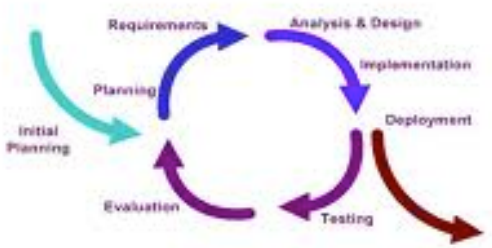
On Requirements Verification for Model Refinements

Paola Spoletini
Università degli Studi dell'Insubria

Carlo Ghezzi	Politecnico di Milano
Claudio Menghi	Politecnico di Milano
Amir Molzam Sharifloo	Politecnico di Milano

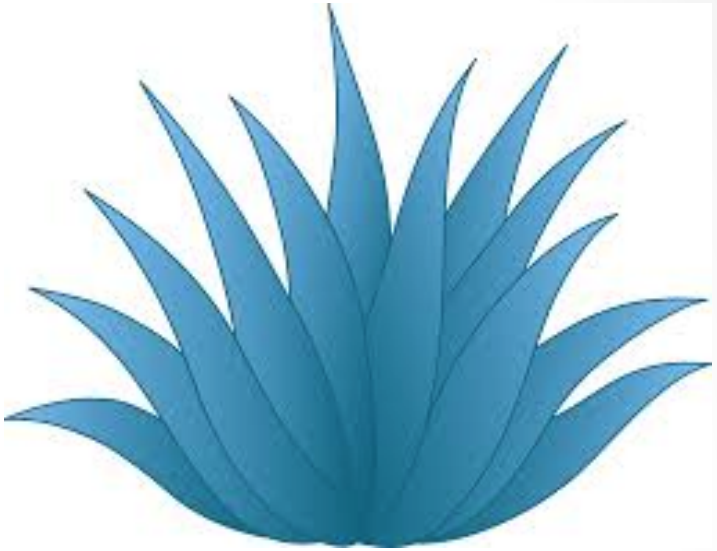
Motivation

- Modern software systems are often developed in an agile fashion
- At each step the model of the system is incomplete …
  - Perform verification at each iteration
  - Wait until the end to verify the system
  - Make assumptions on the missing parts before verifying



How to perform verification **efficiently** and **automatically** at each iteration?

