# Implement Unfolding based POR Algorithm

Huyen Nguyen

October 26, 2015

## 1 Building labelled event structure - parametric semantics

### 1.1 Define the independent relation

**Definition 1** *Commutativity Two transitions t and t' are commuted if at every state (reachable marking) s:*

1. $t, t' \in enable(s) and s \to^{t} s' \Longrightarrow t' \in enable(s')$

   $fire(s,t)$ *is a new state reached by firing t at s*

2. $t, t' \in enable(s) and \exists s" : s \xrightarrow{t.t'} s" then s \xrightarrow{t'.t} s"$

Set up independent set:

- compute all reachable markings (already implemented in simple model checker SMC)
- For all pairs $(t_1, t_2)$, check if $t_1$ and $t_2$ commute or not: $check\_commute(t_1, t_2)$
- if yes, store all independent pairs in a set

**Algorithm 1** Commutativity algorithm

```
function check_commute(a, b)
    for reachable markings s do
        if s enables one of t and t' then
            return false
        else
            if s enables both t and t' then
                s'= fire(s,t)
                s"= fire(s,t')
                if s' enables t' and (fire(s',t') == fire(s",t)) then
                    continue
                else
                    return false
    return true
```

## 1.2 Construct unfolding under the independent relation for a petri net

This part details the process of constructing the unfolding (set of prefix) for a petri net under independent relation defined in previous section.

Unfolding is a LES, tuple $\varepsilon = \langle E, <, \#, h \rangle$, where E is a set of events, $<$ is set of pairs of events in causal relation, $\#$ is set of events in conflict.

**Definition 2** *Construct the unfolding LES*

1. *Initially, LES having one event bottom* $\perp$
2. *Let* $\varepsilon$ *be an prefix containing a history H (one element in the set of possible histories of LES) for some transition* $t \in T$*.*

   $\varepsilon$ is a unfolding prefix, initially $\varepsilon = \perp$
   // `check only maximal event because all extendable events had already been computed and added to the stack extendable in previous step` // can't find $< e_i.lbl, t >$ in independent set

---

**Algorithm 2** Extend LES

---

**procedure** EXTEND
    extStack := $\perp$
    **while** extStack != $\varnothing$ **do**
        Pop up an event e from the stack
        Marking $s := state(e_i)$
        Get all transitions activated (enabled) at s
        **if** $t, e.lbl$ are not independent **then**
            make a new Event with $lbl = t$; $his = \{e\}$
            add it to the stack **extendable**
        **else**
            search in set of events of LES, if exists an event $e'$ with label t
            compute the $state(\sigma)$ with a run $\sigma = t.(e.lbl)$ or $(e.lbl).t$
            find all enabled transitions t' at that state
            creat new event with $\langle t', \{e, e'\} \rangle$ and compute the reached state
            Add event to the stack **extendable**
        add e to LES; **extendable** and add it to LES
        Compute set of transitions in conflict with e

---

To implement the algorithm, we construct a class Event:

**Definition 3** *Event*
*class Event*
{
    *Transition lbl;* \\ *the transition labels the event*
    *Config his;* \\ *histort: set of all predecessors*
    *set⟨Event⟩ cfl;*\\ *set of events conflicting with the event*
    *Marking state;*
    public:
    *void computeCfl();*

*bool is_inHis(Event)*
*bool enable(Transition);*
*void setState();*
}

**Definition 4** *Configuration*
*class Config*
{
*set* $\langle Event \rangle$ *event ; maximal events (no more event which is enabled at them)*
member function:
*Marking computeState();*
*bool is_inConfig(Event);* }

**Definition 5** *Labelled Event Structure LES*
*set of history of a prefix* <
*The process of buiding the computation tree:*
*class LES*
{
*set* $\langle Event \rangle$ *evt;*
*set* $\langle Event, Event \rangle$ *causal; not necessary*
*set* $\langle Event, Event \rangle$ *conflict; not necessary*
*Marking computeState*
}

**Function 1** *Compute set of conflict events in LES*
*Function computeCfl(e)*
{
*for all event* ***e'*** *in LES.evt*
*if* $(e'.is\_inHis(e) == false) \&\& ((e.is\_inHis(e') == false)) \&\& is\_depend(e.lbl, e'.lbl)$
*then*
*add e'to e.cfl;*
*end if*
}

**Function 2** *Check if t and t'independent*
*is_depend(t,t')*
{
*for each pair* $(t_1, t_2)$ *in independent set*
*if* $(t, t') == (t_1, t_2)$
*then return true;*
}

**Function 3** *Check if event e in History of an event*
*is_inHis(e)*
{
$\backslash *$ *return true when e is a event in the calling event.* $* \backslash$
*for each event* $e_i$ *in his*
*if* $(e == e_i)$ *return true;*
*return false;*
}

**Function 4** *Check if t enabled at state of some event*
*enable(Transition t)*
{
    * *return true when t is activated at the state of that event.* *\
    *return state.activates(t);* * *activate(t) is a member function in class Marking* *\
}

# 2 LES exploration

Given a prefix unfolding in terms of sets of events like picture attached.

## 2.1 Overall algorithm

U is a virtual set of all events with attribute $seen = true$

G is a virtual set of all events with attribute $in\_gabage = true$

D is a virtual set of all events with attribute $disbale = true$

A is a virtual set of all events with attribute $add = true$

Attribute *visited* of an event increases by 1 whenever it is visted

C is a configuraiition which also store the maximal events. All events in c are marked with $in\_C = true$

J is a configuration possible to add after C as alternative to D

**Algorithm 3** : POR Exploration algorithm

**procedure** EXPLORE(C,D,A)
    Extend(C)
    **if** $en(C) = \varnothing$ **then return**
    **if** $A = \varnothing$ **then**
        Choose e from en(C)
    **else**
        e form $A \cap en(C)$
    Explore($C \cup \{e\}, D, A \setminus \{e\}$ )
    **if** $\exists J \in Alt(C, D \cup \{e\})$ **then**
        $Explore(C, D \cup \{e\}, J \setminus C)$
    Remove(e,C,D)

### 2.1.1 A configuration

*A configuration is a set of events that are causally closed and conflict free.*
- Declared as a set of maximal events
- Member function:
+ computeState(C) : compute marking reachable by running set of events in C

**Algorithm 4** find alternative event

```
procedure findAlt( )
    possible :=true
    for each event e in U do
        if maximal events of C are in history of e then
            for all events e' in D - disable = true do
                if e not conflict with e' then
                    possbile=false
                    break
            if possible==true then
                J := J + {e}
```

### 2.1.2 Compute en(C)

Compute a set of events enabled at a given configuration C