

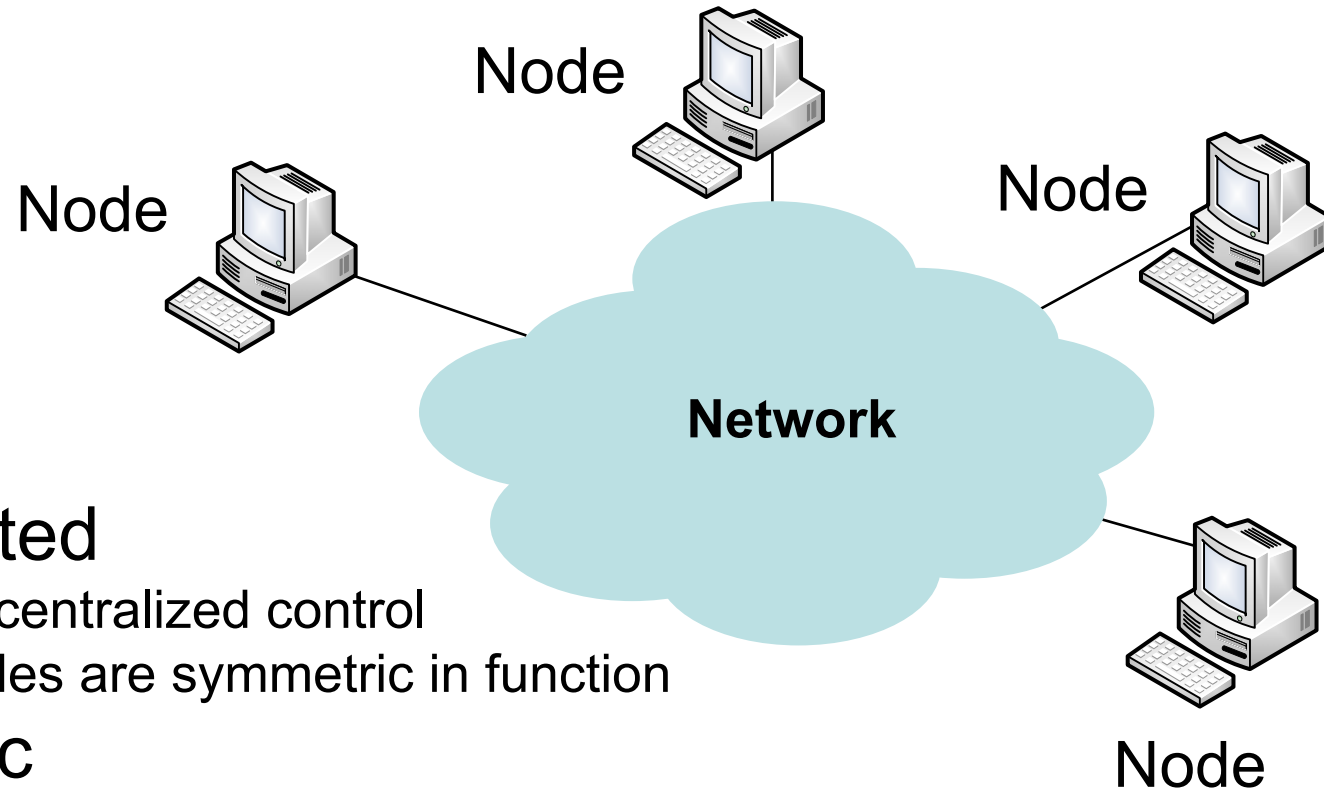
Verification of Peer-to-Peer Algorithms: A Case Study

Rana Bakhshi

Dilian Gurov

Royal Institute of Technology (KTH),
Sweden

What is P2P?



