CADP Tutorial

**Hubert Garavel, Frédéric Lang,
Radu Mateescu, Gwen Salaün,
Wendelin Serwe**

Inria Grenoble – Rhône-Alpes

I. INTRODUCTION

CADP

(**C**onstruction and **A**nalysis of **D**istributed **P**rocesses)

- A modular toolbox for asynchronous systems
- At the crossroads between:
  - concurrency theory
  - formal methods
  - computer-aided verification
  - compiler construction
- A long-run effort:
  - development of CADP started in the mid 80s
  - initially: only 2 tools (CÆSAR and ALDEBARAN)
  - last stable version: CADP 2006
  - today: nearly 50 tools in CADP 2010 (close to stable)

CADP: main features

- Specification languages
  - Formal semantics
  - Based on process calculi
  - User-friendly syntax
- Verification paradigms
  - Model checking (modal μ-calculus)
  - Equivalence checking (bisimulations)
  - Visual checking (graph drawing)
- Verification techniques
  - Reachability analysis
  - On-the-fly verification
  - Compositional verification
  - Distributed verification
  - Static analysis
- Other features
  - Step-by-step simulation
  - Rapid prototyping
  - Test-case generation
  - Performance evaluation

CADP w.r.t. other model checkers

- Parallel programs (rather than sequential programs)
- Message passing (rather than shared memory)
- Languages with a formal semantics (process calculi)
- Dynamic data structures (records, lists, trees…)
- Explicit-state (rather than symbolic)
- Action-based (rather than state-based)
- Branching-time logic (rather than linear-time logic)

Application domains

- Not restricted to a particular application domain
- Case studies cover the following domains:

  avionics, bioinformatics, business processes, cognitive systems, communication protocols, component-based systems, constraint programming, control systems, coordination architectures, critical infrastructures, cryptography, database protocols, distributed algorithms, distributed systems, e-commerce, e-democracy, embedded software, grid services, hardware design, hardware/software co-design, healthcare, human-computer interaction, industrial manufacturing systems, middleware, mobile agents, model-driven engineering, networks, object-oriented languages, performance evaluation, planning, radiotherapy equipments, real-time systems, security, sensor networks, service-oriented computing, software adaptation, software architectures, stochastic systems, systems on chip, telephony, transport safety, Web services

list of case studies: http://cadp.inria.fr/case-studies

Plan

I. Introduction
II. Architecture and verification technology
III. Modeling languages (LNT tutorial)
IV. From languages to models
V. Functional verification
VI. Performance evaluation
VII. Script Verification Language (SVL tutorial)
VIII. Conclusion

RUNNING EXAMPLE: MCS QUEUE LOCK

MCS queue lock

- mutual exclusion protocol for shared memory multiprocessor architectures with coherent caches
- guarantees FIFO ordering, uses “local spinning”
- original pseudo-code [Mellor-Crummey-Scott-91]

shared variable (atomic operations)

locally accessible variable in shared memory

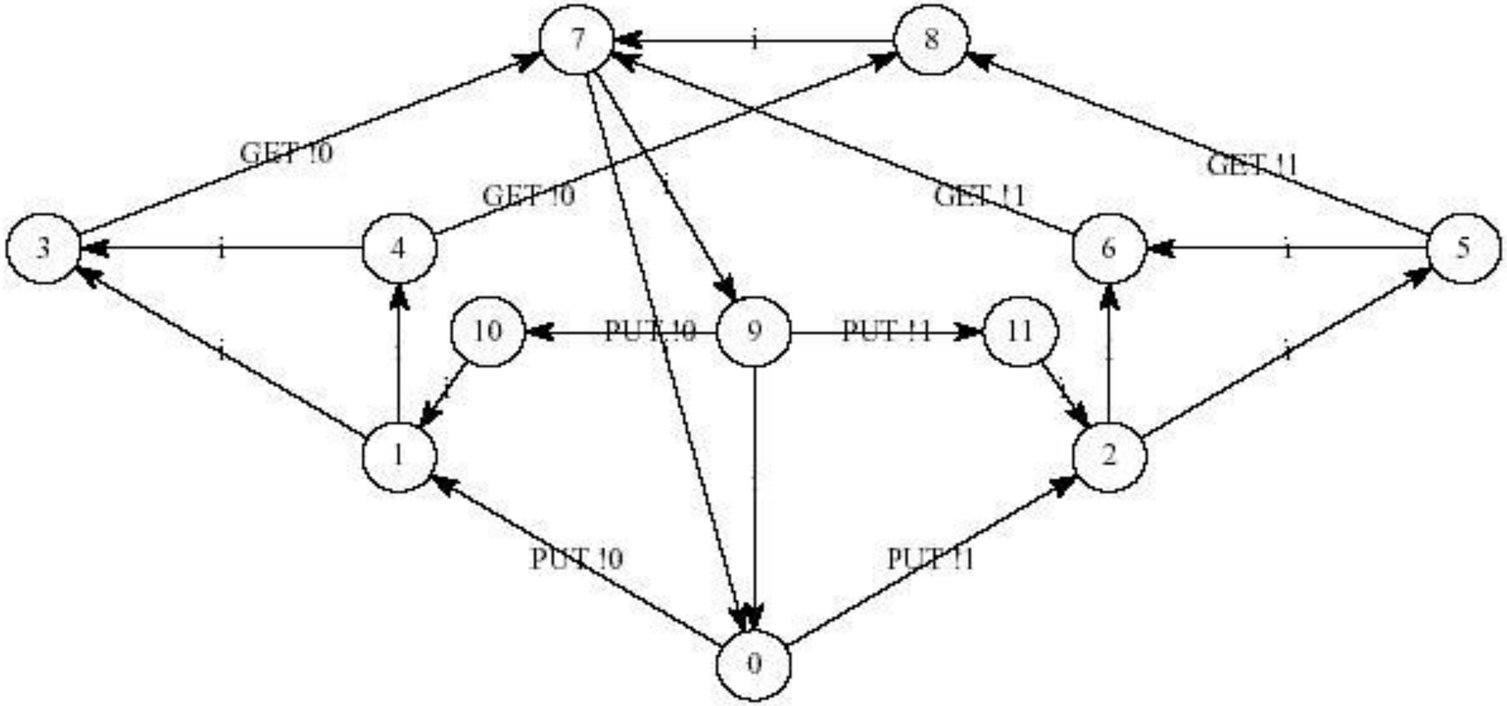
```
type qnode = record
  next      : ^qnode
  locked    : Boolean
type lock = ^qnode

proc acquire_lock (L : ^lock, I : ^qnode)
  I->next := nil
  predecessor : ^qnode := fetch_and_store (L, I)
  if predecessor != nil
    I->locked := true
    predecessor->next := I
    repeat while I->locked  // spin

proc release_lock (L : ^lock, I : ^qnode)
  if I->next = nil  // no known successor
    if compare_and_swap (L, I, nil)
      // true if and only if swapped
      return
    repeat while I->next = nil  // spin
  I->next->locked := false
```

II. ARCHITECTURE AND VERIFICATION TECHNOLOGY

II.1 LTS (*LABELED TRANSITION SYSTEM*)

Labeled Transition Systems

- State-transition graph
- no information attached to states (except the initial state)
- information ("labels" or "actions") attached to transitions



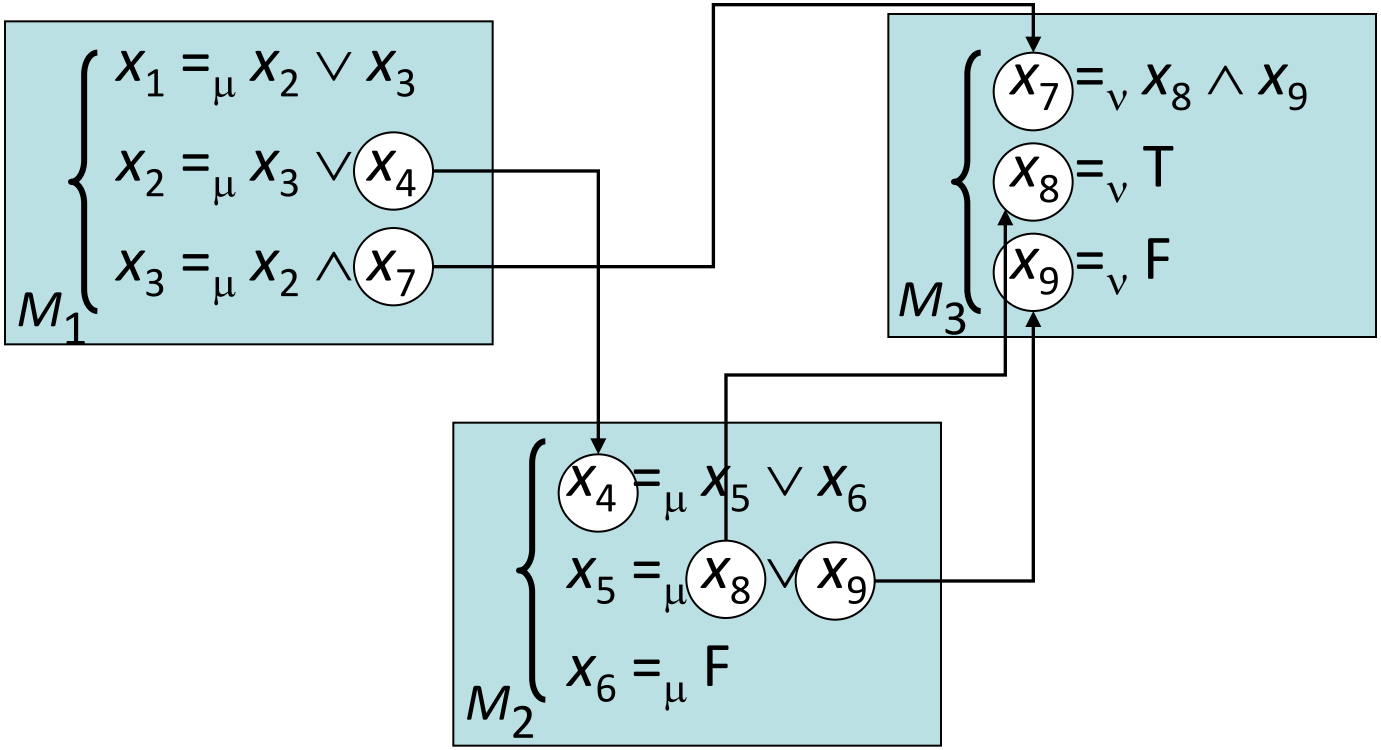
Two kinds of LTS

- **Explicit LTS** (*enumerative, global*)
  - comprehensive sets of states, transitions, labels
  - BCG: a file format for storing large LTSs
  - a set of tools for handling BCG files
  - CADP 2010: BCG limits extended from 2^{29} to 2^{44}
- **Implicit LTS** (*on-the-fly, local*)
  - defined by initial state and transition function
  - Open/Cæsar: a language-independent API
  - many languages connected to Open/Cæsar
  - many tools developed on top of Open/Cæsar

II.2 BES (*BOOLEAN EQUATION SYSTEM*)

Boolean Equation Systems

- least (μ) and greatest (ν) fixed point equations
- DAG (directed acyclic graph) of equation blocks (no cycles – alternation-free)



$$M_1 \begin{cases} x_1 =_\mu x_2 \vee x_3 \\ x_2 =_\mu x_3 \vee x_4 \\ x_3 =_\mu x_2 \wedge x_7 \end{cases}$$

$$M_3 \begin{cases} x_7 =_\nu x_8 \wedge x_9 \\ x_8 =_\nu T \\ x_9 =_\nu F \end{cases}$$

$$M_2 \begin{cases} x_4 =_\mu x_5 \vee x_6 \\ x_5 =_\mu x_8 \vee x_9 \\ x_6 =_\mu F \end{cases}$$

