Towards Compositional Explicit State Model Checking

Damien Zufferey
RiSE MSR / IST Austria

Shaz Qadeer, Ethan Jackson,
Research in Software Engineering
Microsoft Research, Redmond

Sriram Rajamani, Ankush Desai,
Rigorous Software Engineering
Microsoft Research, Bangalore

Overview

- Motivation
- Automata, parallel composition, and properties
- Compositional rule
- (Non-)Compositional verification
- Semantic gap

Motivation

Consider an automaton with n states.

A reachability question can be answered in $O(n)$.
[logspace complexity]

Problem:
m automata running in parallel
The complexity becomes $O(n^m)$.

Can we do better ?
Check one part at the time and assume the rest is correct!
The complexity goes down to $O(m\,n)$.

Compositional verification tells us how we can achieve this.

Automata

Automata are used for implementation and specification.

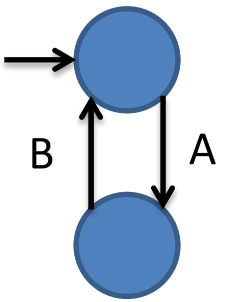
Set of states

Transitions: $\delta \subseteq S \times \Sigma \times S$

$$(S, \Sigma, \delta, i)$$

Alphabet

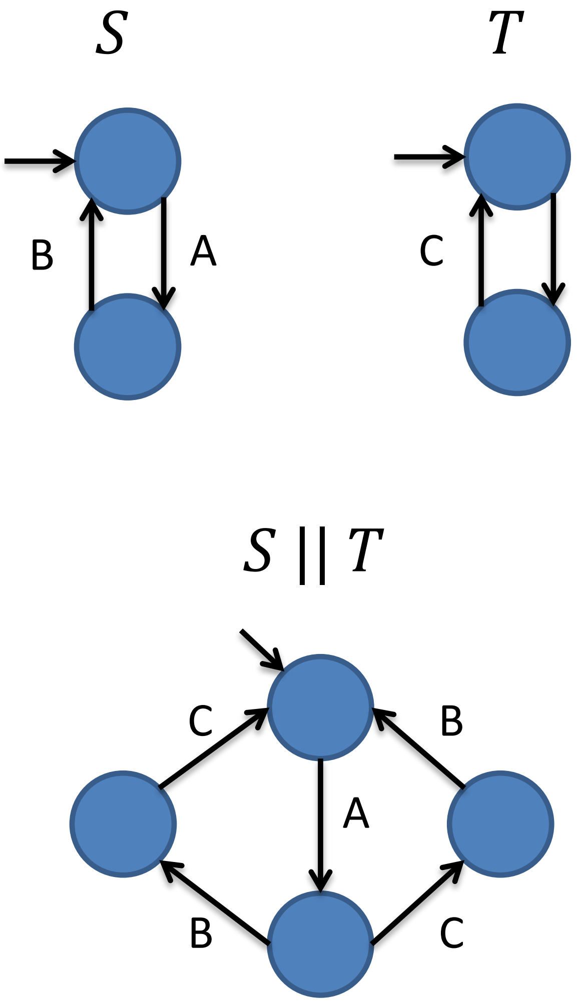
Initial state



$\Sigma = \{ \text{A}, \text{B} \}$

Parallel composition

Parallel composition is the **synchronous product**. (trace intersection)



Shared transition

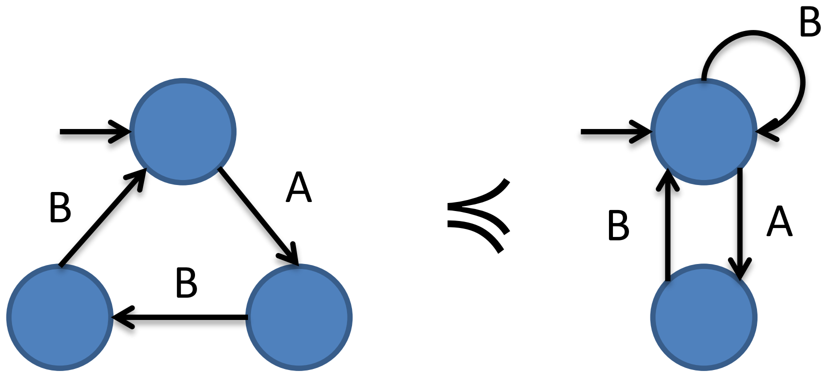
$$\frac{s \xrightarrow{\alpha} s' \qquad t \xrightarrow{\alpha} t'}{(s,t) \xrightarrow{\alpha} (s',t')}$$

