Faithful Simulation of Randomized BFT Protocols on Block DAGs

Hagit Attiya Technion Israel

Constantin Enea Ecole Polytechnique France

Shafik Nassar Technion Israel

CONCUR 2023

Objective

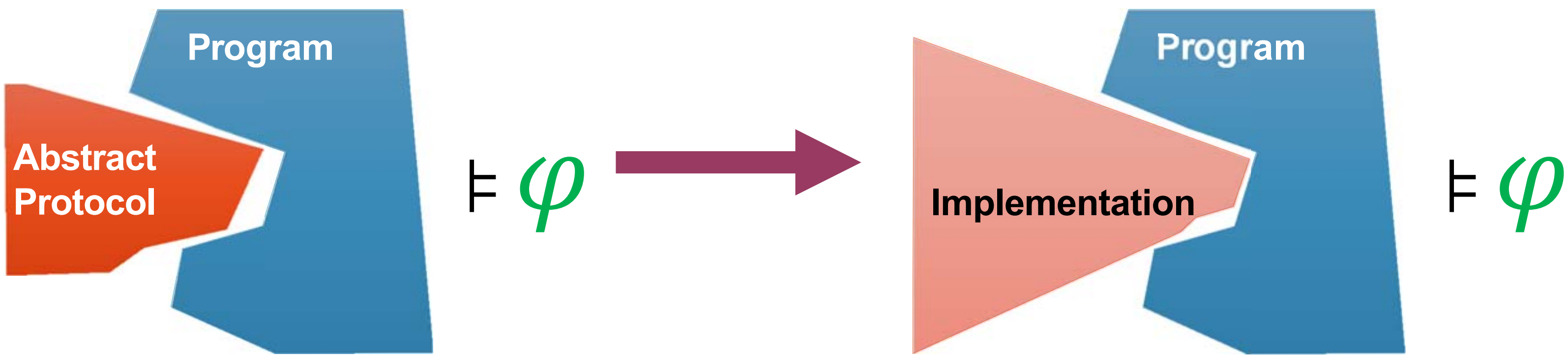
- Exploit blockchain-like concepts for efficient implementations of randomized distributed protocols
- Building on [Schett, Danezis, PODC'21] for deterministic protocols
- Correctness specifications and their guarantees

Plan

- Motivation
- Private-coin block DAG implementations
- Proving their correctness

Correctness: Contextual Refinement

Preserving a property φ in a given class in the context of **every** Program



Standard trace inclusion (refinement):

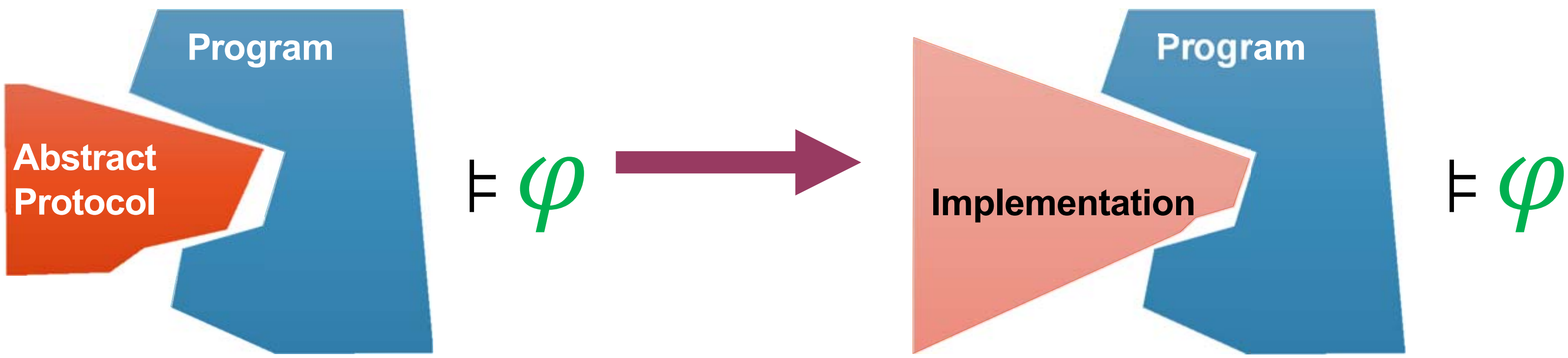
Traces(Program X Imp) $\subseteq$ Traces(Program X Abs)

