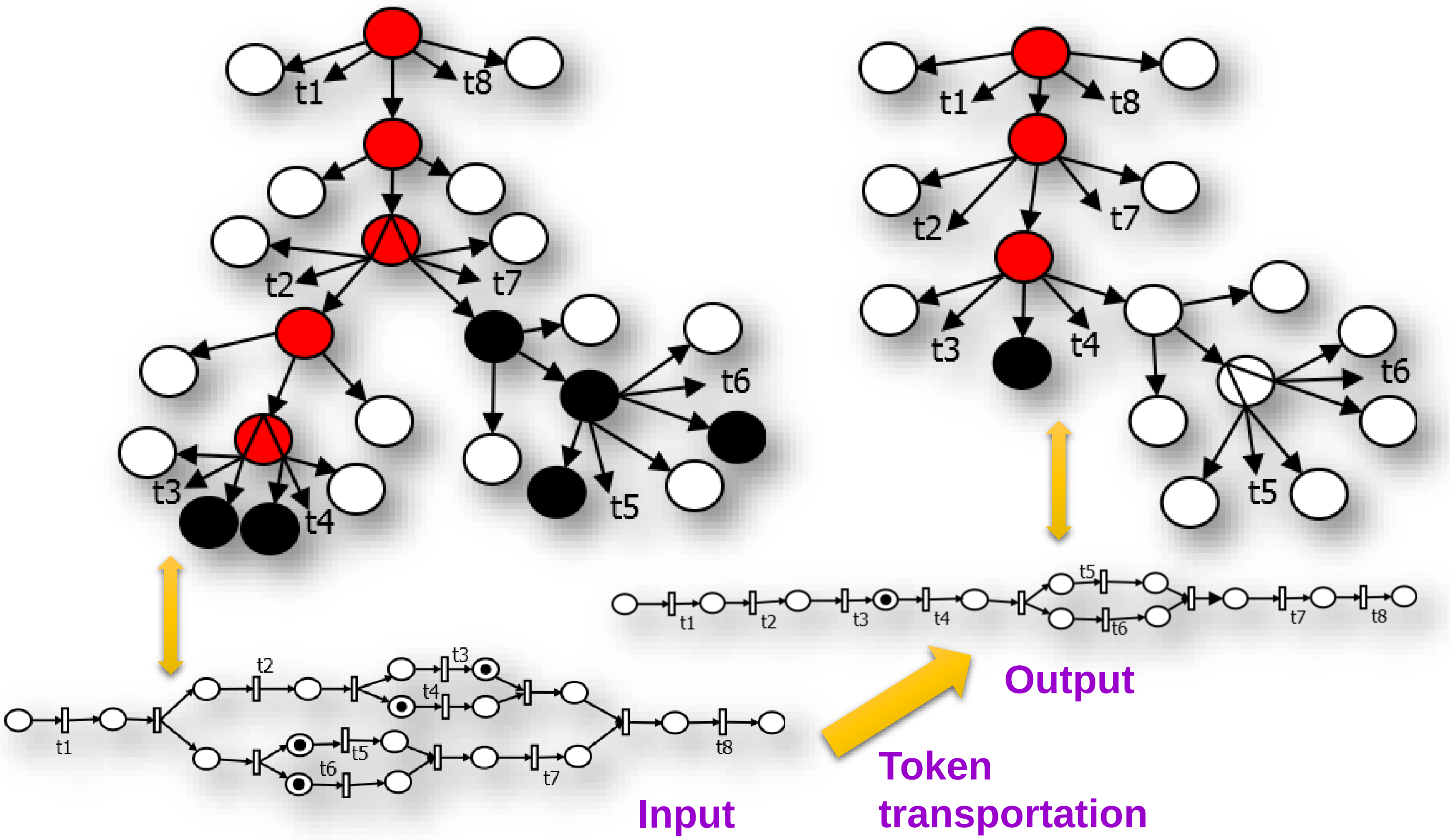
   **else**

   localPropagation(q)



**F**

# YoYo Algorithm




Max. no. of Transportation Steps = no. of patterns ( linear time complexity )

# Correctness Proof

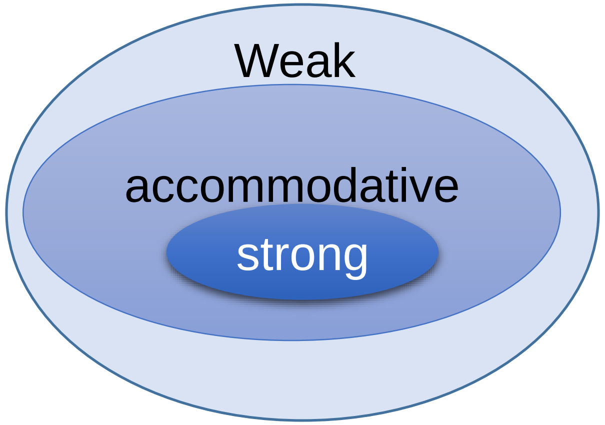
**Catalog Completeness:**
Token transportation catalog is complete w.r.t. the 6 change patterns

**Lemma 1:**
For two Yo-Yo compatible derivation trees, consistent coloring between the top peer patterns guaranties consistent coloring between their immediate child peer patterns

**Lemma 2:**
Lemma 1 can be repeated for all parent-child peer pairs across two Yo-Yo compatible derivation trees

# Lookahead Consistency Models

| Consistency Model Name | Description |
| --- | --- |
| Strong Lookahead | same lookahead trace sets of consistent marking |
| Accommodative Lookahead | old lookahead trace set preserved in new |
| Weak Lookahead | at least one old lookahead trace preserved in new |




***State-transfer*** Algorithms:

1. Determine existence of Weak Lookahead
2. Inferences about Strong/Accommodative Lookahead
3. Given Accommodative Lookahead, enforce Strong Lookahead

# Gist of State-transfer Approaches

Petri Net is not trace-accumulative model, needs some kind of trace-replay in order to obtain trace-based consistent migrations

YoYo algorithm (for existing model of consistency)

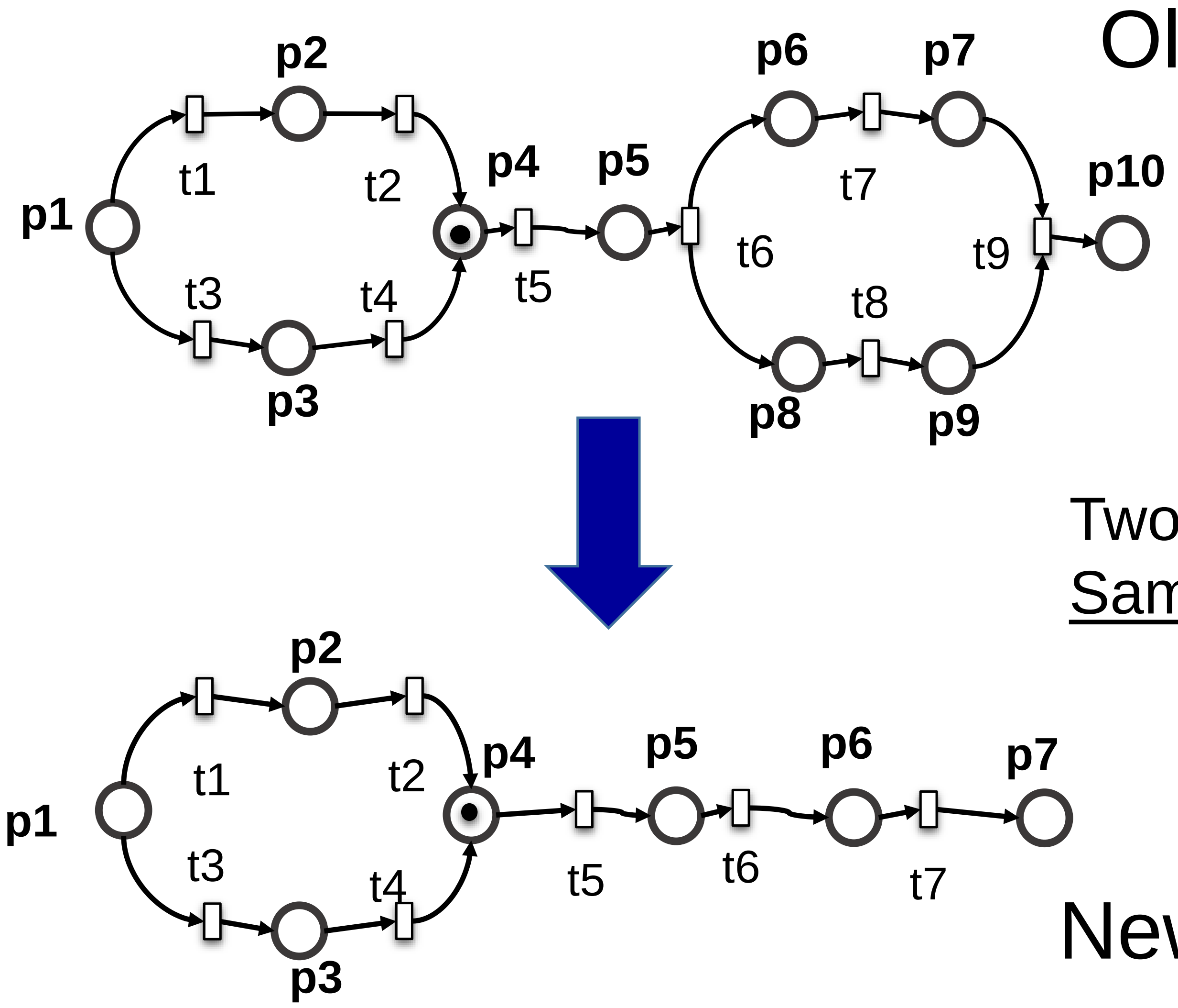
- Vertical trace-reply (through hierarchy of derivation tree) in comparison to traditional horizontal trace-replay (token game)
- Efficient due to ready-made solutions (catalog)
- Restricted scope to CWS-nets and pattern changes

Lookahead algorithms (for new models of consistency) based on trace-reply

# Change Region Approach

# Live Consistency Model

Old Net



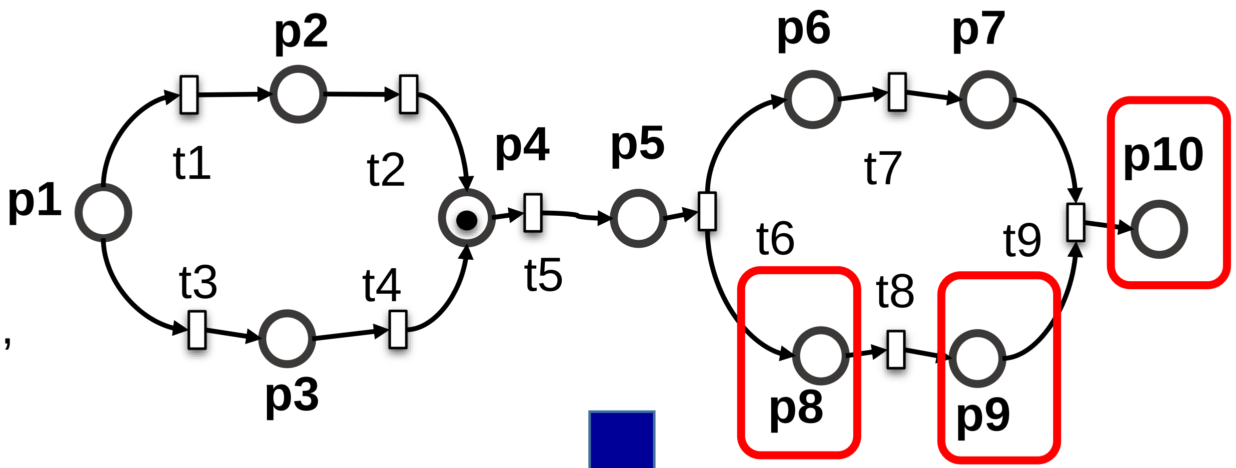

New Net

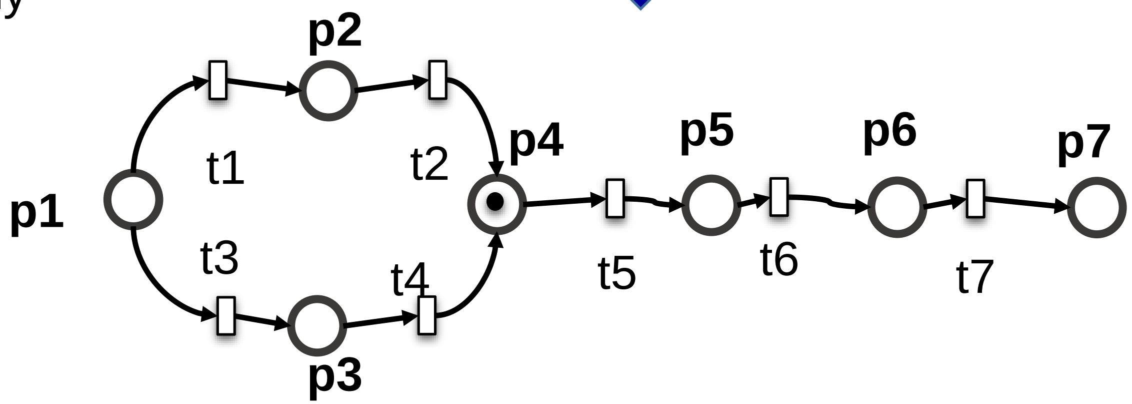
Two consistent markings have the <u>Same set</u> of Marked places

# Change Region

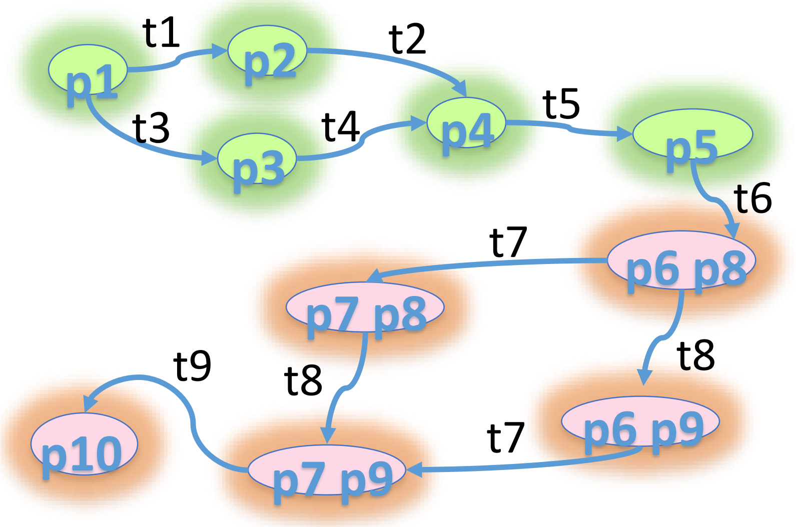
Set of places that causes non-migratable markings

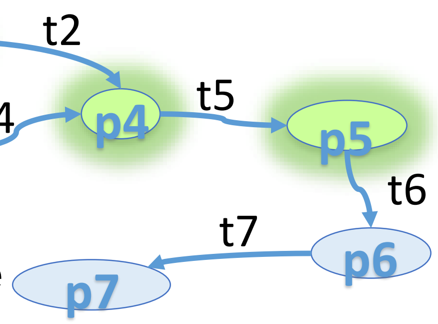
When available before Migration of hundreds of active instances, One can know which of them can be Migrated safely and immediately Without consulting the state space!