Local transition

$$\frac{s \xrightarrow{\alpha} s' \qquad \alpha \notin \Sigma_T}{(s,t) \xrightarrow{\alpha} (s',t)} \qquad \frac{\alpha \notin \Sigma_S \qquad t \xrightarrow{\alpha} t'}{(s,t) \xrightarrow{\alpha} (s,t')}$$

Properties

Specifications are monitors that define the set of allowed traces.
An implementation is correct if it refines the specifications.
Refinement is trace inclusion.



For a **reachability** question we can create a monitor that is respected iff the target is not reachable. A monitor enforces a safety property and reachability is the “dual” of safety.

On the other hand, we are **not checking liveness** properties.

Compositional verification

Let S be a specification and I a set of implementation machines.

We want to prove $I \preccurlyeq S$ (hard to do).

On the other hand, $S \,||\, I \preccurlyeq S$ is trivially valid.
But it does not say anything about I.

Compositional verification tells us how we can do:

$$\frac{S' \,||\, I' \preccurlyeq S \qquad S'' \,||\, I'' \preccurlyeq S \qquad S''' \,||\, I''' \preccurlyeq S \qquad \ldots}{I \preccurlyeq S}$$

where S' are parts of S and I' are subsets of I.

Simple hierarchical case

Some of you may know compositional verification under the name “assume-guarantee reasoning”.

Consider the following proof rule:

$$\frac{I_1 || A \preccurlyeq S \quad I_2 \preccurlyeq A}{I_1 || I_2 \preccurlyeq S}$$

Correct because: $||$ is monotonic, $\preccurlyeq$ is transitive.

More complex version of the rule might look like:

$$\frac{I_1 || A_2 \preccurlyeq A_1 \quad I_2 || A_1 \preccurlyeq A_2 \quad A_1 || A_2 \preccurlyeq S}{I_1 || I_2 \preccurlyeq S}$$

Not general: restrictions on the As.

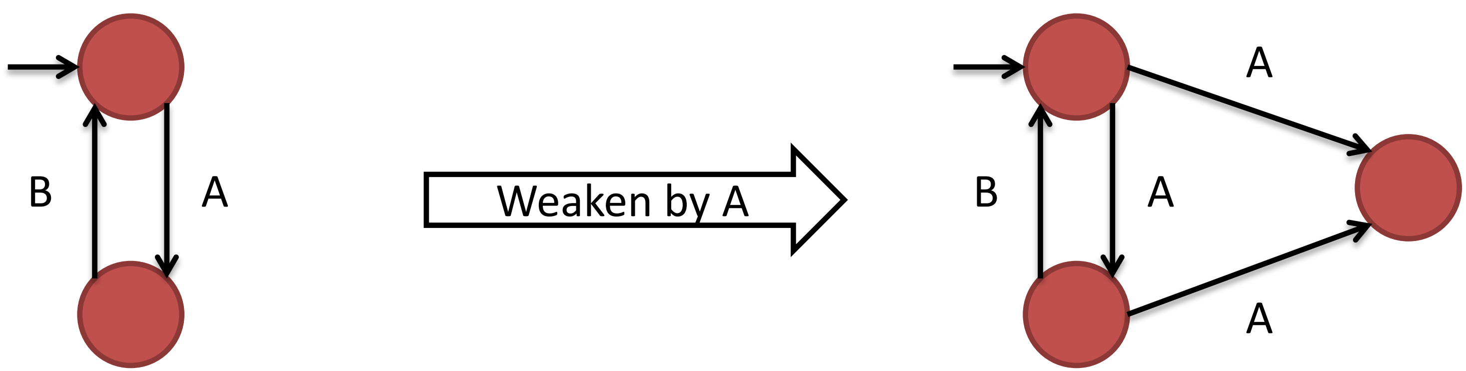
The compositional rule

Given a spec *S*, and a set of implementation machines *I*:

If for all *E* in Σ of *S*,
there is $I' \subseteq I$ such that

$$I' || (S \text{ weakened by } E) \preccurlyeq S$$

Then $I \preccurlyeq S$.