E.g., linearizability preserves safety properties in any program

[Herlihy, Wing][Filipovic, O'Hearn, Rinetzky, Yang]

Correctness: Contextual Refinement

Preserving a property φ in a given class in the context of **every** Program



Standard trace inclusion (refinement):

Traces(Program X Imp) $\subseteq$ Traces(Program X Abs)

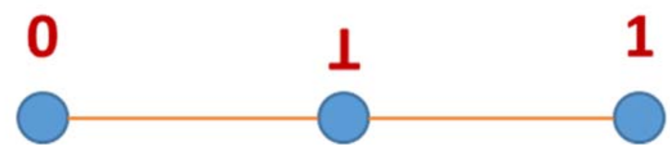
Does not preserve hyperproperties

Example: Binary Crusader Agreement

Binary crusader agreement (BCA) is a weak form of consensus, where processes start with values in {0,1} and return values in {0, 1, ⊥}

- On same or adjacent vertexes (agreement)
- If all start with v, decide on v (validity)

[Dolev, 1982]



More in BA with Welch @ Thursday

Randomized Consensus with BCA

Binary crusader agreement (BCA) is a weak form of consensus, where processes start with values in {0,1} and return values in {0, 1, ⊥}

Every process goes through a sequence of (asynchronous) rounds, each with

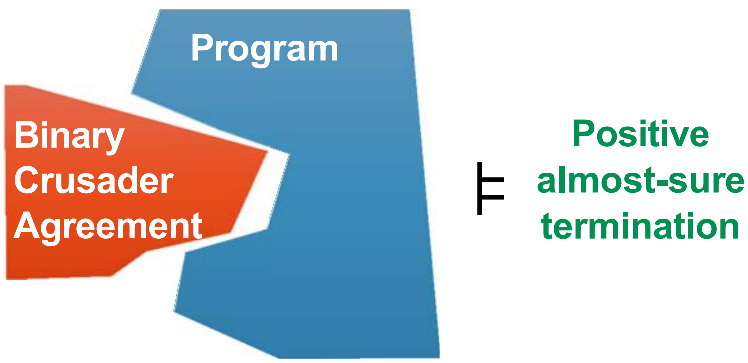
- one instance of BCA &
- one instance of Common Coin Toss

Common Coin Toss: all processes get the same output in {0,1} and it is unpredictable

```
Input: x
r := 0; est := x;
while true do
    r++;
    val := r.BCA(est);
    c := r.Toss();
    if val ≠ ⊥ and c = val then
        output val;
        est := val;
    else if val ≠ ⊥ then
        est := val;
    else
        est := c;
```

Randomized Consensus with BCA

Positive almost-sure termination: termination with probability 1 and in an expected finite number of steps (a hyperproperty)

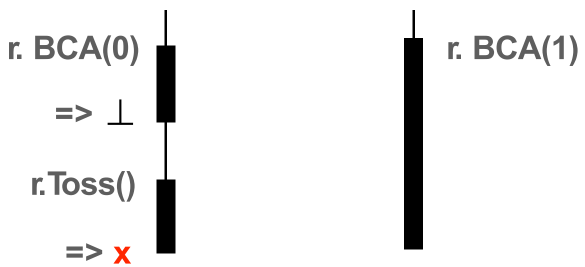


Input: x

```
r := 0; est := x;
while true do
    r++;
    val := r.BCA(est);
    c := r.Toss();
    if val ≠ ⊥ and c = val then
        output val;
        est := val;
    else if val ≠ ⊥ then
        est := val;
    else
        est := c;
```