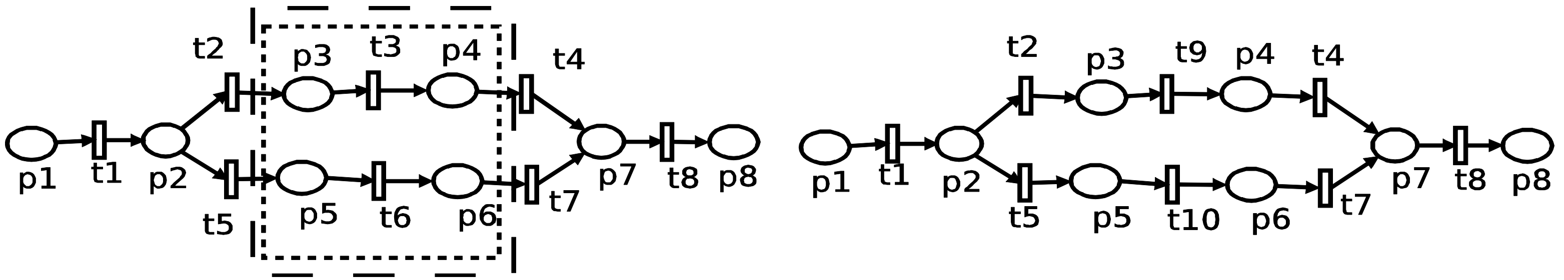

Old State Space
(green states are immediately migratable,i.e. have consistency mappings)



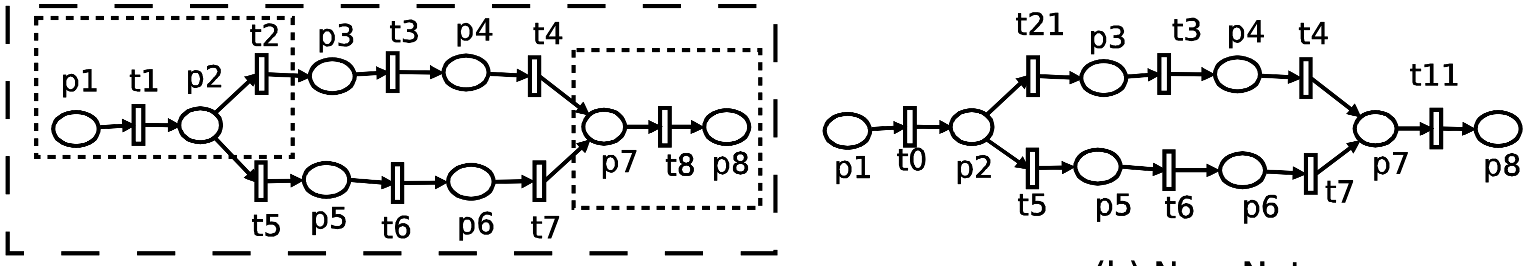




New State Space

# Existing Approach

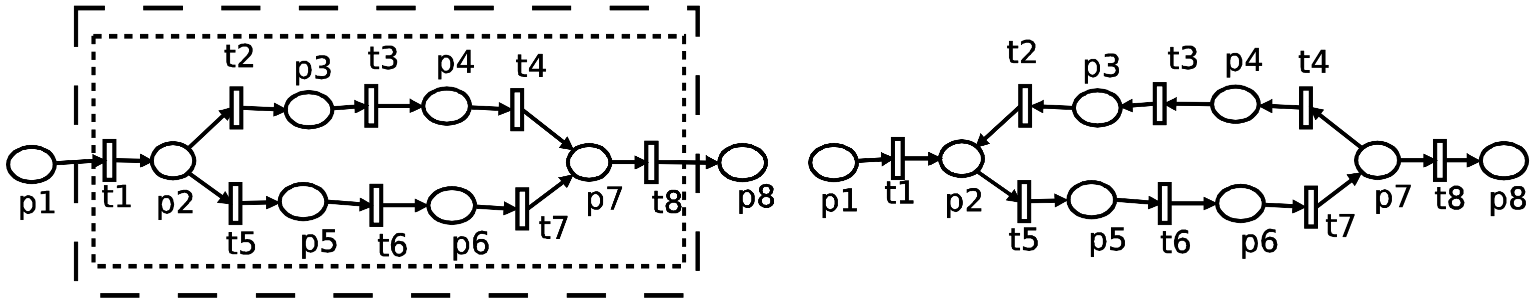



Change Region:
Smallest SESE region covering
Structural changes
(Reasoning:
Whatever change in happens to
State-space, remains confined in
This region)




(a) Old Net

(b) New Net




(a) Old Net

(b) New Net

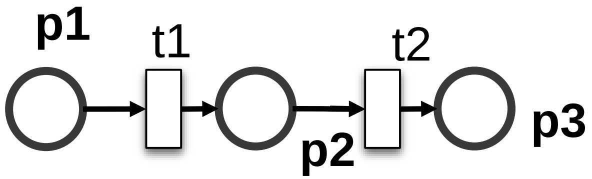
No change in
Reachable states
In the state-space

False-negatives!

# Our Approach

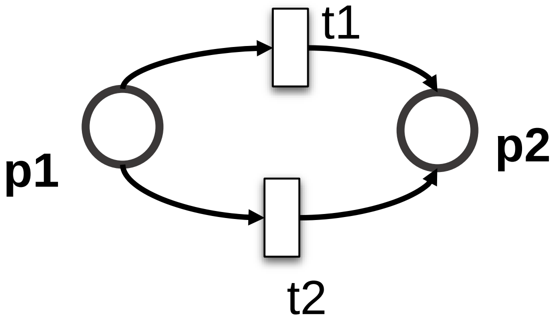
- Structural Model of all possible markings – C-tree
- Analyze causes of non-migratability – Change Properties
- When false-negatives are unavoidable? – overestimation
- Change region devoid of overestimation – PSCR
- Computation of PSCR through change properties

# ECWS Grammar for Workflow Nets



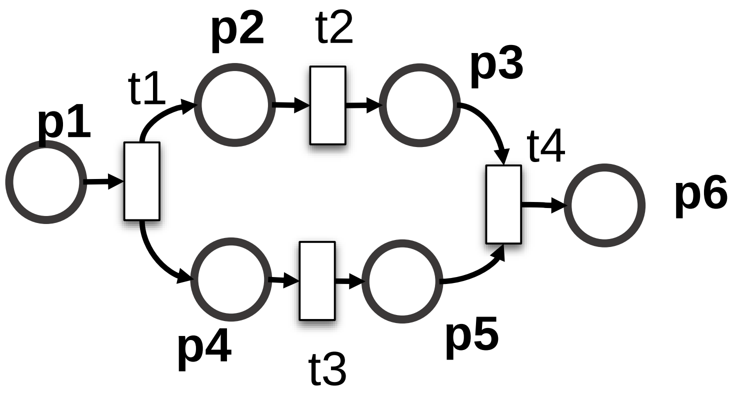

**Sequence**

p1 t1 p2 t2 p3



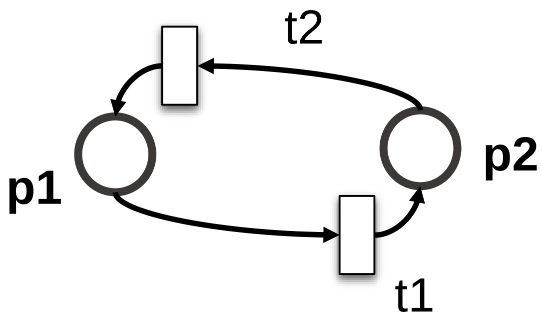

**Choice (XOR)**

p1 [ t1 ] [ t2 ] p2




**Parallel (AND)**

p1 t1 ( p2 t2 p3 ) ( p4 t3 p5 ) t4 p6



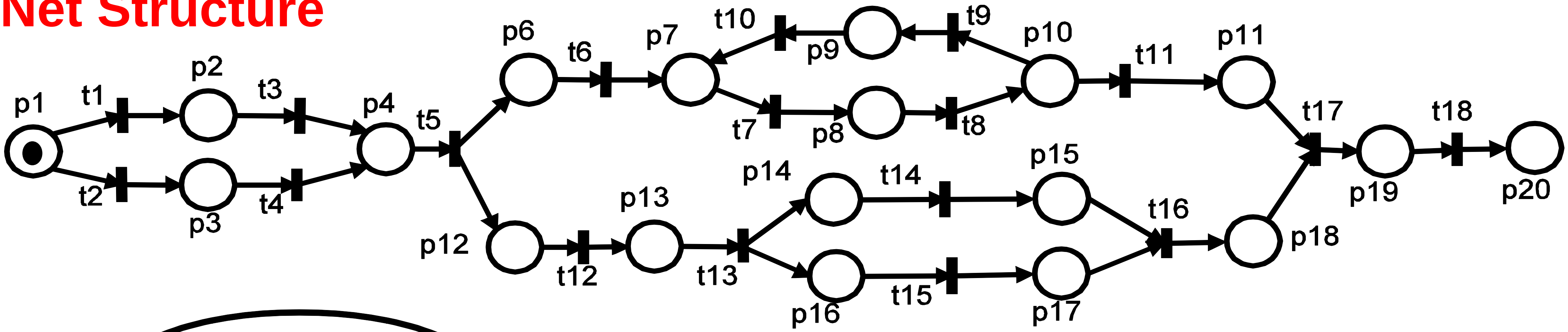

**Loop**

{ p1 t1 p2 } { t2 }

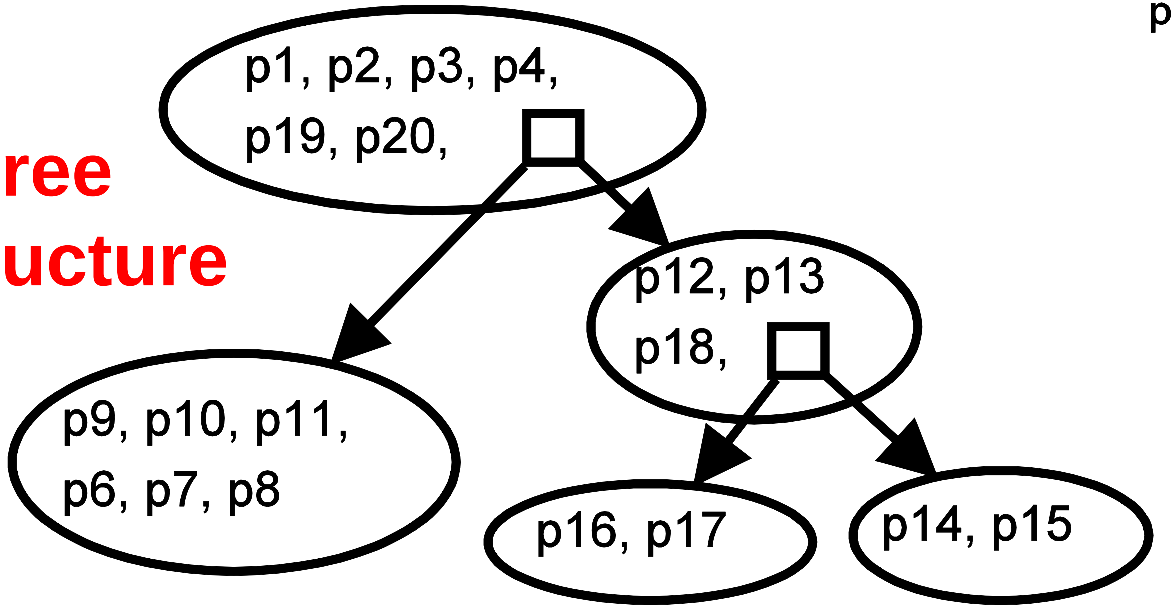
Net → Pnet
Pnet → PLACE
| Pnet TRANS PLACE
| Pnet TRANS loop TRANS Pnet
| Pnet TRANS and TRANS Pnet
| Pnet xor Pnet
Tnet → TRANS | TRANS Pnet TRANS
loop → { Pnet } { Pnet }
xor → [ Tnet ] [ Tnet ] | [ Tnet ] xor
and → ( Pnet ) ( Pnet ) | ( Pnet ) and

# Conjoint Tree (C-tree) abstracts all markings structurally: efficient to compare the old and the new set of markings

## Net Structure




## C-tree Structure



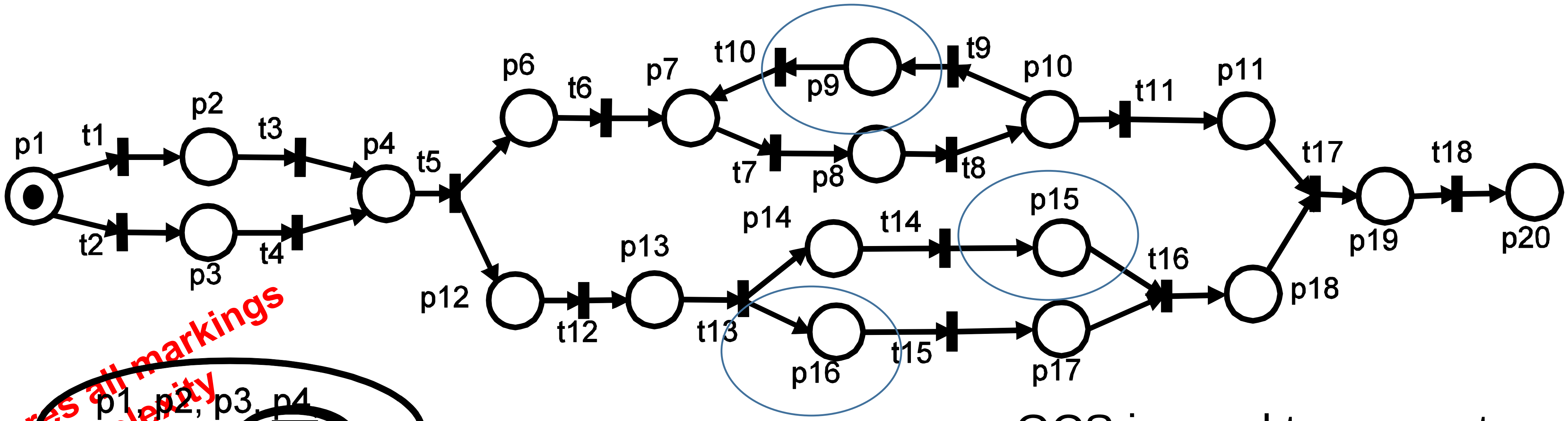

<u>Hierarchy of Nested Concurrency</u>

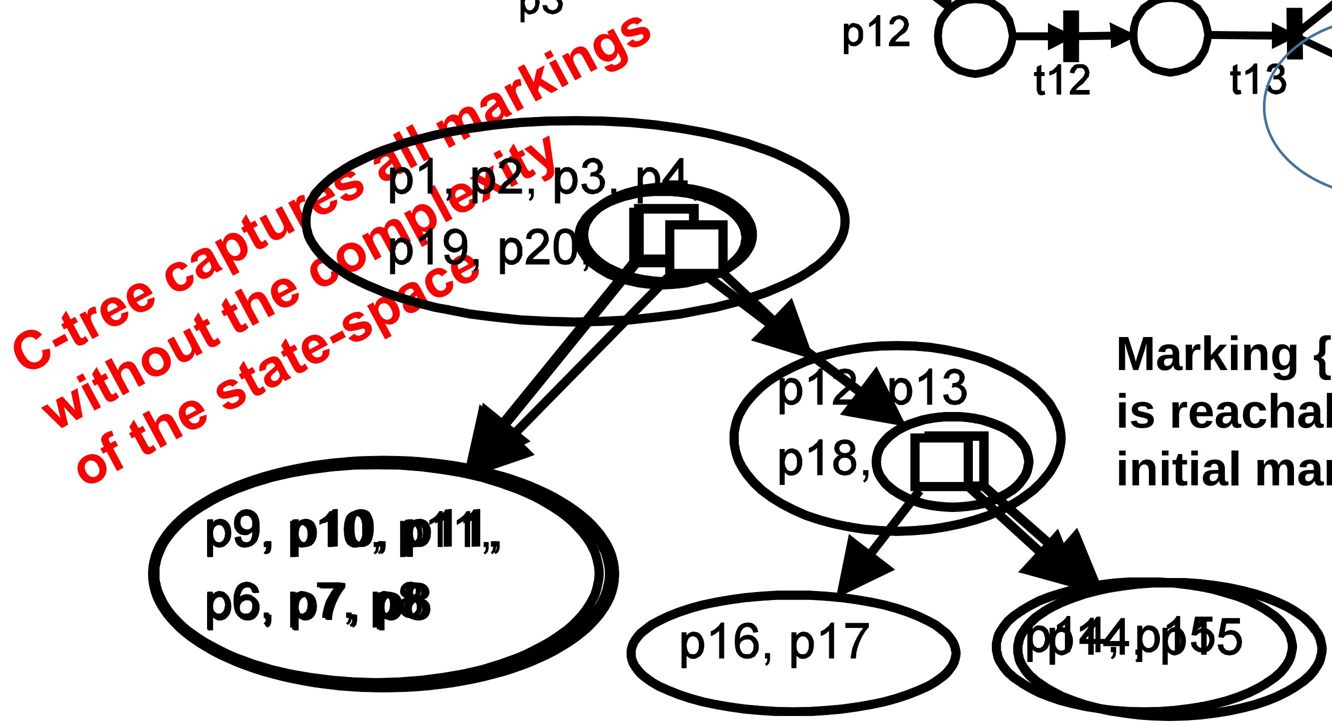
1. sequential places at root
2. Concurrent places at non-root
3. C-block (□) for every AND-block
4. Children of a C-block are AND-branches
5. Recursive structure for nested-AND
6. Places (and C-blocks) in parent-child nodes are non-concurrent to each other

(also places in same node are non-concurrent)

# Generator of Concurrent Submarking (GCS):

captures **the concurrent part of the net w.r.t. a place**



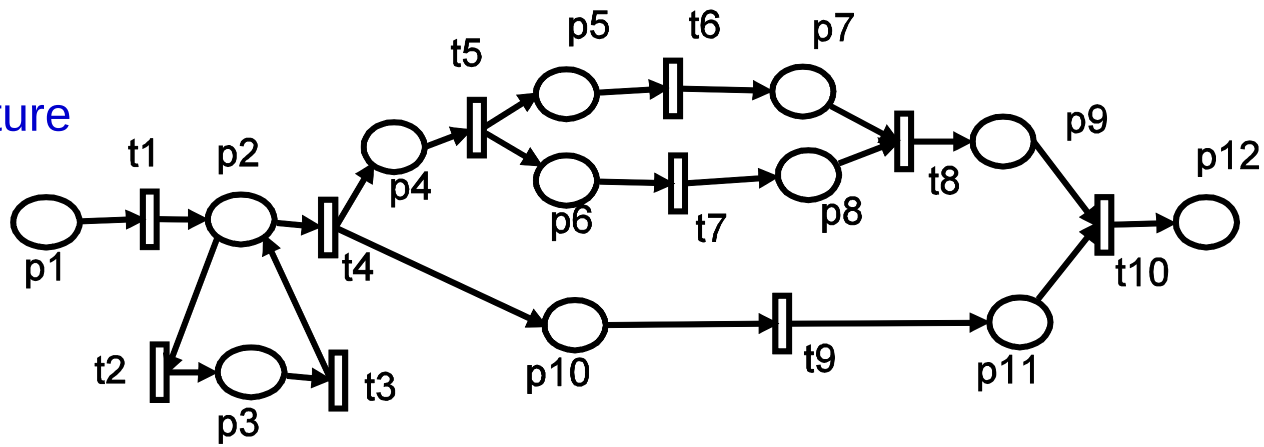

**Marking { p16, p15, p9 } is reachable from initial marking { p1 }**

GCS is used to generate Reachable Markings

GCS is the key to identify any change happened in the Concurrency when two nets are considered