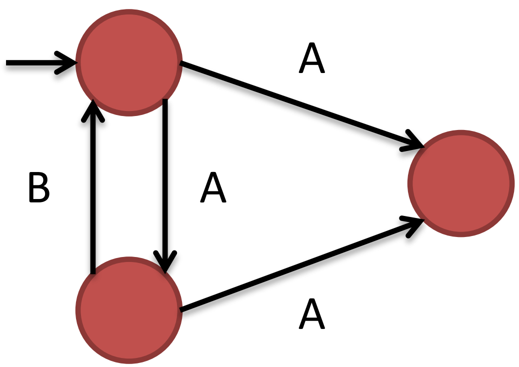
Why does it works ?

The idea:

(S weakened by E) means "assumes that S is true **up to** E".

Proof by induction on the trace:

Assuming that the trace is safe after i steps
we must show it is safe after $i + 1$ steps.

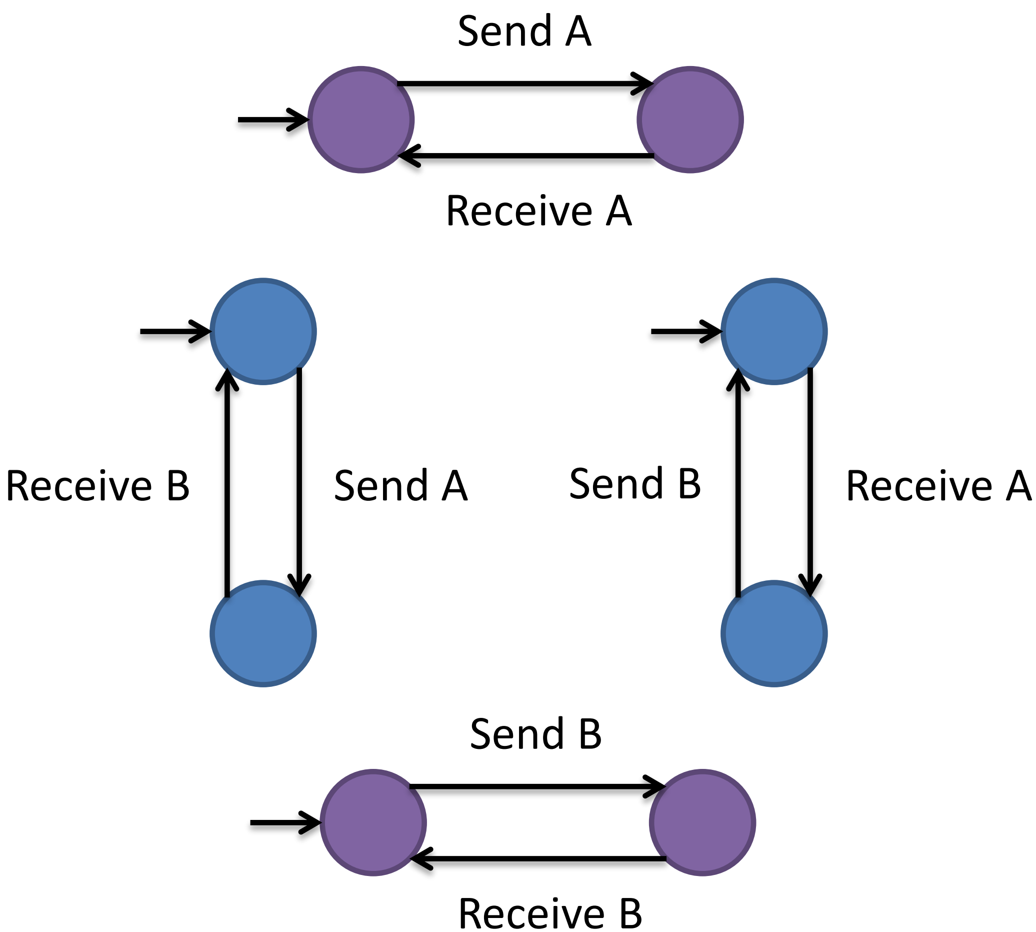


Does not prevent A from occurring at step $i + 1$. So an implementation machine must restrict A.

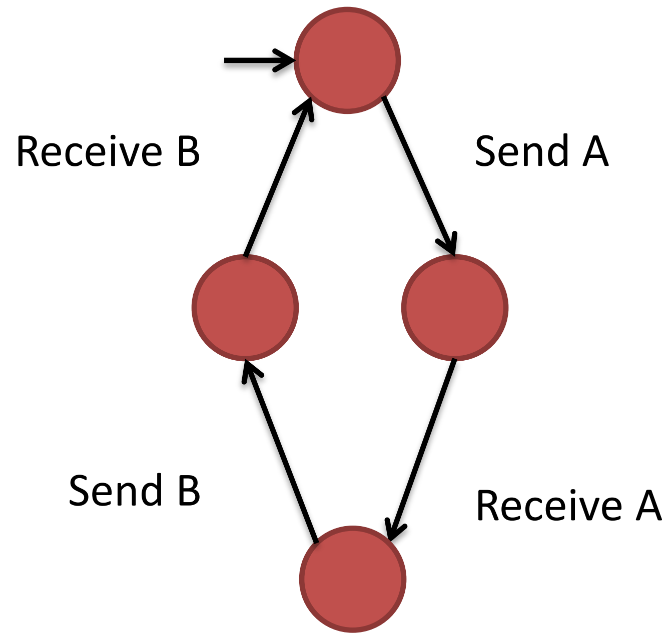
The weakening is done w.r.t. all E.

Example: Pingpong (1)

Implementations (machines and channels)



