A Model Checking Language for Concurrent Value-Passing Systems

Radu Mateescu and Damien Thivolle

INRIA Rhône-Alpes / VASY

http://www.inrialpes.fr/vasy

Context

- Concurrent systems
  - Process algebraic languages (LOTOS, muCRL)
  - Value-passing communication
  - Interleaving semantics (LTSs)
  - Branching-time world (adequate with bisimulations)
- Explicit-state, on-the-fly verification
  - Enumeration of individual states and transitions
  - Incremental construction of the LTS
- CADP toolbox

  http://www.inrialpes.fr/vasy/cadp

Outline

- Standard μ-calculus
- The MCL language
- EVALUATOR 4.0: an implementation
- Demo (analysis of the SCSI-2 protocol)
- Conclusion and future work

Standard modal μ-calculus

- **Assembly** language for temporal logics
- **Models:** Labelled Transition Systems (LTSs)
- Very **simple grammar**:

P ::= mu X . P
  | nu X . P
  | < A > P
  | [A] P
  | X
  | not P
  | P_1 or P_2
  | P_1 and P_2
  | true
  | false

state formula

action formula

propositional variable

A ::= not A
  | A_1 or A_2
  | A_1 and A_2
  | true
  | false
  | *label*

Standard μ-calculus: modalities

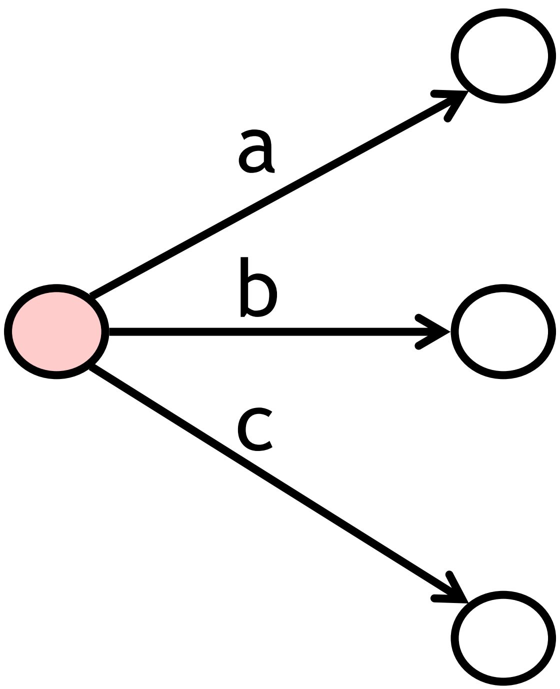
- Possibility: < A > P

  (state formula: P; action formula: A)

  < a > true

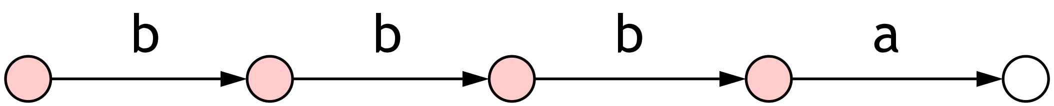
- Necessity: [A] P

  [d] false



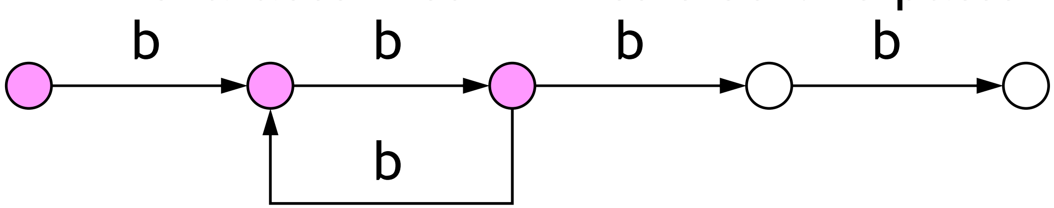
Standard μ-calculus: fixed points

- Intuition: « recursive functions » on the LTS
- Minimal fixed point: mu X . P (X)
  - ➔ characterizes finite tree-like patterns



mu X . < a > true or < not a > X

- Maximal fixed point: nu X . P (X)
  - ➔ characterizes infinite tree-like patterns (cyclic subgraphs)



nu X . < b > X

Syntactic restrictions

- Syntactic monotonicity [Kozen-83]
  - Necessary to ensure the existence of fixed points
  - In every formula mu X . P (X), every free occurrence of X in P falls in the scope of an even number of negations

    mu X . < a > X or not < b > X

- Alternation depth 1 [Emerson-Lei-86]
  - Necessary for efficient (linear-time) verification
  - In every formula mu X . P (X), every maximal subformula nu Y . P’ (Y) of P is closed

    mu X . < a > nu Y . ([b] Y and [c] X)

Extending μ-calculus

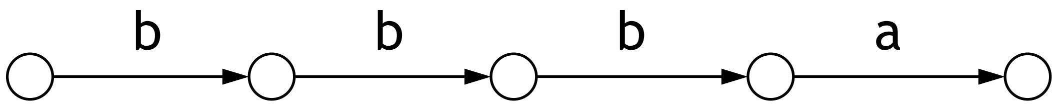
- Temporal logics (CTL, PDL, …) and μ-calculi
  - No data manipulation (basic LOTOS, ACP)
  - Too low-level operators (complex formulas)

  ➔ *extended temporal logics are needed*

- EVALUATOR 3.5
  - On-the-fly model checker included in CADP
  - Standard μ-calculus + regular operators (., |, *, +)
  - Diagnostic generation, libraries of derived operators
  - Extensively used and tested (30 case-studies)
  - *No data handling*

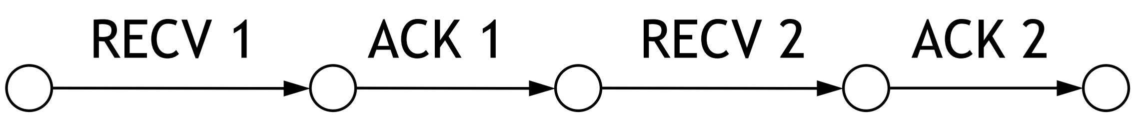
Why do we want to handle data?

- Action counting



→ *cumbersome:* <b><b><b><a>true

- Synchronization with data exchange
  - Common in process algebras
  - Parameterized LTS (PLTS)
    - label = action + data