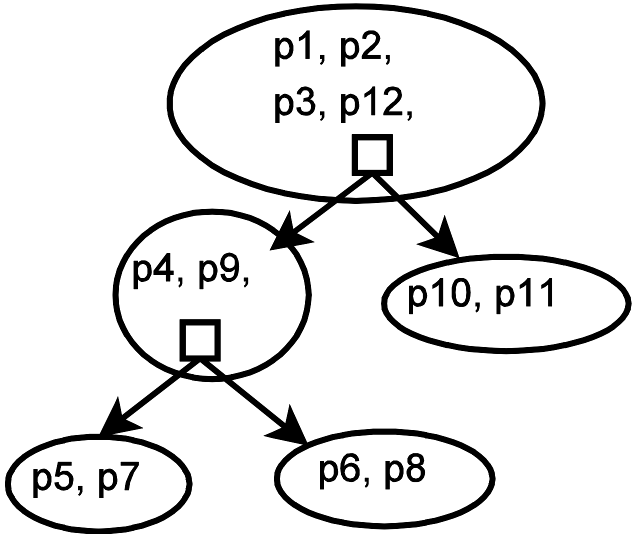
# Dysfunctional C-tree & Break-off Set

Net Structure



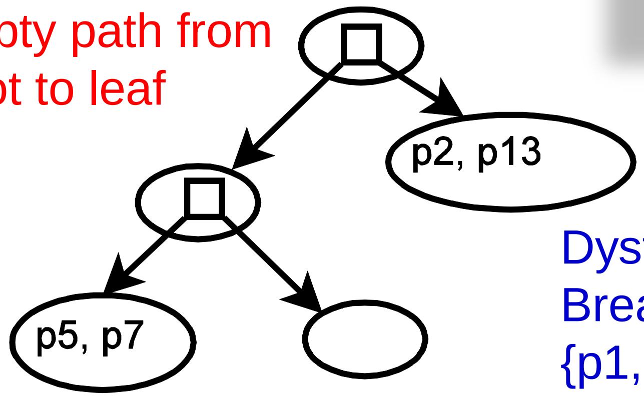

ECWS spec

p1 t1 {p2}{t2 p3 t3} t4 (p4 t5 (p5 t6 p7)(p6 t7 p8) t8 p9)(p10 t9 p11) t10 p12




C-tree

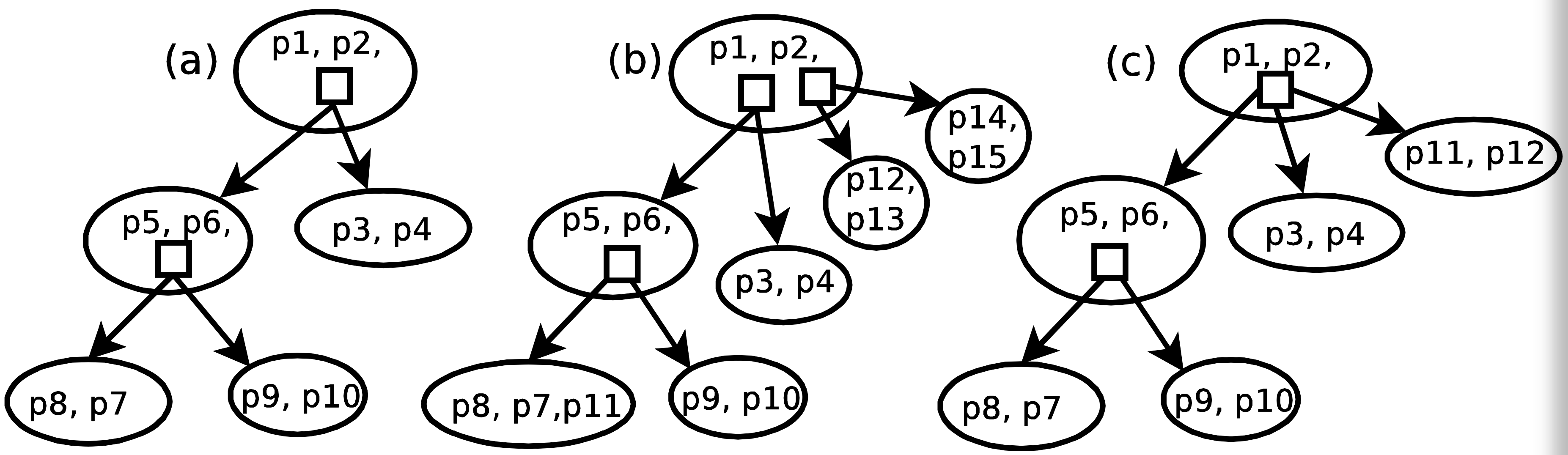
Empty path from
Root to leaf




Dysfunctional C-tree,
Break-off set:
{p1,p2,p3,p4,p6,p8,p9,p12}

If after removal of some places from a C-tree, the mutated C-tree cannot generate a marking that can be generated from the original C-tree, the mutated C-tree is called dysfunctional, and the set of places removal of which renders the tree dysfunctional is called as a break-off set for that C-tree.

# Marking Preserving Embedding (MPE)




**Definition:**

An MPE of C in C' is a mapping from C to C' defined recursively: (i) places(root(C)) is subset of places(root(C′)), and (ii) C-blocks in root(C) are *injectively* mapped to C-blocks in root(C′) such that within each C-block to C-block mapping pair (b,b′), there is a *bijective* MPE of the children C-trees of b to those of b′.

**Non-migratability Lemma:**

Given two C-trees C and C', if an MPE of C in C' does not exists, then at least one marking constructible from C cannot be constructed from C'. The converse is also true.

# Effect of Concurrency of a place on Migratability

| Case id | Concurrent in Old Net | Concurrent in New Net | Migratability |
|---|---|---|---|
| C1 | ✓ | ✓ | Conditionally Migratable |
| C2 | ✓ | ✗ | Non-migratable |
| C3 | ✗ | ✓ | Non-migratable |
| C4 | ✗ | ✗ | Migratable |
| C5 | Don’t care | Absent | Non-migratable |

Example:
C1: {p1,p2} → {p1,p2}; {p1,p2} → {p1,p2,p3}….
C2: {p1,p2} → {p1}
C3: {p1} → {p1,p2}
C4: {p1} → {p1}
C5: {p1} or {p1,p2} → no marking with p1 available

A concurrent place has more than one marking in an ECWS-net

A non-concurrent place has only one marking in an ECWS-net

# Change Properties

Old net N, New net N',
Old C-tree C, New C-tree C'
p is the place to be inspected for its effect on non-migratability

**(C5) Removal**
No marking involving p is reachable in N'. (p is present in C, p is absent in C').

**(C2) Lost Concurrency**
Markings involving p are concurrent in N but not in N' (p in a non-root node in C but in the root node in C').

**(C3) Acquired Concurrency**
Only one standalone marking involving p in N, but concurrent markings in N' (p is in root node of C but in a non-root node in C').

**(C1.1) Weak Reformed Concurrency**
(i) p is in concurrent markings in both N and N' (p in non-root nodes in both C and C')
(ii) at least one concurrent marking involving p in N is not reachable in N' due to addition or reduction of concurrency of p (GCS(p,C) does not have an MPE in GCS(p,C').)

**(C1.2) Strong Reformed Concurrency**
(i) p is in concurrent markings in N and N' (p in non-root nodes both in C and C')
(ii) all concurrent markings involving p in N are not reachable in C'. ( either the set of places {places(GCS(p,C)) - places(GCS(p,C'))} is a break-off set w.r.t. C-tree GCS(p,C), or the set of places {places(GCS(p,C')) - places(GCS(p,C))} is a break-off set w.r.t. C-tree GCS(p,C'). )

Justification: p in non-root nodes implies involvement in concurrency. When condition is satisfied, it means that the places which are concurrent to p in both nets are not capable of generating any common valid marking involving p.

# Characterization of Change Region

**Structural Change Region (SCR)**

Given a migration net pair N and N’, SCR(N,N’) is a subset of places in N s.t.
for every non-migratable marking M from N to N’, M includes a member from SCR(N,N’).

**Perfect Structural Change Region (PSCR)**

Given a migration net pair N and N’, PSCR(N,N’) is a subset of places in N s.t.

i. for every place p in PSCR(N,N’), there exists a non-migratable marking from N to N’ involving p,
ii. for every non-migratable marking M from N to N’, M includes a member from PSCR(N,N’).

**Perfect Member**

A place p in the old net N is a perfect member in N, w.r.t. the new net N’ iff all markings in N involving p are non-migratable.

**Overestimation**

A place p in the old net N is an overestimation w.r.t. the new net N’, iff there exists a migratable marking and also a non-migratable marking involving p in N.

**Safe Member**

A place p in the old net N is a safe member w.r.t. the new net N’, iff every marking involving p in N is migratable.

# Change Region Lemmas

**SCR Lemma**

The union of all overestimations and all perfect members in old net N w.r.t. new net N’ is SCR in N w.r.t. N’.

**PSCR Lemma**

PSCR exists in a given old net N w.r.t. new net N’
iff every non-migratable marking in N includes at least one perfect member.
(the proof constructs the set of perfect members as the PSCR)

**Perfect Member Lemma**

If a place p in old net N satisfies one of Removal, Lost Concurrency, Acquired Concurrency and Strong Reformed Concurrency w.r.t. new net N’, then the place is a perfect member and vice-versa.

**Overestimation Lemma**

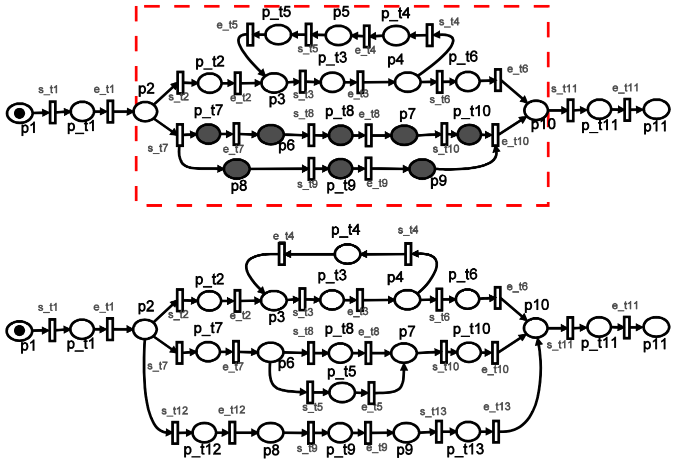
If a place p in old net N satisfies Weak Reformed Concurrency but not Strong Reformed Concurrency w.r.t. new net N’, it is an overestimation w.r.t. N’ and the vice-versa.

# Computation of PSCR

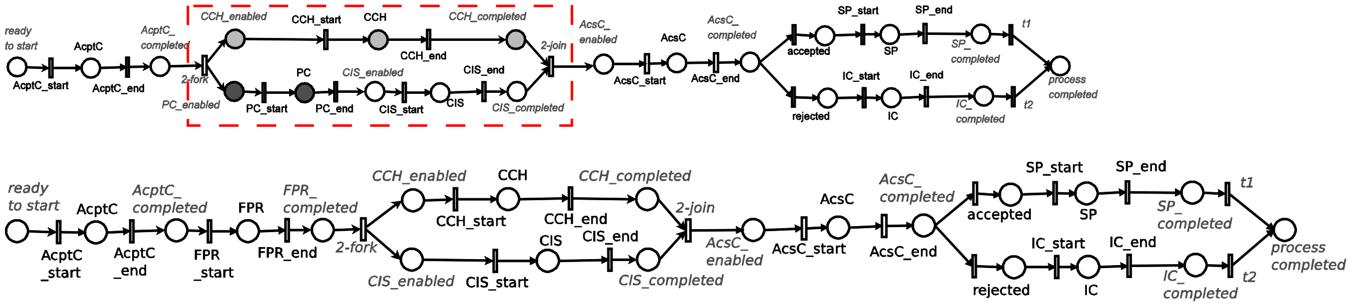
Old C-tree C, New C-tree C', set of Perfect Members in C w.r.t. C' be *Perf.*
If there is no overestimation in C, *PSCR* ← *Perf.*
*Else if* PSCR exists as per the following table, *PSCR* ← *Perf.*

| C has markings without Perfect Members | C' has markings without Perfect Members | All markings without Perfect Members in C can be generated from C' | PSCR Exists |
|---|---|---|---|
| ✗<br>Perf is break-off set for C | Don't care | Don't care | ✓ |
| ✓<br>Perf is not break-off set for C | ✗<br>Perf is break-off set for C' | Don't care | ✗ |
| ✓<br>Perf is not break-off set for C | ✓<br>Perf is not break-off set for C' | ✗<br>delete(C,Perf) doesn't have MPE in delete(C′,Perf) | ✗ |
| ✓<br>Perf is not break-off set for C | ✓<br>Perf is not break-off set for C' | ✓<br>delete(C,Perf) has a MPE in delete(C′,Perf) | ✓ |