With a Distributed BCA Implementation

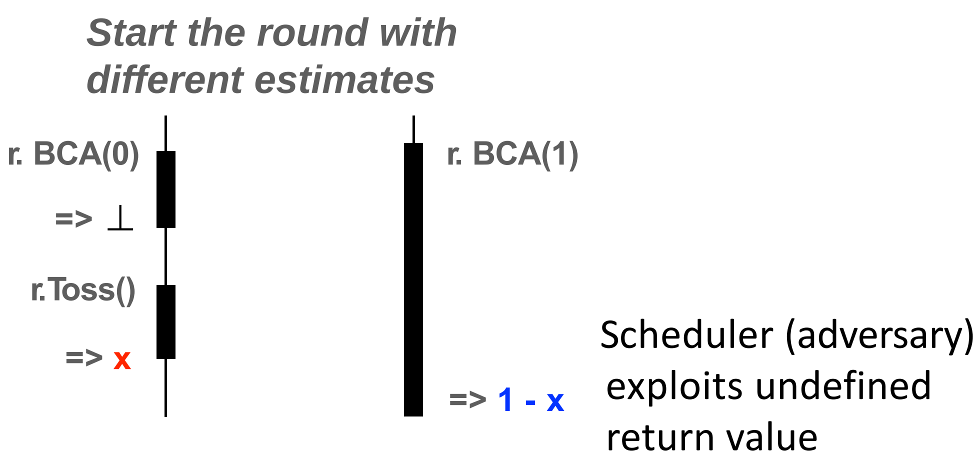
Start the round with different estimates



Input: x

```
r := 0; est := x;
while true do
    r++;
    val := r.BCA(est);
    c := r.Toss();
    if val ≠ ⊥ and c = val then
        output val;
        est := val;
    else if val ≠ ⊥ then
        est := val;
    else
        est := c;
```

With a Distributed BCA Implementation

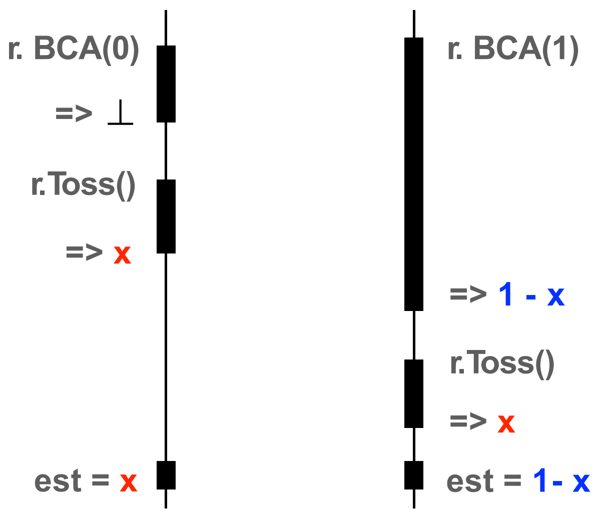


Input: x

$r := 0$; $est := x$;
while *true* **do**
    r++;
    $val :=$ r.BCA(est);
    $c := r$.Toss();
    if $val \neq \perp$ and $c = val$ **then**
        output val;
        $est := val$;
    else if $val \neq \perp$ **then**
        $est := val$;
    else
        $est := c$;

With a Distributed BCA Implementation

Start the round with different estimates



End the round with different estimates

Scheduler (adversary) exploits undefined return value

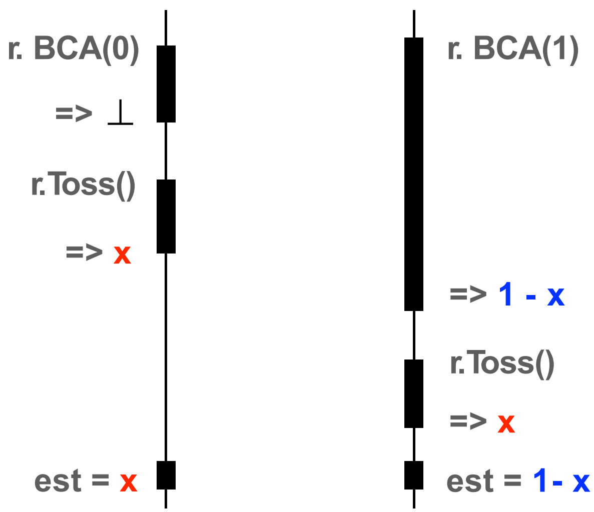
```
Input: x
r := 0; est := x;
while true do
    r++;
    val := r.BCA(est);
    c := r.Toss();
    if val ≠ ⊥ and c = val then
        output val;
        est := val;
    else if val ≠ ⊥ then
        est := val;
    else
        est := c;
```

Binding BCA

When an execution prefix ends in a process returning ⊥, there is a single non-⊥ value that can be returned by a process in any extension

[Abraham, Ben-David, Yandamuri]