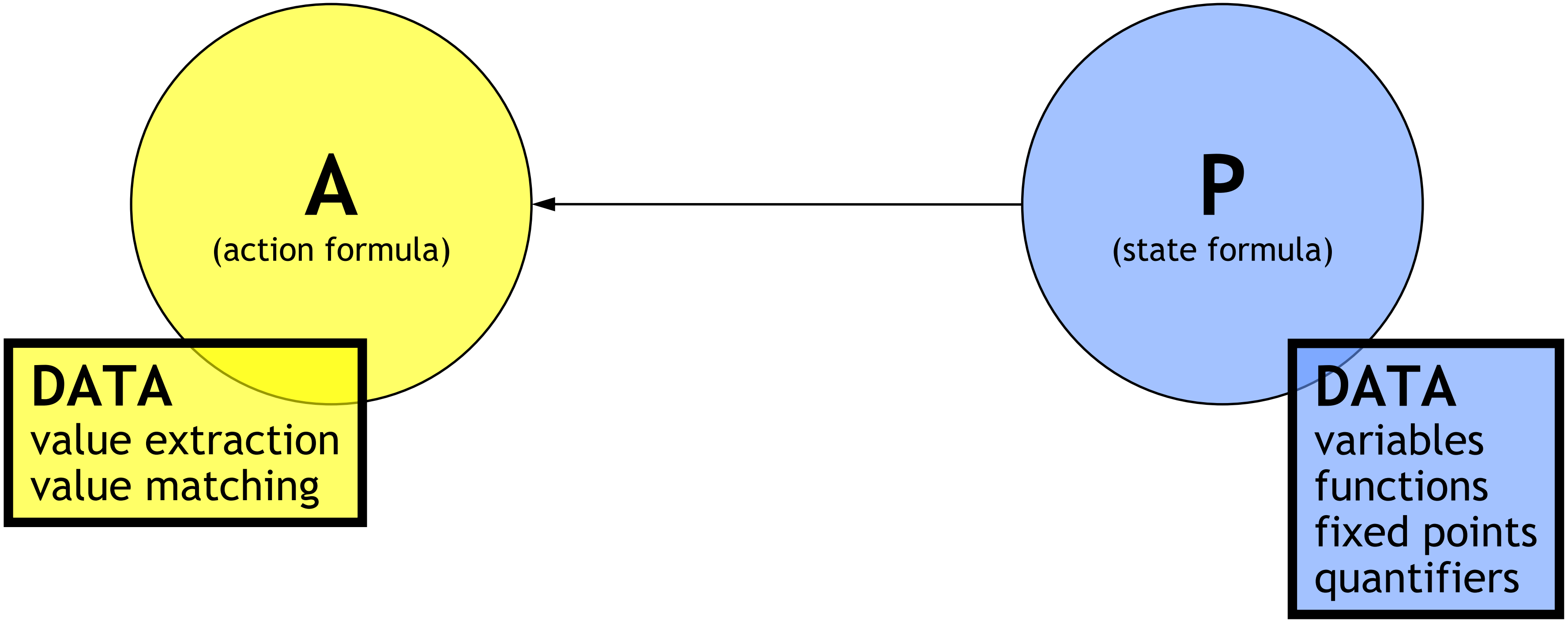


→ *temporal properties on PLTSs require data value extraction*

New Model Checking Language

- Based on EVALUATOR 3.5 input language
  - standard μ-calculus
  - regular operators
- Data-handling mechanisms
  - data extraction from PLTS labels
  - regular operators with counters
  - variable declaration
  - expressions
- Constructs inspired from programming languages

Model Checking Language (1/4)



Grammar extension

P ::= mu X $(y_1{:}T_1{:=}E_1, \ldots, y_n{:}T_n{:=}E_n)$. P
| nu X $(y_1{:}T_1{:=}E_1, \ldots, y_n{:}T_n{:=}E_n)$. P
| [A] P
| < A > P
| X $(E_1, \ldots, E_n)$
| y
| f $(E_1, \ldots, E_n)$
| let y:T:=E in P
| not P
| P_1 or P_2
| P_1 and P_2
| true
| false
| forall y:T among $\{E_1 \ldots E_2\}$ in P
| exists y:T among $\{E_1 \ldots E_2\}$ in P

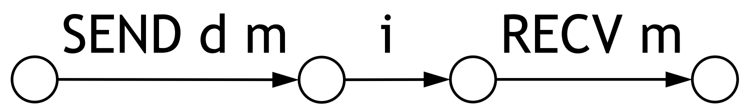
A ::= not A
| A_1 or A_2
| A_1 and A_2
| true
| false
| *label*
| { G $!v_1 \ldots !v_n$ }
| { G $?x_1{:}T_1 \ldots ?x_n{:}T_n$ }

Annotations:

- parameterised fixed point (mu X (…) . P, nu X (…) . P)
- parameterised propositional variable (X (E1, …, En))
- data variable (y)
- function call (f (E1, …, En))
- variable declaration (let y:T:=E in P)
- quantifiers (forall …, exists …)
- value matching ({ G !v1 … !vn })
- value extraction ({ G ?x1:T1 … ?xn:Tn })

Parameterised action formulas

- Parameterised LTS:



- (basic) syntax:

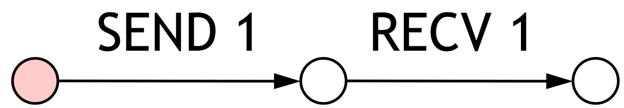
$\{G\ !E_1\ \ldots\ !E_n\}$ *value matching*

gate expressions data variables

$\{G\ ?x_1{:}T_1\ \ldots\ ?x_n{:}T_n\}$ *value extraction*

Parameterised modalities

- Possibility:




```
< {SEND ?msg:Nat} > < {RECV !msg} > true
```

value extraction and propagation

- Necessity:


```
[ {RECV ?msg:Nat} ] (msg < 6)
```

value extraction and propagation

Parameterised fixed points

- (basic) syntax:

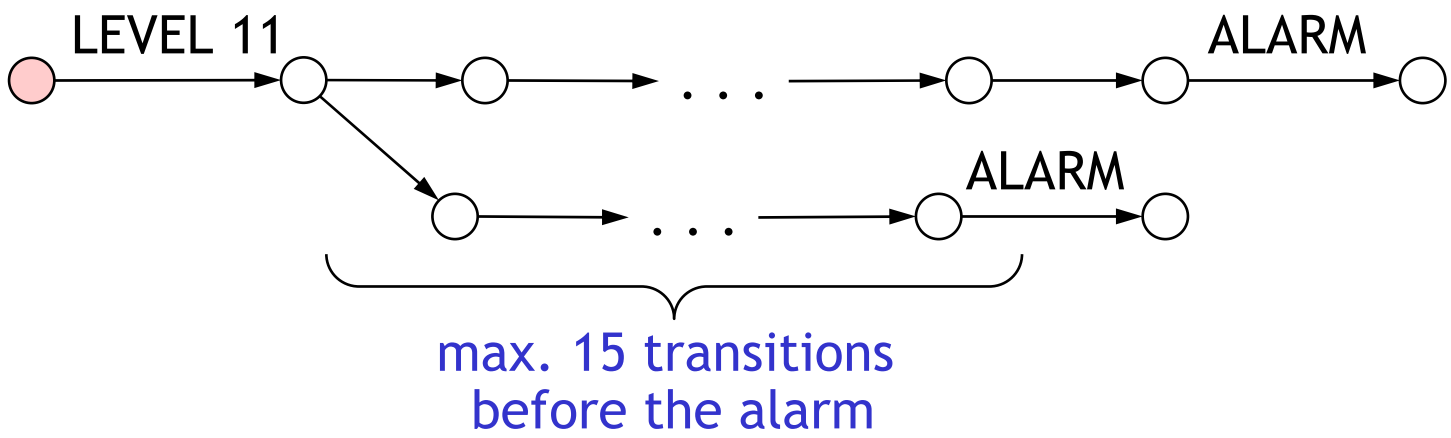
  mu X (y:T := E) . P

  parameter (y:T) — initial value (E) — formula body (P)

  - P contains « calls » X (E')
  - Allows to perform computations and store intermediate results while exploring the PLTS

Example

- Counting of actions (e.g., clock ticks):




```
[ {LEVEL ?l:Nat where l > 10} ]
    nu X (c:Nat := 15) .
        [ not ALARM ] (c > 0 and X (c - 1))
```