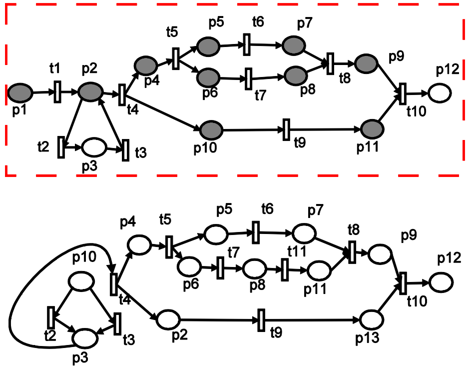
# Experimental Results



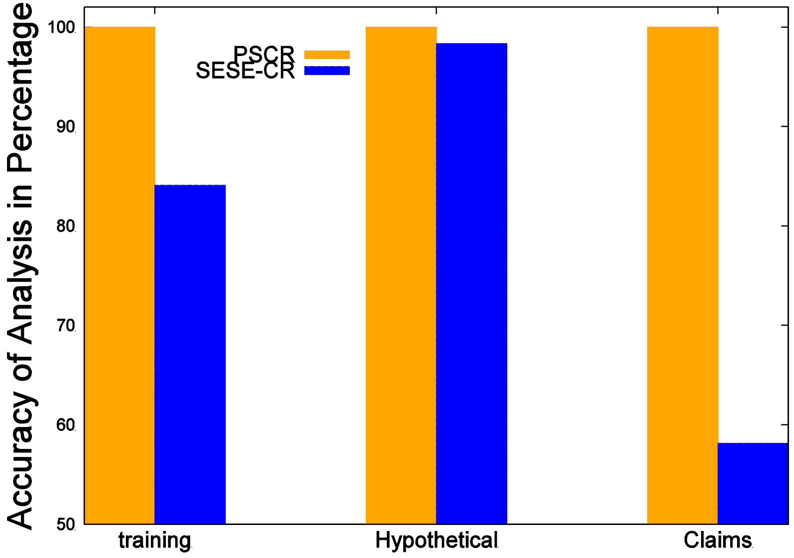
Training Process

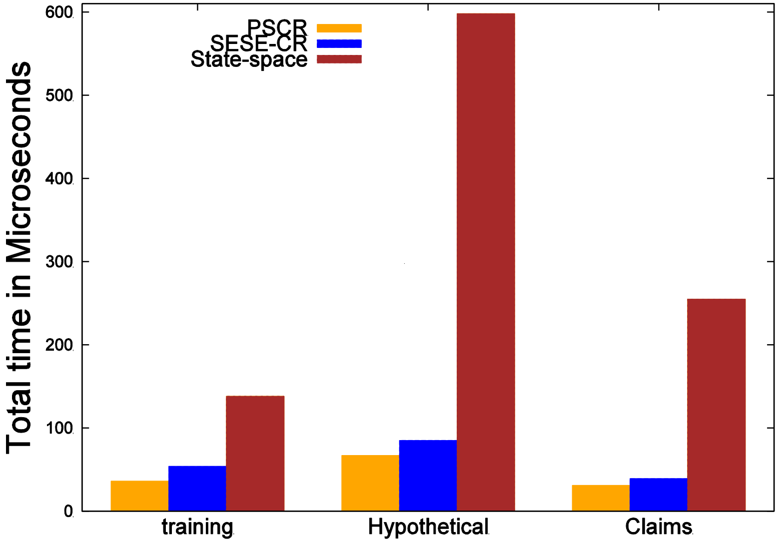



Claims Process



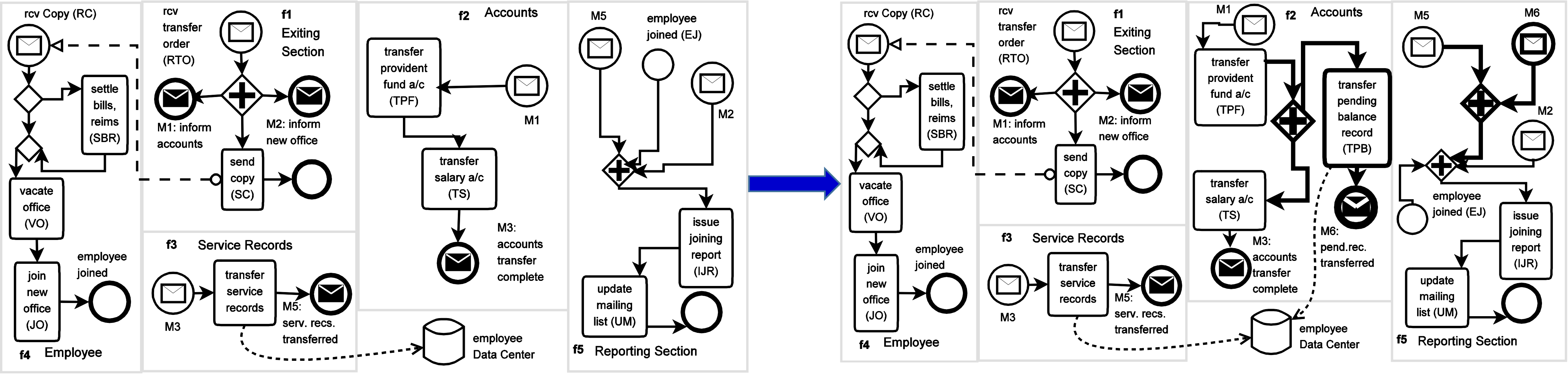
Hypothetical Process



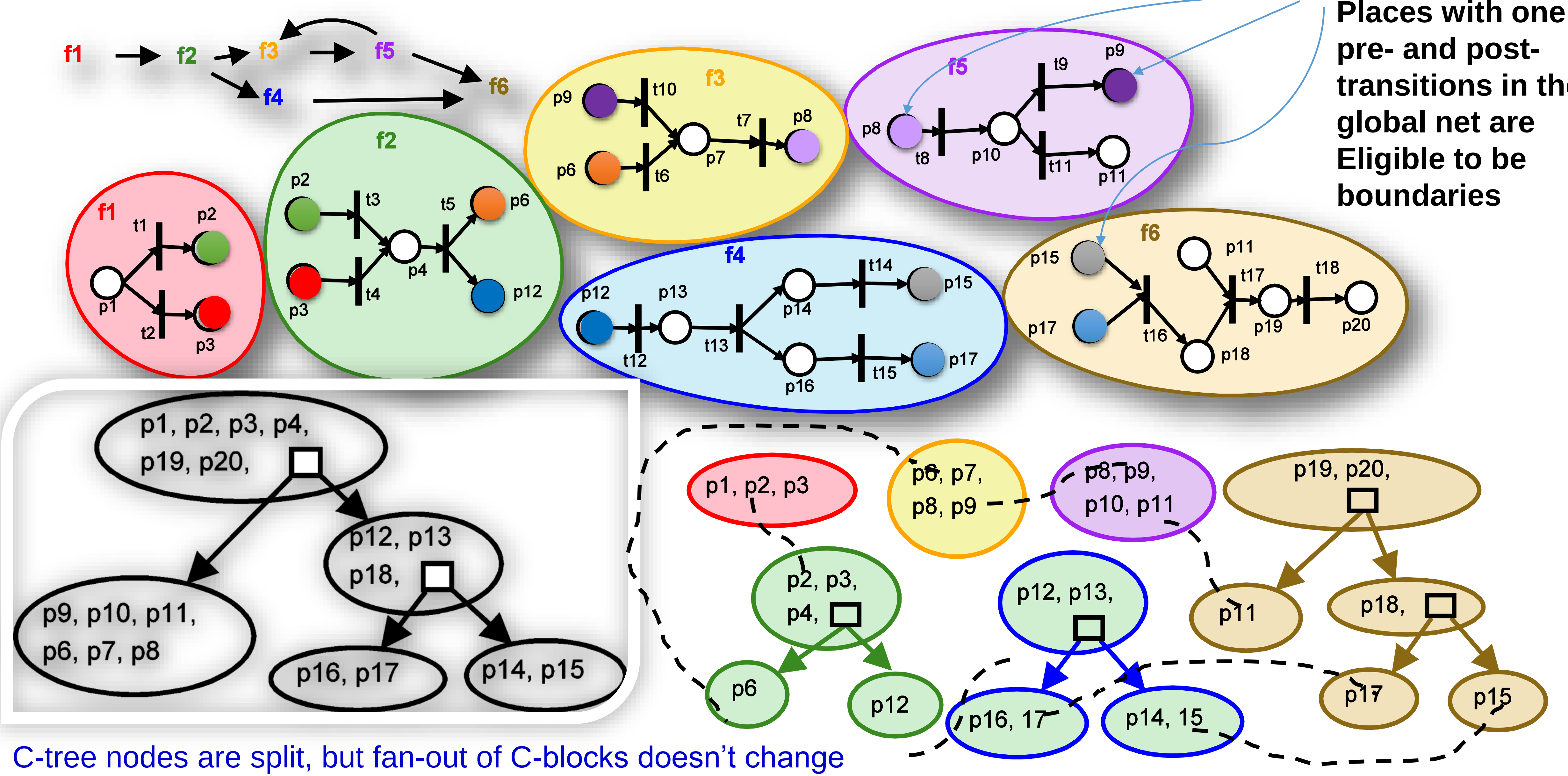
# Distributed Business Processes

- Business processes often cross departments, organizations
- Distributed deployments lack centralized view
- Individual process-fragments may evolve independently
- Individual change regions can be globally conflicting




Employee Transfer Process

# Fragmented Net has no Global View

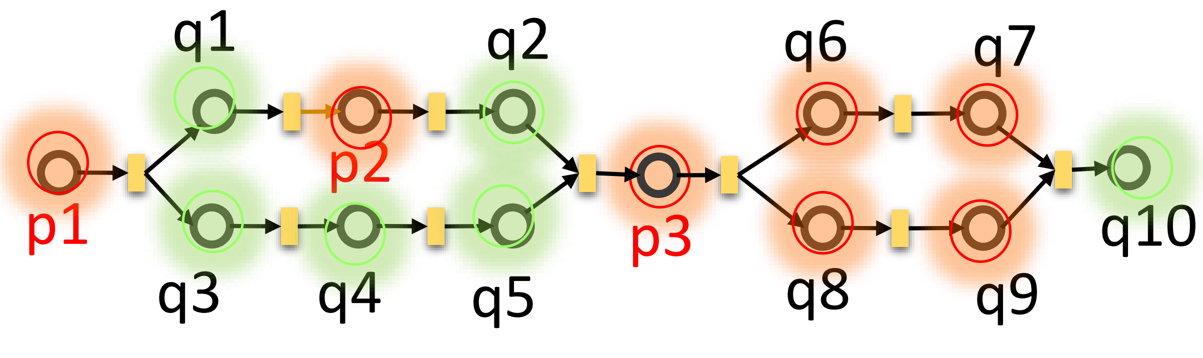


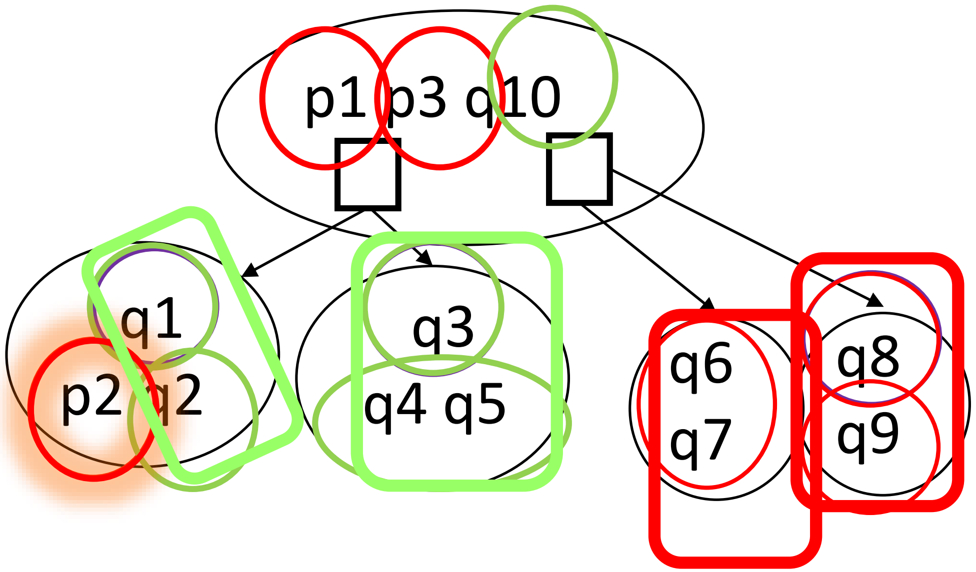

## Boundaries

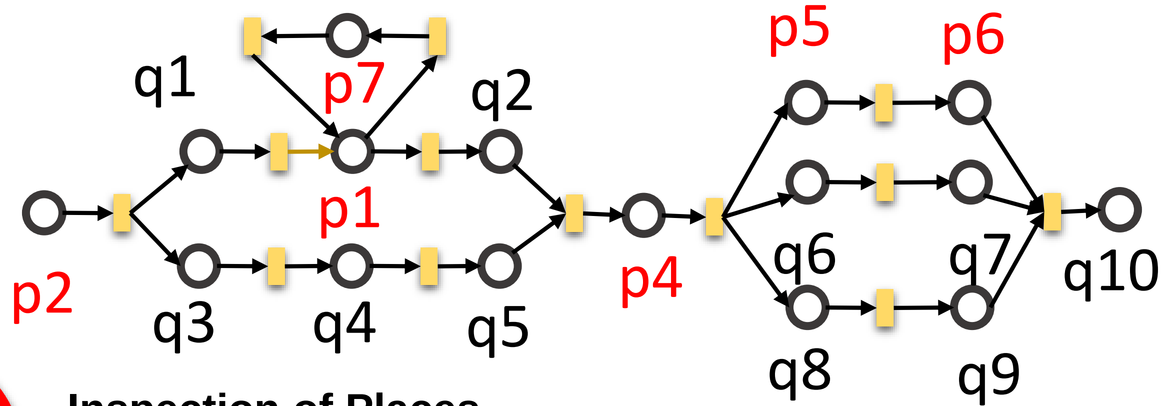
**Places with one pre- and post-transitions in the global net are Eligible to be boundaries**

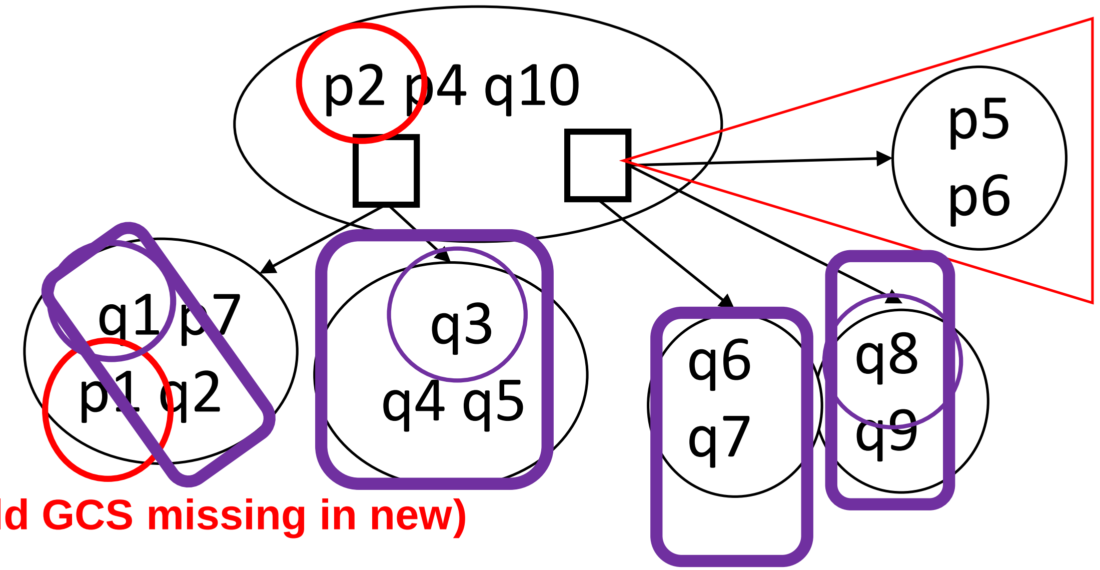
C-tree nodes are split, but fan-out of C-blocks doesn't change