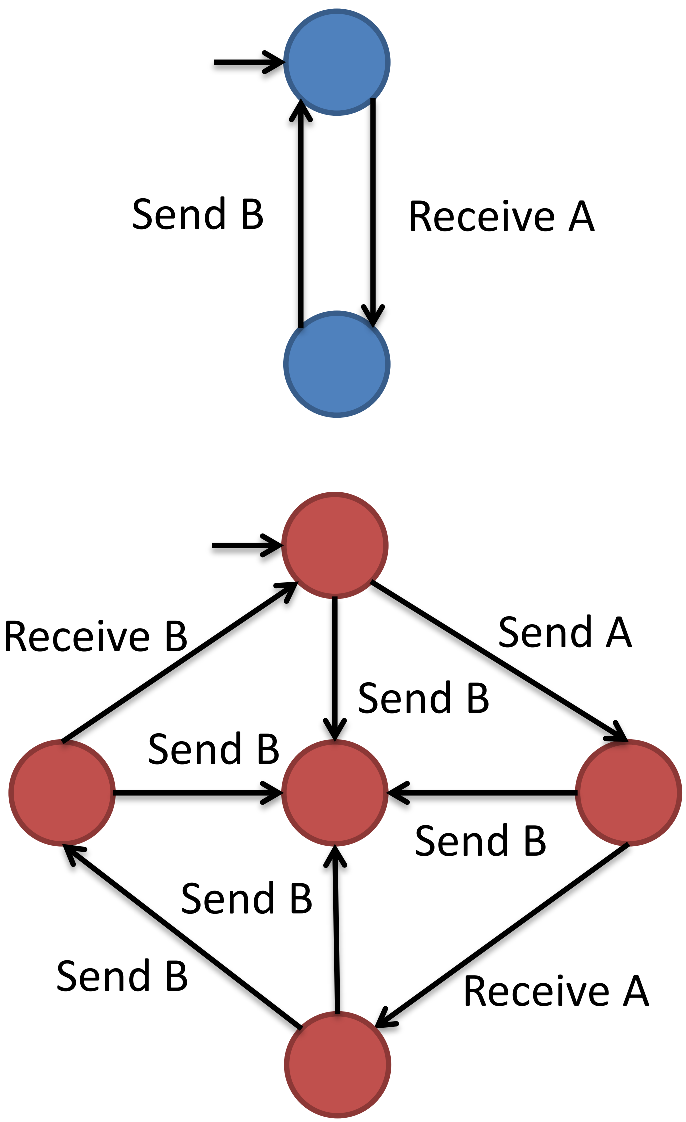
Specification



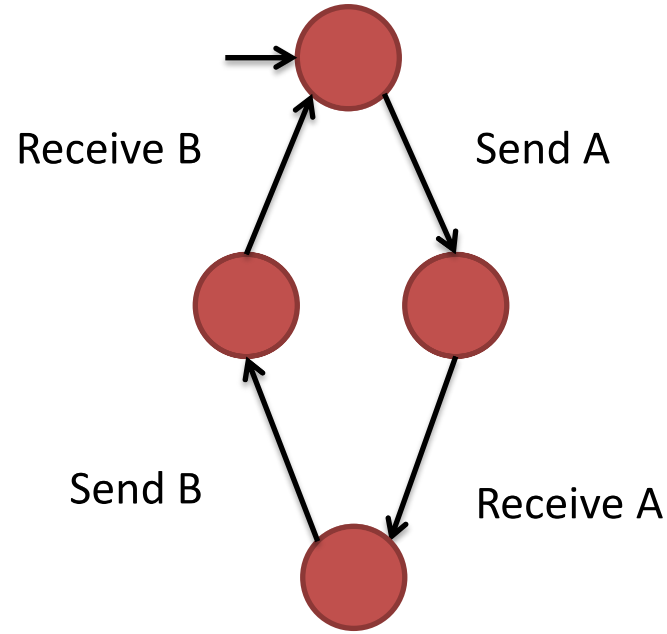
Need to prove 4 lemmas:

I’ || (*S* weakened by *Send A*)
I’’ || (*S* weakened by *Receive A*)
***I’’’* || (*S* weakened by *Send B*)**
I’’’’ || (*S* weakened by *Receive B*)