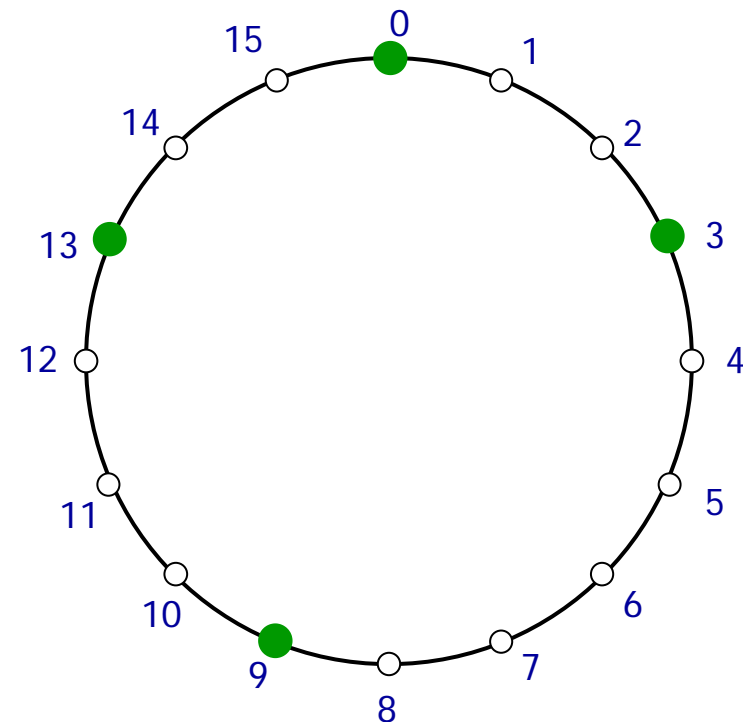
- **Distributed**
 - No centralized control
 - Nodes are symmetric in function
- **Dynamic**
 - nodes to join and leave the network
- **Structured (optional)**
 - network topology maintenance needed

Chord: Protocol

- Provides P2P lookup service
- Given a key, it maps the key onto a node
- Maintains routing information as nodes join and leave the system
 - stabilization algorithm
- and other features...

Chord: Protocol

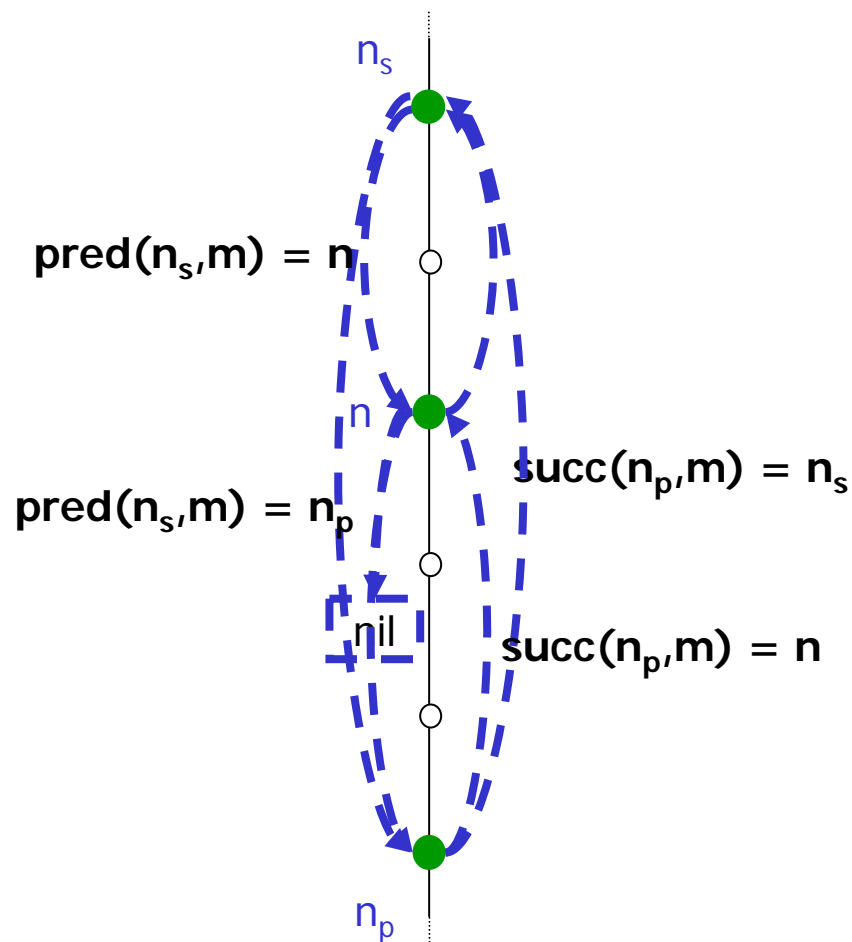
- Chord ring
 - ID are ordered on a ID circle
- Two basic neighbours
 - predecessor and successor
- *Successor*
 - the first node
 - ID is more or equal to the current