Start the round with different estimates



Scheduler (adversary) exploits undefined return value

End the round with different estimates

Input: x

$r := 0;\ est := x;$

while $true$ **do**

  $r{+}{+};$

  $val :=$ $r.\mathsf{BCA}(est);$

  $c := r.\mathsf{Toss}();$

  if $val \neq \bot$ and $c = val$ **then**

    output $val;$

    $est := val;$

  else if $val \neq \bot$ **then**

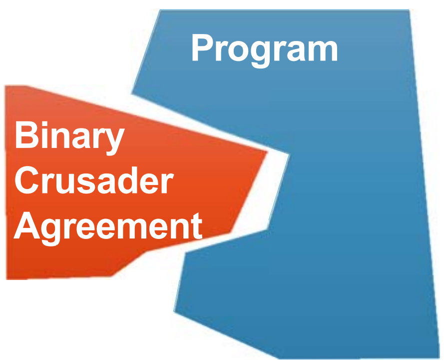
    $est := val;$

  else

    $est := c;$

Binding BCA

When an execution prefix ends in a process returning $\bot$, there is a single non-$\bot$ value that can be returned by a process in any extension
[Abraham, Ben-David, Yandamuri,PODC'22]



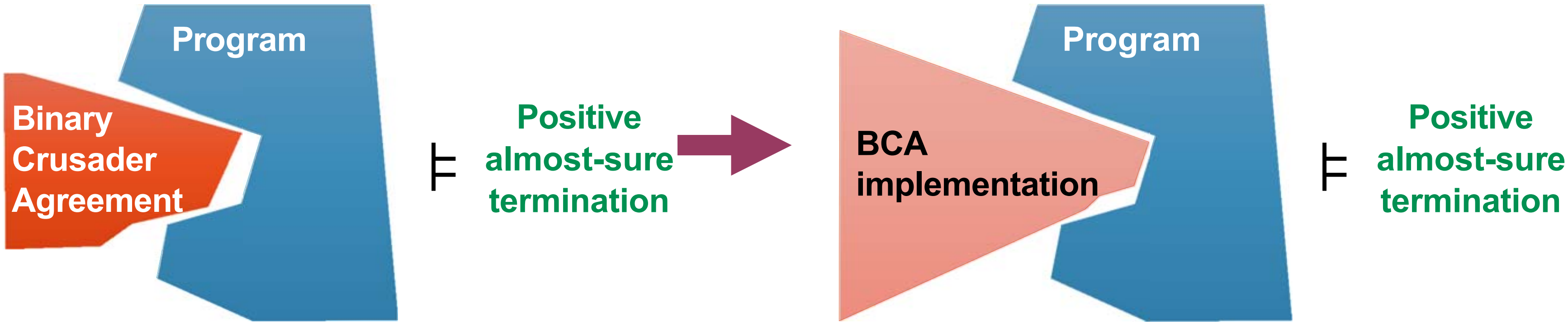
$\models$ **Positive almost-sure termination**

This is a hyperproperty

```
Input: x
r := 0; est := x;
while true do
    r++;
    val := r.BCA(est);
    c := r.Toss();
    if val ≠ ⊥ and c = val then
        output val;
        est := val;
    else if val ≠ ⊥ then
        est := val;
    else
        est := c;
```

Binding BCA

Any implementation of a binding BCA should satisfy binding as well in order to guarantee termination of the consensus algorithm



Preservation of binding can be guaranteed through forward simulations

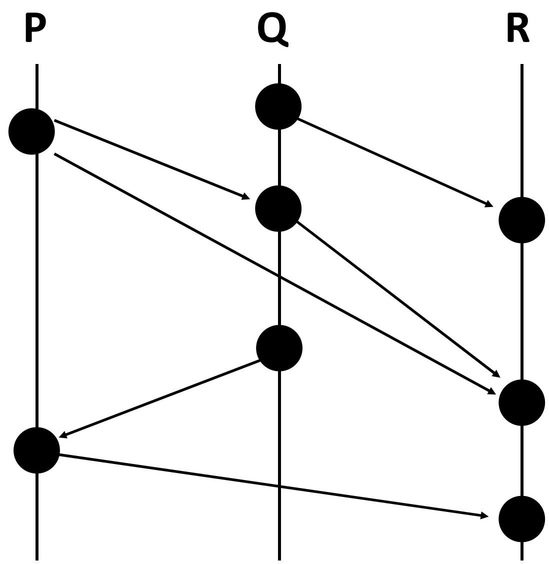
[Attiya&Enea][Dongol Schellhorn,Wehrheim]