Support for BES

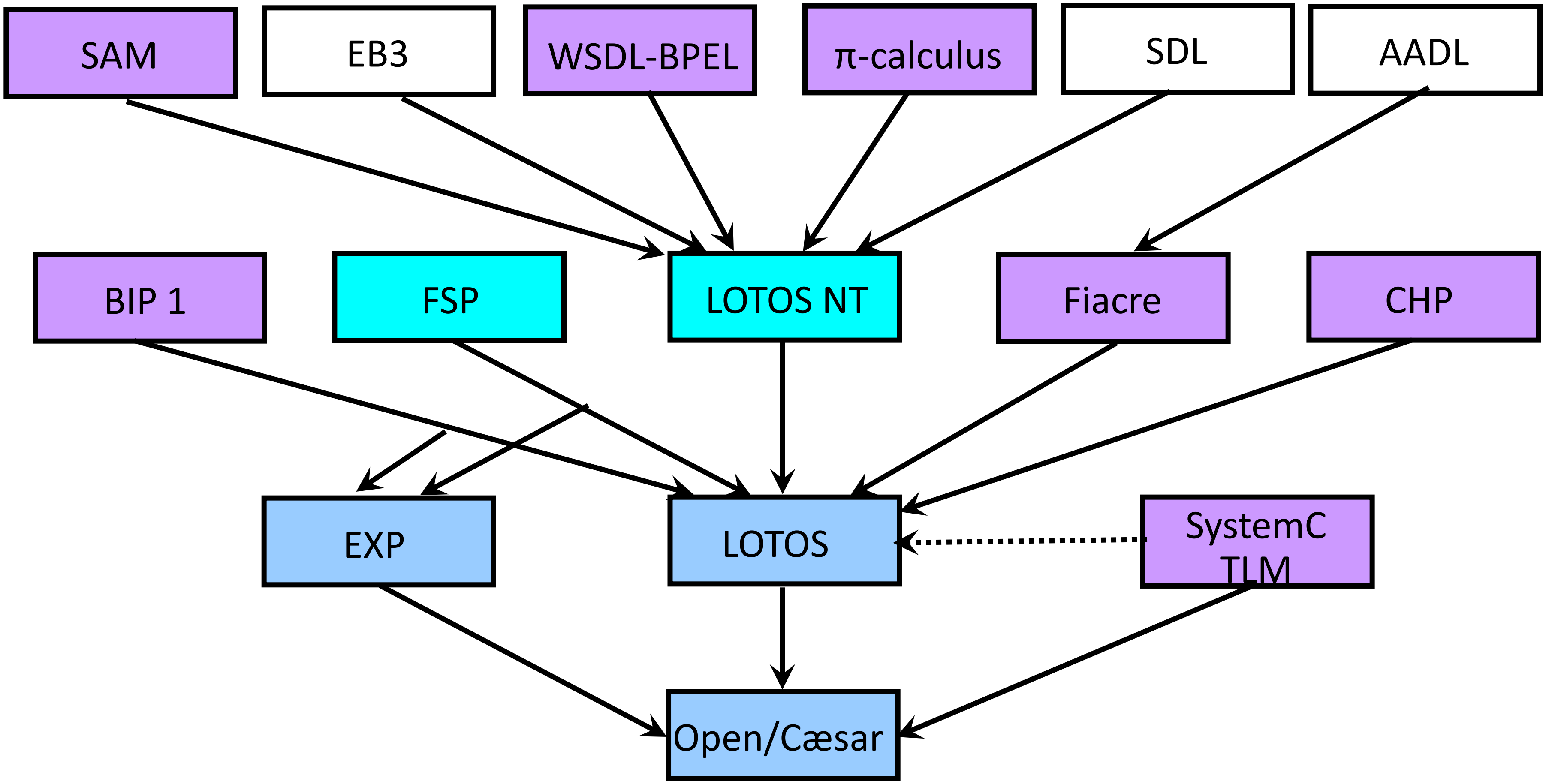
- BES can be given:
  - explicitly (stored in a file)
  - or implicitly (generated on the fly)
- CÆSAR_SOLVE: a solver for implicit BES
  - works on the fly: explores while solving
  - translates dynamically BES into Boolean graphs
  - implements 9 resolution algorithms A0-A8 (general vs specialized)
  - generates diagnostics (witnesses or counterexamples)
  - fully documented API
- BES_SOLVE: a solver for explicit BES

III. MODELING LANGUAGES (LNT TUTORIAL)

Modeling languages

- formal languages for modeling and specification
- CADP 2006: LOTOS only
- CADP 2010: numerous languages
  - wide spectrum from abstract calculi to automata
  - translations to benefit from existing optimized tools
- here: focus on LNT

Languages supported by CADP



Support for LOTOS

- LOTOS (ISO standard 8807):
  - Types/functions: algebraic data types
  - Processes: process algebra based on CCS and CSP
- Tools: CÆSAR, CÆSAR.ADT, CÆSAR.OPEN, etc.
- Features:
  - Optimal implementation of natural numbers
  - Bounded hash tables to canonically store structured types (tuples, unions, lists, trees, strings, sets, etc.)
  - Numerous optimizations of the intermediate Petri net model extended with data
  - Dynamically resizable state tables
  - Code specialization according to the amount of available RAM
  - Rapid prototyping and code generation

Support for FSP

- FSP *(Finite State Processes)* [Magee-Kramer]
  - A simple, concise process calculus
  - Supported by the LTSA tool
- Tools: FSP2LOTOS and FSP.OPEN
  - Translation from FSP to LOTOS + EXP + SVL
  - On-the-fly state space generation for FSP
  - Benefits with respect to LTSA:
    - Non-guarded process recursion is handled
    - 64-bit support for larger state spaces
    - Easy interfacing with all other CADP tools

Motivation behind LNT

- Advantages of process algebras:
  - Appropriate to model asynchronous systems formally
  - Equipped with formal verification tools (took years)
- But unpopular in industry due to
  - Steep learning curve
  - Lack of trained designers/engineers
- Need for new formal description techniques
  - more appropriate for industry (e.g., imperative style)
  - enable reuse of existing tools at minimal cost

⇒ **LNT:**

- subset of E-LOTOS proposed by VASY (since 1995)
- uniform language:
  e.g., functions are a particular case of processes

Short history of LOTOS NT & LNT

- 1995-1998: participation to the standardization of E-LOTOS definition of LOTOS NT by Sighireanu and Garavel
- 2000: release of TRAIAN
  - data part of LOTOS NT into C
  - since then, compiler development of VASY based on TRAIAN: SVL, Exp.Open 2.0, Evaluator 3.0, NTIF, chp2lotos, lnt2lotos, ...
- 2004-2007: FormalFamePlus Contract (VASY – Bull)
  - use of LOTOS NT to model critical parts of Bull's high-end servers
  - funding for the development of a LOTOS NT to LOTOS translator
- 2006: release of lnt2lotos (data part of LOTOS NT)
- 2008: release of lnt2lotos (full LOTOS NT)
- 2010: integration into CADP (release of lnt.open)
- 2011: renaming of LOTOS NT to LNT

LNT tutorial: Plan

- LNT: Language overview
  - Modules
  - Types
  - Functions
  - Processes
- Running example: MCS queue lock

More information in the reference manual:

http://vasy.inria.fr/Publications/Champelovier-Clerc-Garavel-et-al-10.pdf
(regularly updated as $CADP/doc/pdf/Champelovier-Clerc-Garavel-et-al-10.pdf)

III.1 LNT MODULES

LNT modules

- Compilation unit
- One module = one file (of the same name)
- Modules can import other modules:
  currently: no difference between interface and implementation
- *Principal module* containing the *root process* (by default, called "MAIN")
- Case insensitive module names, but
  - all modules in the same directory
  - no two files differing only by case

Sample LNT modules

```
module PLAYER is
      ...
end module
```

file “PLAYER.lnt”

list of imported modules

```
module Team (PLAYER) is
      ...
end module
```

file “TEAM.lnt”

or (one of):
- “Team.lnt”
- “team.lnt”
- “TeAm.lnt”
- …

Module Imports: Naming Conventions

- Problem: LNT case insensitive, but not the OS (except Windows®)
- Chosen approach:
  - all identifiers are converted into upper case
  - for all but the principal module:
    all generated filenames are in uppercase
  - for principal module:
    keep case of case as input file
  - search of imported modules (LNT source):
    - first with the case as in the import line
    - then converted into upper case

III.2 LNT TYPES

LNT types

- Inductive types
  - set of constructors with named and typed parameters
  - special cases: enumerations, records, unions, trees, etc.
  - shorthand notations for arrays, (sorted) lists, and sets
  - subtypes: range types and predicate types
  - automatic definition of standard functions: "==", "<=", "<", ">=", ">" , field selectors and updaters
  - pragmas to control the generated names in C and LOTOS
- Notations for constants (C syntax):
  - natural numbers: 123, 0xAD, 0o746, 0b1011
  - integer numbers: -421, -0xFD, -0o76, -0b110
  - floating point numbers: 0.5, 2E-3, 10.
  - characters: 'a', '0', '\n' , '\\', '\‘'
  - character strings: “hello world”, “hi!\n”

Sample LNT types

- Enumerated type

```
type Weekday is (* LOTOS-style comment *)
        Mon, Tue, Wed, Thu, Fri, Sat, Sun
end type
```

- Record type

```
type Date is -- ADA-style comment (to the end of the line)
        date (day: Nat, weekday: Weekday, month: Nat, year: Nat)
end type
```

- Inductive Type

```
type Nat_Tree is
        leaf (value: Nat),
        node (left: Nat_Tree, right: Nat_Tree)
end type
```

Sample LNT types

- Control of generated LOTOS & C names

```
type BYTE is
   !representedby “LOTOS_BYTE"
   !implementedby “C_BYTE"
   !printedby "PRINT_BYTE"
   BYTE (B0, B1, B2, B3, B4, B5, B6, B7)
end type
```

- Implementation by external C types

```
type INT_32 is -- record type
   !external
   !implementedby "int"
end type
```

Sample LNT types

- Shorthand notation

```
type Nat_List is
   list of Nat
end type
```

instead of

```
type Nat_List is
   nil,
   cons (head: Nat, tail: Nat_List)
end type
```

- Automatic definition of standard functions

```
type Num is
  one, two, three
  with "==", "<=", "<", ">=", ">“
end type
type Date is
  date (d: Nat, wd: Weekday, month: Nat, year: Nat)
  with "get", "set" (* for selectors X.D, ... and updaters X.{D => E} *)
end type
```

Sample LNT types

- One-dimensional array

```
type Vector is -- four-dimensional vector
  array [ 0 .. 3 ] of Int
end type
```

- Two-dimensional array

```
type Matrix is -- four-dimensional square-matrix
  array [ 0 .. 3 ] of Vector
end type
```

- Array of records

```
type Date_Array is
  array [ 0 .. 1 ] of DATE
end type
```

Sample LNT types

- Range types (intervals)

```
type Index is
  range 0 .. 5 of Nat
  with “==“, “!=“
end type
```

further automatically definable functions: first, last, card

- Predicate types

```
type EVEN is
      n: NAT where n mod 2 == 0
end type
type PID is
  i: Index where i != 0
end type
```

MCS queue lock: data types

```
type Index is
  range 0 .. 5 of Nat
  with "==", "!="
end type

type Pid is
  pid: Index where pid != 0
  with "==", "!="
end type

type Operation is
  Read_next, Read_locked,
  Write_next, Write_locked,
  Fetch_and_Store, Compare_and_Swap
end type
```

```
type Qnode is
  Qnode (next: Index, locked: Bool)
  with "get", "set"
end type

type Memory is
  array [ 1 .. 5 ] of Qnode
end type
```

```
type qnode = record
  next        : ^qnode
  locked      : Boolean
type lock = ^qnode
```

LNT Module Pragmas

- Automatic generation of predefined functions
  module M with “get”, “set”, “card” is …

more functions:
see type definition

- Width and range of predefined types
  module M is !nat_bits 3 …
  - nat_bits/int_bits:
    bits for storing Nat/Int type
  - nat_inf/int_inf & nat_sup/int_sup:
    lower & upper bound of Nat/Int type
  - nat_check/int_check:
    (de)activate bound checks for Nat/Int type
  - string_card:
    maximum number of strings (size of the hash table)

0: deactivate
1: activate (default)

III.3 LNT FUNCTIONS

LNT functions

- Pure functions (without side effects) in imperative syntax ensured by type checking and initialization analysis
- Functions defined using standard algorithmic statements:
  - Local variable declarations and assignments: “var”
  - Sequential composition: “;”
  - Breakable loops: “while” and “for”
  - Conditionals: “If-then-else”
  - Pattern matching: “case”
  - (Uncatchable) exceptions: “raise”
- Three parameter passing modes:
  - “in” (call by value)
  - “out” and “inout” (call by reference)
- Function overloading
- Support for external implementations (LOTOS and C)

call syntax requires “eval” keyword

Sample LNT functions

- Constants

```
function pi: Real is
  return 3.14159265
end function
```

- Field access
  -
    ```
    function get_weekday (d: Date): Weekday is
      return d.wd
    end function
    ```
  -
    ```
    function set_weekday (inout d: Date, new_wd: Weekday) is
      d := d.{wd => new_wd}
    end function
    ```

Sample LNT functions

- Update of the element (i,j) of a matrix M

```
function update (inout M: Matrix, i, j: Nat, new_e: Nat) is
  var v: Vector in
    v := M[i];
    v[j] := new_e;
    M[i] := v
  end var
end function
```

- Access to the first element of a list L

```
function get_head (L: Nat_List) : Nat raises Empty_List: none is
  case L in var head: Nat in
     nil ->                              raise Empty_List
   | cons (head, any Nat_List) ->        return head
  end case
end function
```

Sample LNT functions

```
function reset_diagonal_elements (M: Matrix) : Matrix is
  var
    result: Matrix,
    i: Nat
  in
    result := M;
    for i := 0 while i < 3 by i := i + 1 loop
      eval update (!?result, i, i, 0)
    end loop;
    return result
  end var
end function
```

MCS queue lock: functions

-
  ```
  function nil: Index is (* constant definition *)
    return Index (0)
  end function
  ```

-
  ```
  function Nat (pid: Pid) : Nat is (* explicit type cast *)
    return Nat (Index (pid))
  end function
  ```

-
  ```
  function _!=_ (p: Pid, i: Index) : Bool is (* infix comparison *)
    return (Index (p) != i)
  end function
  ```

III.4 LNT PROCESSES

LNT processes

- Processes are a superset of functions *(except return)*:
  - symmetric sequential composition
  - variable assignment, "if-then-else", "case", "loop", etc.
- Additional operators:
  - communication: rendezvous with value communication
  - parallel composition: "par"
  - gate hiding: "hide"
  - nondeterministic choice: "select"
  - "disrupt", etc.
- Static semantics constraints
  - variable initialization
  - typed channels (with polymorphism and "any" type)

LOTOS style
(see next slide)

LNT rendezvous

G $(O_1, \ldots, O_{n\geq0})$ where V

O_i ::= V | !V | ?P

- Polymorphic channel types
- Exchange of several values (*offers* O_i)
- Combination of inputs and outputs
- Value matching / constraint solving
- Pattern matching
- For short: LOTOS-style rendezvous plus
  - pattern matching
  - polymorphic gate typing (channel)

Sample LNT channels

- channel None is
  ```
  ()
  ```
  end channel
- channel C1 is
  ```
  (Nat)
  ```
  end channel
- channel C2 is
  ```
  (Signal, Nat),
  (Signal, Nat, Nat)
  ```
  end channel
- predefined channel:
  any
  rendezvous without type-check for offers (LOTOS style)

MCS queue lock: channels

```
channel Resource_Access is
  (Pid)
end channel
channel Memory_Access is
  (Operation, Pid, Index, Pid),              -- read/write field next
  (Operation, Pid, Bool,  Pid)               -- read/write field locked
end channel
channel Lock_Access is
  (Operation, Index, Index, Pid),            -- fetch-and-store
  (Operation, Index, Index, Bool, Pid)       -- compare-and-swap
end channel
channel Latency is
  (Pid),
  (Operation)
end channel
```

“Hello, world!”