# Change Region Computation when centralized view is available (includes overestimates)



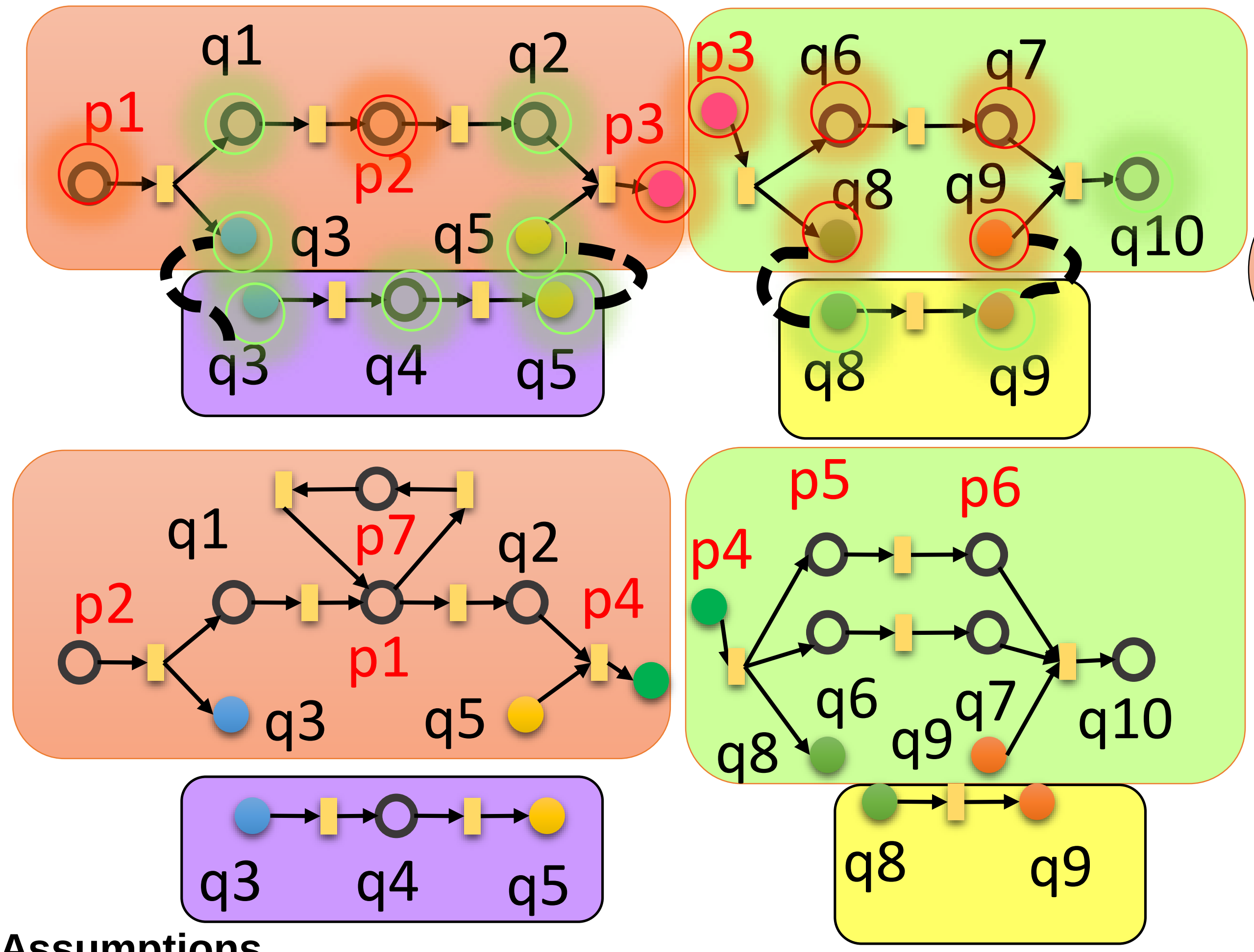



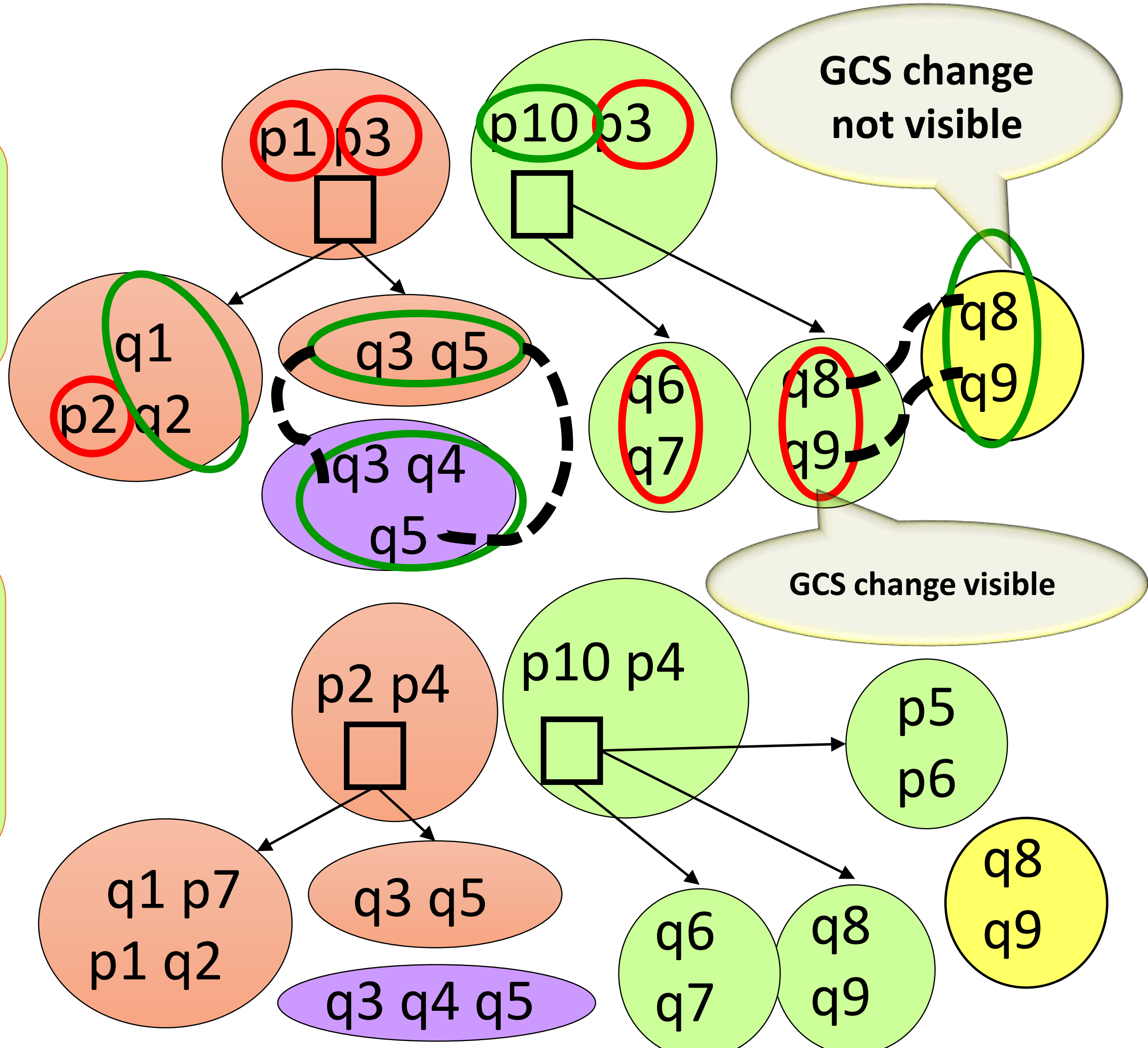

**Change in GCS**

**Inspection of Places**

1. **{ p3 } → no marking with p3 available (deleted)**
2. **{ p1 } → { p1, _ } ( root → non-root )**
3. **{ p2, _ } → { p2 } (non-root → root )**
4. **{ q3, _ } → { q3, _ }, but { q3, p2 } not available (places of old GCS missing in new)**
5. **{ q8, _ } → { q8, _ , _ } (new branches in GCS)**

# Effect of Fragmentation



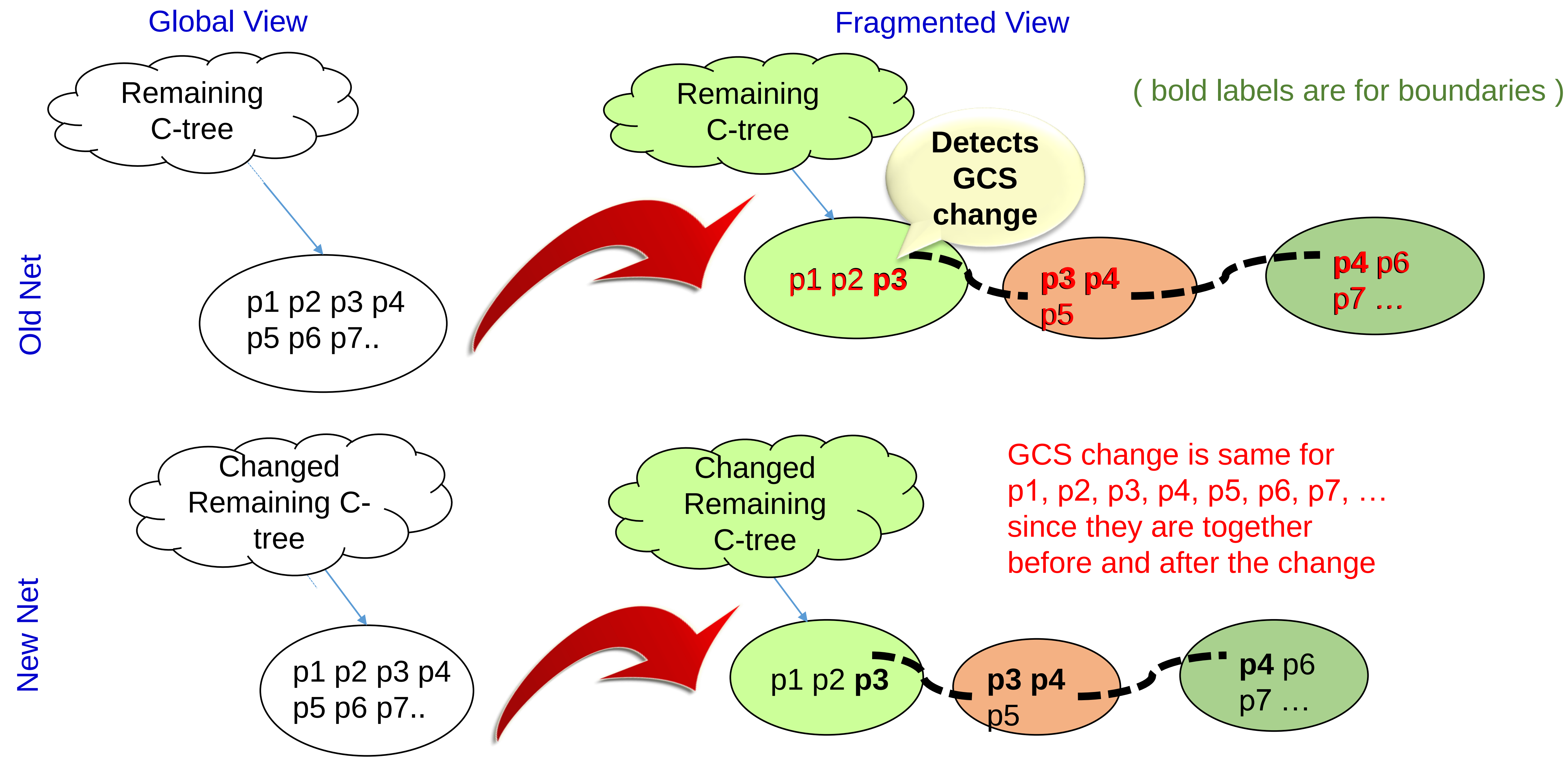

**<u>Assumptions</u>**

**A place does not jump to another fragment (root ←→ non-root locally visible)**

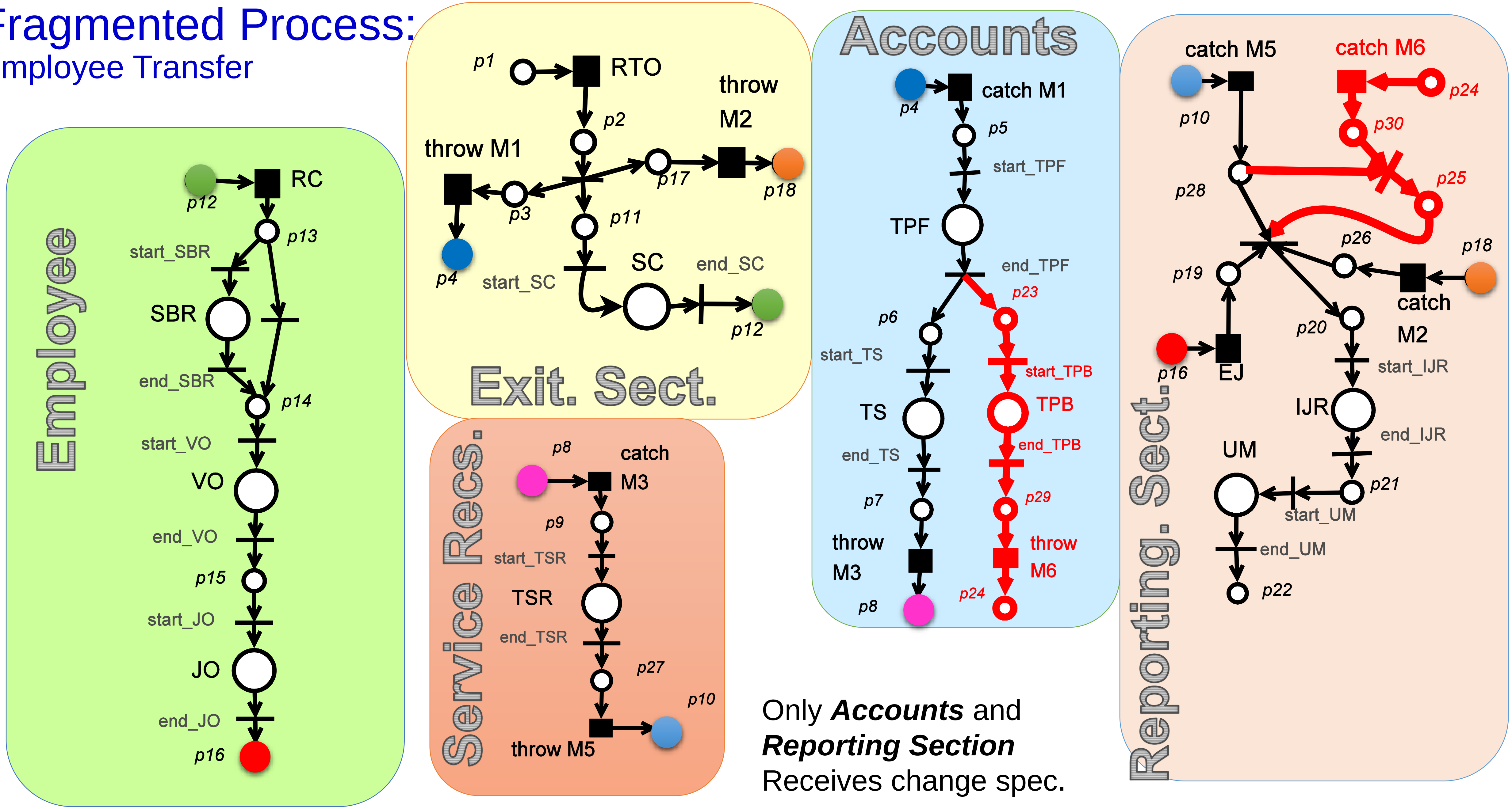
**No boundary place becomes internal**

**Deletion of boundary is consistent between peer fragments } (place deletion locally visible)**

# Conflict resolution between fragments using boundaries

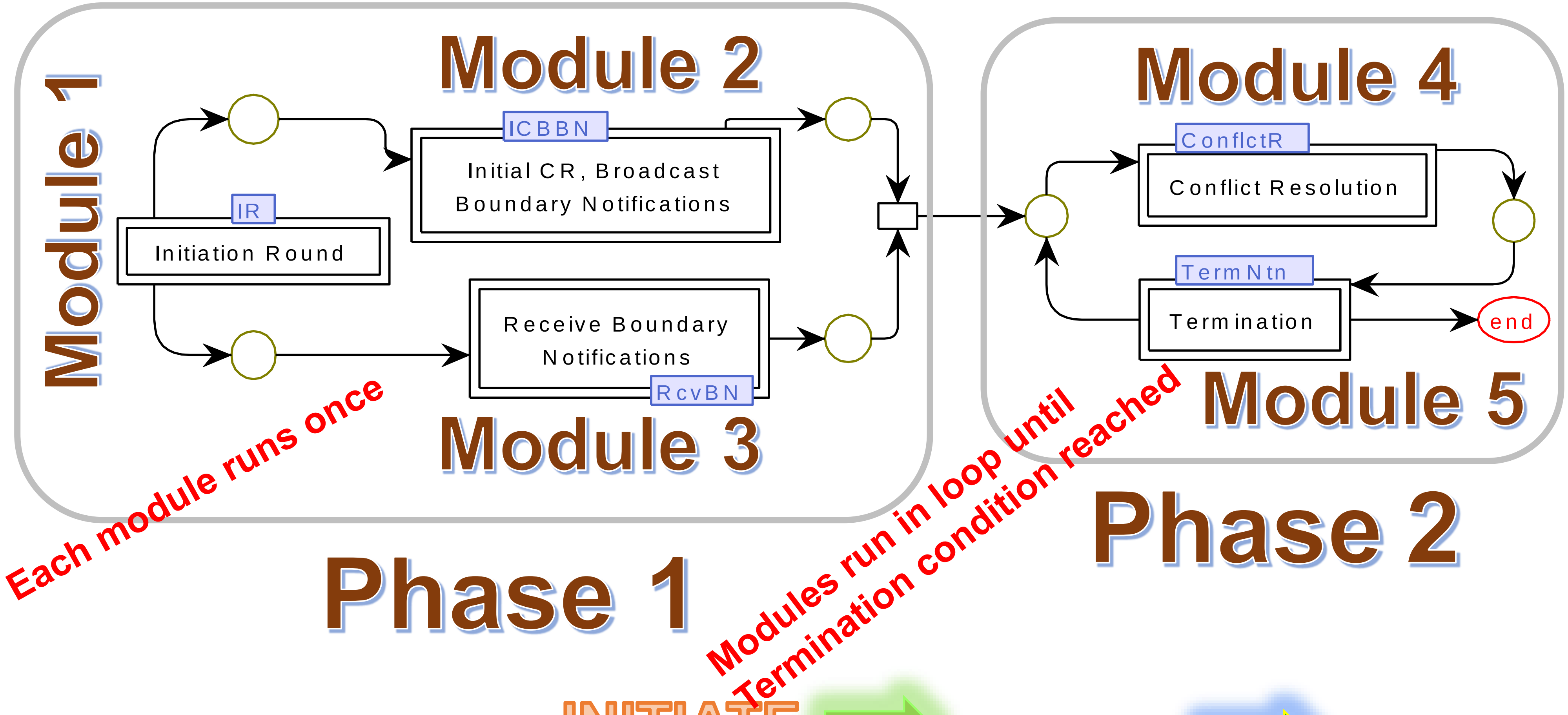


# Fragmented Process:
## Employee Transfer

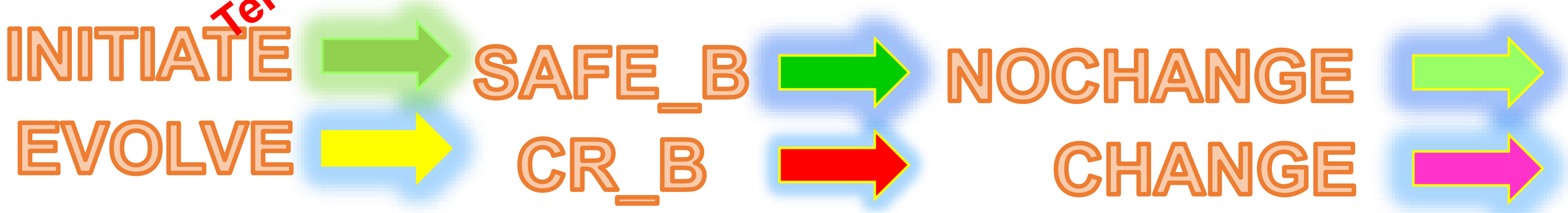



Only ***Accounts*** and ***Reporting Section*** Receives change spec.