Chord: Stabilization algorithm

- All nodes information must be up to date
 - for correctly executing lookups
- Each node periodically runs a stabilization algorithm
 - to update the successor information
- Case study
 - Verification of the stabilization algorithm

Chord: Stabilization algorithm



- **n runs *join()*:**
 1. predecessor = nil
 2. n acquires n_s as successor from n'
- **n runs *stab()*:**
 1. n notifies n_s being the new predecessor
 2. n_s acquires n as its predecessor
- **n_p runs *stab()*:**
 1. n_p asks n_s for its predecessor (now n)
 2. n_p acquires n as its successor
 3. n_p notifies n
 4. n acquires n_p as its predecessor
- **predecessor and successor information are now updated**

Motivation

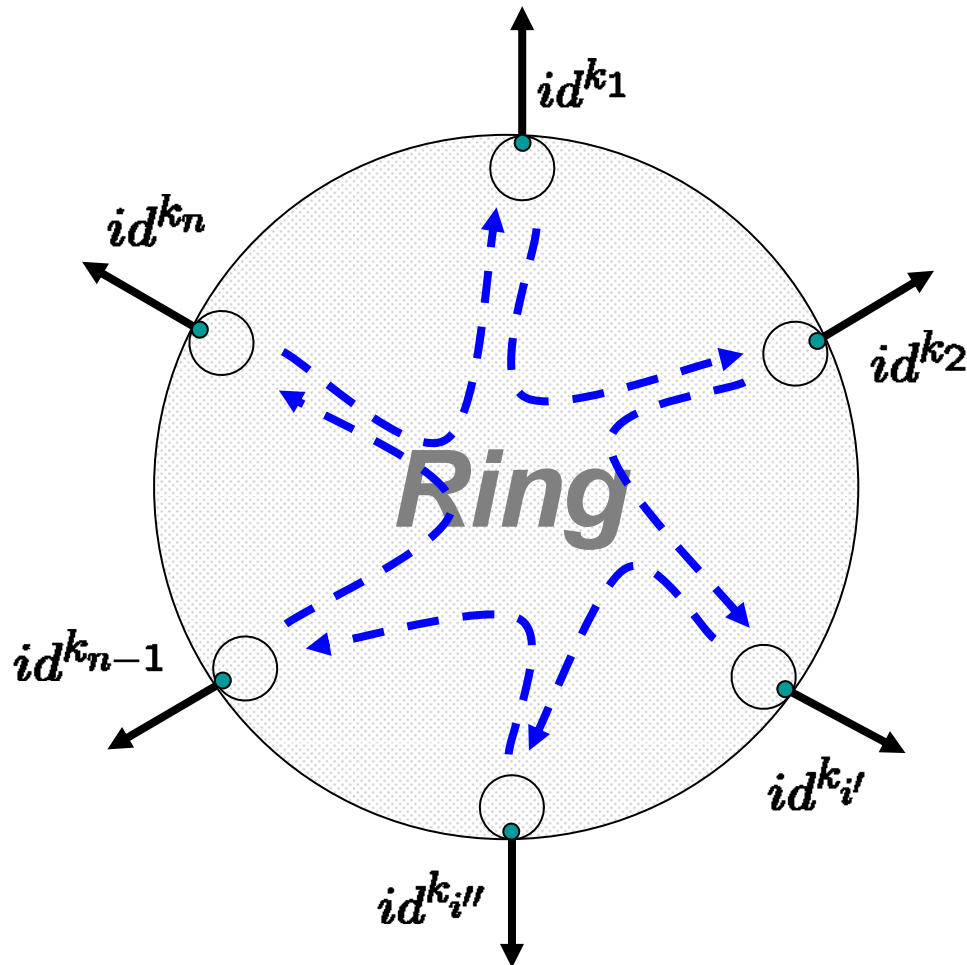
- A reasonable level of abstraction at which to perform the verification
 - Correctness proofs
 - sketch at the high level of abstraction
 - tend to provide no operational semantics
 - Model checking techniques
 - not directly applicable
 - systems are inherently dynamic
 - have infinite state behaviour