Example: Pingpong (2)

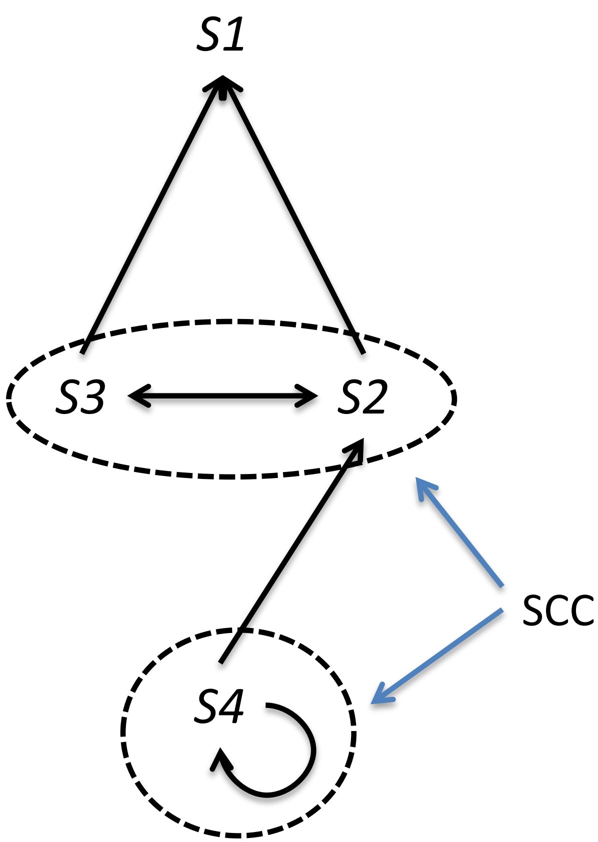


refines



(Non-)Compositional verification

Dependency graph for the specifications



The rule generalizes to more than one spec. However, it tends to generates many lemmas. We want to apply it only when it is needed.