Idea

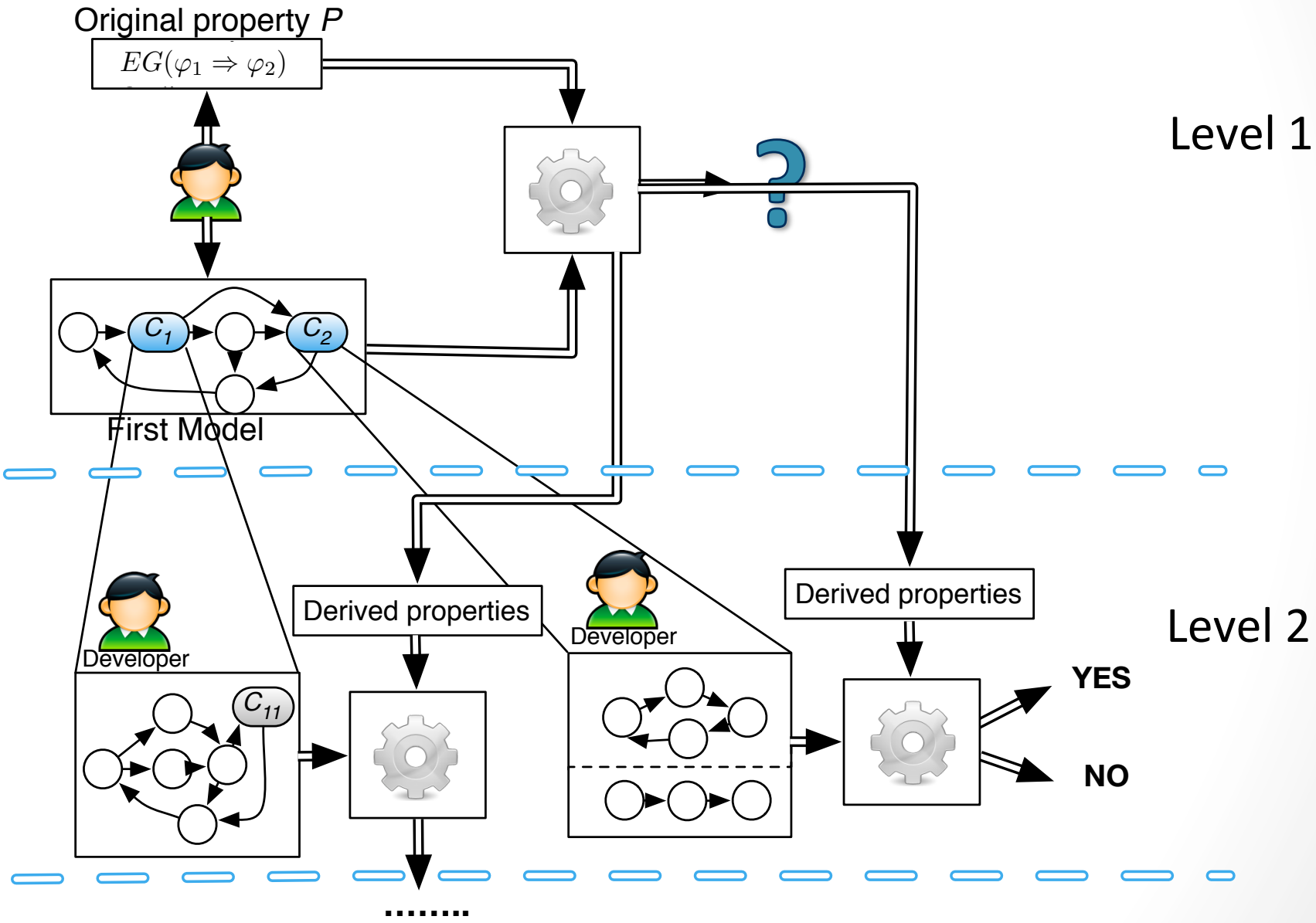


AGaVE: AGile Verification Environment

Verification technique

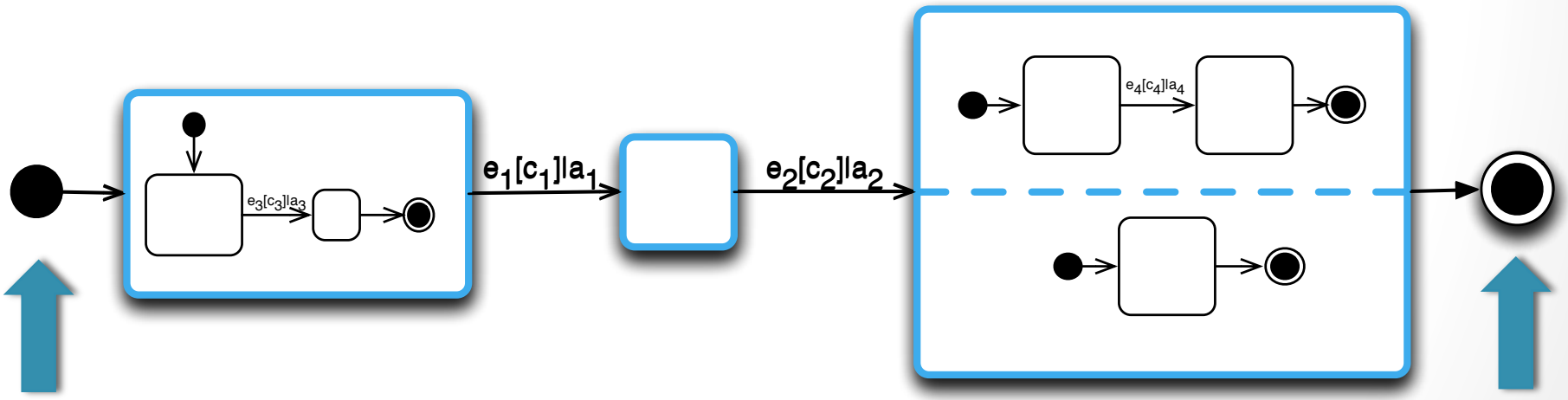
- to check whether a specification satisfies a given property
- to (automatically) generate sub-properties that the missing components have to respect

Approach overview



Statecharts

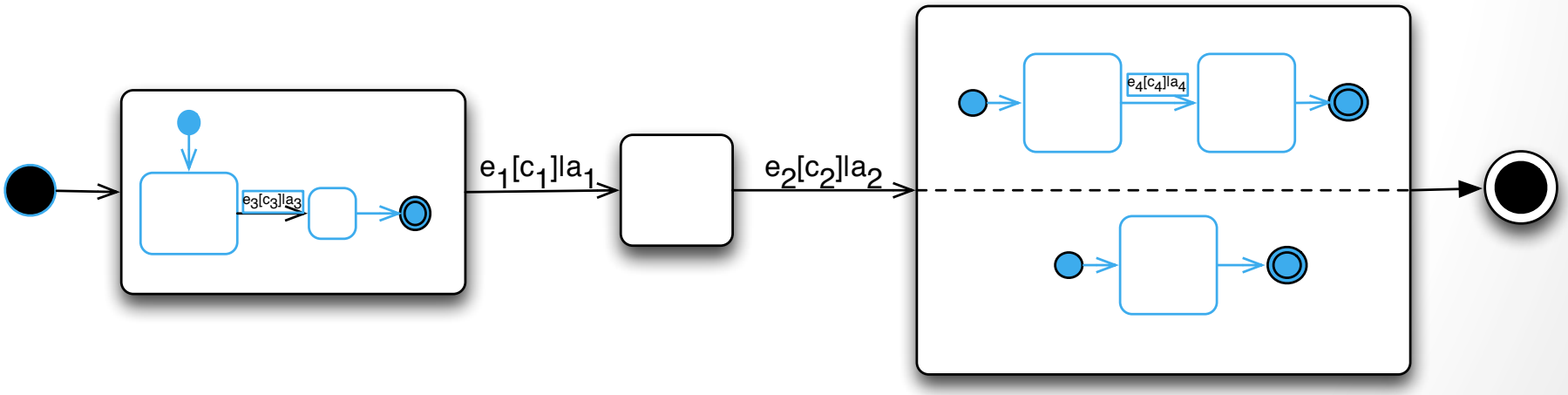