- without channel typing

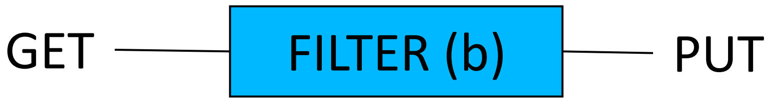
```
module hello_world is
process MAIN [G:any] is
    G (”Hello, world!\n”)
end process
end module
```

- with channel typing

```
module hello_world is
channel String_channel is (String) end channel
process MAIN [G:String_channel] is
    G (”Hello, world!\n”)
end process
end module
```

Sample LNT process

```
type option is none, some (x: Nat) end type

channel option_channel is (o: Option) end channel

channel nat_channel is (n: Nat) end channel

process FILTER [GET: option_channel, PUT: nat_channel] (b: Nat) is
   var opt: Option in
     loop L in
       GET (?opt) ;
       case opt in var x: Nat in
          none                    -> null
        | some (x) where x > b    -> PUT (x)
       end case
     end loop
   end var
end process
```



MCS queue lock: competing process

```
process P [NCS, CS_Enter, CS_Leave: Resource_Access,
              L: Lock_Access, M: Memory_Access]
             (pid: Pid) is
 loop
  NCS (pid);
  acquire_lock [L, M] (pid);
  CS_Enter (pid); CS_Leave (pid);
  release_lock [L, M] (pid)
 end loop
end process
```

MCS queue lock: acquire_lock

```
process acquire_lock [L: Lock_Access, M: Memory_Access] (pid: Pid) is
  var predecessor: Index, locked: Bool in
    M (W_next, pid, nil of Index, pid);
    L (Fetch_and_Store, ?predecessor, Index (pid), pid);
    if (predecessor != nil) then
      M (W_locked, pid, true, pid);
      M (W_next, Pid (predecessor), Index (pid), pid);
      loop L in
        M (R_locked, pid, ?locked, pid);
        if not (locked) then break L end if
      end loop
    end if
  end var
end process
```

```
proc acquire_lock (L : ^lock, I : ^qnode)
  I->next := nil
  predecessor : ^qnode :=
    fetch_and_store (L, I)
  if predecessor != nil
    I->locked := true
    predecessor->next := I
    repeat while I->locked // spin
```

MCS queue lock: release_lock

```
process release_lock [L: Lock_Access, M: Memory_Access] (pid: Pid) is
  var next: Index, swap: Bool in
    M (R_next, pid, ?next, pid);
    if next == nil then
      L (Compare_and_Swap, Index (pid), nil of Index, ?swap, pid);
      if swap == false then
        loop L in
          M (R_next, pid, ?next, pid);
          if next != nil then break L end if
        end loop;
        M (W_locked, Pid (next), false, pid)
      end if
    else
      M (W_locked, Pid (next), false, pid)
    end if
  end var
end process
```

```
proc release_lock (L : ^lock, I : ^qnode)
 if I->next = nil  // no known successor
  if compare_and_swap (L, I, nil)
   // true iff swapped
   return
  repeat while I->next = nil // spin
 I->next->locked := false
```

MCS queue lock: Global variable

```
process Lock [L: Lock_Access] is
  var i, new_i, j: Index in
    i := nil;
    loop select
      L (Fetch_and_Store, i, ?new_i, ?any Pid);
      i := new_i
     []
      L (Compare_and_Swap, ?j, ?new_i, true, ?any Pid) where i == j;
      i := new_i
     []
      L (Compare_and_Swap, ?j, ?new_i, false, ?any Pid) where i != j
      -- ignore new_i
    end select end loop
  end var
end process
```

MCS queue lock: Shared variables

```
process Memory [M: Memory_Access] is
  var m: Memory, pid: Pid, next: Index, locked: Bool in
    m := Memory (Qnode (nil, false));
    loop select
         M (Read_next, ?pid, ?next, ?any Pid)
            where next == m[Nat (pid)].next
     []  M (Read_locked, ?pid, ?locked, ?any Pid)
            where locked == m[Nat (pid)].locked
     []  M (Write_next, ?pid, ?next, ?any Pid);
         m[Nat (pid)] := m[Nat (pid)].{next => next}
     []  M (Write_locked, ?pid, ?locked, ?any Pid);
         m[Nat (pid)] := m[Nat (pid)].{locked => locked}
    end select end loop
end var end process
```

MCS queue lock for five processes

```
process Protocol [NCS, CS_Enter, CS_Leave: Resource_Access,
                      L: Lock_Access, M: Memory_Access] is
  par M, L in
    par
        P [NCS, CS_Enter, CS_Leave, L, M] (Pid (1))
    ||  P [NCS, CS_Enter, CS_Leave, L, M] (Pid (2))
    ||  P [NCS, CS_Enter, CS_Leave, L, M] (Pid (3))
    ||  P [NCS, CS_Enter, CS_Leave, L, M] (Pid (4))
    ||  P [NCS, CS_Enter, CS_Leave, L, M] (Pid (5))
    end par
  ||
    par Lock [L]  ||  Memory [M] end par
  end par
end process
```

MCS queue lock: service (1/3)

```
type Pid_list is
 list of Pid with "==", "!="
end type
function _is_in_ (pid: Pid, fifo: Pid_list) : Bool is
 -- return true iff pid is in the list fifo
 case fifo in
 var head: Pid, tail: Pid_list in
   nil ->                        return false
 | cons (head, tail) ->          if (head == pid) then
                                   return true
                                 else
                                   return pid is_in tail
                                 end if
 end case
end function
```

MCS queue lock: service (2/3)

```
function pop (inout fifo: Pid_list, out pid: Pid)
raises Empty_list: none
is  -- remove last element of the list fifo
  case fifo in
  var head: Pid, tail: Pid_list in
    {} ->
        raise Empty_list
  | { head } ->
        pid := head; fifo := {}
  | cons (head, tail) ->
        eval pop (!?tail, ?pid); fifo := cons (head, tail)
  end case
end function
```

MCS queue lock: service (3/3)

```
process Service [CS_Enter, CS_Leave: Resource_Access] is
 var pid: Pid, fifo: Pid_list, current: Index in
  fifo := nil; current := nil;
  loop select
   pid := any Pid where (not (pid is_in fifo)) and (pid != current);
   fifo := cons (pid, fifo); i
  []
   if (current == nil) and (fifo != nil) then
    eval pop (!?fifo, ?pid); CS_Enter (pid); current := Index (pid)
   else stop end if
  []
   if current != nil then
    CS_Leave (Pid (current)); current := nil
   else stop end if
  end select end loop
end var end process
```

Check of semantic constraints

- Semantic checks performed by lnt2lotos
  - Correct declaration (variables, gates)
  - Correct initialization (variables / parameters)
  - Non-ambiguous overloading
  - Breaks inside matching loops
  - Path constraints (e.g., presence of a return)
  - Parameters usage
- Semantic checks performed by Cæsar(.adt) / CC
  - Type constraints (expressions and gates)
  - Availability of used types, functions, and processes
  - Exhaustiveness of case statements
  - Availability of external code (LOTOS, C)
  - Range/overflow checks for numbers

by lnt_check on the C code generated by Cæsar(.adt)

See the reference manual for details!

IV. FROM LANGUAGES TO MODELS

IV.1 BCG (*BINARY CODED GRAPH*)

BCG format

- Text-based formats are not satisfactory to store large LTSs in computer files
  - disk space consuming (Gbytes)
  - slow (read/write operations are costly)
- BCG (*Binary-Coded Graphs*):
  - **a compact file format** for storing LTSs
  - **a set of APIs**
  - **a set of software libraries**
  - **a set of tools (binary programs and scripts)**

BCG libraries and APIs

- BCG_WRITE
  API to create a BCG file
- BCG_READ
  API to read a BCG file
- BCG_TRANSITION
  API to store a transition relation in memory:
  - successor function, or
  - predecessor function, or
  - successor and predecessor functions

Basic BCG tools

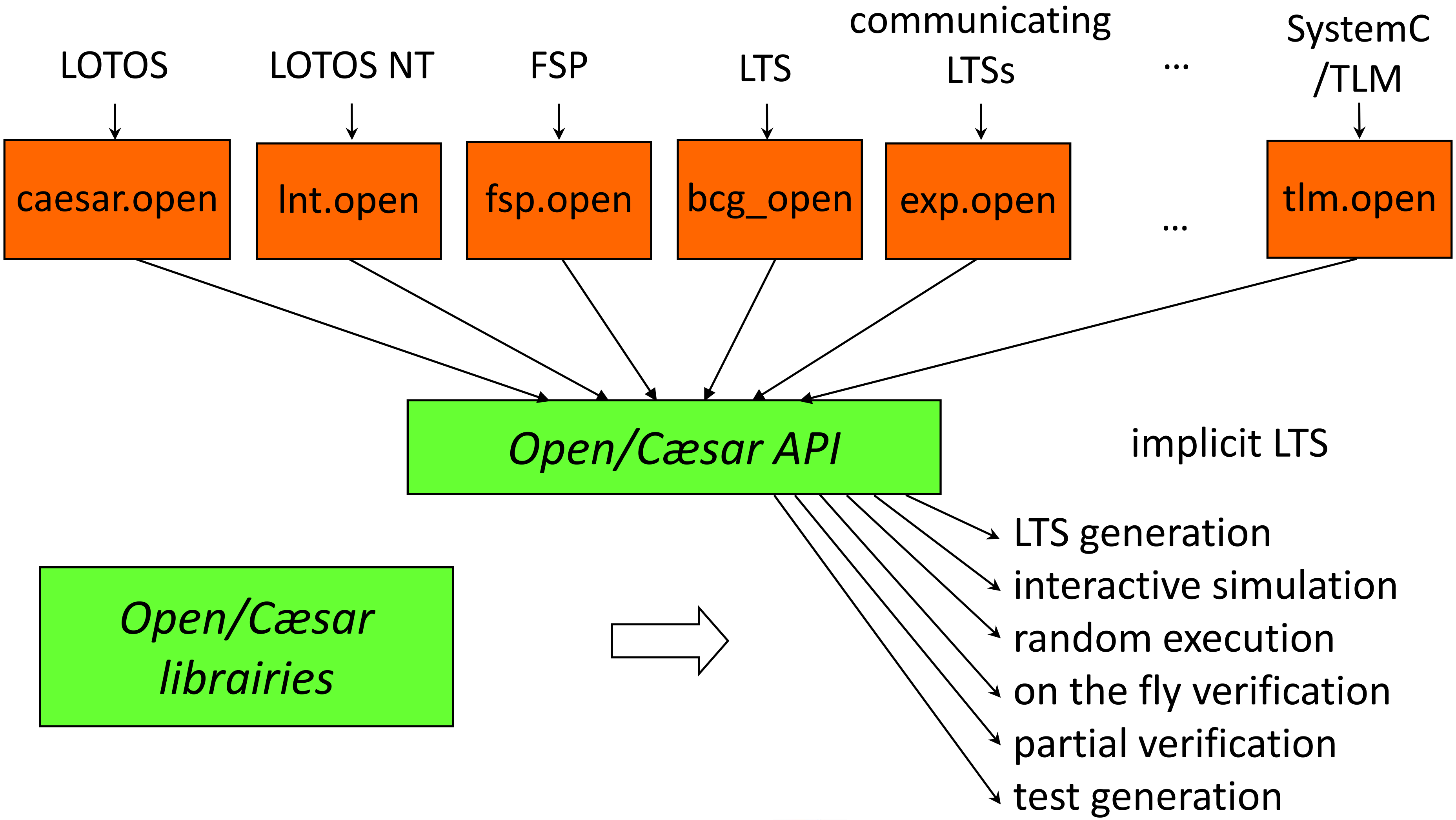
- bcg_info: extract info from a BCG file
- bcg_io: convert BCG from and to other formats
- bcg_labels: hide and/or rename labels
- bcg_draw, bcg_edit: visualize LTSs
- bcg_graph: generation of particular BCG graphs (chaos automata, FIFO buffers, bag automata)
- bcg_open: connection to Open/Cæsar applications