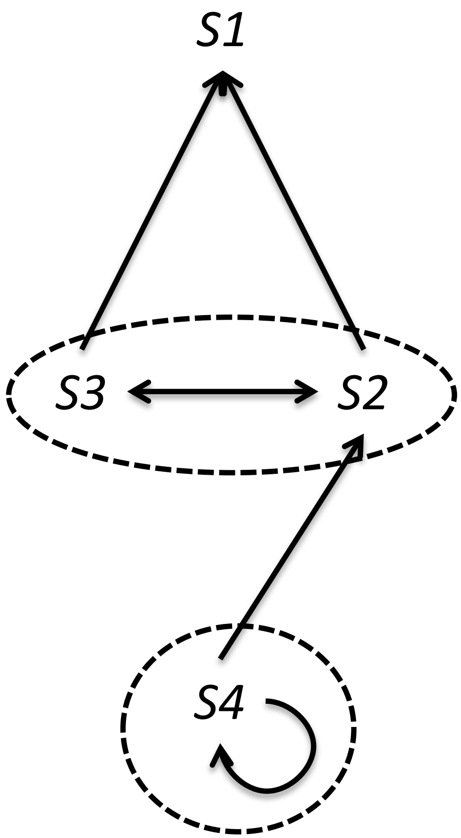
We use a dependency graph of specification. The LHS of a lemma for some *S* can only use other specification if *S* depends on it.

Within the strongly connected components (SCC), the compositional rule is applied.

For the acyclic part the dependencies can be assumed without modification.

(Non-)Compositional example

Dependency graph for the specifications



What the graph means:
S1 do not use any other spec.
{*S2*,*S3*} uses {*S1*,*S2*,*S3*}.
S4 uses {S2,S4}.

{*S2*,*S3*} and *S4* are weakened for their respective lemmas.
{*S2*,*S3*} can use *S1* without weakening.
S4 can use {*S1*,*S2*,*S3*} without weakening.

The system generates the proof obligations and the user pick the LHS for each lemma.