- Statecharts extend finite state machines with
  - Hierarchy
  - Concurrency
- Formally, S = $\langle Q, q_0, q_F, St, \rho, \tau \rangle$



Statecharts

- Statecharts extend finite state machines with
  - Hierarchy
  - Concurrency
- Formally, $S = \langle Q, q_0, q_F, St, \rho, \tau \rangle$

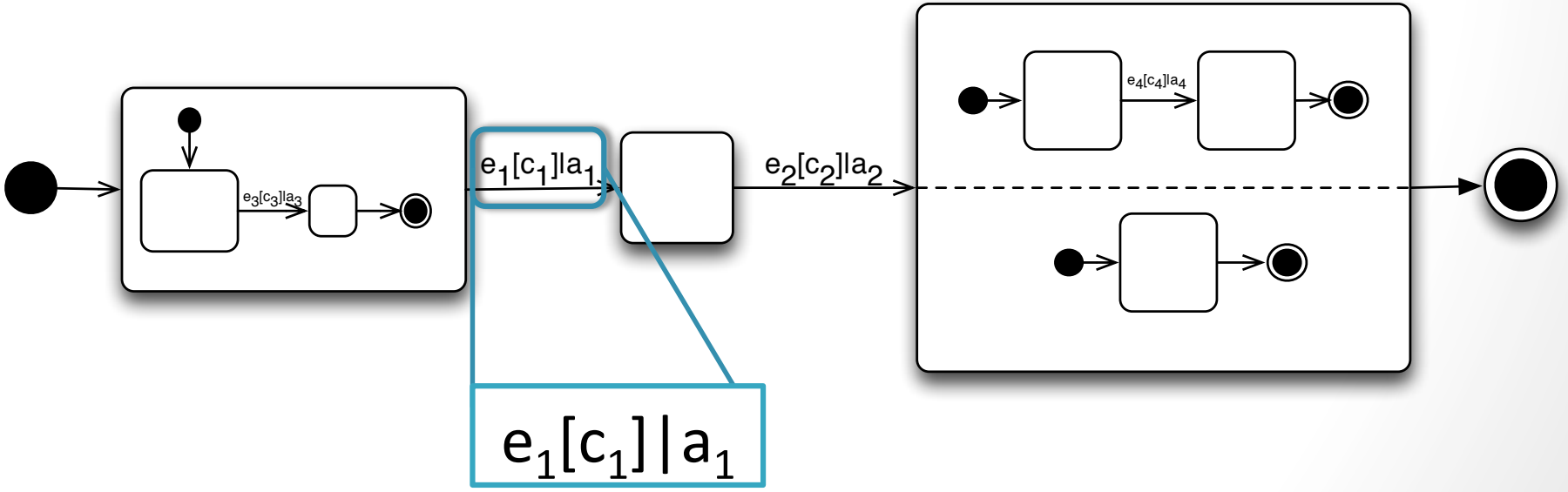
$$\rho \subseteq (Q - \{q_0, q_F\}) \times \{AND, OR\} \times \wp(St)$$