Plan

- Motivation
- Private-coin block DAG implementations
- Proving their correctness

Block DAG Implementations

[Schett, Danezis, PODC'21]



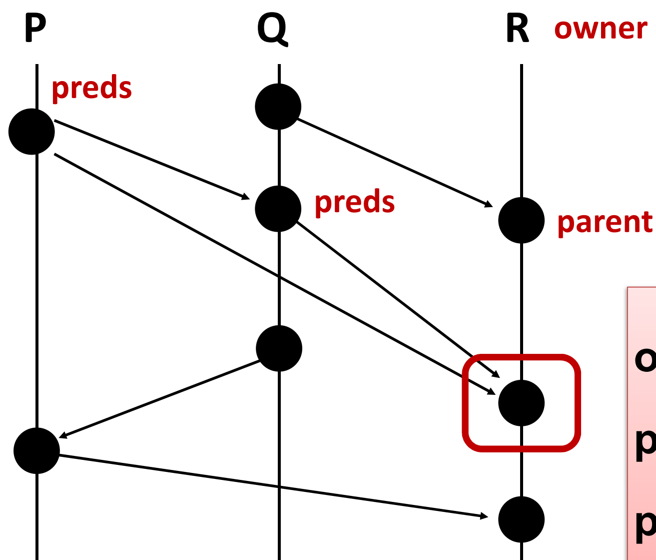
Protocol behavior = DAG of compute nodes

Ordered by Lamport's happens-before relation

A block DAG implementation = Agree on a joint DAG + Interpret DAG based on a protocol P (can use the same DAG to interpret multiple protocols)

Tolerates Byzantine failures

Blocks: Terminology



owner: process id

parent: hash of previous block generated by owner

preds: hashes of blocks ≠ ancestors of the parent

data: inputs, shared objs. return values, **random string**

Implementation of a protocol P

Local state: set of **valid** blocks (the joint DAG)

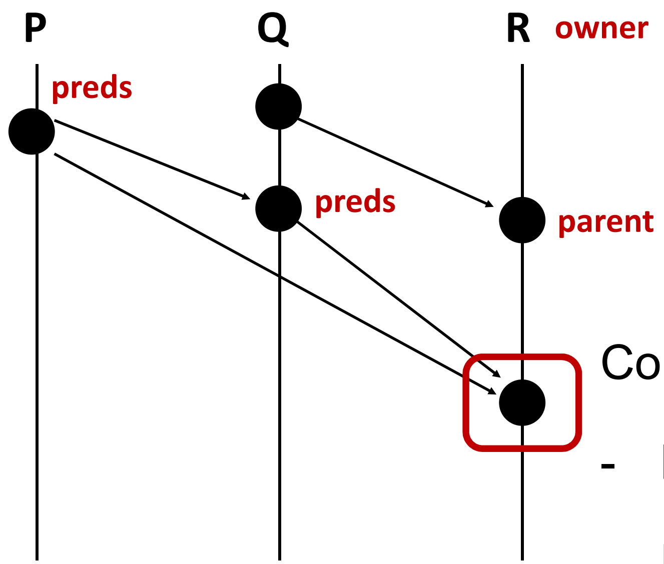
+ interpretations of blocks w.r.t. P (protocol configurations)

```
Generate block (based on the current joint DAG)

If new blocks are received, interpret them according to P

Exchange blocks
```

Interpretation of Blocks



Compute new local state of R:

- Based on its state in **parent**
- Receiving messages sent in compute steps of **preds**
- Using inputs, random choices, shared objects, return values in **data**

Exchanging Blocks

Guarantee: if some correct process adds a block to its DAG, then every correct process eventually adds the same block

- Every block is signed (Byzantine failures) before being broadcasted
- A block is valid if it is correctly signed and all its predecessors are valid (ensures acyclicity)
- If a predecessor block is missing, send a forwarding request (pull) to its owner