Quantifiers

- Existential quantifier:

  exists x:T among { E_1 ... E_2 } . P

  limits of the subdomain of T

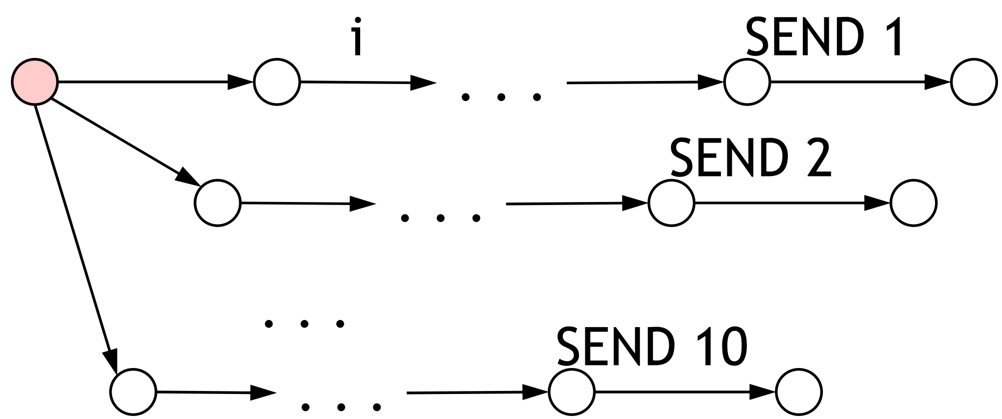
- Universal quantifier:

  forall x:T among { E_1 ... E_2 } . P

➔ *shorthands for large disjunctions and conjunctions*

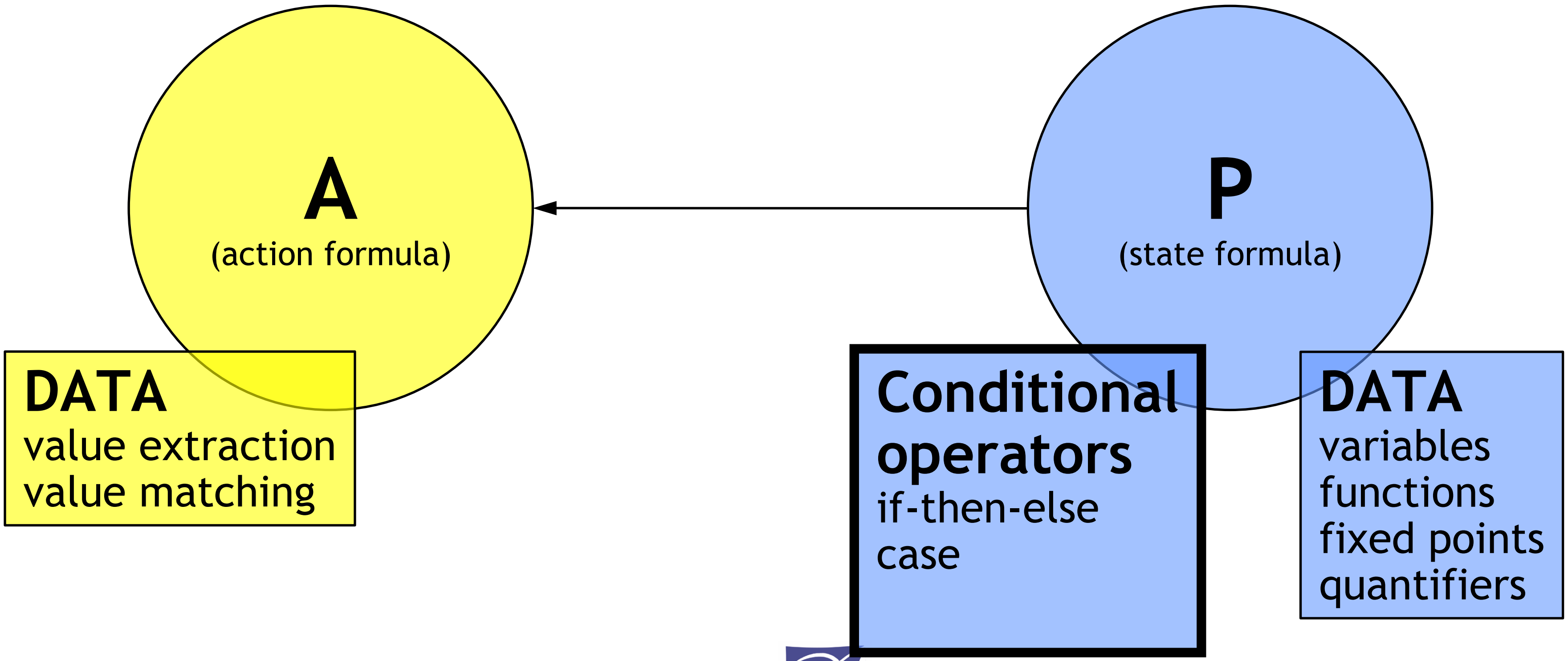
Example

- Broadcast of messages:



forall msg:Nat among { 1 … 10 } .
mu X . (< {SEND !msg} > true or < true > X)

Model Checking Language (2/4)



Conditional operators

- Branching operator:

$$\text{if } P_1 \text{ then } P_1' \text{ elsif } P_2 \text{ then } P_2' \ldots \text{ else } P_n' \text{ end if}$$

```
if P1 then P1'
   elsif P2 then P2'
   ...
   else Pn'
end if
```

propositionally closed subformulas in the branch conditions (to ensure syntactic monotonicity)

mandatory clause (to avoid exceptions)

- Selection operator:

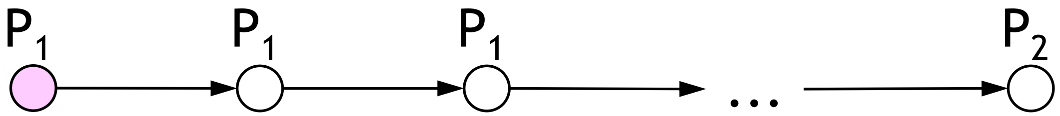
```
case E is
      M1 -> P1
|     ...
|     any -> Pn
end case
```

pattern

mandatory exhaustiveness (to avoid exceptions)