Statecharts

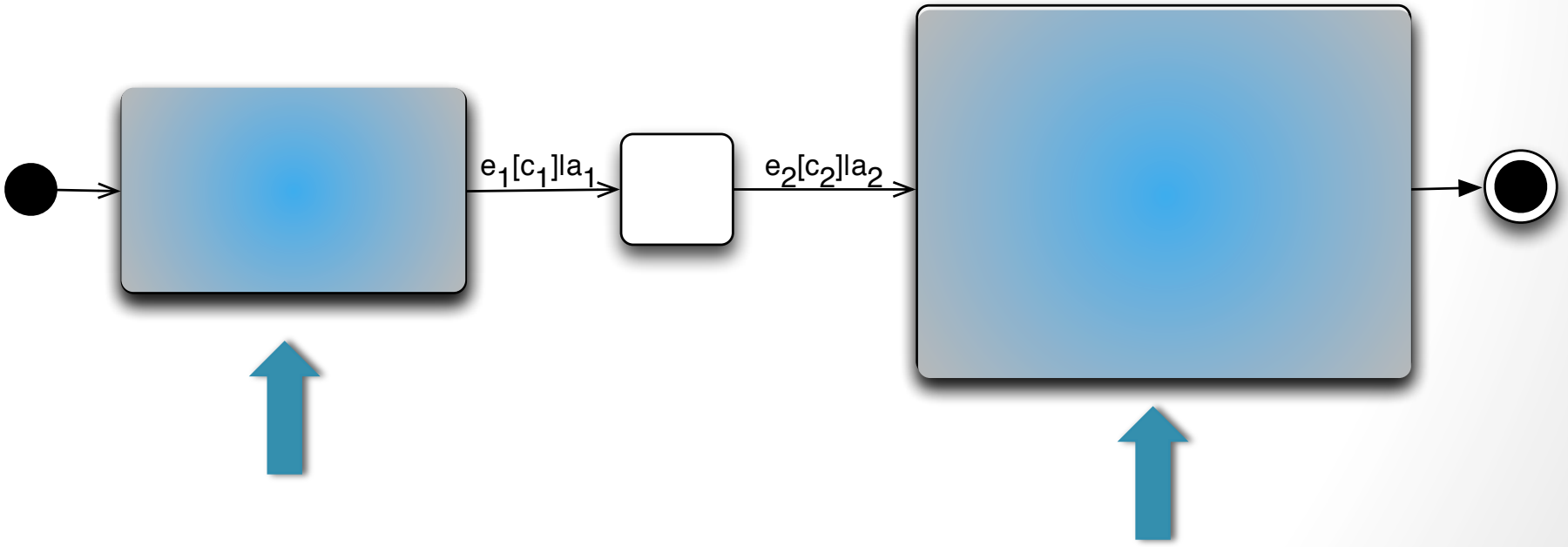
- Statecharts extend finite state machines with
  - Hierarchy
  - Concurrency
- Formally, $S = \langle Q, q_0, q_F, St, \rho, \tau \rangle$

$$\tau:(Q-\{q_F\})\times E\times C(I)\rightarrow(Q-\{q_0\})\times A(I)$$

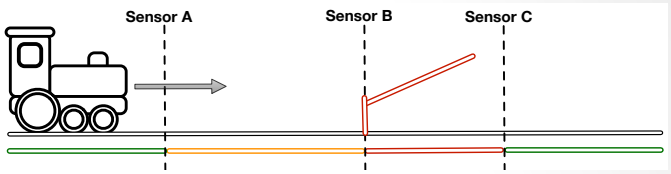


Statecharts

- Statecharts extend finite state machines with
  - Hierarchy
  - Concurrency
- Formally, $S = \langle Q, q_0, q_F, St, \rho, \tau \rangle$