IV.2 OPEN/CÆSAR API

Motivations

- Most model checkers dedicated to one particular input language (e.g. Spin, SMV, …)
- They can't be reused easily for other languages
- Idea: introduce modularity by separating
  - **language-dependent aspects:** compiling language into LTS model
  - **language-independent algorithms**: algorithms for LTS exploration

OPEN/CÆSAR



LOTOS

LOTOS NT

FSP

LTS

communicating LTSs

…

SystemC /TLM

caesar.open

lnt.open

fsp.open

bcg_open

exp.open

…

tlm.open

Open/Cæsar API

implicit LTS

- LTS generation
- interactive simulation
- random execution
- on the fly verification
- partial verification
- test generation

Open/Cæsar librairies

OPEN/CÆSAR API

- Primitives to represent an implicit LTS
  - Opaque type for the representation of a state
  - Initial state function
  - Successor function
  - etc.
- Provided by Open/Cæsar compilers
- Used by Open/Cæsar compliant tools

OPEN/CÆSAR libraries

- **A set of predefined data structures**
  - EDGE: list of transitions (e.g., successor lists)
  - HASH: catalog of hash functions
  - STACK_1: stacks of states and/or labels
  - DIAGNOSTIC_1: set of execution paths
  - TABLE_1: hash table for states, labels, strings, etc.
  - BITMAP: Holzmann’s "bit state" tables
  - RENAME_1: handling of label renaming options
- **Specific primitives for on the fly verification**
  - possibility to attach additional information to states
  - stack or table overflow => backtracking
  - etc.

Some OPEN/CÆSAR applications

- EXECUTOR: random walk
- OCIS: interactive simulation (graphical)
- GENERATOR: exhaustive LTS generation
- REDUCTOR: LTS generation with reduction
- PROJECTOR: LTS generation with constraints
- TERMINATOR: Holzmann's bit-space algorithm
- EXHIBITOR: search paths defined by reg. expr.
- EVALUATOR: evaluation of mu-calculus formulas
- TGV: test sequence generation
- DISTRIBUTOR: distributed state space generation
- CUNCTATOR: Markov chain steady-state simulator
- …

Example: GENERATOR (1/2)

```
#include "caesar_graph.h"
#include "caesar_edge.h"
#include "caesar_table_1.h"
#include "bcg_user.h"

int main (int argc, char *argv[]) {
    char *filename;
    CAESAR_TYPE_TABLE_1 t; CAESAR_TYPE_STATE s1, s2;
    CAESAR_TYPE_EDGE e1_en, e; CAESAR_TYPE_LABEL l;
    CAESAR_TYPE_INDEX_TABLE_1 n1, n2, initial_state ; CAESAR_TYPE_POINTER dummy;
    filename = argv[0];
    CAESAR_INIT_GRAPH ();
    CAESAR_INIT_EDGE (CAESAR_FALSE, CAESAR_TRUE, CAESAR_TRUE, 0, 0);
    CAESAR_CREATE_TABLE_1 (&t, 0, 0, 0, 0, TRUE, NULL, NULL, NULL, NULL);
    if (t == NULL) CAESAR_ERROR ("not enough memory for table");
    CAESAR_START_STATE ((CAESAR_TYPE_STATE) CAESAR_PUT_BASE_TABLE_1 (t));
    CAESAR_PUT_TABLE_1 (t);
    initial_state = CAESAR_GET_INDEX_TABLE_1 (t);
    BCG_INIT ();   BCG_IO_WRITE_BCG_BEGIN (filename, initial_state, 2, "", 0);
```

Example: GENERATOR (2/2)

```
    while (!CAESAR_EXPLORED_TABLE_1 (t)) {
        s1 = (CAESAR_TYPE_STATE) CAESAR_GET_BASE_TABLE_1 (t);
        n1 = CAESAR_GET_INDEX_TABLE_1 (t);
        CAESAR_GET_TABLE_1 (t);

        CAESAR_CREATE_EDGE_LIST (s1, &e1_en, 1);
        if (CAESAR_TRUNCATION_EDGE_LIST () != 0)
            CAESAR_ERROR ("not enough memory for edge lists");

        CAESAR_ITERATE_LN_EDGE_LIST (e1_en, e, l, s2) {
            CAESAR_COPY_STATE ((CAESAR_TYPE_STATE) CAESAR_PUT_BASE_TABLE_1 (t), s2);
            (void) CAESAR_SEARCH_AND_PUT_TABLE_1 (t, &n2, &dummy);
            BCG_IO_WRITE_BCG_EDGE (n1, CAESAR_STRING_LABEL (l), n2);
        }
        CAESAR_DELETE_EDGE_LIST (&e1_en);
    }
    BCG_IO_WRITE_BCG_END ();
    return (0)
}
```

IV.3 TOOLS FOR STATE SPACE GENERATION

State space generation

- **Motivation:** generate an explicit LTS (BCG) from an implicit one (Open/Cæsar), for verification
- Use GENERATOR for direct generation
- **Problem:** possible state explosion, e.g. when the number of concurrent processes grows
- Several solutions to fight against state explosion:
  - Compositional verification
  - Distributed state space generation
  - (Combined with static analysis, partial order reductions, ...)

Compositional verification

- *"Divide and conquer" to fight state explosion*
  - Partition the system into subsystems
  - Minimize each subsystem modulo a strong or weak bisimulation preserving the properties to verify
  - Recombine the subsystems to get a system equivalent to the initial one
- Refined compositional verification:
  - Tightly-coupled processes constrain each other
  - Separating them may lead to explosion
  - "Interfaces" used to model synchronization constraints
- SVL (*Script Verification Language*) provides high-level support for compositional verification (see later)

Minimization tools

- Aldebaran
  - no longer supported after July 2008 (64-bit issue)
  - functionalities retained with Aldebaran 7.0 script
- BCG_MIN
  - minimization of explicit LTSs
  - strong and branching bisimulation
  - new signature-based algorithm
  - supports LTS with $10^9 – 10^{10}$ states
- Reductor
  - on-the-fly (partial) reduction of implicit LTSs
  - 8 equivalence relations supported:
    strong, branching, tau*.a, safety, trace (aka automata determinization), weak trace, tau-confluence, tau-compression, and tau-divergence

EXP.OPEN 2.0

- A language for describing networks of LTS
  - LTS encoded in AUT or BCG format
  - synchronization vectors + parallel composition operators (LOTOS, CCS, CSP, mCRL, etc.)
  - label hiding, renaming, cutting (using regexps)
  - "priority" operator
- An Open/Cæsar compiler
  - on-the-fly partial order reductions (branching eq., weak trace eq., stochastic/probabilistic eq.)

PROJECTOR 3.0

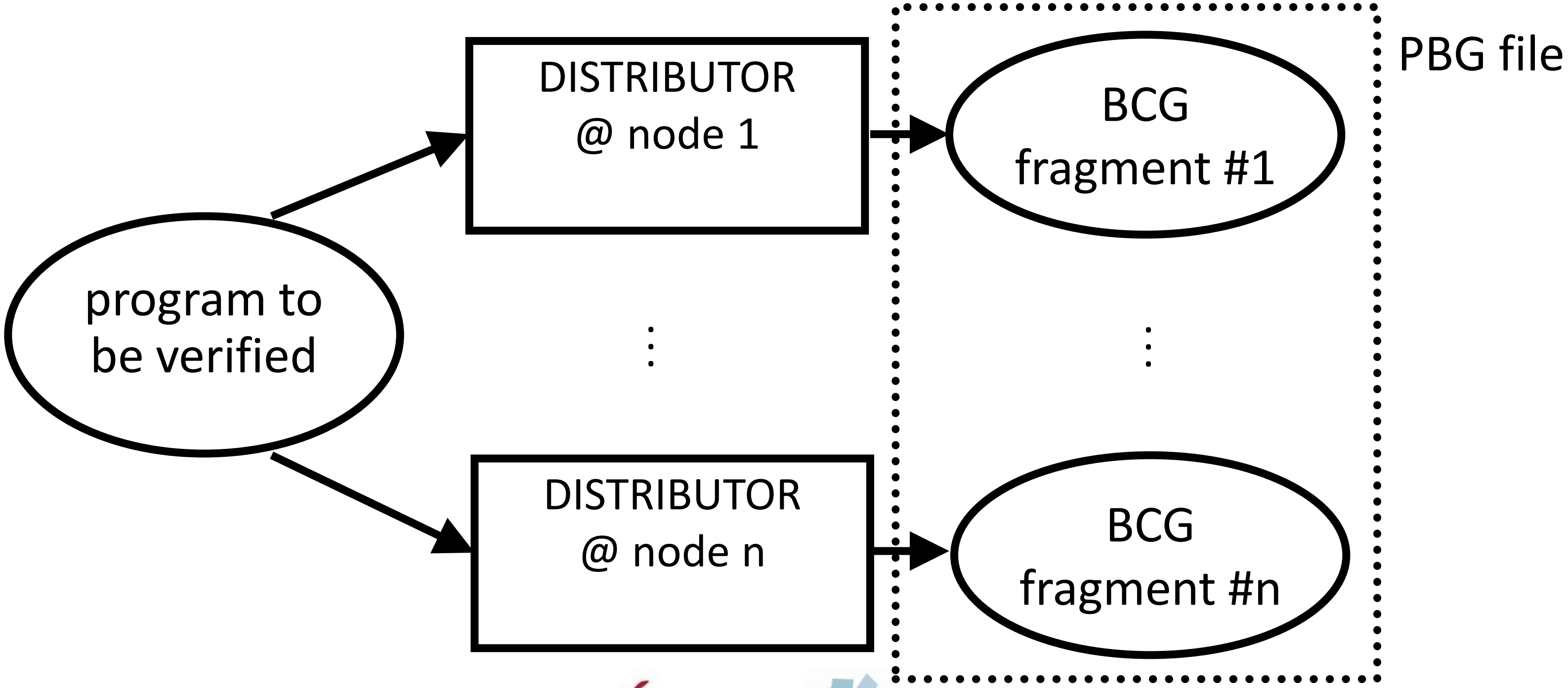
- To achieve refined compositional verification
- Implements ideas of Graf & Steffen, Krimm & Mounier
- Computes on the fly the restriction of an LTS modulo interface constraints
  - Interface = LTS understood as a set of traces
  - Eliminates states and transitions of a process never reached while following all traces of its interface
  - User-given interfaces involve predicate generation to check their correctness

Distributed state space generation

- Exploit workstation networks, clusters and grids
- Cumulate CPU and RAM across the network
- GCF (*Grid Configuration File*) to configure:
  - number and names of machines
  - local directories
  - CADP installation directories
  - communication protocols, addresses
- Socket-based internal communication library (SSH connections, TCP sockets)

DISTRIBUTOR

- Distributed state space generation
- Generates distributed BCG fragments referenced in a PBG (*Partitioned BCG graph*) file
- Enables tau-compression and tau-confluence (partial order) reductions preserving branching bisimulation