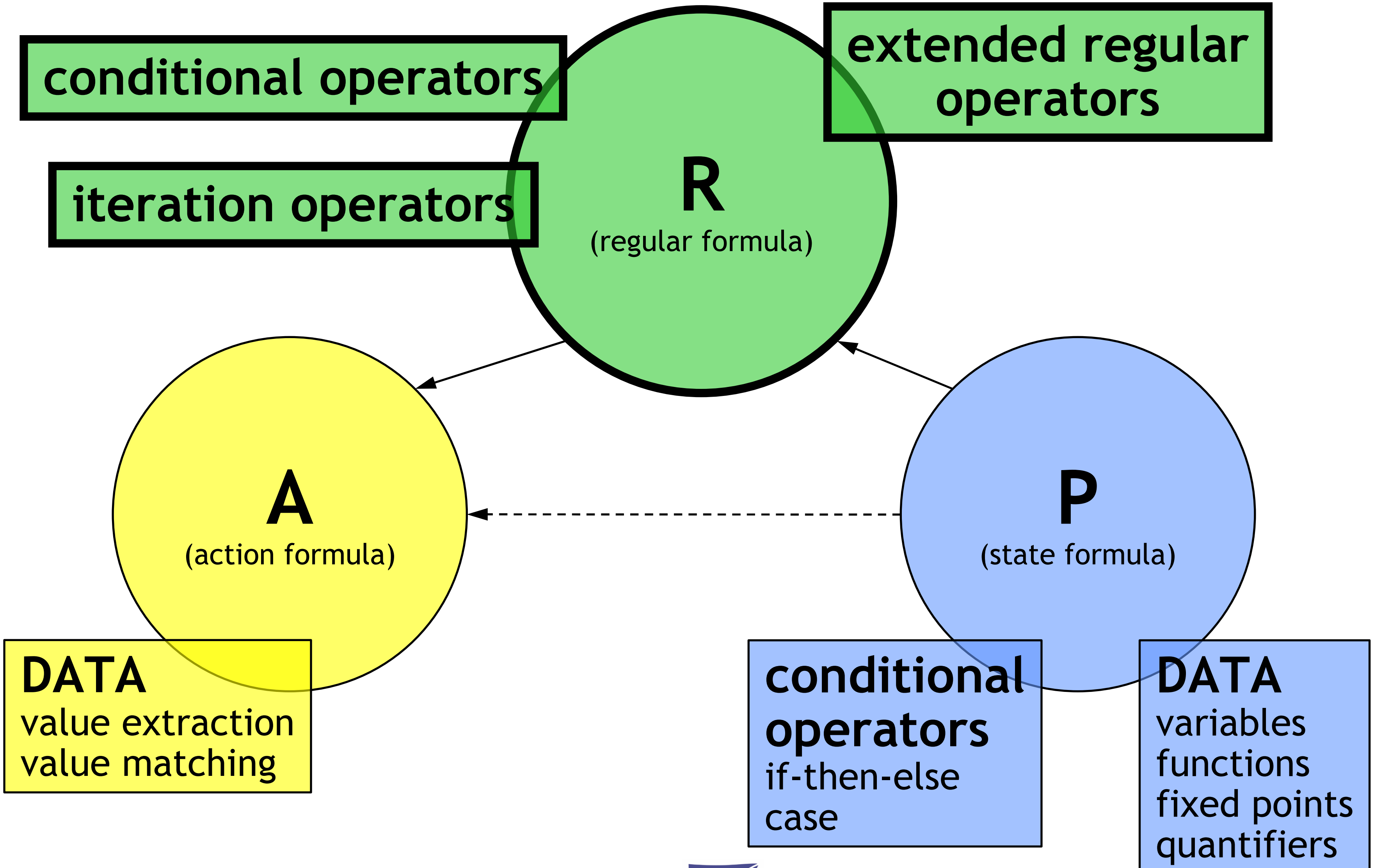
Example

- Operator E (P_1 U P_2) of CTL:



E (P_1 U P_2) =
mu X . if P_1 then <true> X else P_2 end if

Model Checking Language (3/4)



Grammar extension

P ::= ...
| ~~[A] P~~
| ~~< A > P~~
| [R] P
| < R > P
| ...

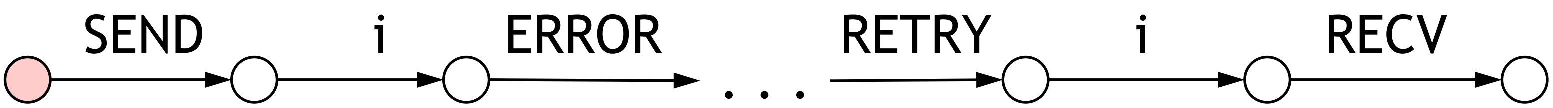
regular formulas implemented by EVALUATOR 3.5

R ::= nil	*empty sequence*
| A	*one-step sequence*
| R_1 . R_2	*concatenation*
| R_1 | R_2	*choice*
| R *	*0 or more occurrences*
| R +	*1 or more occurrences*
| R ?	*choice (0 or 1 occurrences)*
| R { n }	*exactly n occurrences*
| R { m ... n }	*between m and n occurrences*
| R { m ... }	*at least m occurrences*

| if P_1 then R_1 ... else R_2 end if
| case E is M_1 -> R_1 ... end case
| while P do R end while
| repeat R until P end repeat

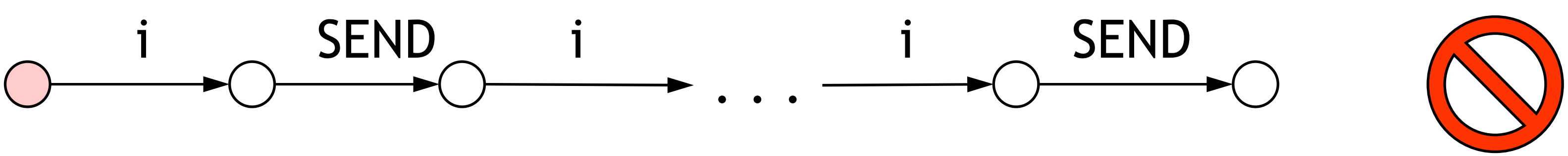
Basic regular formulas
(Evaluator 3.5)

- Liveness properties (existence of sequences):



< SEND . (i*. ERROR . i*. RETRY)*. i*. RECV > true

- Safety properties (interdiction of sequences):



[true*. SEND . (not RECV)*. SEND] false

Conditional operators

- Branching operator:

  if P_1 then R_1
  elsif P_2 then R_2
  ... else R_n
  end if

- Selection operator:

  case E is
  M_1 -> R_1 | ... | M_n -> R_n
  end case

propositionally closed subformulas in the branch conditions (to ensure syntactic monotonicity)

optional exhaustiveness (missing branches equivalent to nil)

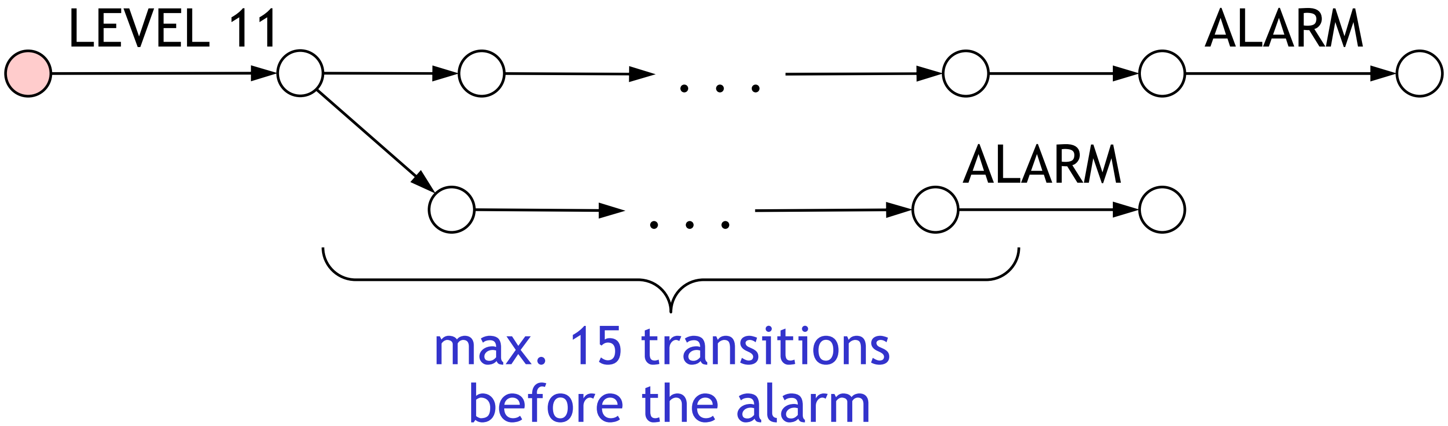
Counting operators

R { E }	*repetition E times*
R { E_1 ... }	*repetition at least E_1 times*
R { E_1 ... E_2 }	*repetition between E_1 and E_2 times*

- Some identities:

nil = false *	R + = R . R*
R * = R { 0 ... }	R ? = R { 0 ... 1 }
R + = R { 1 ... }	R { E } = R { E ... E }

Example: action counting (revisited)



- Formulation using counting operators:

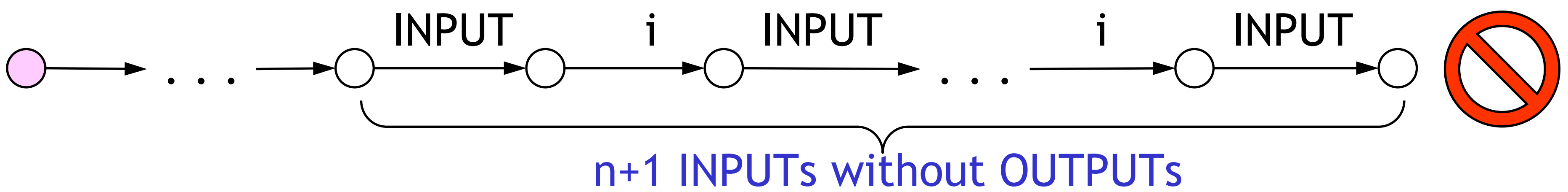
  [{LEVEL ?l:Nat where l > 10} .