Example



RCS

Train

s_1

t_1: e_1[traveling]/ approaching ¬traveling

t_3: e_3[crossing]/ travelling ¬crossing

s_3

t_2: e_2[approaching ¬open]/ cross ¬appro

s_2 T

Gate

s_4

t_4: [approaching]/ ¬open

t_5: [traveling]/ open

s_5

t_6: []/ askPermit

s_7

t_8: e_4[]/ permit

s_8

t_9: [permit]/ ackToCentralStation

t_{10}: Timeout [¬permit]/

s_9 T

t_{11}

t_{11}: e_5[]/ failure

t_{12}: []/ ¬failure

t_7: []/ askPermit

s_{10}

t_{13}: []/ alarm

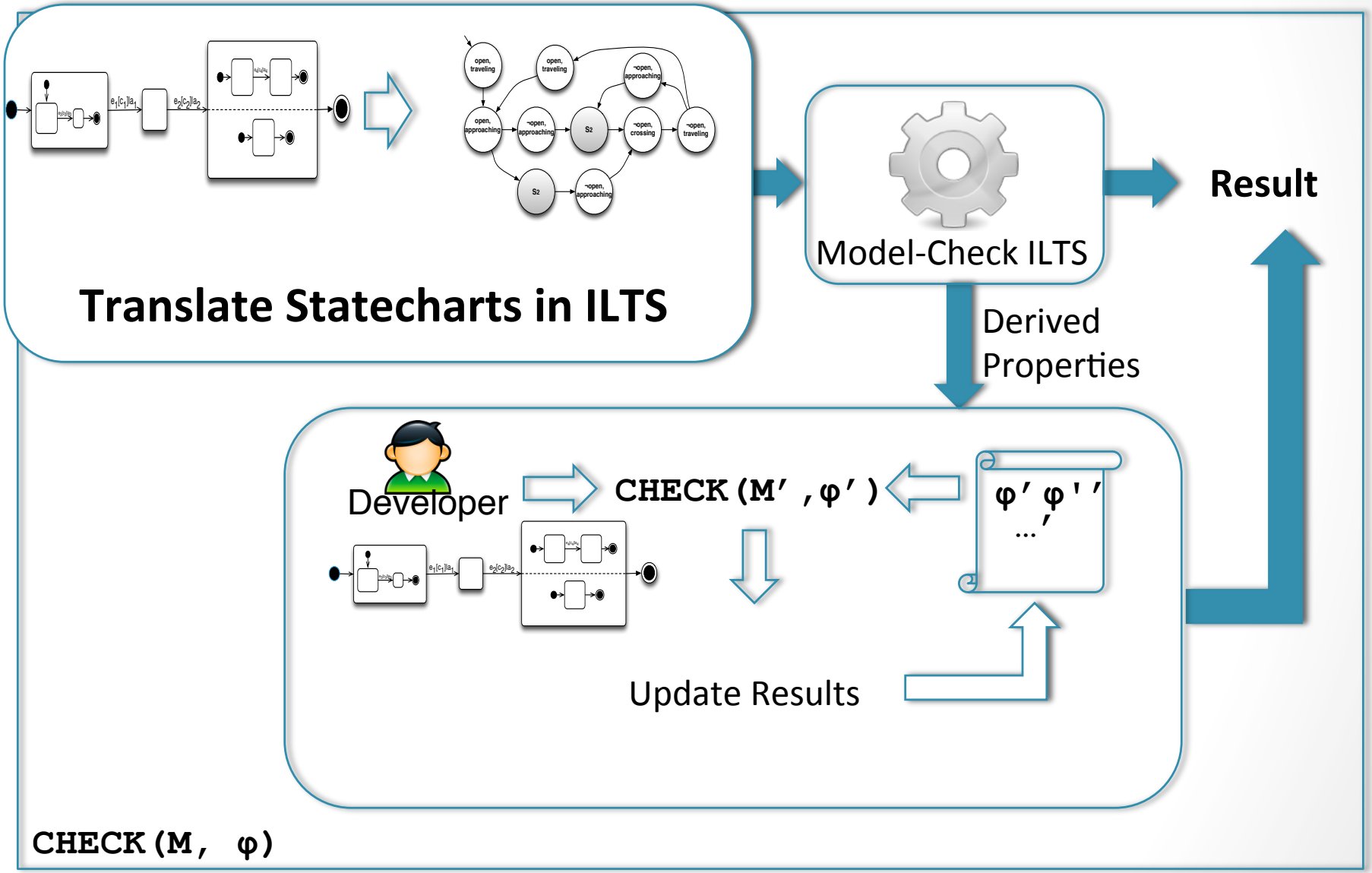
Path-qCTL

- qCTL = **qualitative** CTL
- Path-qCTL = qCTL + operator on a finite path
- Its syntax is defined as

$$\phi \rightarrow \phi \wedge \phi \mid \neg\phi \mid E\phi U\phi \mid EG\phi \mid p \mid E_pG\phi$$

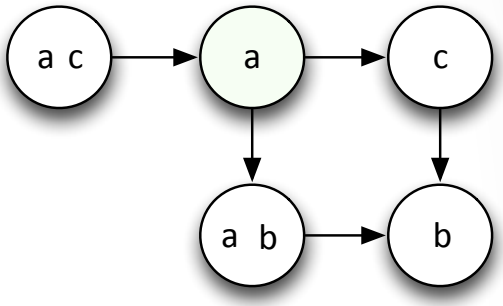
  - $E_pG\phi$ = "There exists a path that reaches the final state in which ϕ always holds"
- Example
  - ϕ = ¬E(¬permit U crossing)

The Verification Algorithm

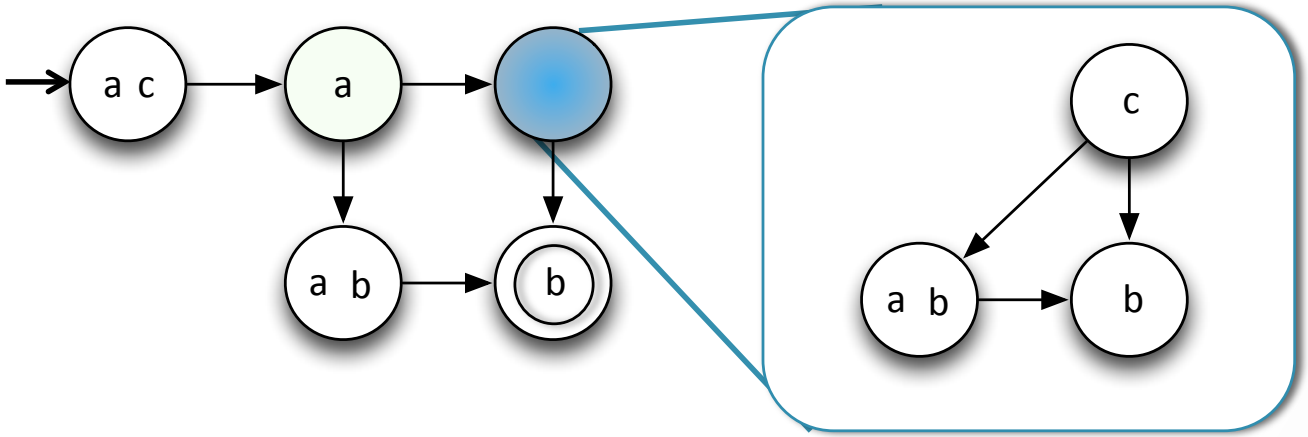


ILTS (with initial and final states)