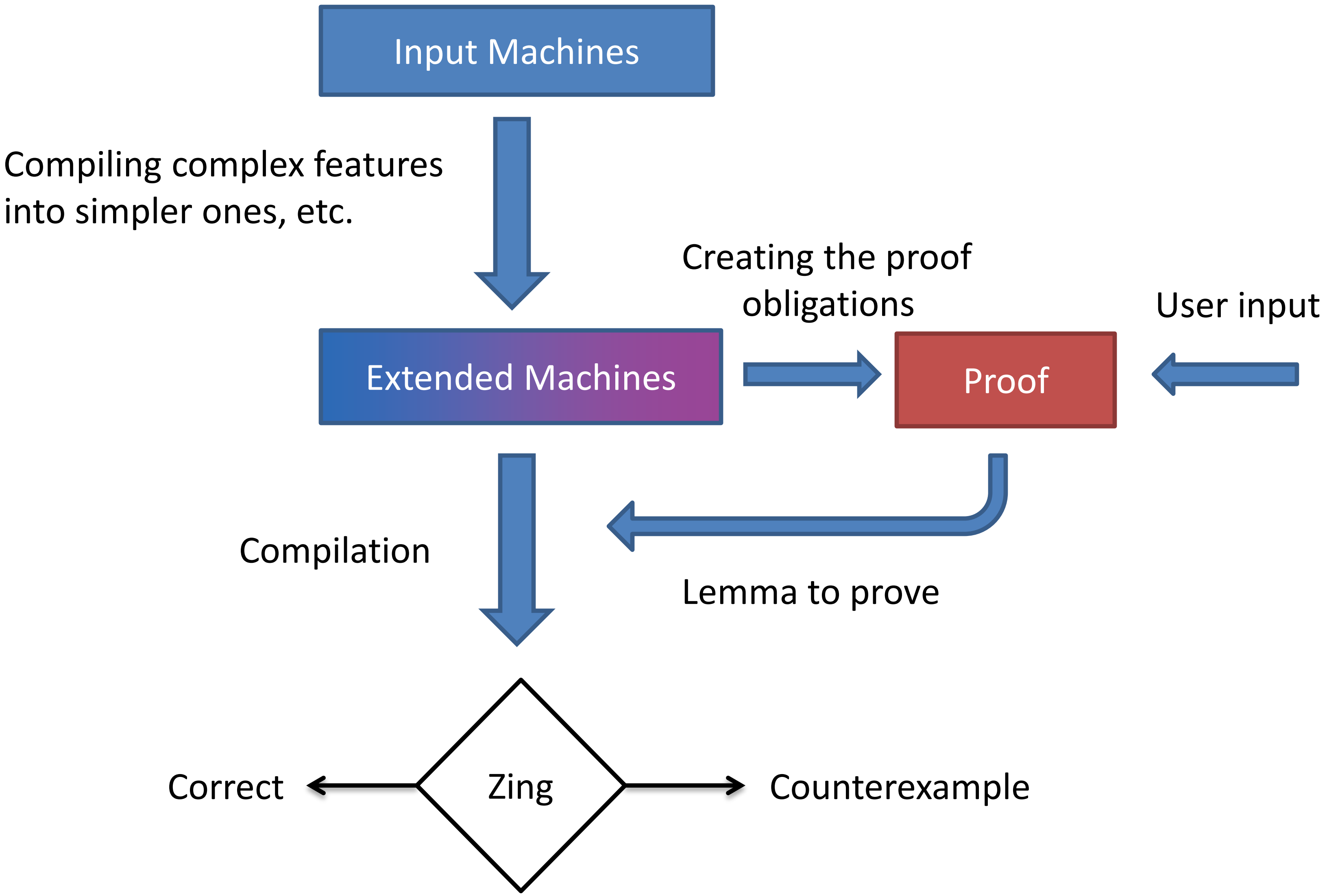
For example:
I || *S1* || *S2* || (*S4* weakened by *E*) refines *S4*.

Semantic gap

Matching the semantics of the machine to the automata used in the compositional rules:

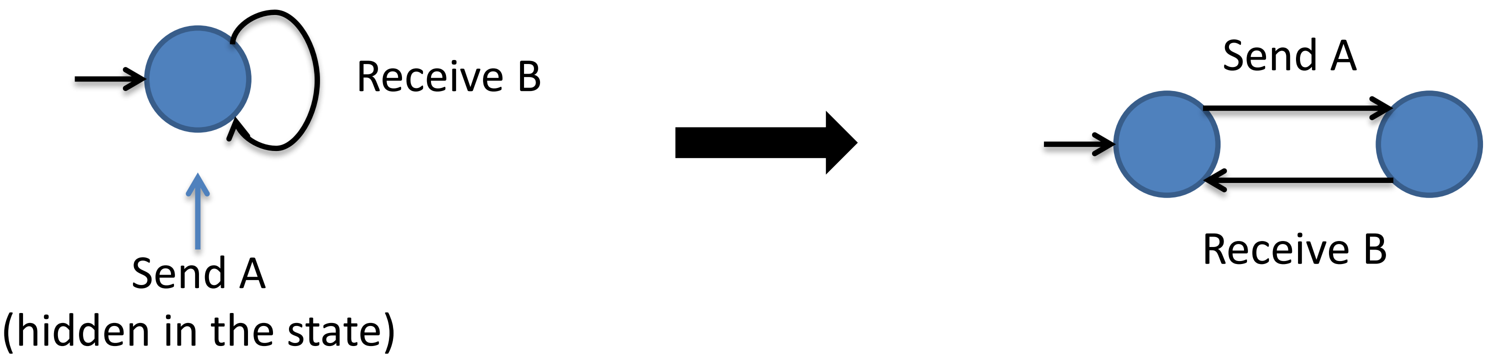
- Asynchronous message-passing
- Unbounded mailboxes
- …