  (not ALARM) { 16 }] false

Example: safety of a n-place buffer

- Formulation using extended regular operators:

```
[ true* . ((not OUTPUT)* . INPUT) { n + 1 } ] false
```



- Formulation using parameterized fixed points:

```
nu X . (nu Y (c:Nat:=0) . (
              [not OUTPUT] Y (c) and
              if c = n+1 then [INPUT] false
                   else [INPUT] Y (c+1)
              end if)
   and [ true ] X)
```

Iteration operators

- Cycle with initial test:

  while P do

  R

  end while

- Cycle with final test:

  repeat

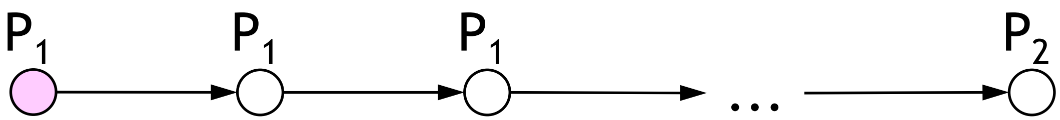
  R

  until P end repeat

propositionally closed subformulas in the cycle conditions (to ensure syntactic monotonicity)

Example

- Operator E (P_1 U P_2) of CTL (revisited):



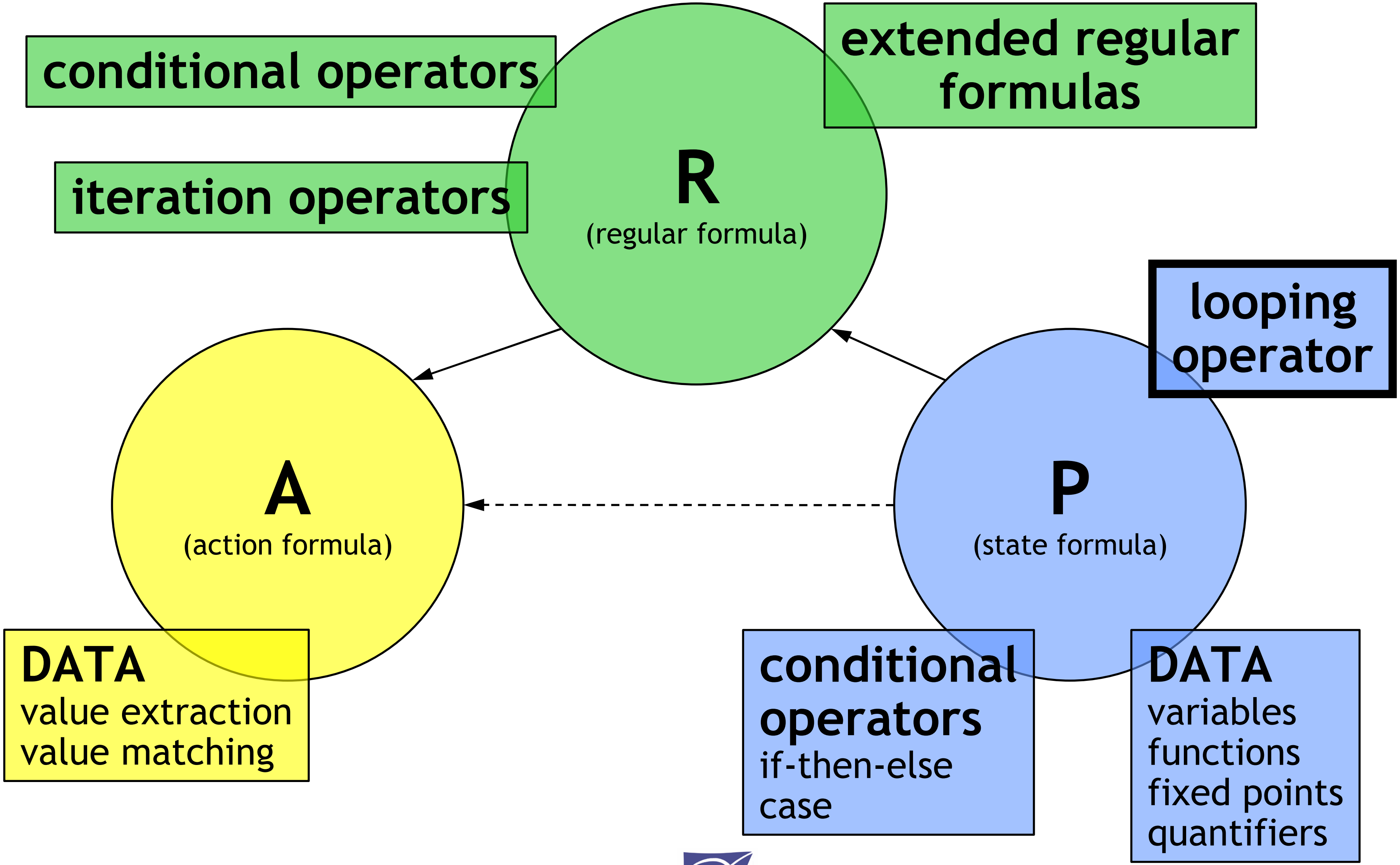
E (P_1 U P_2) =

< while P_1 and not P_2 do

true

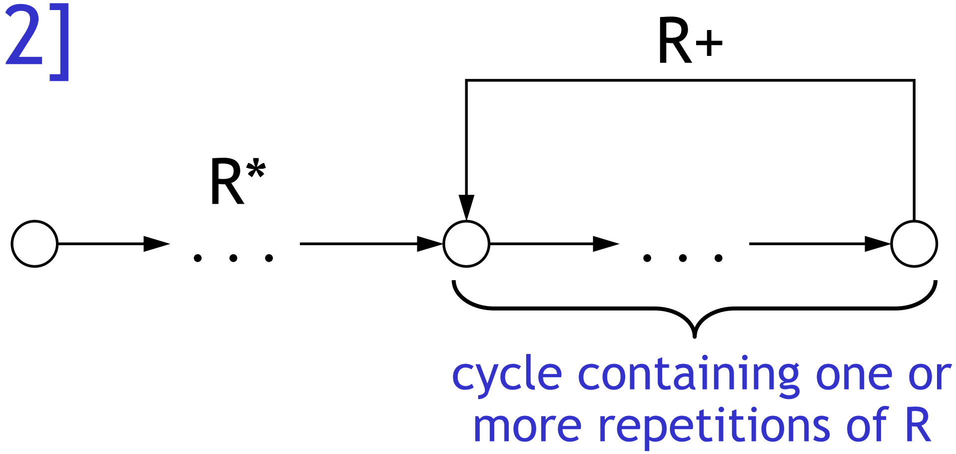
end while > P_2

Model Checking Language (4/4)



Looping operator (from PDL-delta)

- *Δ* R operator added to PDL to specify infinite behaviours [Streett-82]
- MCL syntax: < R > @



- Examples:
  - process overtaking

    [REQ_0] < (not GET_0)* . REQ_1 . (not GET_0)* . GET_1 > @
  - Büchi acceptance condition

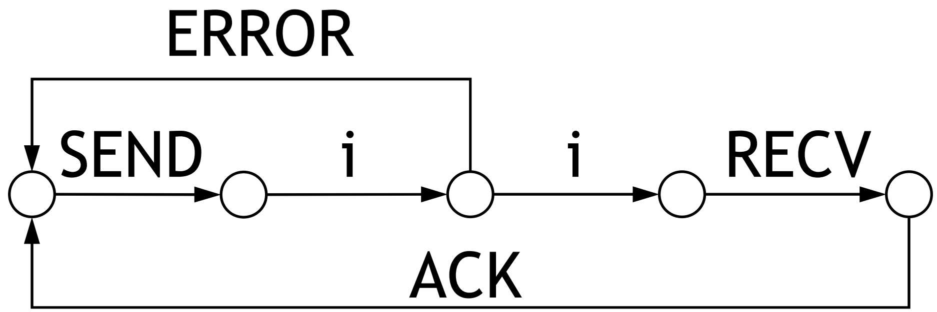
    < true* . if $P_{accepting}$ then true end if > @