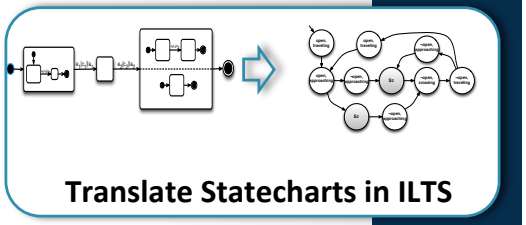
- LTS on an alphabet AP = ⟨Q, τ, L⟩
  - Q = finite set of states
  - τ ⊆ Q x Q = transition relation
  - L: Q → ℘(AP) = labeling function



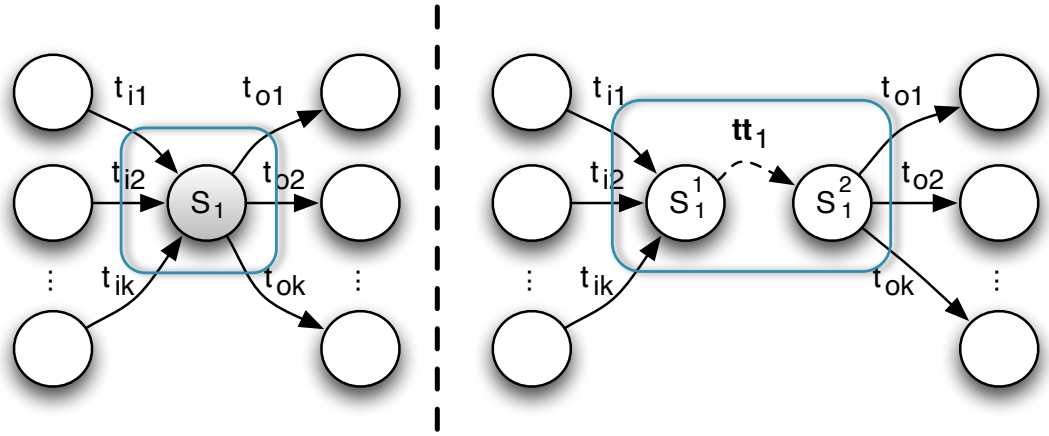
- Incomplete LTS on an alphabet AP = ⟨Q, τ, L⟩
  - Q is partitioned in regular and transparent states
  - Transparent states represent components



From Statecharts to ILTS

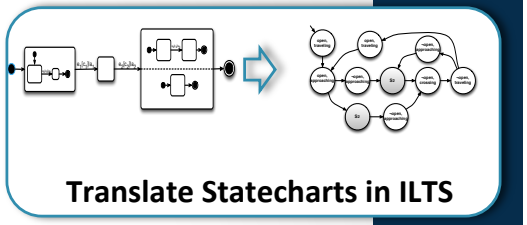


1. Preprocess transparent states



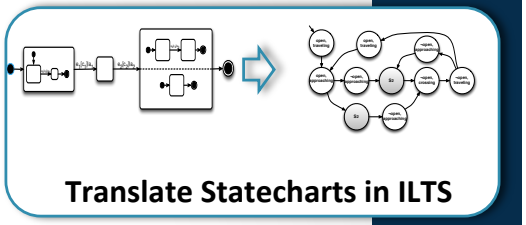
From a state to a transition

From Statecharts to ILTS



1. Preprocess transparent states
2. Eliminate AND-states
   - states = cartesian product of the states of the sub-Statecharts
   - Transitions = all possible interleaving steps