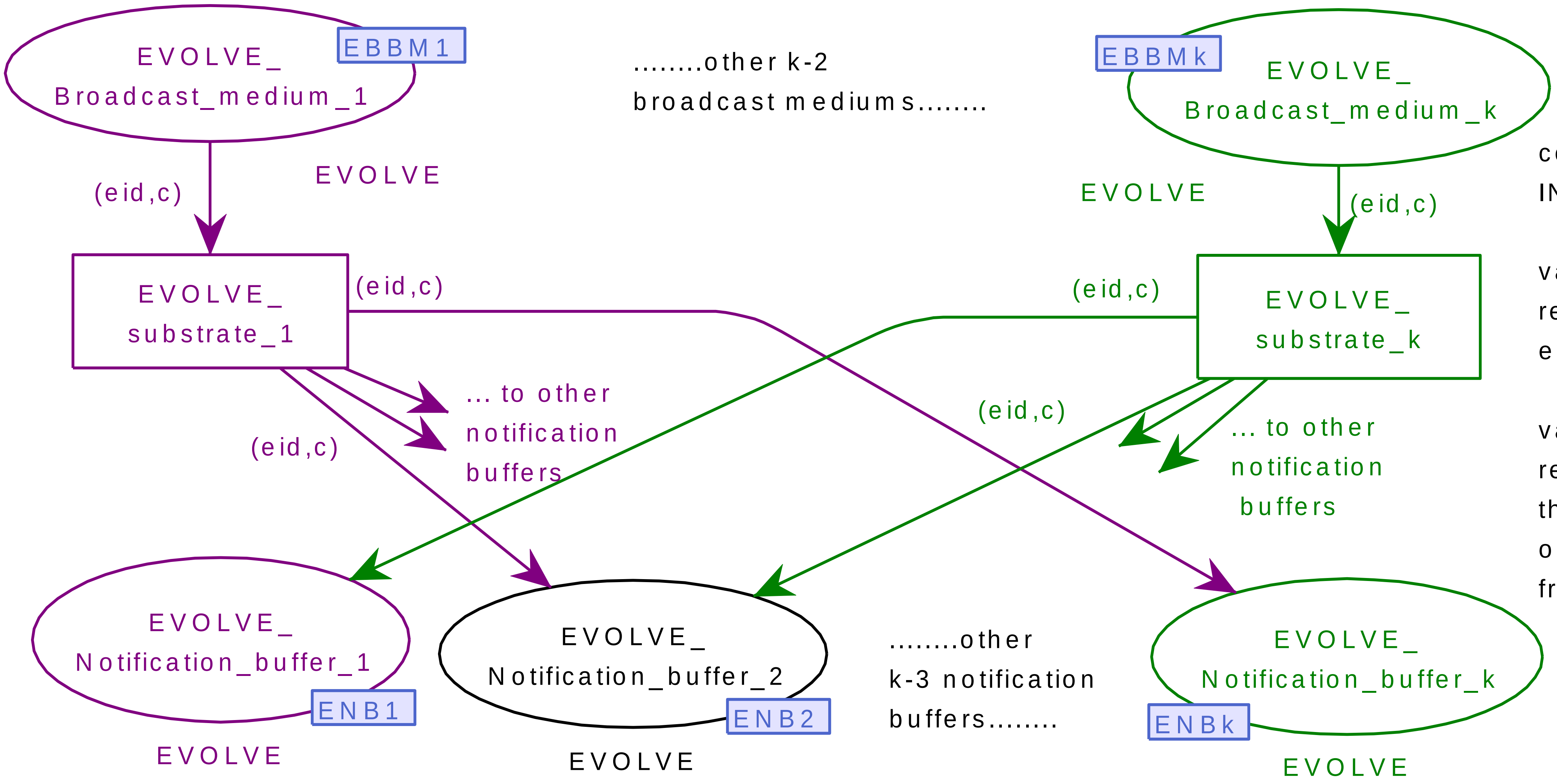
# The Distributed Algorithm using Asynchronous Events:
## Hierarchical CPN based description




**EVENT TYPES → Color definitions.**
**Event parameters → values in tokens**



# Event Substrate: example for EVOLVE event type

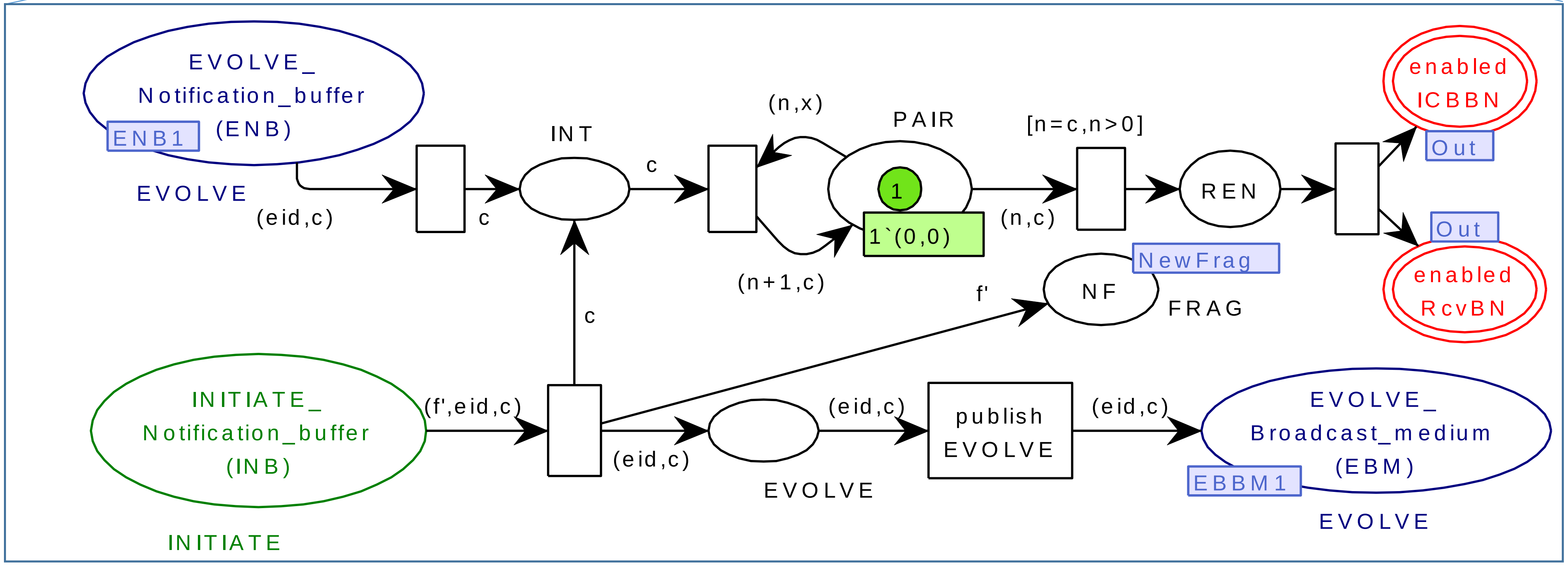



color EVOLVE is INT x INT;

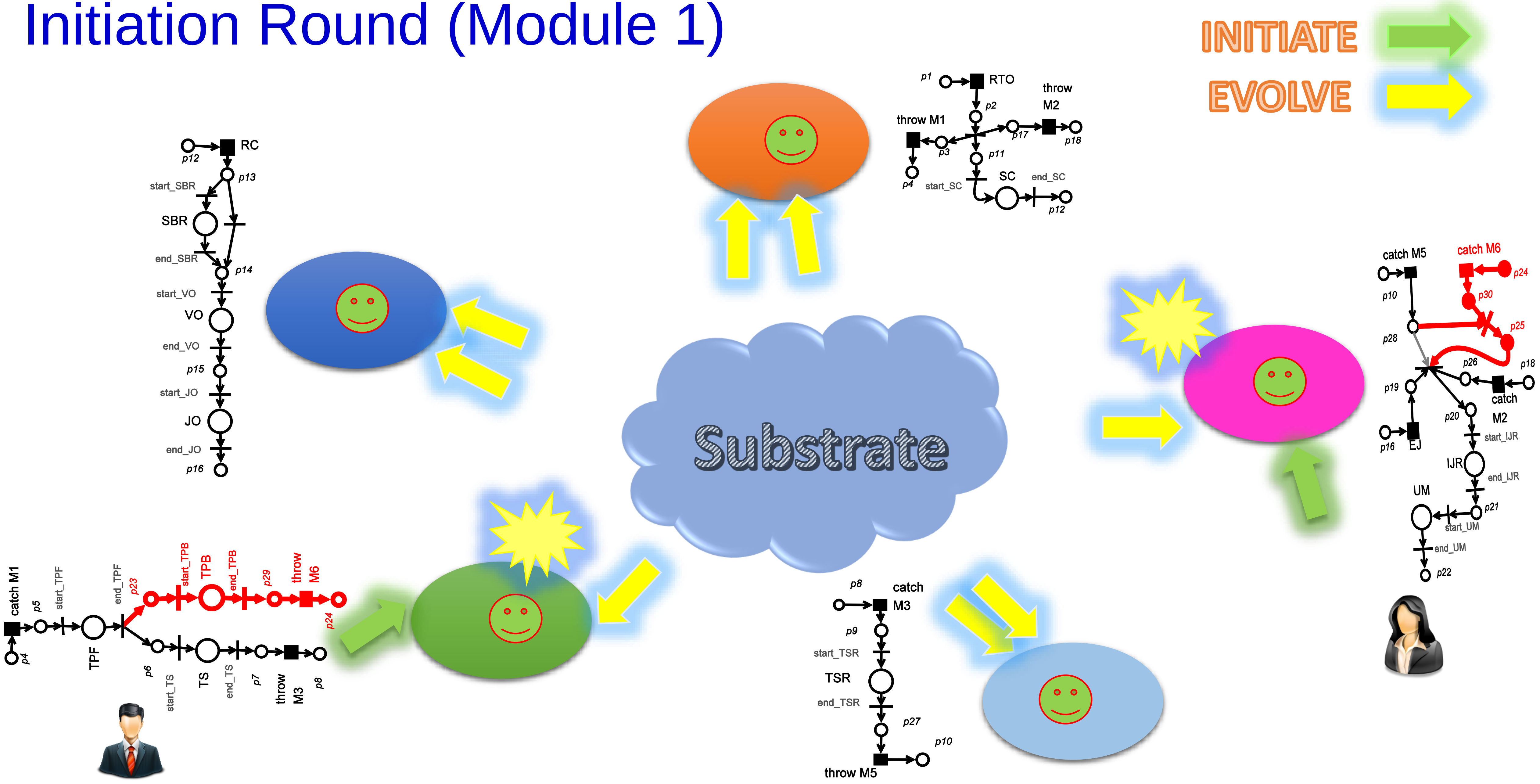
var eid: INT, represents evolution id;

var c: INT, represents the number of changing fragments;

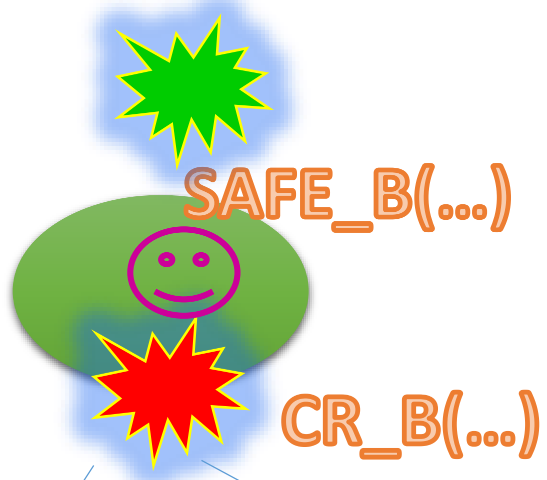
# Initiation Round (Module 1)

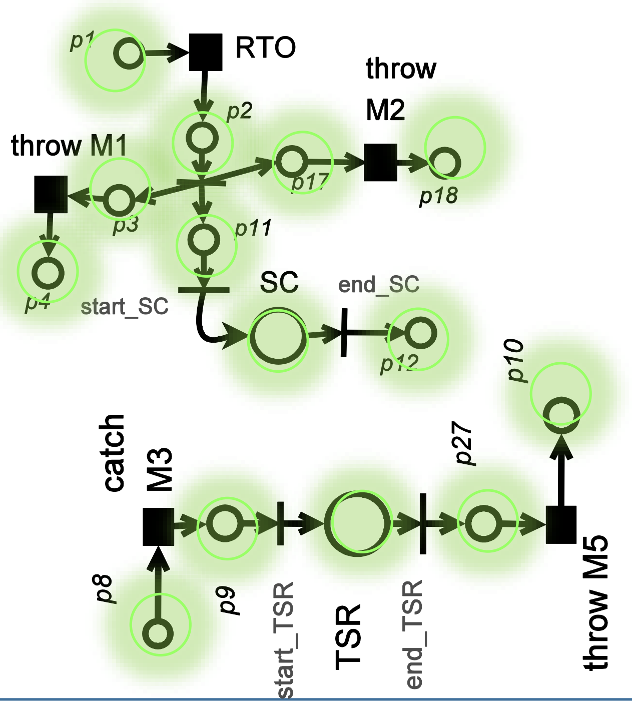



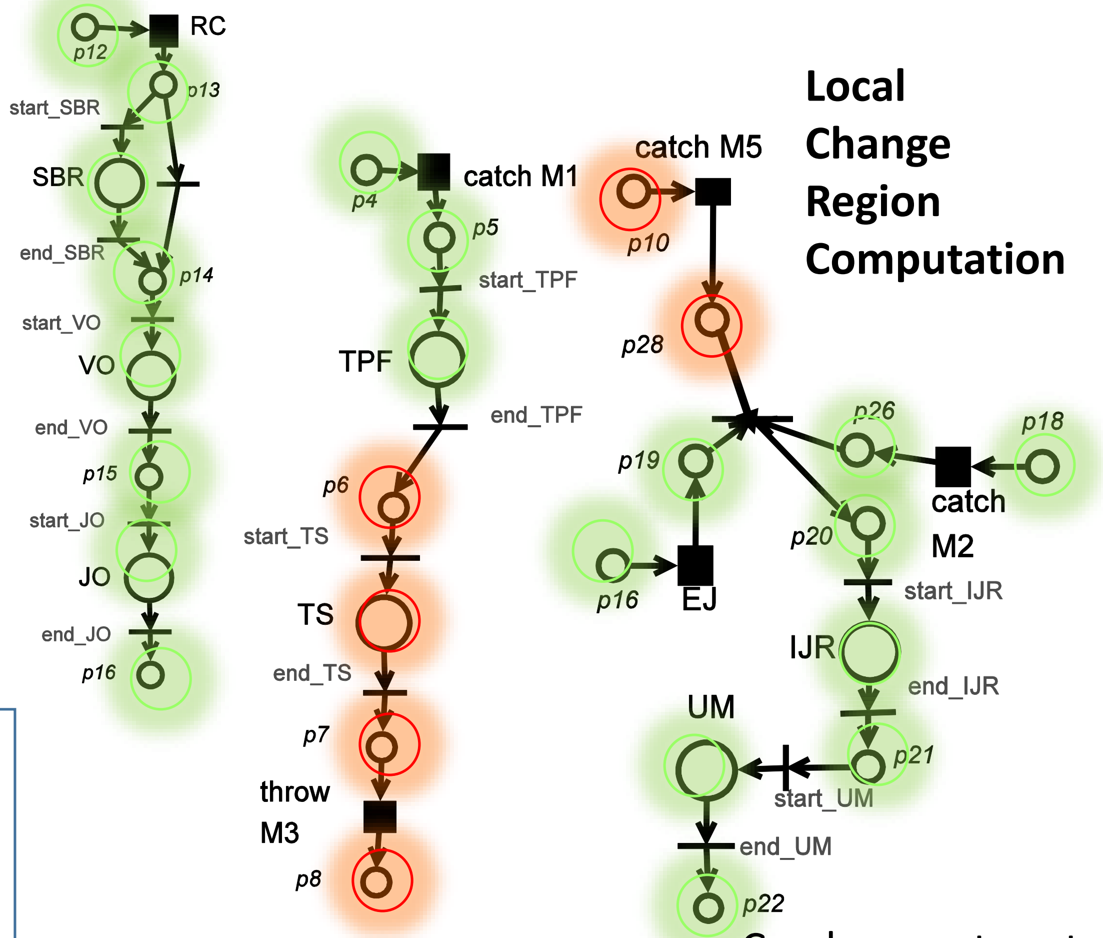
# Initiation Round (Module 1)