Modelling

- Assumptions
 - No finger tables and successors' lists
 - Pure Join Model
- Formalism
 - π -calculus
- Strategy
 - establishing weak bisimulation

Specification

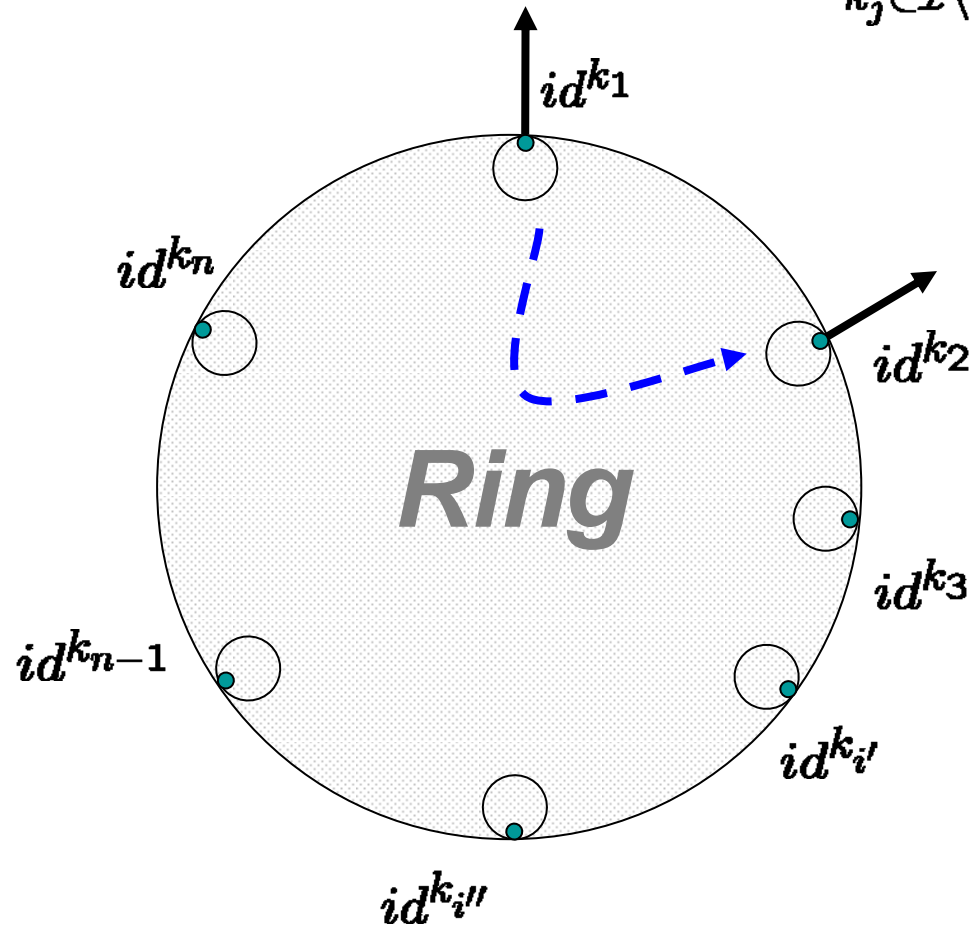
$$Ring(k_i, m) \triangleq \tau.\overline{id^{k_i}}.Ring(\text{succ}(k_i, m), m) + \dots$$



- Ring behaviour:
 - A node performs an output on its channel
 - and passes the token to its successor