PBG file

program to be verified

DISTRIBUTOR @ node 1

⋮

DISTRIBUTOR @ node n

BCG fragment #1

⋮

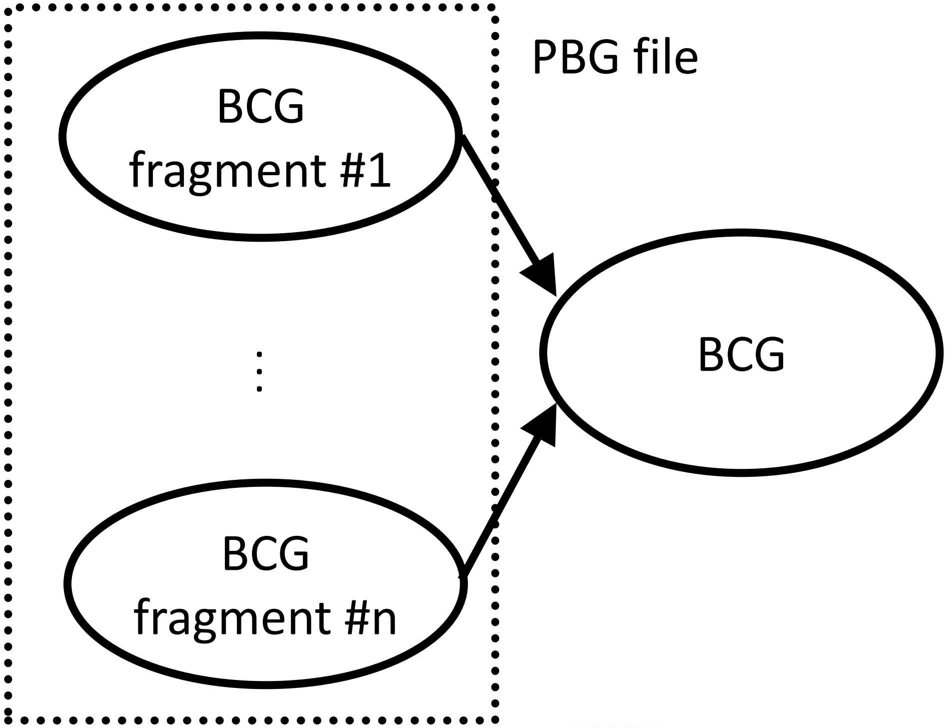
BCG fragment #n

Tools to handle PBG files

- pbg_info:
  - compute global state space information by combining state space information of the fragments
  - check consistency of the PBG file
- pbg_cp, pbg_mv, and pbg_rm:
  - convenient handling
  - single command to modify all fragments of a PBG
- pbg_open: connection to the Open/Cæsar API

BCG_MERGE

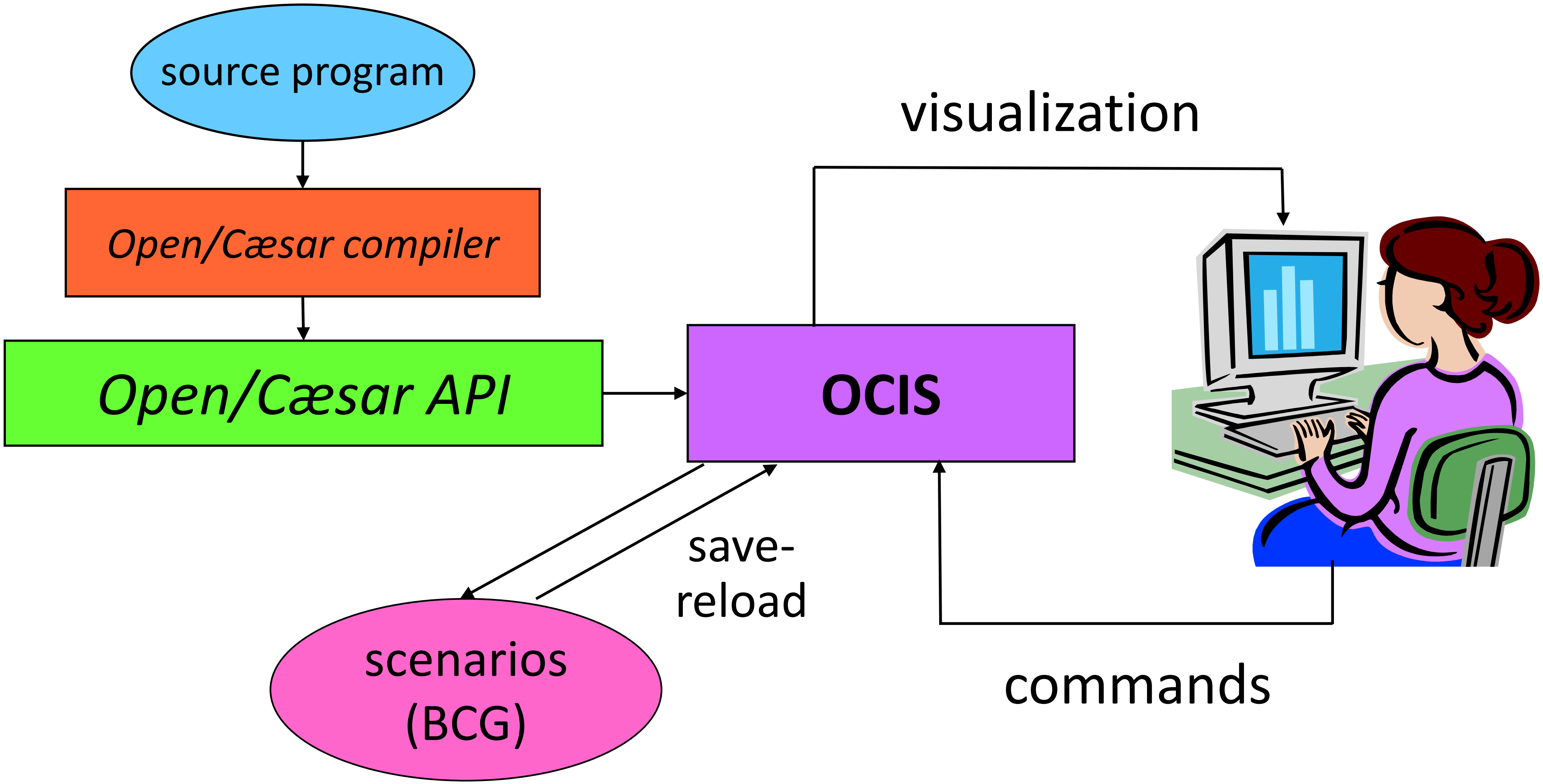
- Merges a distributed state space produced by DISTRIBUTOR into a monolithic labelled transition system
- Same functionality as pbg_open/generator but more efficient



V. FUNCTIONAL VERIFICATION

V.1 VISUAL CHECKING

OCIS (*Open/Cæsar Interactive Simulator*)



source program

Open/Cæsar compiler

Open/Cæsar API

OCIS

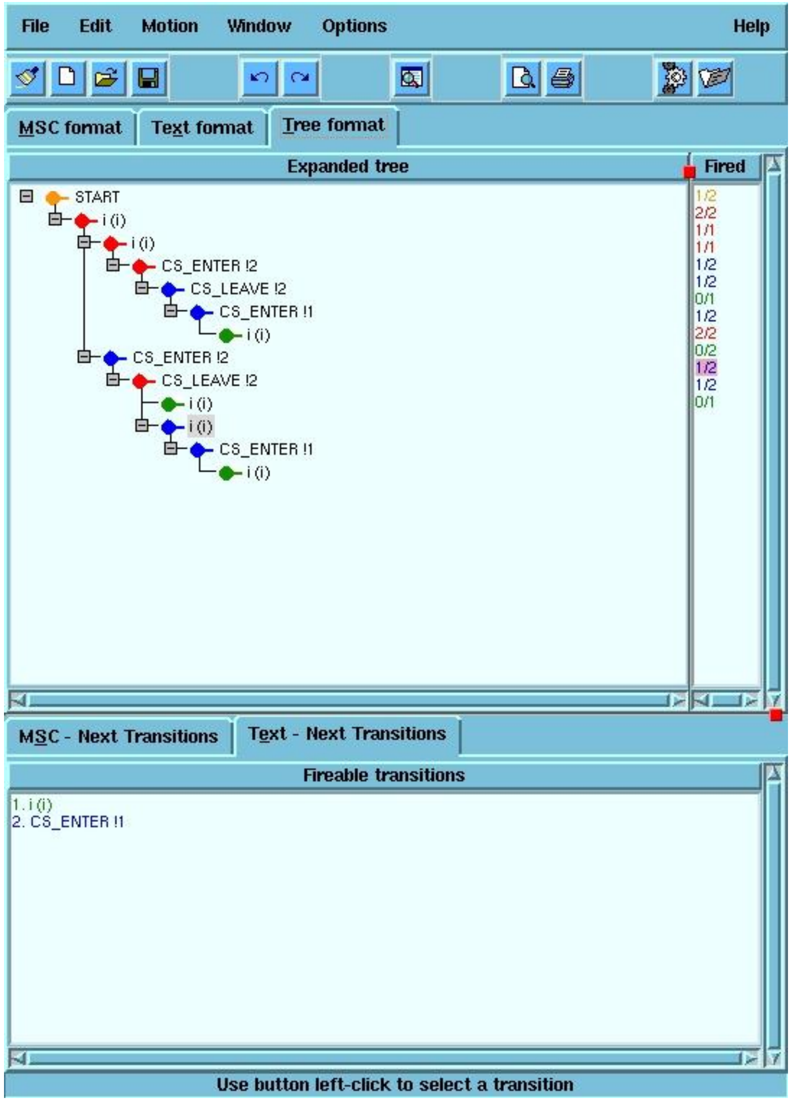
visualization

save-
reload

scenarios
(BCG)

commands

OCIS (*Open/Cæsar Interactive Simulator*)



```
START                          1/2
  i (i)                        2/2
    i (i)                      1/1
      CS_ENTER !2              1/1
        CS_LEAVE !2            1/2
          CS_ENTER !1          1/2
            i (i)              0/1
    CS_ENTER !2                1/2
      CS_LEAVE !2              2/2
        i (i)                  0/2
        i (i)                  1/2
          CS_ENTER !1          1/2
            i (i)              0/1
```

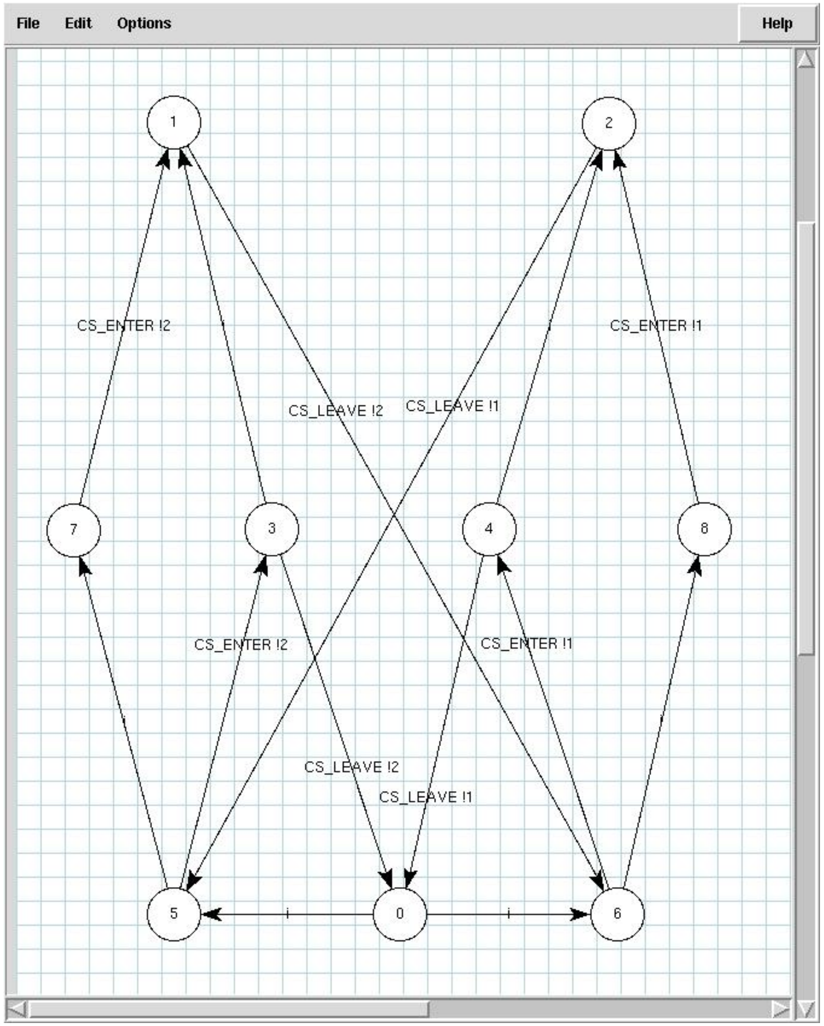
```
1. i (i)
2. CS_ENTER !1
```

Use button left-click to select a transition

- language-independent
- tree-like scenarios
- save/load scenarios
- source code access
- dynamic recompile

Bcg_Draw and Bcg_Edit

- View BCG graph
- Edit postscript interactively
- Applicable to small LTSs (e.g., after hiding internal actions & minimization)



V.2 EQUIVALENCE CHECKING

BISIMULATOR

- On-the-fly comparison of an implicit LTS (Open/Cæsar graph) and an explicit LTS (BCG graph)
- Uses Boolean Equation Systems (CÆSAR_SOLVE)
- Checks equivalence (=) or inclusion (≤ or ≥)
- Seven equivalence relations supported (strong, branching, observational, tau*.a, safety, trace, and weak trace)
- Generates counterexamples (common LTS fragments leading to differences)

V.3 MODEL CHECKING WITH MCL

MCL language

- Extended temporal logic

  Alternation-free mu-calculus

  + Regular sequences
  + Fairness operators (alternation 2)
  + Data handling
  + Libraries of derived operators
- Supported by the EVALUATOR 4.0 tool
  - BES resolution (CÆSAR_SOLVE)
  - Several optimized resolution algorithms
  - Tau-confluence reduction
  - Diagnostic generation

MCL examples (1/4)