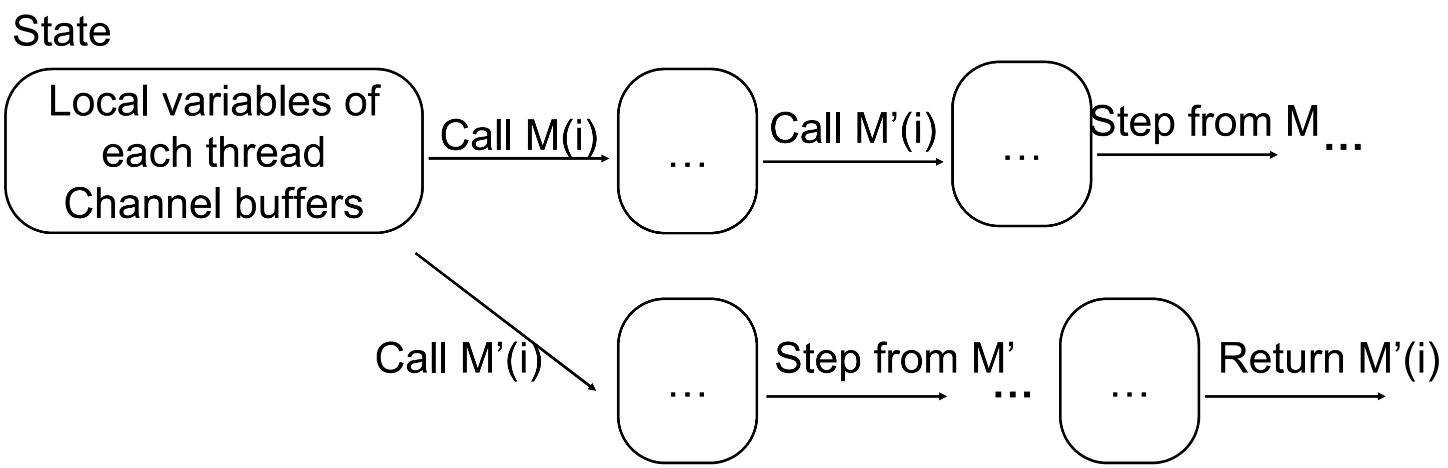
Plan

- Motivation
- Private-coin block DAG implementations
- Proving their correctness

Labeled Transition Systems (LTSs)

Model nondeterministic protocols as Labeled Transition Systems (LTSs)

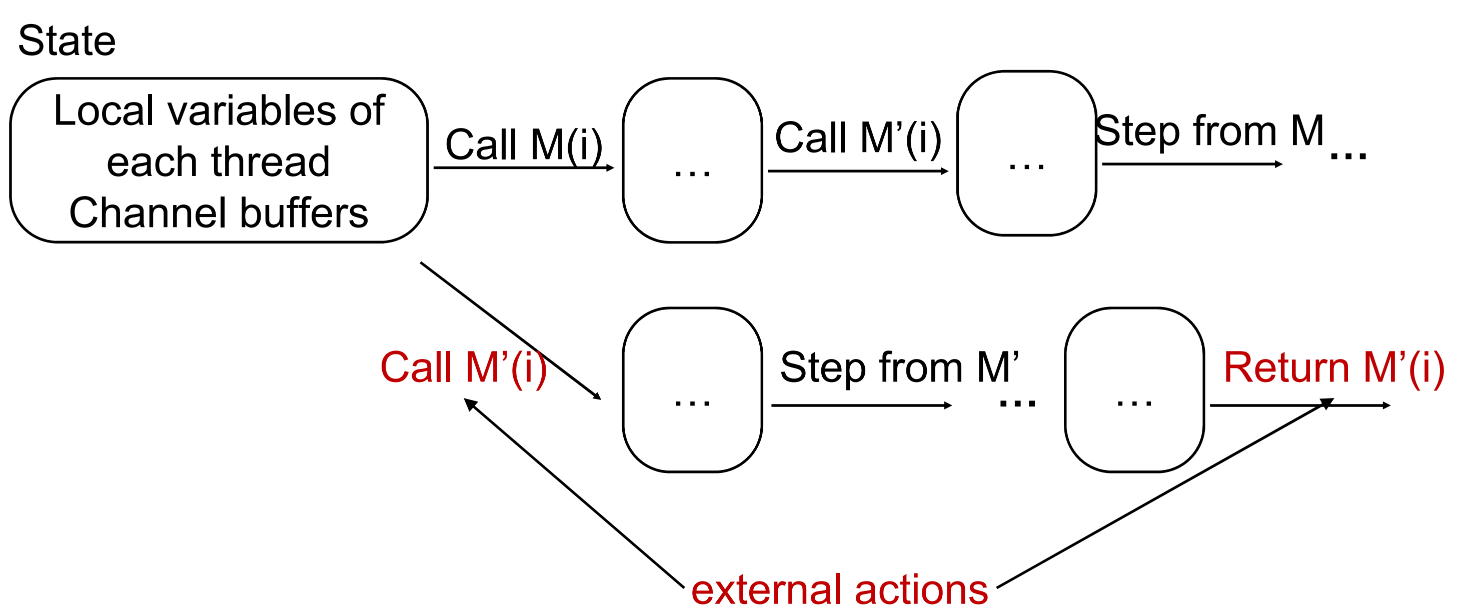
[Keller, CACM'76]