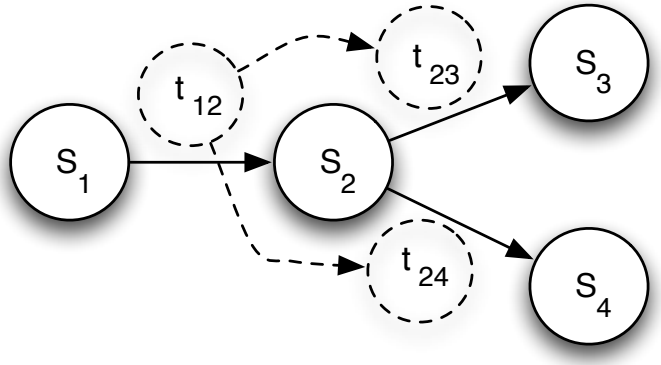
From Statecharts to ILTS



1. Preprocess transparent states
2. Eliminate AND-states

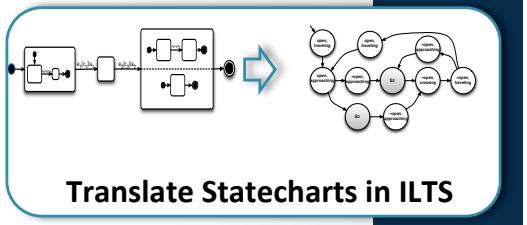
Preprocessing

3. Build the ILTS structure



Statechart transitions become ILTS states

From Statecharts to ILTS

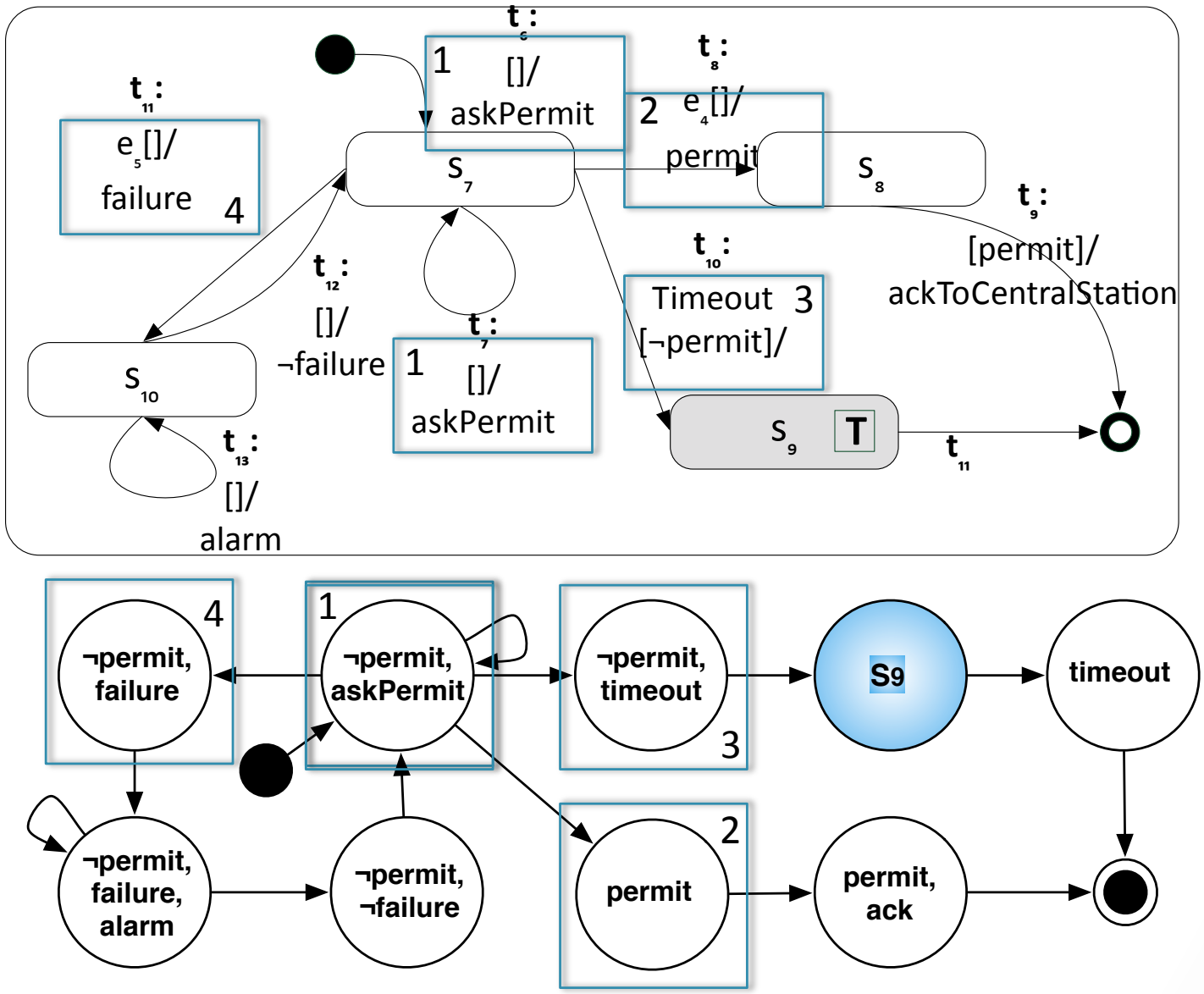

Translate Statecharts in ILTS

1. Preprocess transparent states
2. Eliminate AND-states

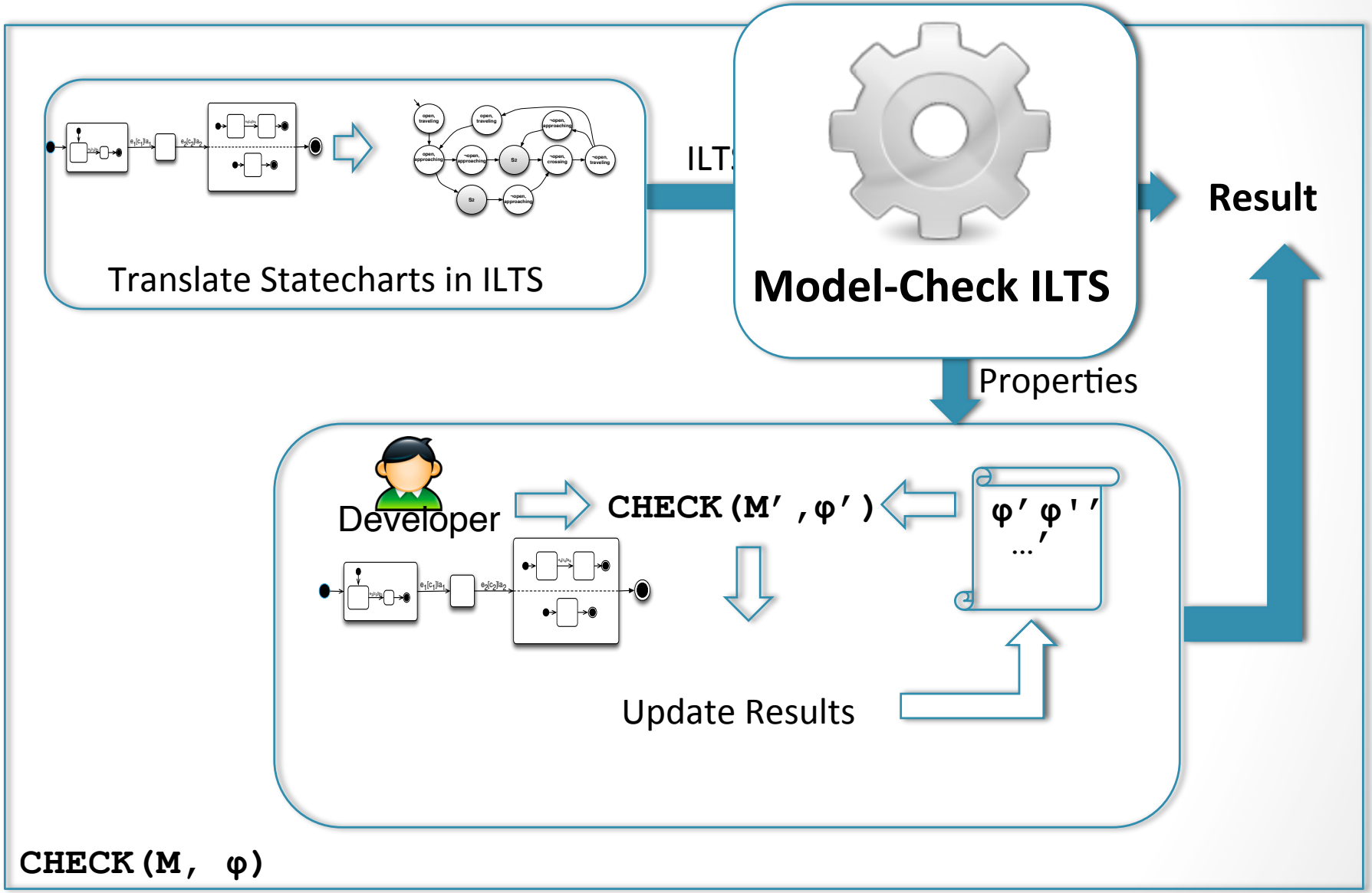
Preprocessing

3. Build the ILTS structure
4. Label the ILTS
   - Event = label
   - Values assigned by actions = label
   - Unchanged values are propagated
     - Split of states
     - Assumption of no-side effect for transparent states

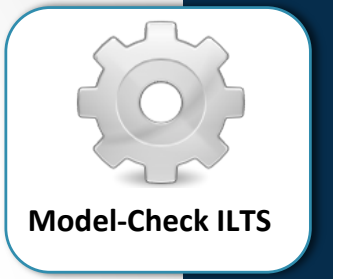
Example


t6 :
1
[]/
askPermit
t8 :
2
e4[]/
permit
t11 :
e5[]/
failure
4
s7
s8
t9 :
[permit]/
ackToCentralStation
t10 :
Timeout 3
[¬permit]/
t12 :
[]/
¬failure
t7 :
1
[]/
askPermit
s10
s9
T
t11
t13 :
[]/
alarm
4
¬permit,
failure
1
¬permit,
askPermit
¬permit,
timeout
3
S9
timeout
¬permit,
failure,
alarm
¬permit,
¬failure
2
permit
permit,
ack