- **Deadlock freeness**

```
[ true* ] < true > true
```

- **Mutual exclusion**

```
[ true* .
      { CS !"ENTER" ?i:Nat } .
      (not { CS !"LEAVE" !i })* .
      { CS !"ENTER" ?j:Nat where j <> i }
] false
```

MCL examples (2/4)

- **Independent progress** (N == number of processes)

  (*if a process stops in its non-critical section, the other processes can still access their critical sections)*

```
[ true* ] forall j:Nat among { 1 .. N } .  (
      < { NCS !j } > true
      implies
      [ (not { ... !j })* ] forall i:Nat among { 1 .. N } . (
                (i <> j) implies
                < (not { ... !j })* > < { ... !i }* . { CS ... !i } > @
      )
)
```

MCL examples (3/4)

- **Bounded overtaking**
  (process *j* overtakes process *i* exactly *max* times)

```
< true* . { NCS !i } .
      (not { ?G:String ... !i where (G <> "NCS") and (G <> "CS") })* .
      { ?G:String ... !i where (G <> "NCS") and (G <> "CS") } .
      (  for k:nat from 0 to n-1 do
            (not { CS ... !i })* .
            { ?G:String ... !k where (k = i) implies (G <> "CS") }
         end for .
         (not { CS ?any !i })* . { CS !"ENTER" !j }
      ) { max }
> true
```

MCS examples (4/4)

- **Livelock freedom**

  (*there is no cycle in which each process executes an instruction but no one enters its critical section)*

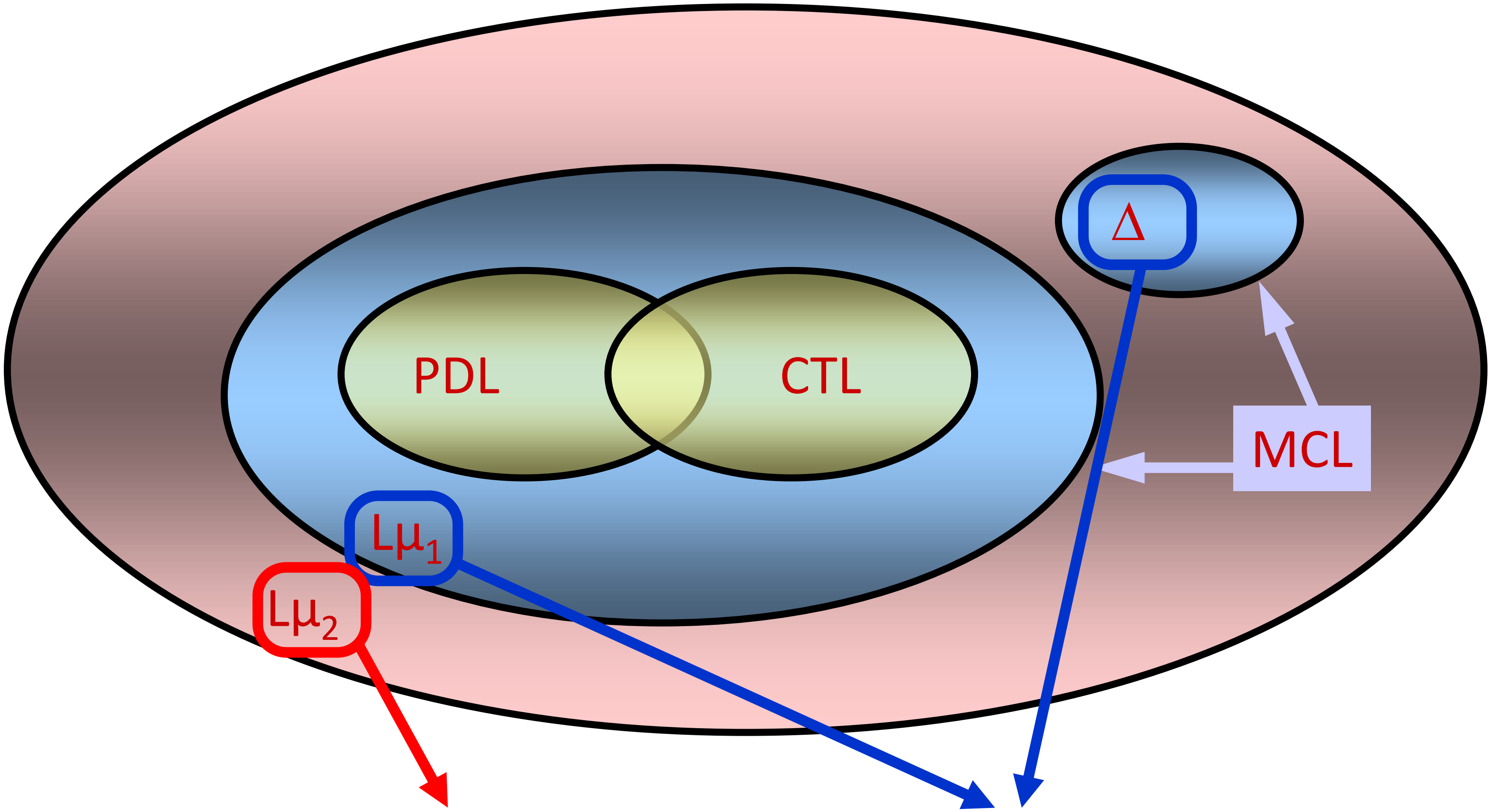
```
[  true* . { NCS ?j:Nat } .
   (not { ?any ?"READ"|"WRITE" ... !j })* .
   { ?any ?"READ"|"WRITE" ... !j }
] not < for j:Nat from 0 to n – 1 do
                    (not { CS ... })* .
                    { ?G:String ... !j where G <> "CS" }
          end for
        > @
```

complex cycle containing a set of events (generalized Büchi automaton)

MCL summary

- Characterization of finite trees using cascading of (strong/weak) regular modalities
- Characterization of infinite trees using ***infinite looping*** operator < R > @ and the dual ***saturation*** operator [R] -|
- Subsumes HML, ACTL, PDL, temporal patterns of Dwyer, and Transition-Based Generalized Büchi Automata (for LTL verification)
- Allows simulation of pushdown automata (context-free properties)

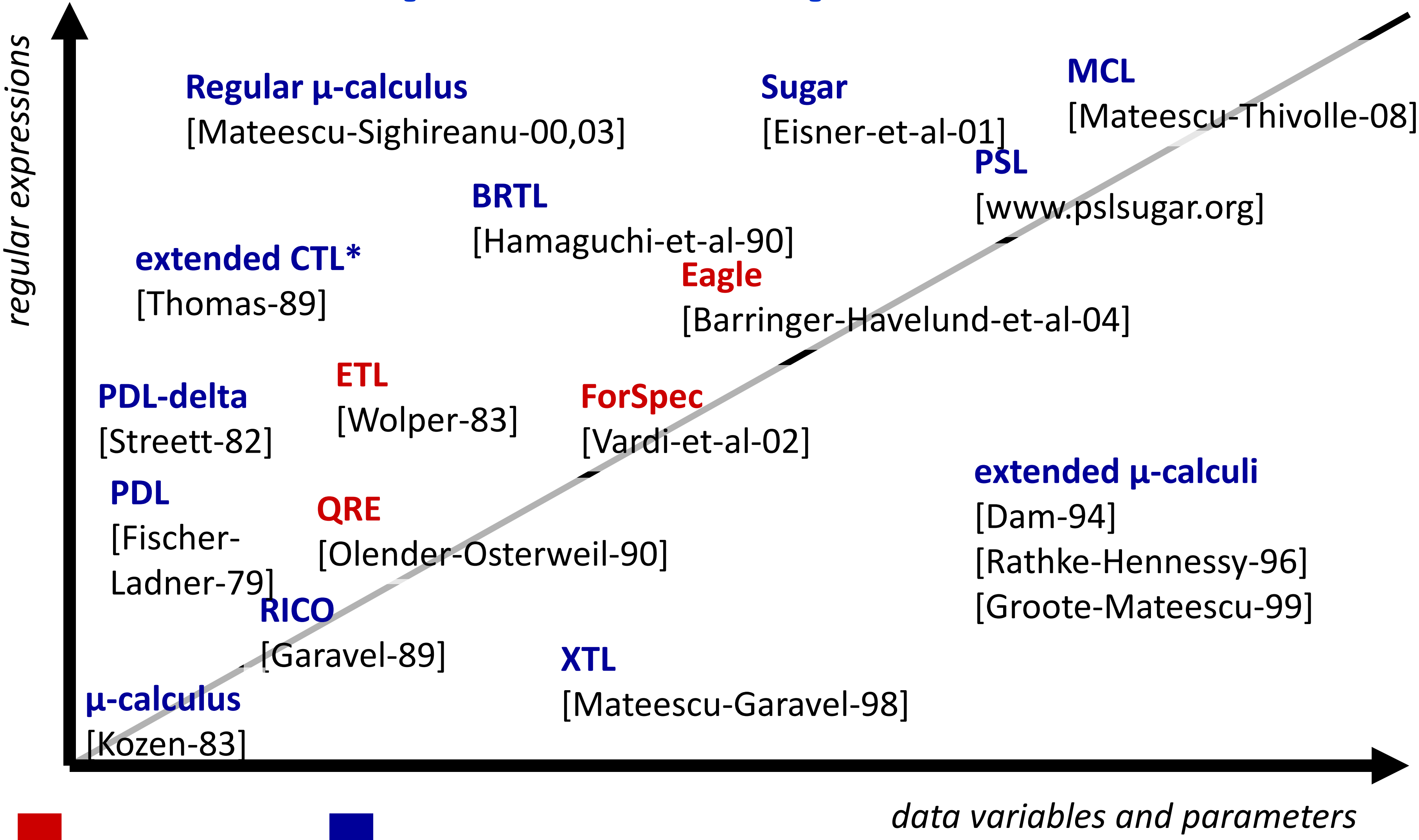
Expressiveness and complexity



quadratic-time complexity
(LTS size × formula size)^2

linear-time complexity
LTS size × formula size

The quest for a powerful TL



Regular μ-calculus
[Mateescu-Sighireanu-00,03]

Sugar
[Eisner-et-al-01]

MCL
[Mateescu-Thivolle-08]

PSL
[www.pslsugar.org]

BRTL
[Hamaguchi-et-al-90]

extended CTL*
[Thomas-89]

Eagle
[Barringer-Havelund-et-al-04]

PDL-delta
[Streett-82]

ETL
[Wolper-83]

ForSpec
[Vardi-et-al-02]

extended μ-calculi
[Dam-94]
[Rathke-Hennessy-96]
[Groote-Mateescu-99]

PDL
[Fischer-Ladner-79]

QRE
[Olender-Osterweil-90]

RICO
[Garavel-89]

XTL
[Mateescu-Garavel-98]

μ-calculus
[Kozen-83]

🟥 linear-time

🟦 branching-time

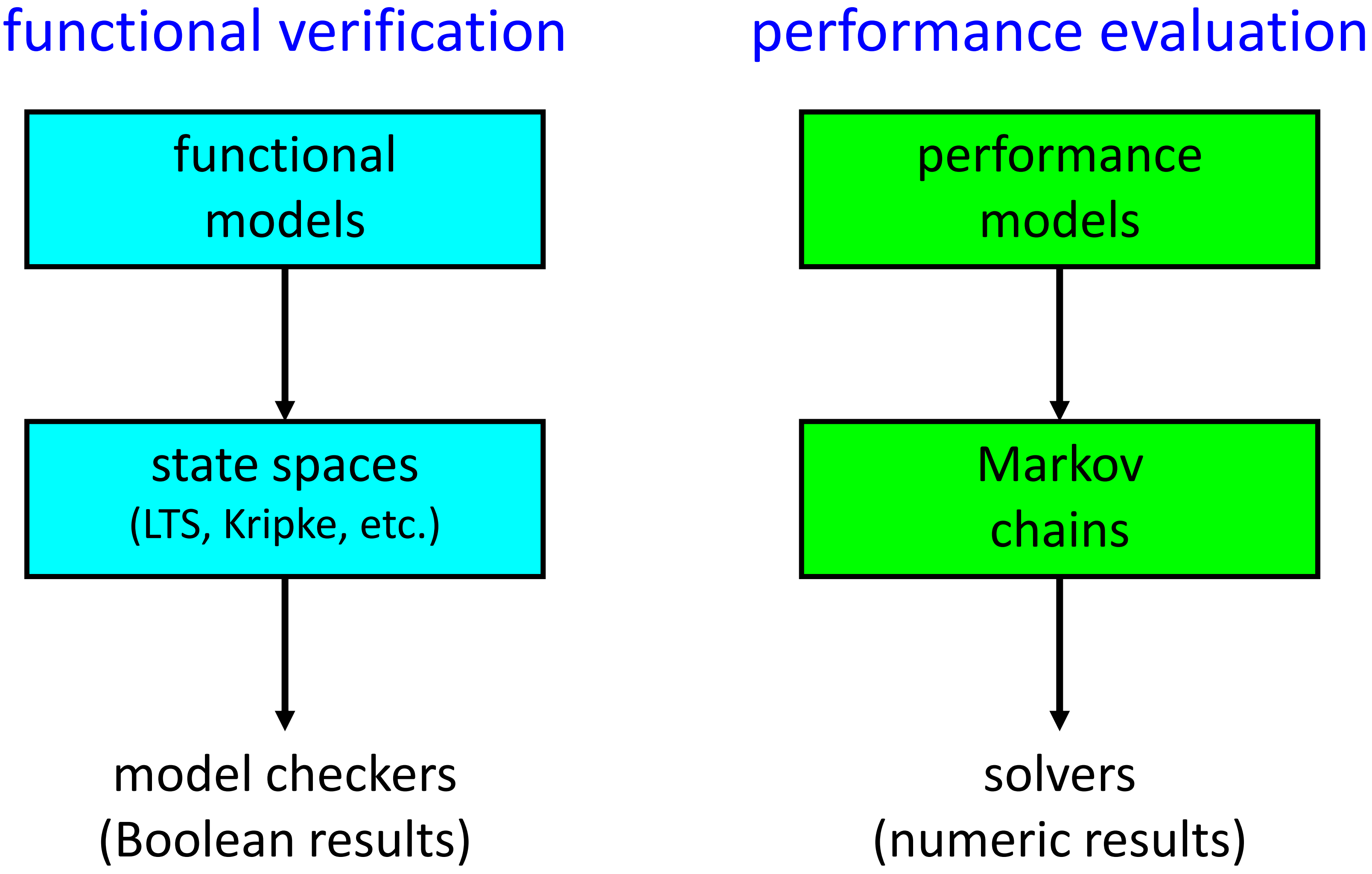
VI. PERFORMANCE EVALUATION

Performance evaluation

- Answer to *quantitative questions* such as:
  - Is the system efficient? *(performance estimation)*
  - Which probability for a failure? *(dependability)*