    ➔ *allows to encode LTL model checking*

On-the-fly verification

- Principle
  - Translate the MCL formula into a (parameterised) HMLR equation system
  - Translate the verification of a HMLR system on a PLTS into a PBES resolution
  - Expand the PBES into a plain BES on-the-fly
  - Solve the BES locally (Caesar_Solve library of CADP)
- Optimizations
  - Operators of CTL, ACTL, PDL-delta
    - Compiled into disjunctive/conjunctive BESs
    - Memory-efficient algorithms A3 and A4 [Mateescu-03]
- Diagnostic generation

Example



- Every SEND is followed by a RECV after 2 steps:

```
[ true* . SEND ] < true { 2 } . RECV > true =
nu X . ( [ SEND ] mu Y (c:Nat := 2) .
                          if c = 0 then < RECV > true
                               else < true > Y (c - 1)
                          end if
       and
       [ true ] X )
```

Translation into HMLR

```
nu X . [ SEND ]

        and [ true ] X
```

→

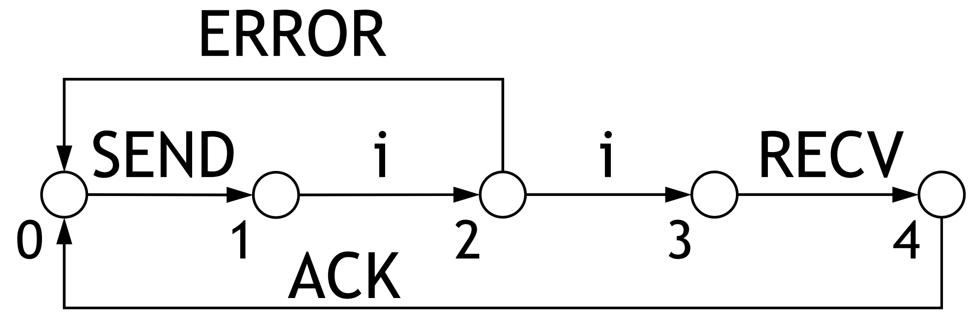
```
mu Y (c:Nat := 2) .
  if c = 0 then < RECV > true
          else < true > Y (c - 1)
  end if
```

```
{ X =_nu
      [ SEND ] Y (2)
      and
      [ true ] X
}
```

→

```
{ Y (c:Nat) =_mu
  if c = 0 then < RECV > true
          else < true > Y (c - 1)
  end if
}
```

Translation into BESs + resolution



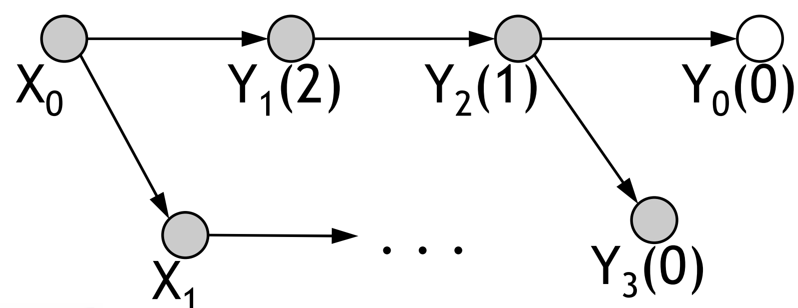

```
{ X =nu
      [ SEND ] Y (2)
      and
      [ true ] X
}
```

```
{ Y (c:Nat) =mu
    if c = 0 then < RECV > true
         else < true > Y (c - 1)
    end if
}
```

- Encoding scheme:

  X_s = "s |= X"

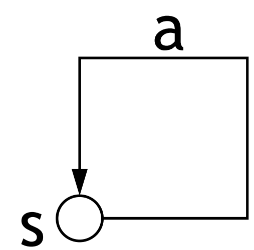
  Y_s (c) = "s |= Y (c)"



Divergence

- In presence of data parameters of infinite types, termination of model checking is not guaranteed anymore
- (pathological) property:

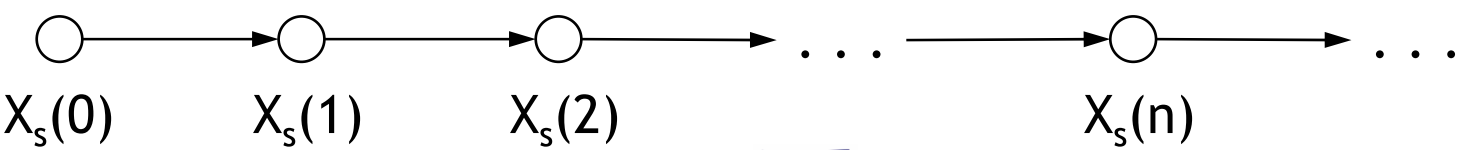
LTS:



mu X (n:Nat := 0) . < a > X (n + 1)

BES : $\{ X_s (n:Nat) =_{mu} \text{OR}_{s \to_a s'} X_{s'} (n + 1) \} =$

$\{ X_s (n:Nat) =_{mu} X_s (n + 1) \}$



Linear-time model checking
(looping operator)

- Translation in mu-calculus of alternation depth 2 [Emerson-Lei-86]:

  `< R > @ = nu X . < R > X`

  if R contains *-operators, the formula is of alternation depth 2

- But still checkable in linear-time:
  - Mark LTS states potentially satisfying X
  - Computation of SCCs containing marked states
    - Can serve for LTL model checking
    - Allows linear-time handling of repeated invocations
  - $A4_{cyc}$ algorithm for local BES resolution