Labeled Transition Systems (LTSs)

Model nondeterministic protocols as Labeled Transition Systems (LTSs)

[Keller, CACM'76]



Labeled Transition Systems (LTSs)

Trace (history): sequence of external actions in an execution of the LTS

Trace inclusion: For an implementation **Imp** and an abstract protocol **Abs**

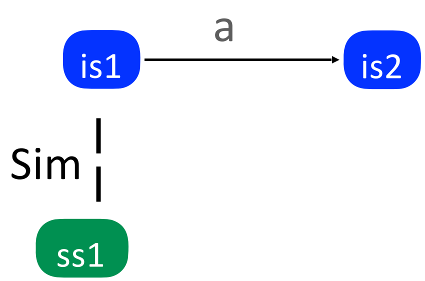
$$\textbf{Traces(Imp)} \subseteq \textbf{Trace(Abs)}$$

Forward Simulation for LTSs

Prove trace inclusion by induction via a simulation relation between states of implementation and abstract protocols

[Lynch, Vaandrager, 1996]

Forward simulation

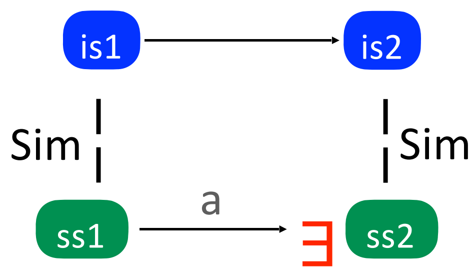


Forward Simulation for LTSs

Prove trace inclusion by induction via a simulation relation between states of implementation and abstract protocols

[Lynch, Vaandrager, 1996]

Forward simulation

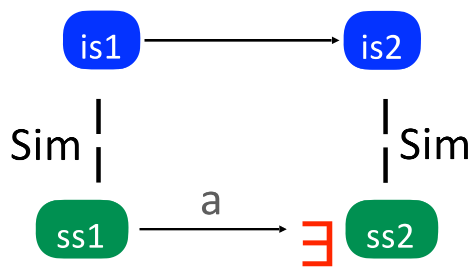


Forward Simulation for LTSs

Prove trace inclusion by induction via a simulation relation between states of implementation and abstract protocols

[Lynch, Vaandrager, 1996]

Forward simulation



Preserve **hyperproperties** w.r.t deterministic scheduler (strong adversary) in every context

[Attiya, Enea][Dongol, Schellhorn, Wehrheim]

For randomized protocols, include probabilities in transition labels

⇨ weak probabilistic simulation [Segala, CONCUR'95] which has same guarantees

Main Transitions in Block-DAG

validateBlock(i → j): p_i validates a block issued by p_j

compute(i,ρ): p_i produces and disseminates a new block with ρ as its randomness, and then interprets the new block (and other previously uninterpreted blocks)

sendFWD(i → j) p_i pulls (requests a block) from p_j

replyFWD(j → i) denotes a transition where p_j responds with a block to p_i

deliverBlocks(i → j) all the blocks in the output buffer i → j are moved to the input buffer i → j

indicate(i, w) a response w from shared service is returned to p_i

Theorem. There is a forward simulation from the block DAG implementation of a protocol P to the original protocol P (as LTSs)

Proof idea: Relate configurations of the block DAG implementation with configurations of the original protocol:

- local state of process p = local state derived by interpreting the most recent block issued by p
- messages in transit from p to q: sent by interpreting a block issued by p which is not yet validated by q

Conclusion

- A block DAG implementation of randomized distributed protocols, which extends the deterministic one [Schett, Danezis, PODC'21]

- Faithfulness of the implementation = forward simulations (preserving trace distributions, or hyperproperties)

Future Work:

- Private-coin DAG-based protocols
- Other cryptographic protocols