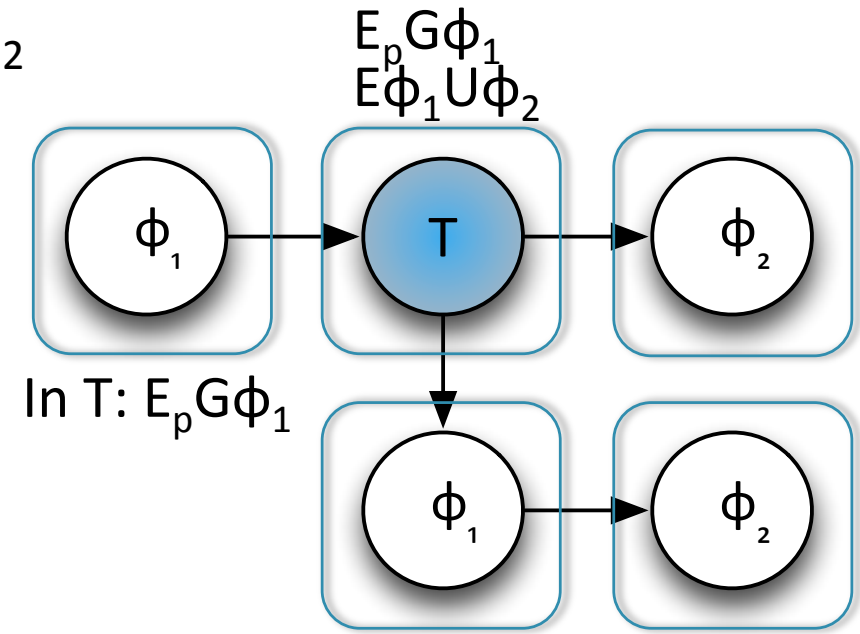
The Verification Algorithm



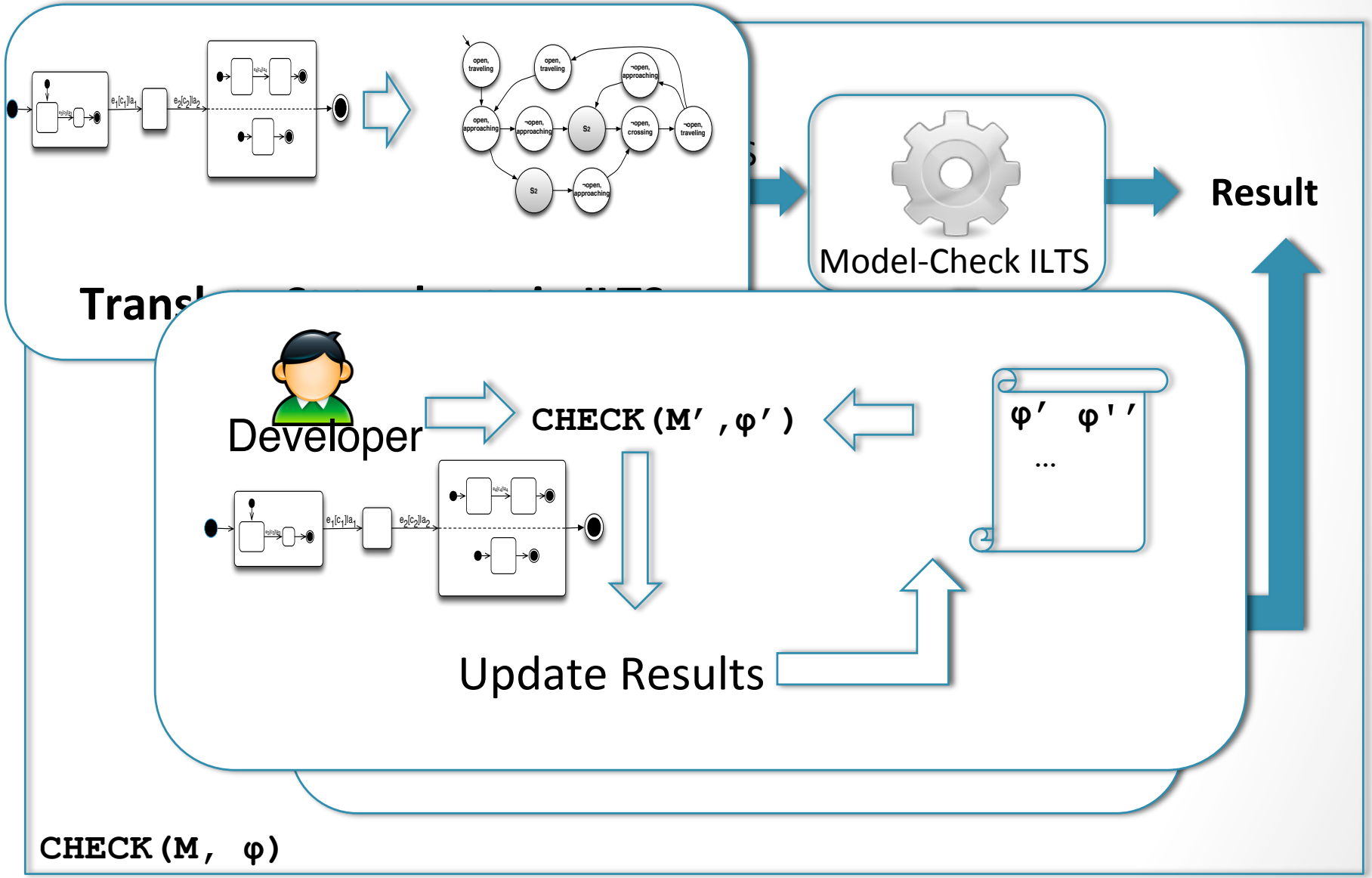
Verifying ILTS



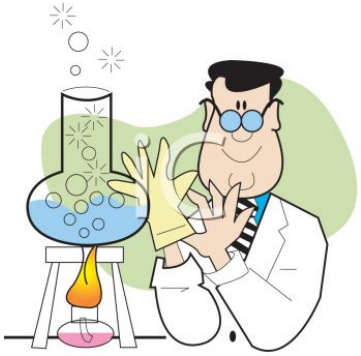
- Formulae are processed by navigating their parsing tree from the leaves to the root
- Labeling procedure as in explicit model checking
  - $E\phi_1 U\phi_2$

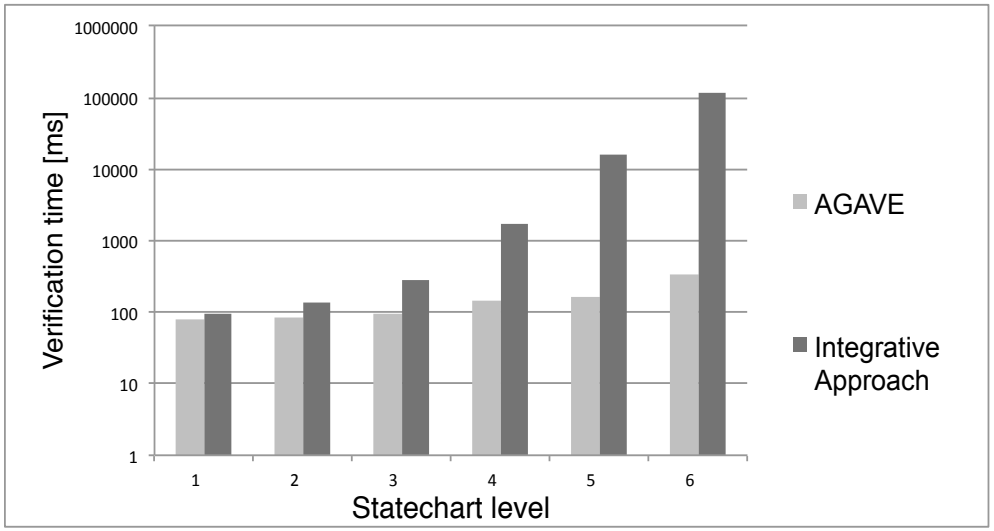


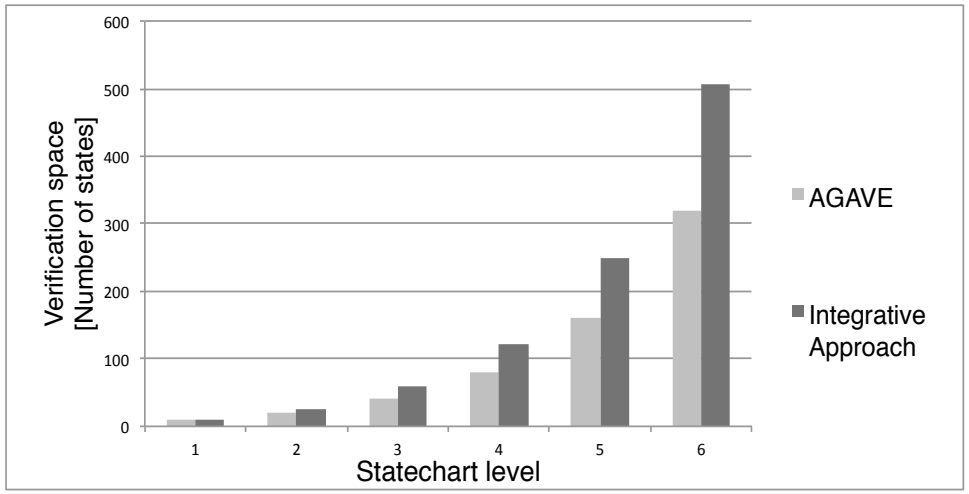
The Verification Algorithm



Experimental Results







- Prototype verification tool
  - Java standalone application
- Experiments
  - Railway crossing case study
  - Traditional approaches check the whole system at each step
  - AGaVE checks the new components against the new constraints

Conclusion

- AGaVE = AGile Verification Environment
  - Verification technique dependent on the modeling language
  - Independent methodology
- Benefits in verifying
  - Incomplete specifications
  - Analysis of different alternatives
  - Adaptive systems
  - Hierarchical specifications
    - Composite states can be considered as transparent states

On-going and Future Work

- Optimizing the current verification algorithm
  - Optimize the AND-state management
  - Efficiently remove the no-side effect assumption
- Extending the verification and the properties
  - Consider also operators AX and EX
  - Deal with metric operators
  - Add timed transitions