Specification

$$Ring(k_i, m) \triangleq \dots + \sum_{k_j \in \mathcal{I} \setminus m} \tau. Ring(k_i, m \oplus k_j)$$



- Ring behaviour:
 - A new node joins the ring
 - and passes the token to its successor

Implementation

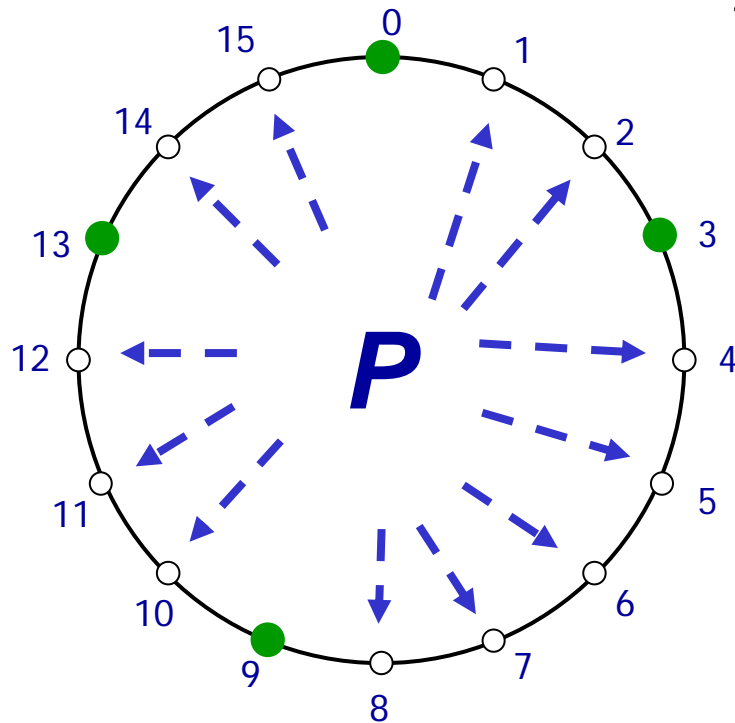
- A collection of concurrent processes
 - nodes **A**
 - nodes **P** ready to join a network
 - a token

$$\text{Impl}(k_i, m) \triangleq (\nu \vec{in}^{k_j}, \vec{in}^{k_i}, \vec{in}^{k_l}) \left(\left(\vec{in}^{k_i} \cdot \mathbf{0} \right) \mid \left(\prod_{k_j \in \mathcal{I} \setminus m} \sum_{k_v \in m'} P \langle \vec{in}^{k_j}, \vec{in}^{k_v} \rangle \right) \right)$$

$$\left(\prod_{k_{i'} \in m'' \subseteq m} A \langle \vec{id}^{k_{i'}}, \vec{in}^{k_{i'}}, \vec{s}^{k_{i'}}, \vec{p}^{k_{i'}} \rangle \mid \prod_{k_l \in m' \subseteq m} A \langle \vec{id}^{k_l}, \vec{in}^{k_l}, \vec{s}^{k_l}, \vec{p}^{k_l} \rangle \right)$$

Implementation

$$Impl(k_i, m) \triangleq \dots | \left(\prod_{k_j \in \mathcal{I} \setminus m} \sum_{k_v \in m'} P(\overrightarrow{in}^{k_j}, in_4^{k_v}) \right) | \dots$$