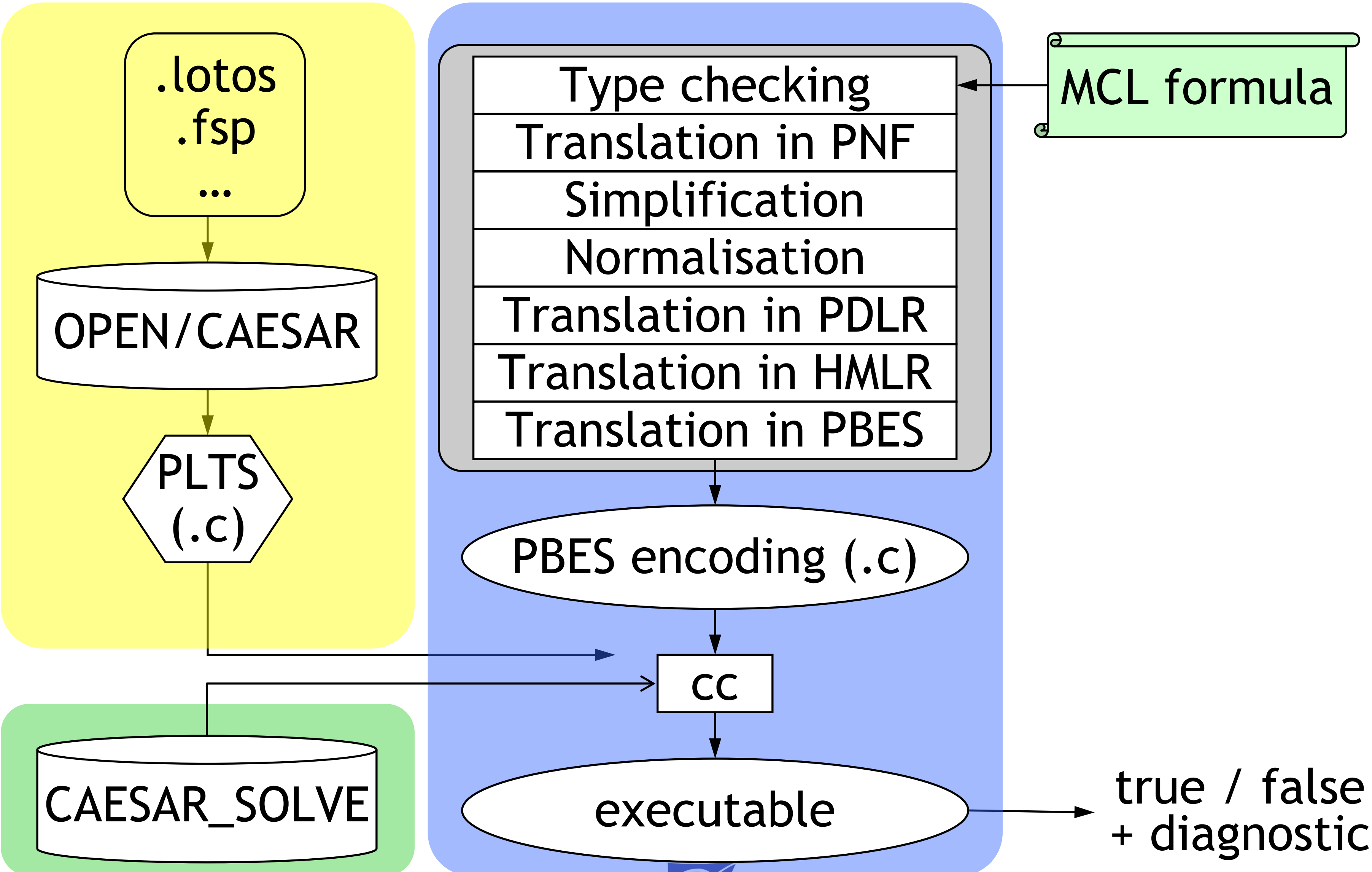
Model checking complexity

- Formulas without data:
  - Linear-time w.r.t. the PLTS and formula size
  - Problem rephrased in terms of a BES
  - On-the-fly resolution
    - Linear algorithms of the Caesar_Solve library [Mateescu-06]
    - New linear algorithm $A4_{cyc}$ for the looping operator
- Formulas with data:
  - Additional complexity depending on the number of actual parameter values computed during verification
  - Risk of divergence (same as for recursive functions in programming languages; however cycles are allowed)
  - All counting regular operators are convergent

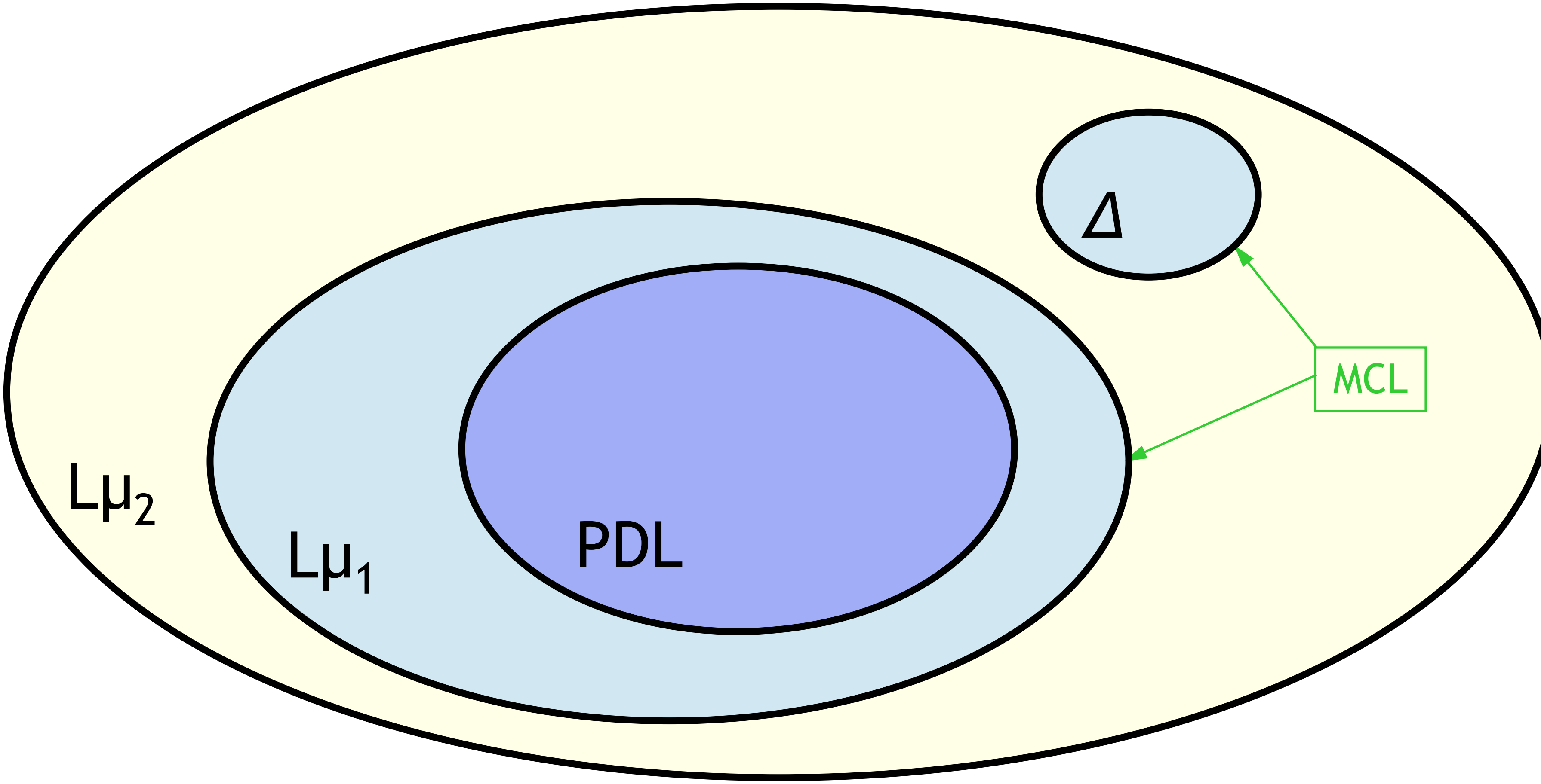
EVALUATOR 4.0

- Implemented in CADP
  - Benefits from OPEN/CAESAR environment: interface with languages (LOTOS, FSP...) and formats (AUT, EXP...) supported by CADP
- Front-end
  - Evaluation of an MCL formula on a PLTS is translated by 6 successive phases into a parameterized boolean equation system (PBES)
- Back-end
  - PBES is expanded into a BES and solved on-the-fly (with the Caesar_Solve library of CADP)
- Stats
  - 50,000 lines of code (Syntax, LOTOS NT, C)