# Initial Change Region, Broadcast Boundary Notifications (Module 2)



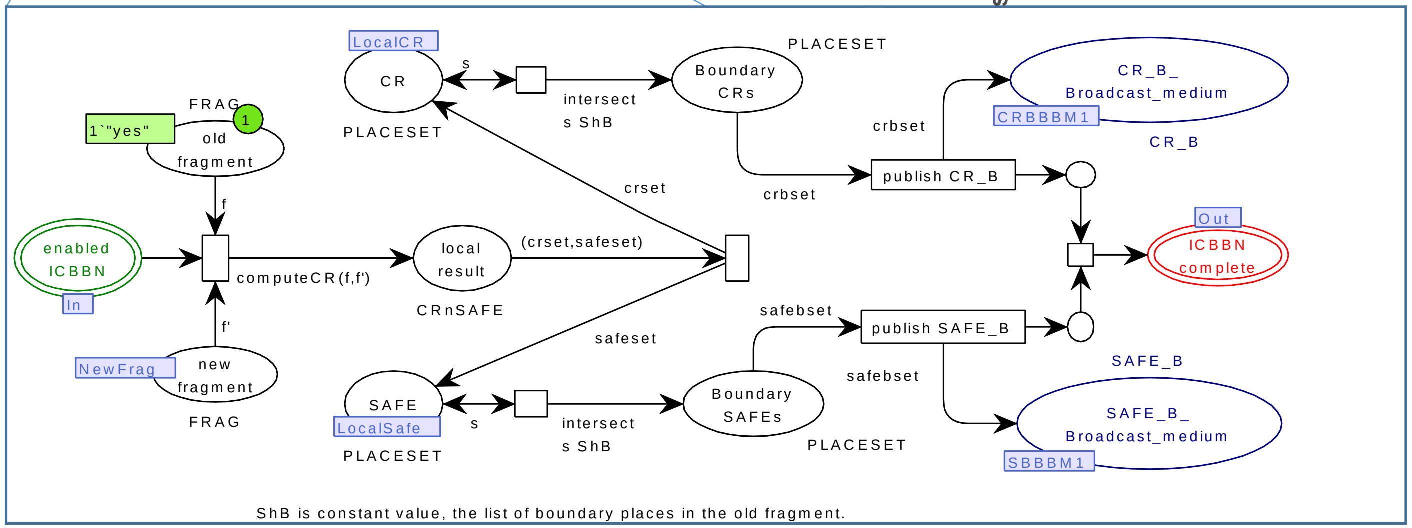




**Local Change Region Computation**

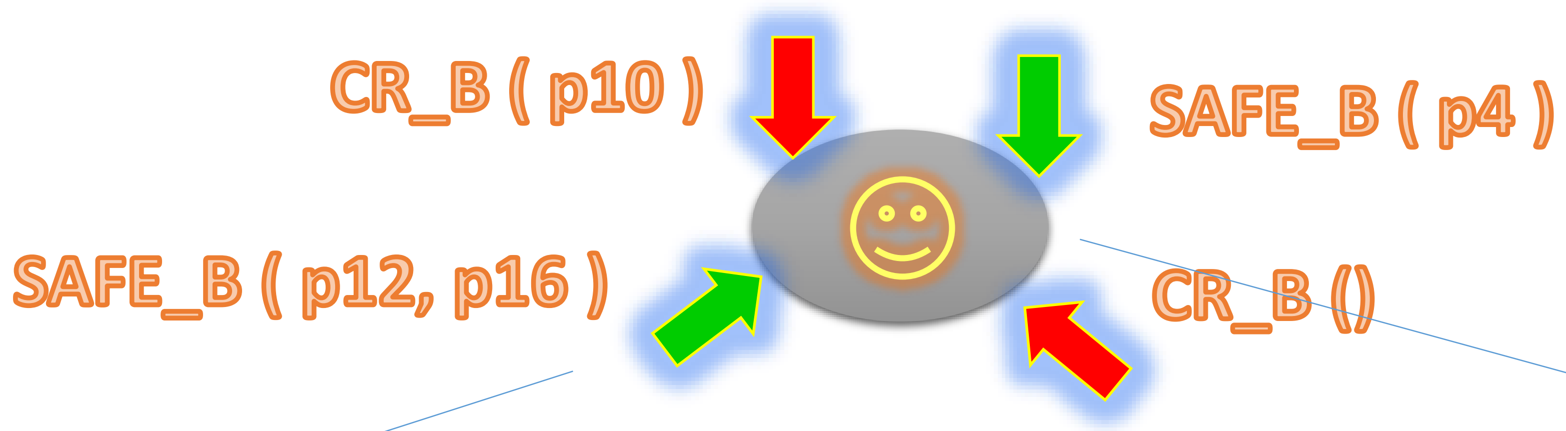
Can be empty set

SAFE_B(<set of places>)

CR_B(<set of places>)

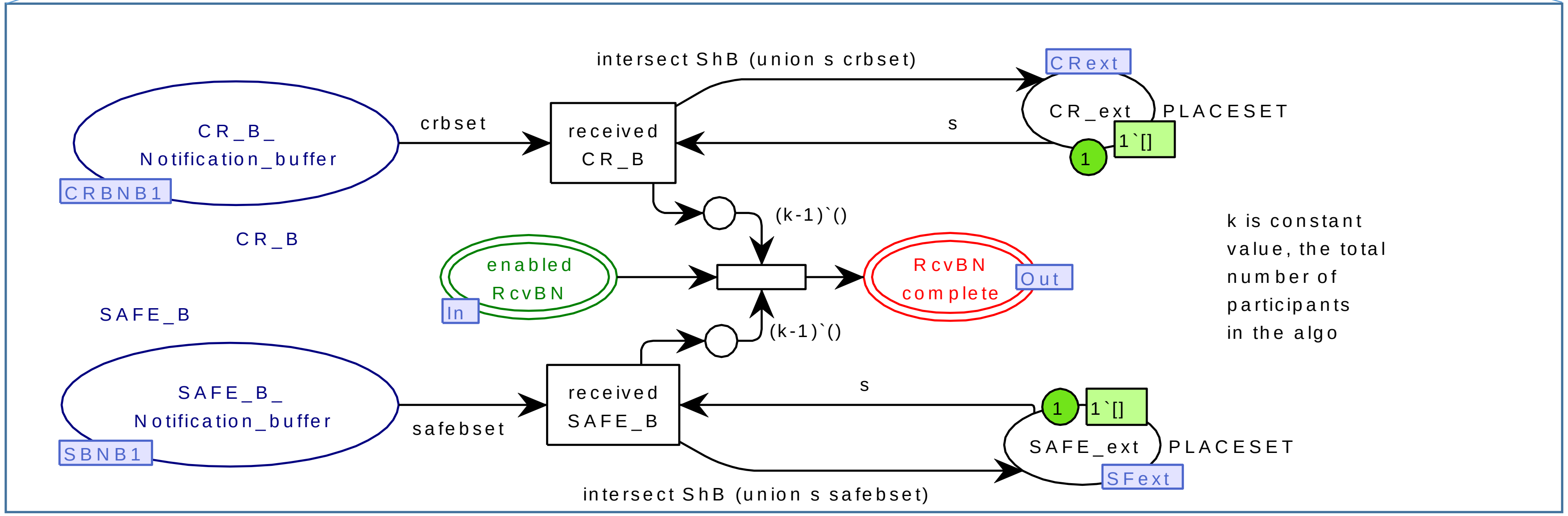


# Receive Boundary Notifications (Module 3)

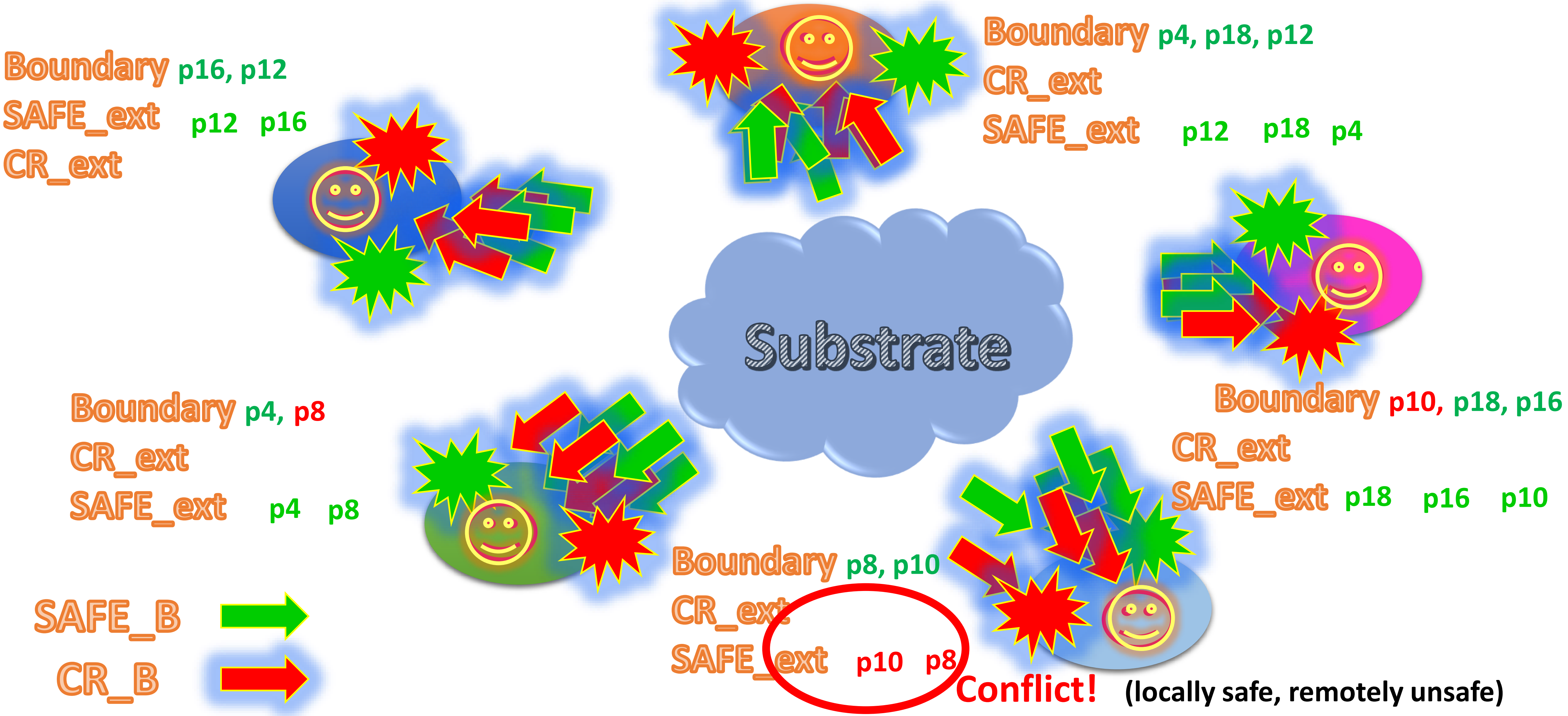



Boundaries p4, p8
CR_ext
SAFE_ext p4

**Records remote-status of Local boundaries**, ignores other Incoming parameters



# Initial Change Region, Broadcast Boundary Notifications, Receive Boundary Notifications (Modules 2+3)

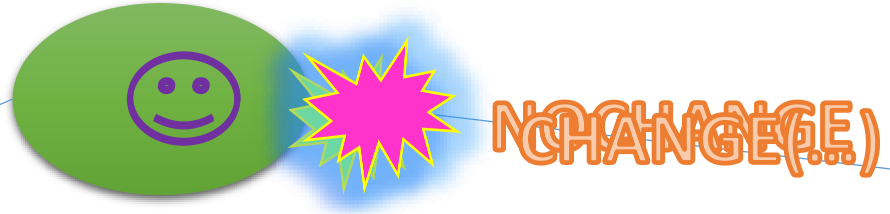


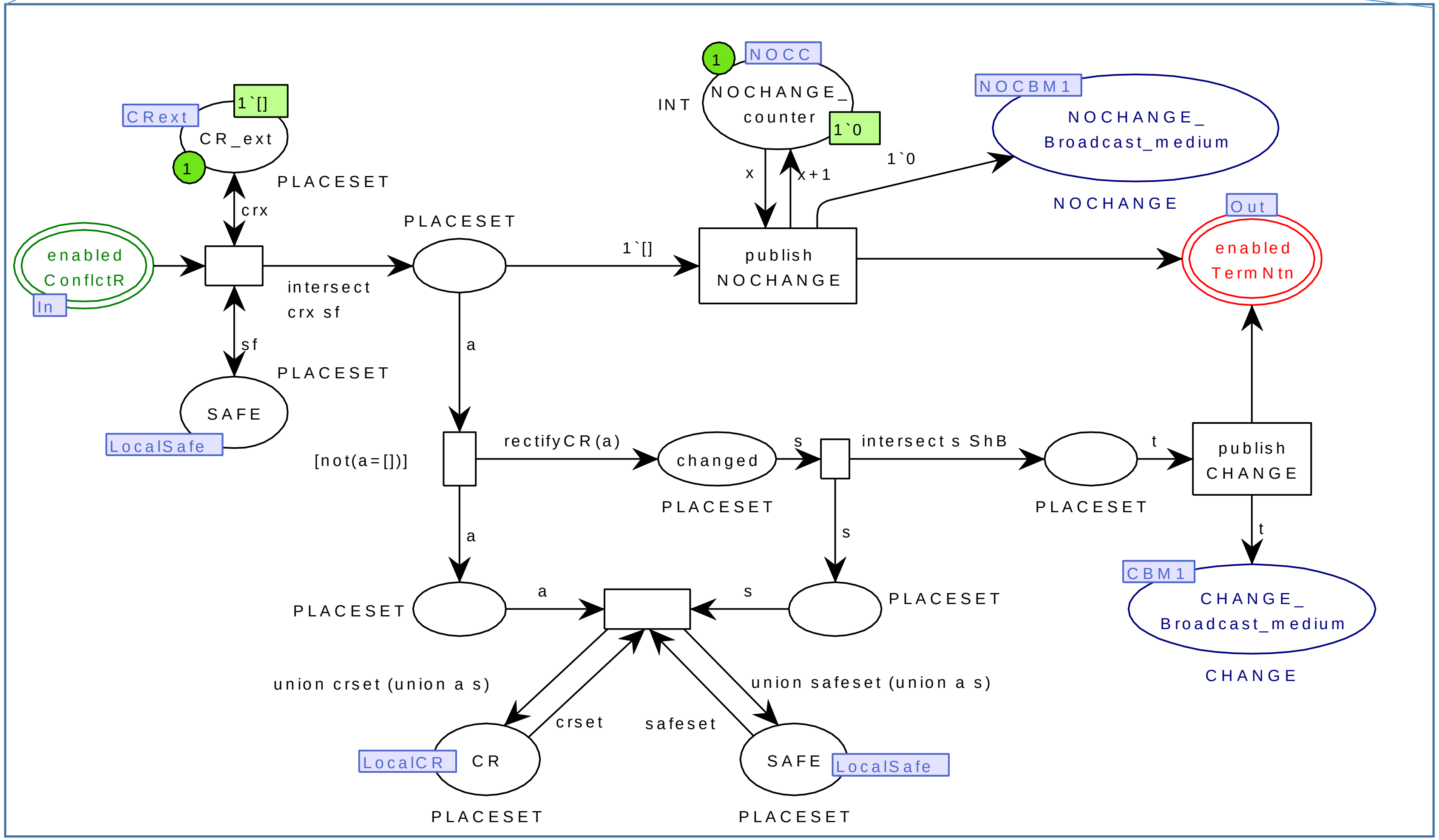
# Conflict Resolution (Module 4)

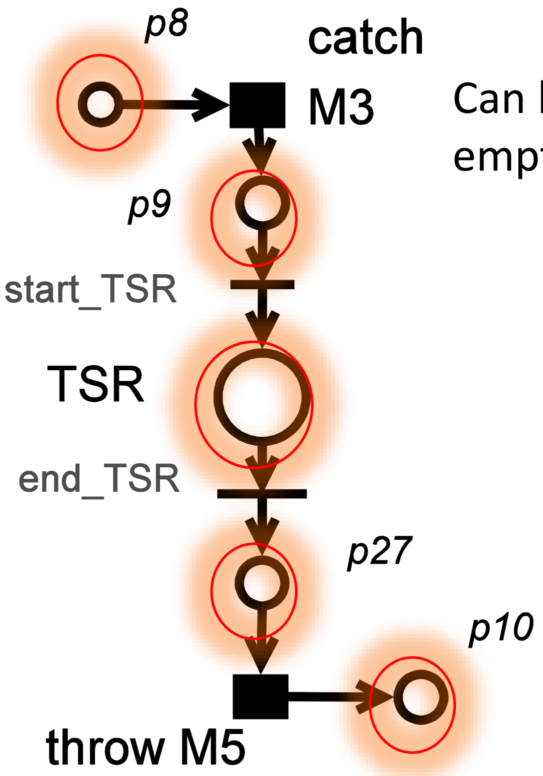
NOCHANGE

CHANGE(<set of places>)

Can be empty set



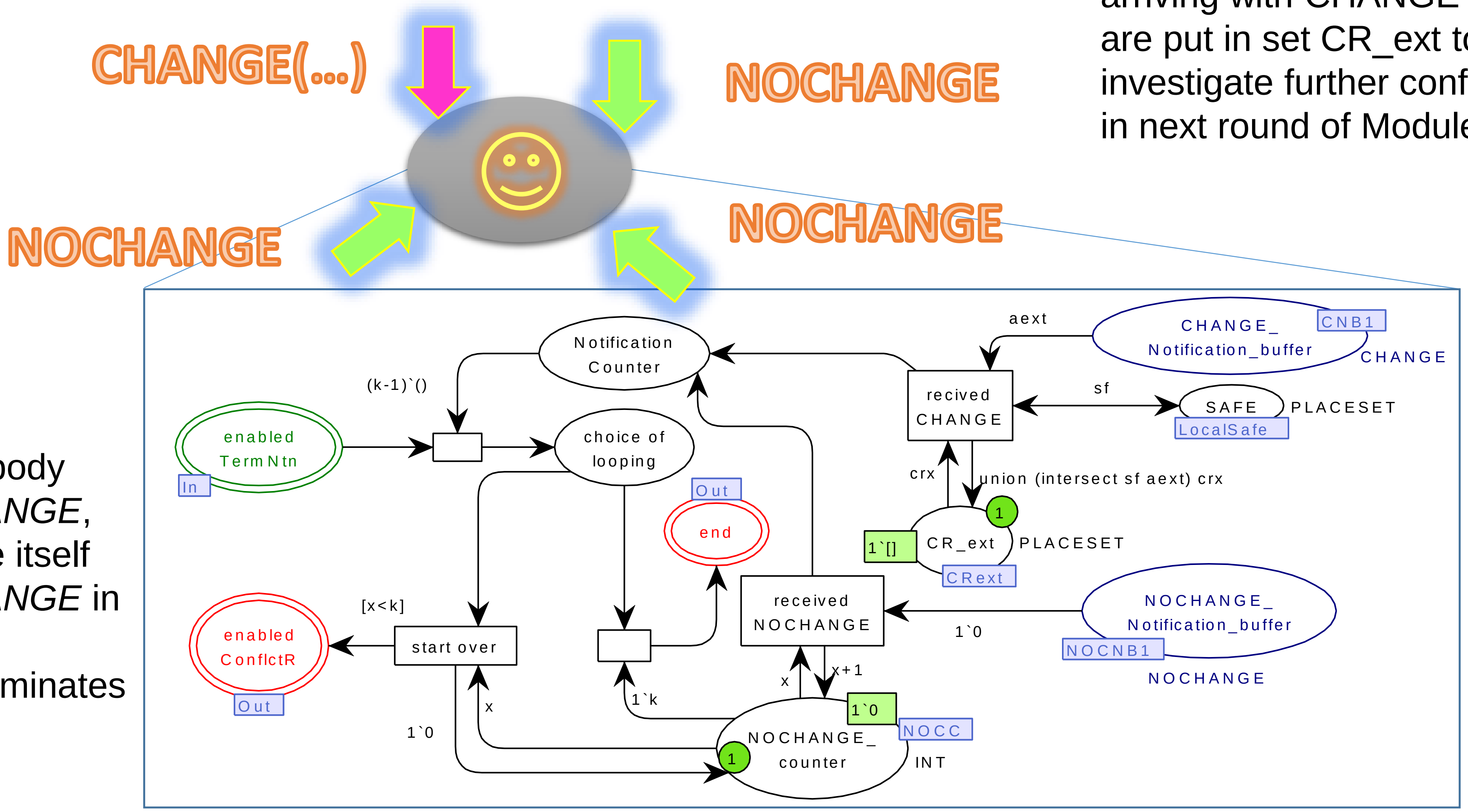



If a safe boundary & other relevant places are made unsafe, **CHANGE** event is published **with additional boundaries that became unsafe**