- Use *extended Markovian models* combining
  - **Functional models** specified in high-level languages (e.g., LOTOS or LNT)
  - **Performance data** based on Markov chains

The initial picture



functional verification

functional models

state spaces
(LTS, Kripke, etc.)

model checkers
(Boolean results)

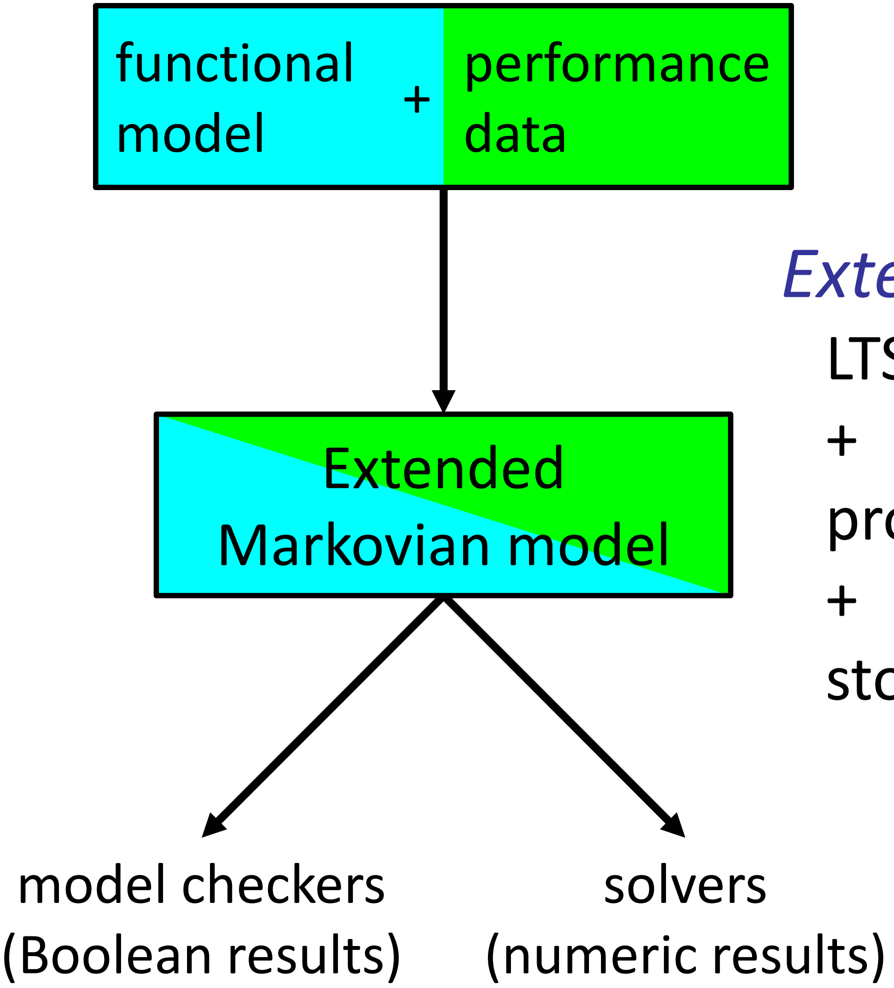
performance evaluation

performance models

Markov chains

solvers
(numeric results)

Extended Markovian models



functional model + performance data

Extended Markovian model

model checkers (Boolean results)

solvers (numeric results)

Extended Markovian model =

LTS (*Labeled Transition System*)

\+

probabilistic transitions ("prob 0.8")

\+

stochastic transitions ("rate 3.1")

BCG: supported Markovian transitions

- *ordinary transitions*

  a
- *stochastic transitions*

  "**rate** *r*" ($r \in R^+$)
- *labeled stochastic* transitions

  "*a*; **rate** *r*" ($r \in R^+$)
- *probabilistic transitions*

  "**prob** *p*" ($p \in]0, 1]$)
- *labeled probabilistic transitions*

  "*a*; **prob** *p*" ($p \in]0, 1]$)

Markovian models supported by CADP

Model	LTS transitions	Stochastic transitions	Probabilistic transitions
LTS (*Labeled Transition System*)	✓	✗	✗
CTMC (*Continuous Time Markov Chain*)	✗	✓	✗
DTMC (*Discrete Time Markov Chain*)	✗	✗	✓
IMC (*Interactive Markov Chain*) [Hermanns 02]	✓	✓	✗
IPC (*Interactive Probabilistic Chain*) [Coste 10]	✓	✗	✓
Extended Markovian models [CADP]	✓	✓	✓

Models subsumed by CADP's extended Markovian models (among others)

Performance evaluation techniques

- Technique #1:
  - Generation of a Markovian model
  - Analysis using a Markovian solver

  State explosion sometimes occurs!

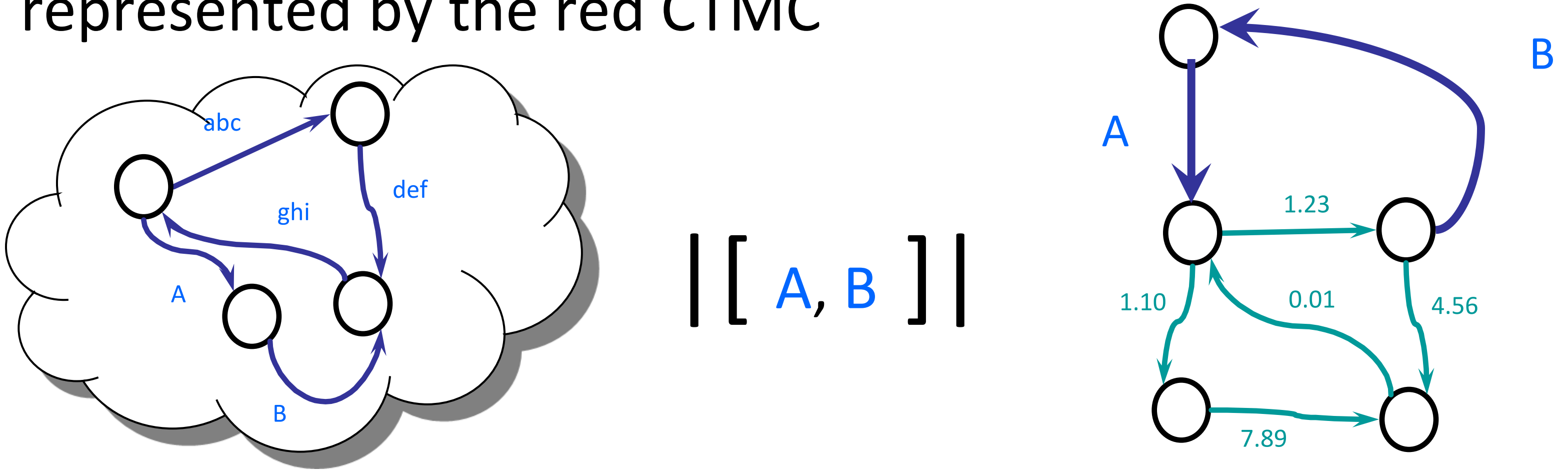
- Technique #2:
  - Random simulation and on-the-fly analysis

VI.1 MARKOVIAN MODEL GENERATION TOOLS

High-level Markovian models

- Functional model (e.g. in LNT)
- Two ways to model performance aspects
  - Symbolic rate transitions with ordinary labels, later on instantiated (i.e., renamed) with actual rates
  - Constraint-oriented compositional delay insertion

    Example: insert between successive actions A and B a delay represented by the red CTMC



MCS queue lock: delay insertion (1/2)

compositionnal delay-insertion between operations

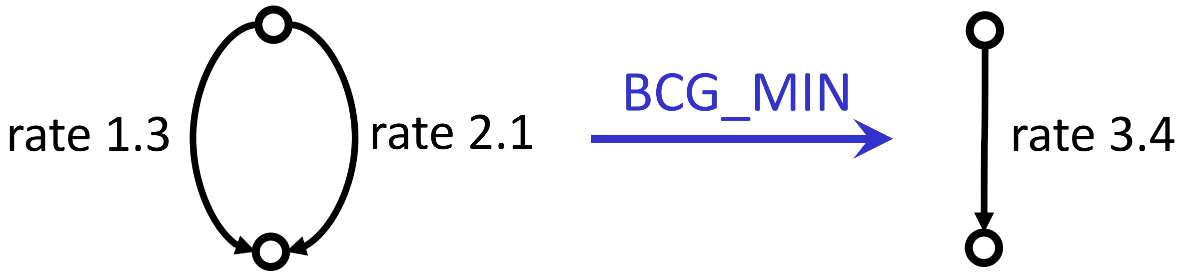
```
process Main [NCS, CS_Enter, CS_Leave: Resource_Access,
              L: Lock_Access, M: Memory_Access,
              Lambda, Mu, Nu: Latency]
is
  par NCS, CS_Enter, CS_Leave, L, M in
    Protocol [NCS, CS_Enter, CS_Leave, L, M]
  ||
    Latency [NCS, CS_Enter, CS_Leave, L, M, Lambda, Mu, Nu]
  end par
end process
```

MCS queue lock: delay insertion (2/2)

```
process Latency [NCS, CS_Enter, CS_Leave: Resource_Access,
                 L: Lock_Access, M: Memory_Access,
                 Lambda, Mu, Nu: Latency] is
 var pid: Pid, op: Operation in
   loop select
      NCS (?pid); Lambda (pid)
     [] L (?op, ?any Index, ?any Index, ?any Pid); Mu (op)
     [] L (?op, ?any Index, ?any Index, ?any Bool, ?any Pid); Mu (op)
     [] M (?op, ?any Pid, ?any Index, ?any Pid); Mu (op)
     [] M (?op, ?any Pid, ?any Bool, ?any Pid); Mu (op)
     [] CS_Enter (?pid); Nu (pid)
     [] CS_Leave (?any Pid) -- no delay
   end select end loop
end var end process
```

Extensions of EXP.OPEN and BCG_MIN

- BCG_MIN:
  - stochastic and probabilistic equivalences: strong and branching bisimulation + lumpability



  - recent improvements (for extended Markovian models):
    - 500 times faster and 4 times less memory than BCG_MIN 1.0
    - minimization of graphs up to 10^7 states and 10^8 transitions
- EXP.OPEN:
  - parallel composition of extended Markovian models
  - no synchronization on "**rate**"/"**prob**" transitions
  - on-the-fly reduction for stochastic and probabilistic equivalences

DETERMINATOR

- On-the-fly Markov chain generation
  - local transformations to remove stochastic non-determinism
  - determinacy check ("well specified" stochastic process)
  - algorithm: variant of [Deavours-Sanders-99]
- Input:
  - On-the-fly extended Markovian model
- Output:
  - either BCG graph (extended CTMC)
  - or an error message

VI.2 NUMERICAL ANALYSIS OF EXTENDED MARKOVIAN MODELS

BCG_TRANSIENT

- Numerical solver for Markov chains
- Transient analysis
- Inputs:
  - Extended Markovian model in the BCG format
  - List of time instants
- Outputs:
  - Numerical data usable by Excel, Gnuplot…
- Method:
  - BCG graph converted into a sparse matrix
  - Uniformisation method to compute Poisson probabilities
  - *Fox-Glynn* algorithm [Stewart-94]