EVALUATOR 4.0

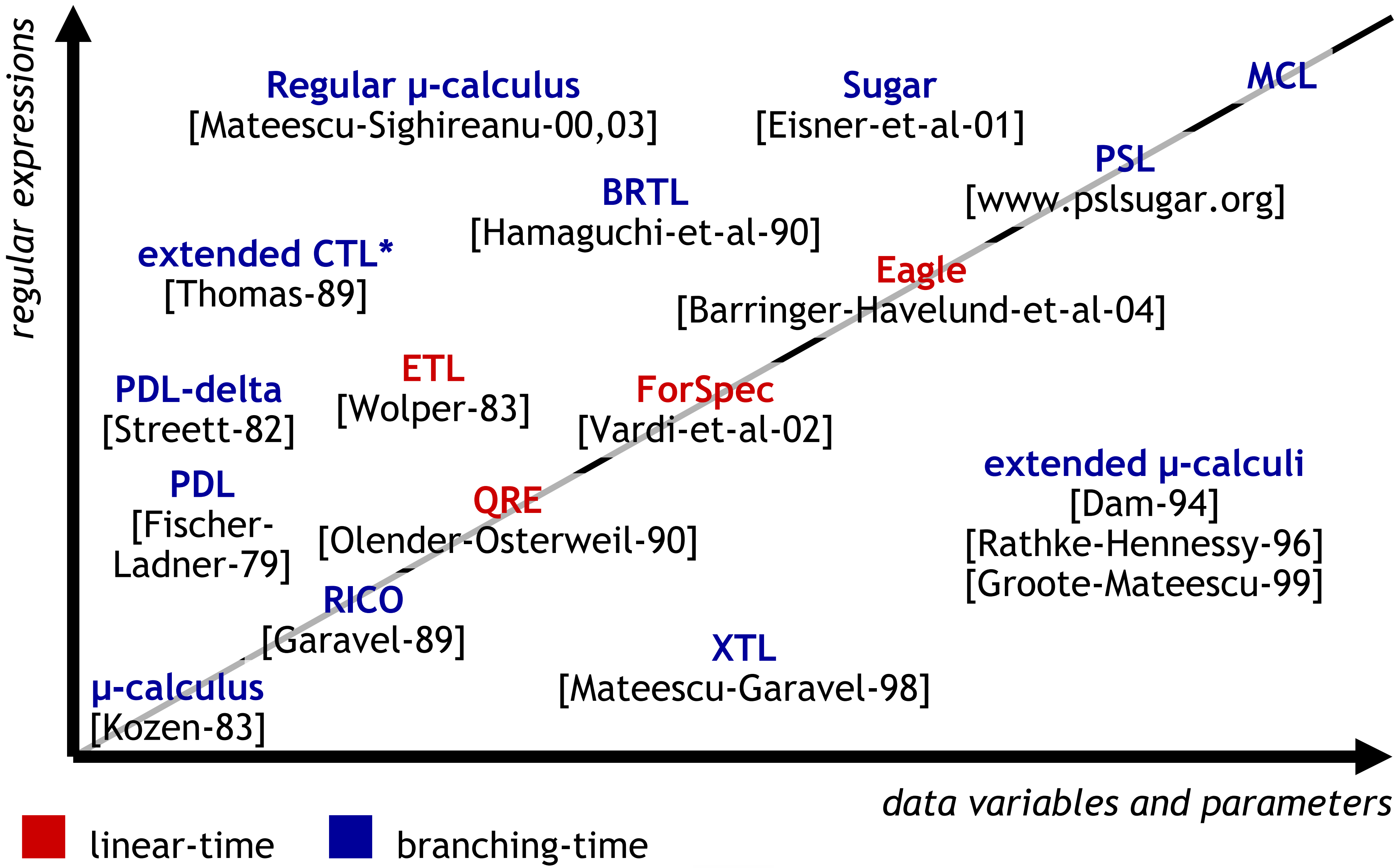


MCL expressiveness (dataless part)



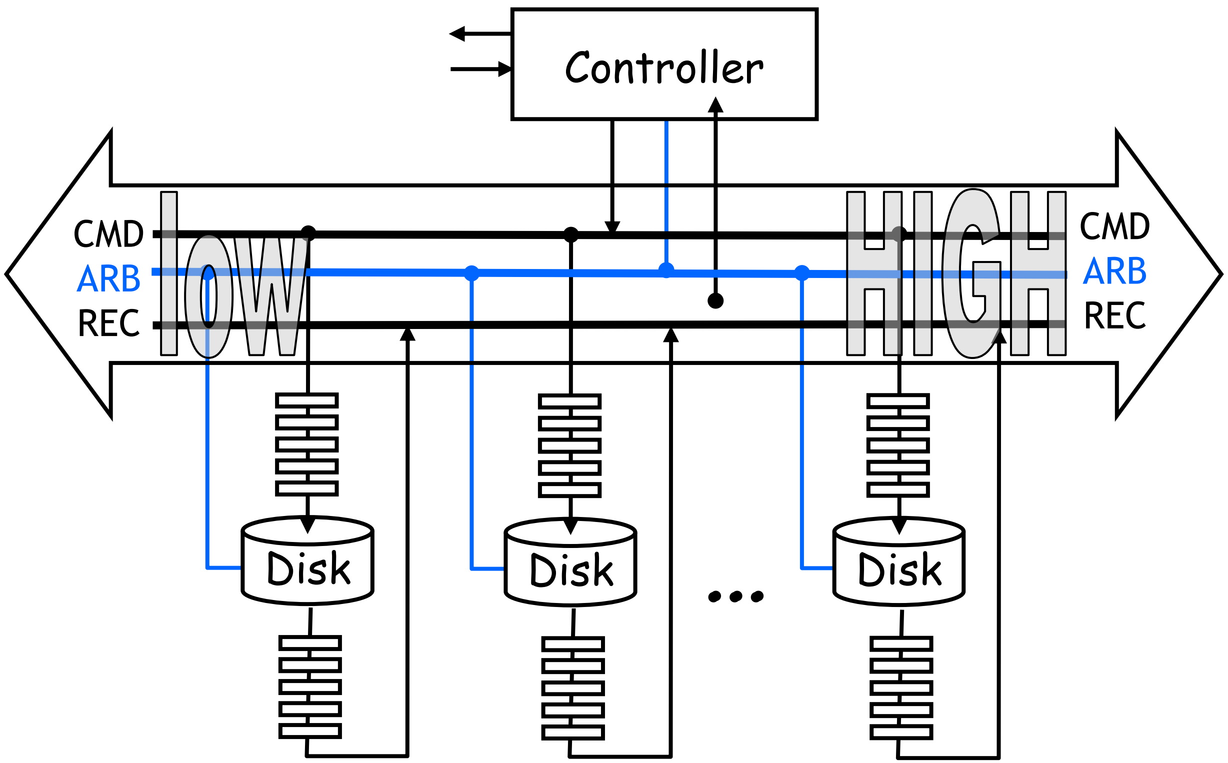
CTL* $\subseteq$ PDL $\cup$ Δ $\subseteq$ MCL
[Wolper-82]

Related work (TL extensions)



Demo
(SCSI-2 bus arbitration protocol)

- Prioritized arbitration mechanism, based on static IDs on bus (devices numbered from 0 to n – 1)
- Fairness problem (starvation of low-priority disks)



Starvation property

"Every time a disk i with priority lower than the controller nc receives a command, its access to the bus can be continuously preempted by any other disk j with higher priority"

```
[ true*. {cmd ?i:Nat where i < nc} ]
        forall j:Nat among { i + 1 ... n − 1 } .
                (j <> nc) implies
                < (not {rec !i})*. {cmd !j} .
                   (not {rec !i})*. {rec !j} > @
```

Safety property

"The difference between the number of commands received and reconnections sent by a disk i varies between 0 and 8 (the size of the buffers associated to disks)"

```
forall i:Nat among { 0 … n – 1 } .
    nu Y (c:Nat:=0) . (
        [ {cmd !i} ] ((c < 8) and Y (c + 1))
        and
        [ {rec !i} ] ((c > 0) and Y (c − 1))
        and
        [ not ({cmd !i} or {rec !i}) ] Y (c)
    )
```

Conclusion and future work

- Already available:
  - Formal definition of the MCL language
    - Syntax and semantics
    - Translation into more primitive forms
  - EVALUATOR 4.0
    - Currently under testing, integrated in the next CADP release
    - Used in case studies with BULL (multiprocessor architectures) and EC-MOAN European project (bioinformatics)
    - Linear time model checking (dataless part), stores only states
- Ongoing and future:
  - Packaging EVALUATOR 4.0 (testing and demos)
  - Allow types and functions definition
  - Interface with LOTOS-defined data types