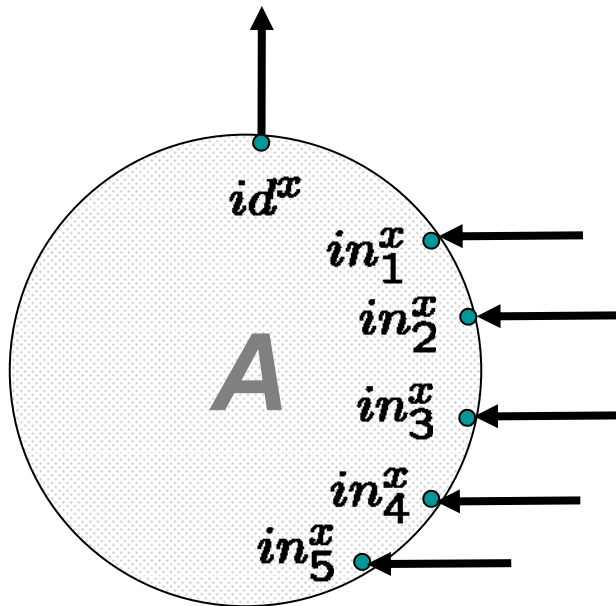
- Process **P**

- A node that is not in the network but may join it
- Implements ***join()***

$$P(\overrightarrow{in}^{k_j}, in_4^{k_i}) \triangleq (\nu \overrightarrow{in}^{k_j}) \left(\overline{in}^{k_i} \langle \overrightarrow{in}^{k_j} \rangle . in_4^{k_j} (\overrightarrow{in}^z) . A \langle id^{k_j}, \overrightarrow{in}^{k_j}, in_4^{k_z}, \overline{\perp} \rangle \right)$$

Implementation

$$Impl(k_i, m) \triangleq \dots | \prod_{k_l \in m' \subseteq m} A(\overrightarrow{id}^{k_l}, \overrightarrow{in}^{k_l}, \overrightarrow{s}^{k_l}, \overrightarrow{p}^{k_l})$$



- Process **A**

- 5 ports for different type of messages and 1 port for output ID
- stores info of itself and neighbours
- implements ***stab()***

Conclusion

- Extends results on lookup algorithm for the static case
- A model for P2P networks with a ring topology
 - Process *Ring*
- π -calculus offers a suitable theory

Future Work

- Other aspects of P2P networks
 - including presence of failures in the network
 - adding finger tables and successors lists
- Theorem proving systems
 - models can be formalized
 - the bisimulation proof can be carried
 - e.g. Isabelle/HOL

Thank you!



Chord: Stabilization algorithm

```
%new node joins a Chord ring containing
node n'
join() ->
n'! {"find_successor",n}
receive
  {"find_successor",n'} ->
    succ := n'; pred := nil;
end.

%in parallel with

loop
  receive
    {"find_successor",n'} ->
      if n'∈(n,succ) ->
        n'! {"find_successor",succ}
      n'∉(n,succ) ->
        succ! {"find_successor",n'}
  end
end.
```

```
%stabilization algorithm
stab() ->
succ! {"req_predecessor",n}
receive
  {"resp_predecessor",n'} ->
    if n'∈(n,succ) ->
      succ := n';
    succ! {"notify",n}
end.

%in parallel with

loop
  receive
    {"req_predecessor",n'} ->
      n'! {"resp_predecessor",pred}
    {"notify",n'} ->
      if pred = nil ∨ n'∈(pred,n) ->
        pred :=n';
      end
  end
end.
```

Chord: Stabilization algorithm

```
%%new node join a Chord ring containing
node n'
join() ->
n'! {"find_successor",n}
receive
{"find_successor",n'} ->
succ := n'; pred := nil;
end.

%in parallel with

loop
receive
{"find_successor",n'} ->
if n'∈(n,succ] ->
n'! {"find_successor",succ}
n'∉(n,succ] ->
succ! {"find_successor",n'}
end
end.

%%stabilization algorithm
stab() ->
succ! {"req_predecessor",n}
receive
{"resp_predecessor",n'} ->
if n'∈(n,succ) ->
succ := n';
succ! {"notify",n}
if (n'∉(n,succ) ∧ (n'≠n)) ->
succ! {"notify",n}
end.

%in parallel with

loop
receive
{"req_predecessor",n'} ->
n'! {"resp_predecessor",pred}
{"notify",n'} ->
if pred = nil ∨ n'∈(pred,n) ->
pred :=n';
if (pred = n) ∧ (succ = n)
->
succ := n';
succ! {"notify",n}
end
end.
```