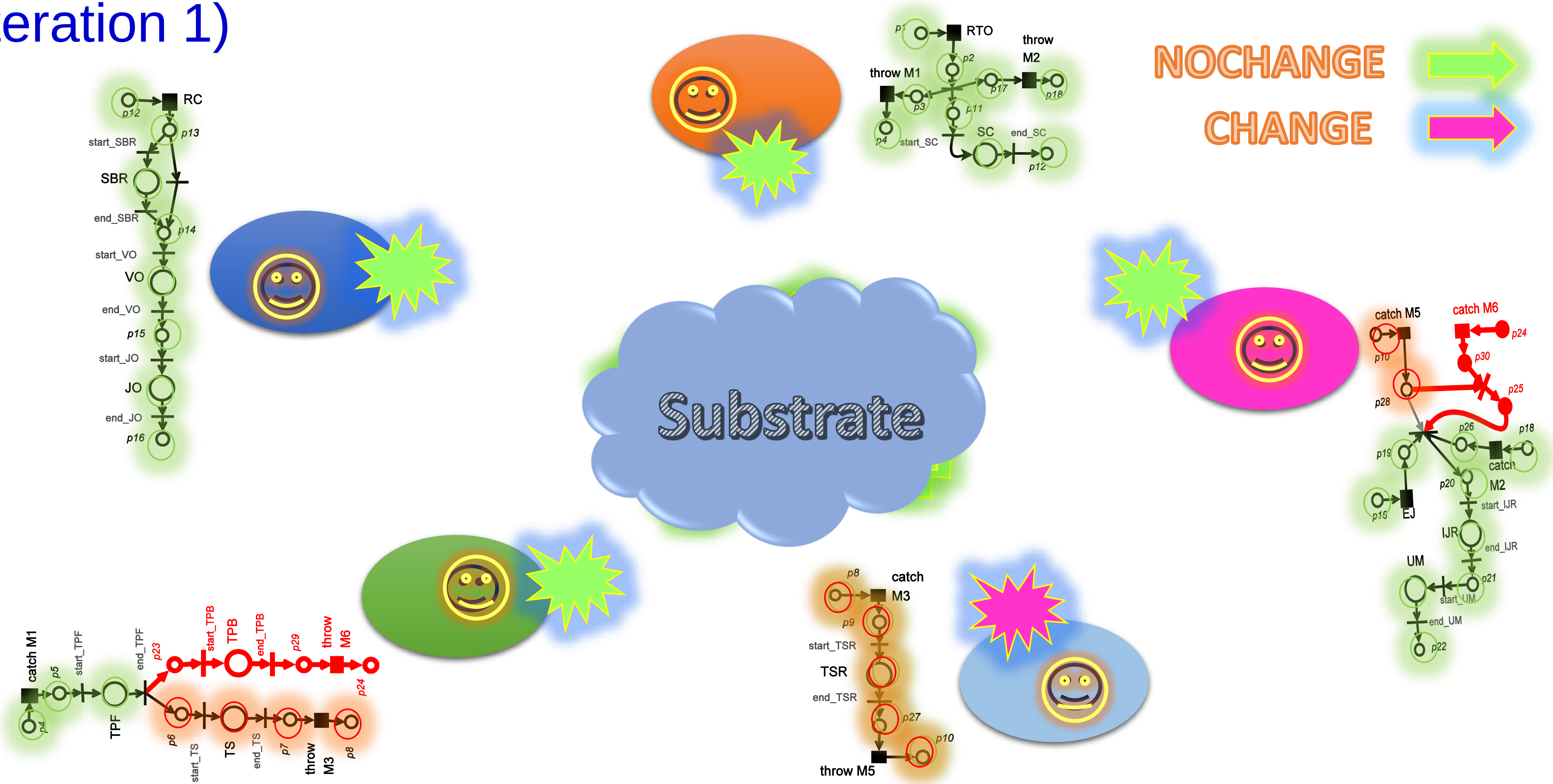
# Termination (Module 5)

Local boundaries arriving with CHANGE event are put in set CR_ext to investigate further conflict in next round of Module 4 (loop)

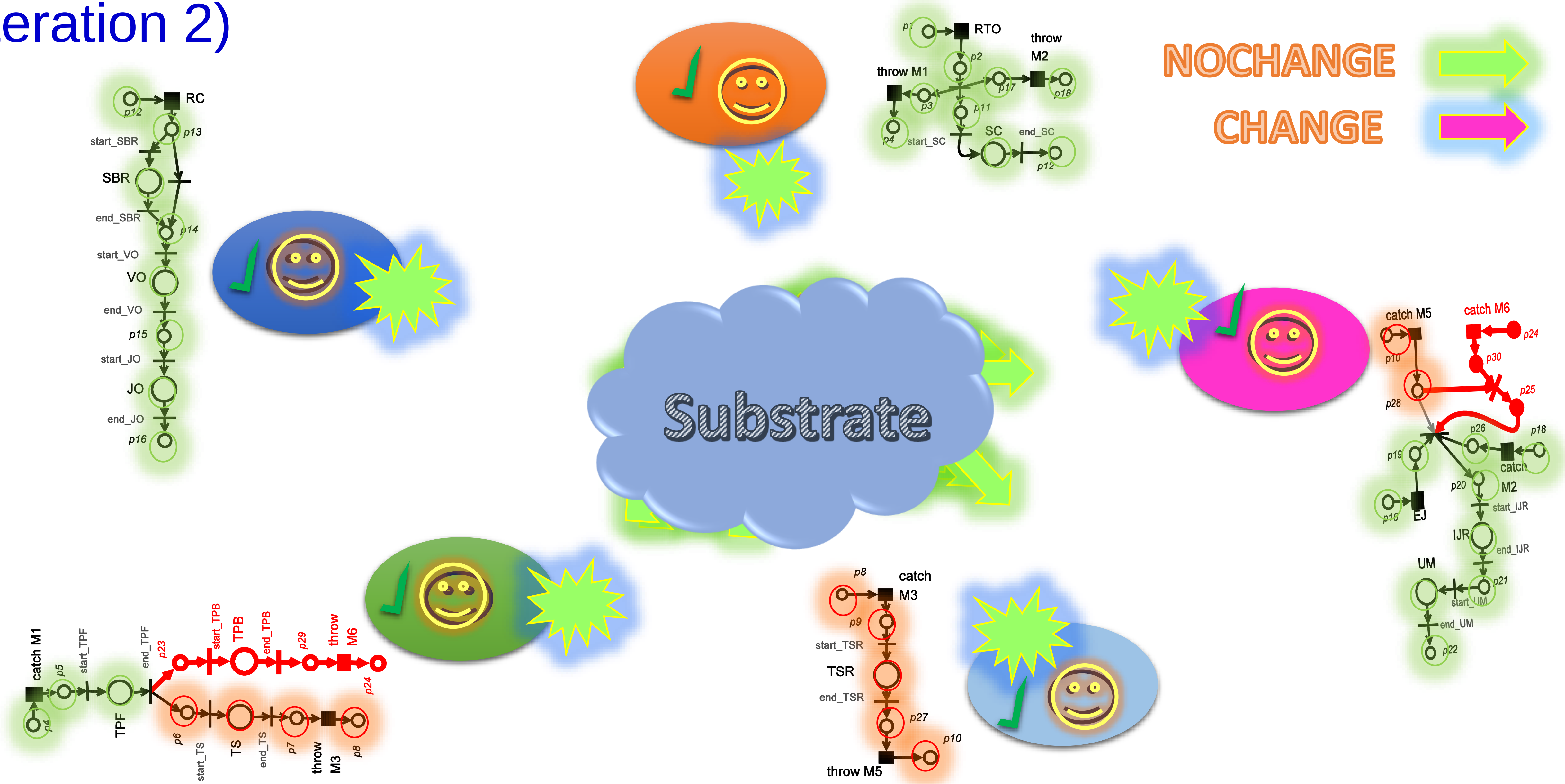



When everybody sent *NOCHANGE*, and the node itself sent *NOCHANGE* in Module 4, Algorithm terminates

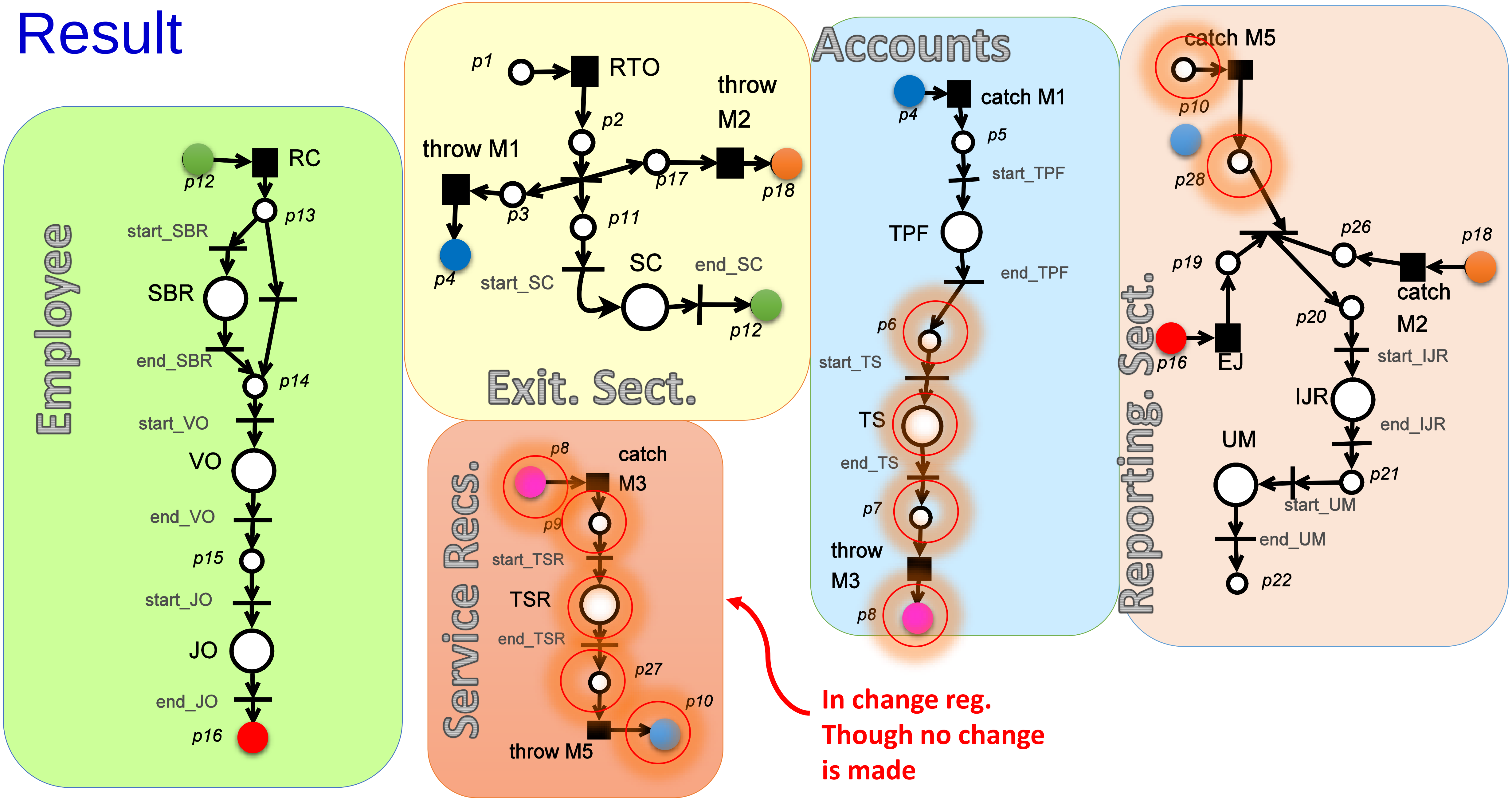
# Conflict Resolution & Termination (Module 4+5) (iteration 1)



# Conflict Resolution & Termination (Module 4+5) (iteration 2)



# Result




**In change reg. Though no change is made**

# Proof of Correctness

Lemma 1

The effect of deletion of a place in the change region in a fragment is fully covered by the fragment in which the place is deleted.

Lemma 2

If there is any change in concurrency of a place in the global net, some fragment detects it.

Bounded Wait

Termination

# Gist of Change Region Approach

- Focus on Marking Reachability (effect of changes) rather than structural changes
- Structural Modeling of markings through C-tree
- Capturing non-migratability through properties
- Approach restricted to structured nets (ECWS grammar)

structured AND is mandatory for C-tree construction, unstructured XOR, LOOP can be handled in C-tree since they are all sequential (w.r.t. token game) regions. Specification through balanced parenthesis is convenient for programming.

- Centralized and Distributed Computation algorithms developed

# Future Work

- Distributed instance migration
- Interplay among consistency models and change operations
- Extending the theory for unstructured workflows
- Implementation Issues

# Publications

**Presented Works:**

- **Distributed Change Region Detection in Dynamic Evolution of Fragmented Processes** by Ahana Pradhan and Rushikesh K. Joshi in the **International Workshop on Petri Nets and Software Engineering 2016**, co-located with the 37th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2016 and the 16th International Conference on Application of Concurrency to System Design ACSD 2016, Torun, Poland, June 20-21, 2016.
- **Lookahead Consistency Models for Dynamic Migration of Workflow Processes** by Ahana Pradhan and Rushikesh K. Joshi in the **International Workshop on Petri Nets and Software Engineering 2015**, co-located with the 36th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2015 and the 15th International Conference on Application of Concurrency to System Design ACSD 2015, Brussels, Belgium, June 22-23, 2015.
- **Catalog-based Token Transportation in Acyclic Block-Structured WF-nets** by Ahana Pradhan and Rushikesh K. Joshi in the **International Workshop on Petri Nets and Software Engineering 2015**, co-located with the 36th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2015 and the 15th International Conference on Application of Concurrency to System Design ACSD 2015, Brussels, Belgium, June 22-23, 2015.
- **Token transportation in Petri net models of workflow patterns** by Ahana Pradhan and Rushikesh K. Joshi in the **7th India Software Engineering Conference 2014**, Chennai, ISEC 2014, Chennai, India, February 19-21, 2014.

**Unpublished Works:**

- **A Structural Approach to Dynamic Migration in Petri Net Models of Structured Workflows** by Ahana Pradhan and Rushikesh K. Joshi [under review in IEEE TSE]
- **A Survey of Consistency Models for Dynamic Workflow Migration** by Ahana Pradhan and Rushikesh K. Joshi [in preparation]

# Thank You

# Outcomes

**Algorithms**

YoYo algorithm for instance migration
Algorithm for weak lookahead
Accept/reject branching algorithm for strong lookahead
PSCR computation algorithm
Change region computation algorithm
Distributed change region computation algorithm

**Properties**

YoYo compatibility, peer patterns
Generator of Concurrent Submarking (GCS)
Dysfunctional C-tree and Break-off Set
Marking Preserving Embedding (MPE)
Change properties
Perfect Member and Overestimation
Perfect Structural Change Region (PSCR)
Fragmentation

**Taxonomy Framework for Consistency Models**

Structural equivalence
Trail-based models
- history equivalence
- trace equivalence
- purged-history equivalence
- purged-trace equivalence

Live model
Lookahead models
- strong
- accommodative
- weak

**Proofs**

Non-migratability lemma
Perfect Member lemma
Overestimation lemma
SCR & PSCR lemma
Proof of correctness for algorithms

**Representation Techniques for Analysis & Application**

C-tree, Derivation Tree
Token transportation catalog
Token transportation bridge

**New Consistency Models**

Strong lookahead
Accommodative lookahead
Weak lookahead

**Workflow Specification Languages**

CWS, ECWS