Overview

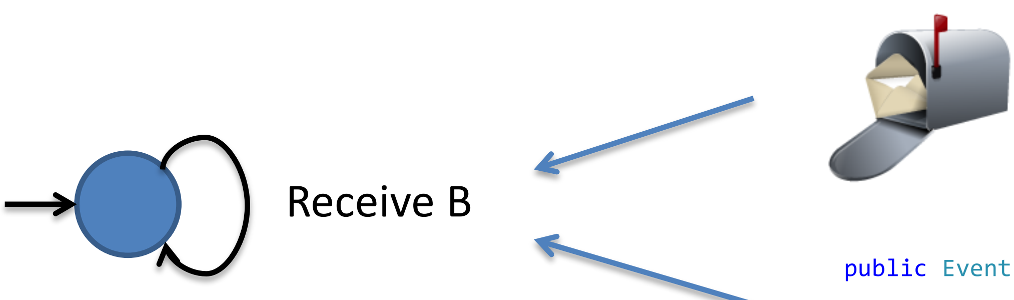

Input Machines
Compiling complex features
into simpler ones, etc.
Creating the proof
obligations
User input
Extended Machines
Proof
Compilation
Lemma to prove
Correct
Zing
Counterexample

Emphasis on the Reactive Aspect

Only the “receive” events are shown on the edges.
Sending occurs within the states.

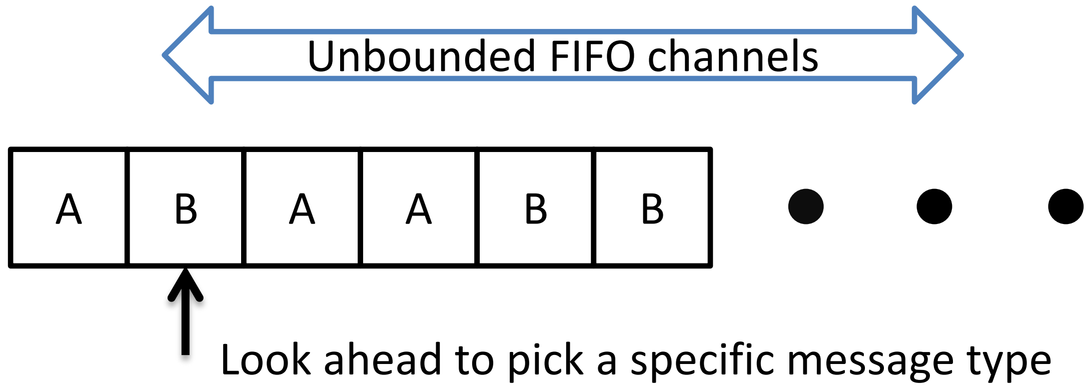


External events come from the input buffer;
Internal events are the result of calling some function.

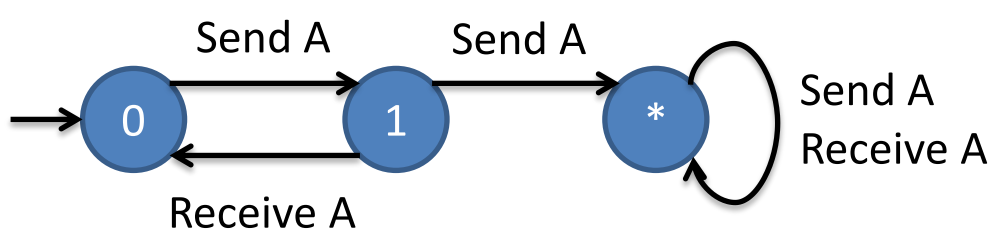



```
public Event foo(Bar arg) {
    …
    return e;
}
```

Communication Channels



Unbounded channels can give a hard time to an explicit-state model checker.
Fortunately, the proof (usually) requires only simple lemma about the channel:



These lemmas (finite abstraction of the channel) are generated per event.
No ordering between events
Precision of the event is up to a bound.

Thanks And Questions!