π -calculus

u, v	$::=$	o, p, x	names
		in, id	channels
		\perp	undefined value
		k_i, k_j	integers, IDs
		$succ(x, m)$	successor
		$pred(x, m)$	predecessor
e, e', e''	$::=$	u	expressions
ϕ, ψ	$::=$	$e = e'$	boolean tests
		$e \in (e', e'']$	interval check
m, m', m''			sets, subsets
π	$::=$	$\tau \mid \bar{p} \langle \vec{v} \rangle \mid p(\vec{v})$	prefix
		$(\vec{v}).P$	abstractions
		$\vec{u} \langle \vec{v} \rangle .P$	concretions
Q, R	$::=$		processes
		M	summation
		$(Q \mid R)$	parallel composition
		$(\nu \vec{p}) Q$	restriction
		if ψ then Q else R	if statement
		$!Q$	replication
		$Q \langle \vec{v} \rangle$	process constant
M, M'	$::=$	$\mathbf{0}$	inaction
		$\pi.Q$	process action
		$M + M'$	choice

π -calculus

$$\text{succ}(x, \hat{m}) = \{y \in \hat{m} \mid (y \boxminus x) = \min\{(z \boxminus x) \mid z \in \hat{m}\} \wedge \hat{m} \subseteq \mathcal{I}\}$$

$$\text{pred}(x, \hat{m}) = \{y \in \hat{m} \mid (y \boxminus x) = \max\{(z \boxminus x) \mid z \in \hat{m}\} \wedge \hat{m} \subseteq \mathcal{I}\} \cup \{x \notin \hat{m} \mid \perp\}$$

Implementation

$$\begin{aligned}
A(\overrightarrow{id^o}, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^p}) &\triangleq \overrightarrow{in^s_1}(\overrightarrow{in^o}).A(\overrightarrow{id^o}, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^p}) \\
&+ \overrightarrow{in^o_1}(\overrightarrow{in^z}).\overrightarrow{in^z_2}(\overrightarrow{in^p}).A(\overrightarrow{id^o}, \overrightarrow{in^o}, \overrightarrow{in^z}, \overrightarrow{in^p}) \\
&+ \overrightarrow{in^o_2}(\overrightarrow{in^z}).(\text{if } z \in (o, s) \text{ then } \overrightarrow{in^z_3}(\overrightarrow{in^o}).A(\overrightarrow{id^o}, \overrightarrow{in^o}, \overrightarrow{in^z}, \overrightarrow{in^p}) \\
&\quad \text{else (if } z = o \text{ then } A(\overrightarrow{id^o}, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^p}) \\
&\quad \quad \text{else } \overrightarrow{in^s_3}(\overrightarrow{in^o}).A(\overrightarrow{id^o}, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^p})) \\
&+ \overrightarrow{in^o_3}(\overrightarrow{in^z}).(\text{if } p = \perp \vee z \in (p, o) \\
&\quad \text{then (if } (p = o) \wedge (s = o) \text{ then } \overrightarrow{in^z_3}(\overrightarrow{in^o}).A(\overrightarrow{id^o}, \overrightarrow{in^o}, \overrightarrow{in^z}, \overrightarrow{in^z}) \\
&\quad \quad \text{else } A(\overrightarrow{id^o}, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^z})) \\
&\quad \text{else } A(\overrightarrow{id^o}, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^p})) \\
&+ \overrightarrow{in^o_4}(\overrightarrow{in^z}).(\text{if } z \in (o, s] \text{ then } \overrightarrow{in^z_4}(\overrightarrow{in^s}).A(\overrightarrow{id^o}, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^p}) \\
&\quad \text{else } \overrightarrow{in^s_4}(\overrightarrow{in^z}).A(\overrightarrow{id^o}, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^p})) \\
&+ \overrightarrow{in^o_5}(\overrightarrow{id^o}.A(\overrightarrow{id^o}, \overrightarrow{in^o}, \overrightarrow{in^s}, \overrightarrow{in^p}) | \overrightarrow{in^s_5}.0)
\end{aligned}$$

where: $\overrightarrow{in^x} = \{x, in^x_1, in^x_2, in^x_3, in^x_4, in^x_5\}$.