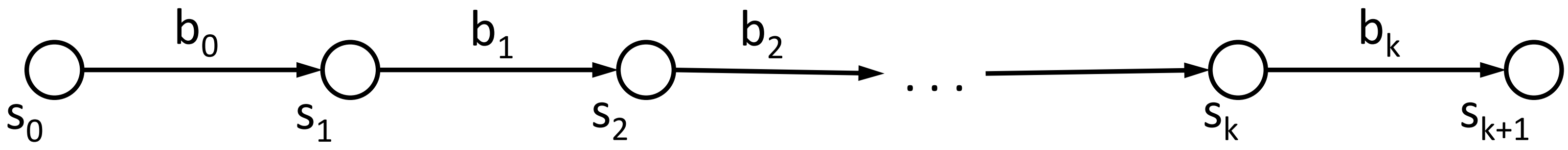
BCG_STEADY

- Numerical solver for Markov chains
- Steady-state analysis (equilibrium)
- Inputs:
  - Extended Markovian model in the BCG format
  - No deadlock allowed
- Outputs:
  - Numerical data usable by Excel, Gnuplot…
- Method:
  - BCG graph converted into a sparse matrix
  - Computation of a probabilistic vector solution
  - Iterative algorithm using *Gauss-Seidel* [Stewart-94]

VI.3 ON-THE-FLY SIMULATION OF EXTENDED MARKOVIAN MODELS

CUNCTATOR

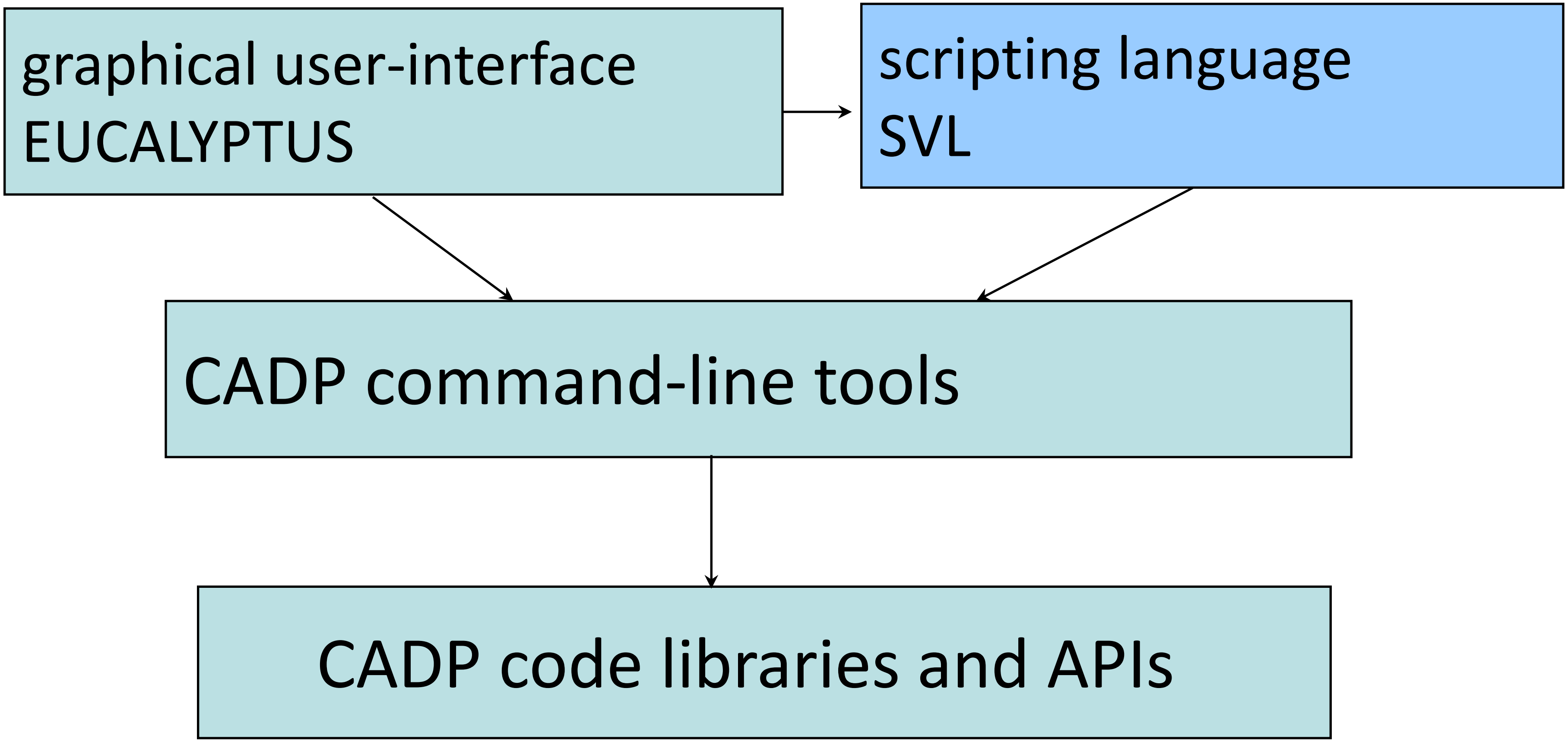
- A steady-state random simulator for IMCs
- On-the-fly label hiding and renaming to produce a (labeled) CTMC with internal actions
- On-the-fly exploration of a sequence:



- Compute the throughput of each stochastic action “*a*; rate *r*”
- Different scheduling strategies for internal acions
- Save/restore context of simulation
- Caching of internal sequences of transitions

VII. SVL (*SCRIPT VERIFICATION LANGUAGE*)

Interface: Graphics vs Scripts



graphical user-interface
EUCALYPTUS

scripting language
SVL

CADP command-line tools

CADP code libraries and APIs

Why Scripting ?

- Verification scenarios can be complex
- They can be repetitive
- Many objects/formats to handle:
  - High-level process descriptions (e.g., LNT, FSP, LOTOS)
  - Networks of communicating LTSs
  - Explicit and implicit LTSs
- Many operations to perform:
  - LTS generation of a process, a network of LTSs
  - Label hiding, label renaming
  - LTS minimization/comparison modulo equivalences
  - Verification (deadlock, livelock, temporal logic formula)
- Various verification techniques:
  - enumerative, on-the-fly, compositional, etc.

What is SVL?

- An acronym: *Script Verification Language*
- A language for describing (compositional) verification scenarios
- A compiler (SVL 2.1) for executing scenarios writen in this language
- A software component of CADP

SVL Components

Two types of components can be mixed

- SVL verification statements (written *S*)
  - Compute and store an LTS or network of LTSs in a file
  - Verify temporal properties
  - Compare LTSs, etc.
- Bourne shell constructs (lines starting with **%**)
  - Variables, functions, conditionals, loops, …
  - All Unix commands

SVL Behaviours

- Algebraic expressions used in statements
- Several operators
  - Parallel composition
  - LTS generation and minimization
  - Label hiding and renaming, etc.
- Several types of behaviours
  - LTSs (several formats)
  - Networks of communicating LTSs
  - LNT, LOTOS, and FSP descriptions
  - Particular processes in LNT, LOTOS, and FSP descriptions

Abstract Syntax of Behaviours

B ::= *"F.bcg"* | *"F.aut"* | *"F.seq"* | *"F.exp"*
| *"F.lnt"* | *"F.lnt"* : P [$G_1, \ldots, G_n$]
| *"F.lotos"* | *"F.lotos"* : P [$G_1, \ldots, G_n$]
| *"F.lts"* | *"F.lts"* : P [$G_1, \ldots, G_n$]
| $B_1 \;|[G_1, \ldots, G_n]|\; B_2 \mid B_1 \;|||\; B_2 \mid B_1 \;||\; B_2$
| **par** $G_1, \ldots, G_n$ **in**
$[G_{0,1}, \ldots, G_{0,m1} \rightarrow]\; B_0 \;||\; \ldots \;||\; [G_{p,1}, \ldots, G_{p,mp} \rightarrow]\; B_p$ **end par**
| **generation of** B_0
| R **reduction** [***with*** T] **of** B_0
| [S] **hide** [**all but**] $L_1, \ldots, L_n$ **in** B_0
| [S] **rename** $L_1 \rightarrow L_1', \ldots, L_n \rightarrow L_n'$ **in** B_0
| [**user**] **abstraction** B_1 [**sync** $G_1, \ldots, G_n$] **of** B_2

Explicit LTSs

- States and transitions listed exhaustively
- LTSs in several formats

B ::= *"F.bcg"*	Binary Coded Graphs
| *"F.aut"*	Aldébaran ASCII format
| *"F.seq"*	Set of traces

- Format conversions are fully automatic

Implicit LTSs

- LNT, LOTOS, or FSP descriptions (*"F.lnt", "F.lotos", "F.lts"*)
- Particular LNT, LOTOS, or FSP processes (*"F.lnt" : P [G_1, ..., G_n], ...*)
- Networks of communicating automata (*"F.exp"*)

Explicit vs Implicit LTSs

SVL principles:

- Keep LTSs implicit as long as possible
  - Explicit LTS generation is expensive (state explosion)
  - Not all properties necessitate to explore the whole LTS
- Explicit LTS generation is done only if required explicitly by the user

LTS Generation

Conversion from an implicit LTS to an explicit LTS

B ::= **generation of** B_0

Examples

- **generation of** *"spec.lnt"*
  Use LNT.OPEN and GENERATOR
- **generation of** *"spec.lnt" : P [G]*
  Use LNT.OPEN (option –root) and GENERATOR
- **generation of** *"spec.exp"*
  Use EXP.OPEN and GENERATOR
- **generation of par** G_1 **in** "$spec_1$.bcg" || "$spec_2$.aut" **end par**
  Use EXP.OPEN and Generator

Parallel Composition

$$\left.\begin{array}{l} B \; ::= \; B_1 \;\; |[G_1, \ldots, G_n]| \;\; B_2 \\ \qquad \mid \;\; B_1 \; ||| \; B_2 \;\; \mid \;\; B_1 \; || \; B_2 \end{array}\right\} \text{LOTOS}$$

$$\left.\begin{array}{l} \qquad \mid \;\; \textbf{par } G_1, \ldots, G_n \textbf{ in } [G_{0,1}, \ldots, G_{0,m0} \rightarrow] \; B_0 \\ \qquad\quad || \ldots || \; [G_{p,1}, \ldots, G_{p,mp} \rightarrow] \; B_p \end{array}\right\} \text{LNT}$$

$$\qquad\quad \textbf{end par}$$

- LOTOS and LNT operators
- B_1, B_2, ... can be LTSs, but also any SVL behaviour
- Generation of intermediate EXP.OPEN files

Label Hiding

B ::= [M] **hide** $L_1, \ldots, L_n$ **in** B_0
| [M] **hide all but** $L_1, \ldots, L_n$ **in** B_0

- An extension of LOTOS hiding, where
  - *L* is either
    a gate name
    a label string (e.g. "G !3.14 !TRUE")
    a regular expression (e.g. "G !.* !TRUE")
  - *M* ::= **gate** | **total** | **partial** is a *matching semantics* for regular expressions
  - **all but** means complementation of the set of labels
- Tools used: BCG_LABELS or EXP.OPEN

Label Hiding: Examples

[**gate**] **hide** *G, H* **in** *"test.bcg"*

invokes BCG_LABELS (-hide) and returns an LTS in which labels whose gate is G or H are hidden

total hide *"G ![AB].*"* **in** *"test.bcg"*

invokes BCG_LABELS and returns an LTS in which labels matching "G ![AB].*" are hidden

partial hide *G* **in** *"test.bcg"*

invokes BCG_LABELS and returns an LTS in which labels containing G are hidden

Label Renaming

$$B ::= [M]\ \textbf{rename}\ L_1 \rightarrow L_1', \ldots, L_n \rightarrow L_n'\ \textbf{in}\ B_0$$

where

- each $L \rightarrow L'$ is a Unix-like substitution containing regular expressions
- M is a matching semantics

$$M ::= \textbf{gate} \mid \textbf{total} \mid \textbf{single} \mid \textbf{multiple}$$

- Tools used: BCG_LABELS or EXP.OPEN

Label Renaming: Examples

[**gate**] **rename** *G* → *H, H* → *G* **in** *"test.bcg"*

invokes BCG_LABELS (-rename) and returns LTS

in which gate G is renamed into H and H into G

total rename *"G !A !TRUE"* → *"A_TRUE"* **in** *"test.bcg"*

invokes BCG_LABELS and returns an LTS in which
label "G !A !TRUE" is renamed into A_TRUE

total rename *"G !\(.*\) !\(.*\)"* → *"G \2 \1"* **in** *"test.bcg"*

invokes BCG_LABELS and returns an LTS in which
offers of labels whose gate is G are swapped

Reduction (Minimization)

LTS Minimization modulo an equivalence relation

B ::= R **reduction** [**with** T] **of** B_0

- Several relations R

  [**probabilistic**|**stochastic**] **strong**, **branching**, **safety**, **tau*.a**, **(weak) trace**, **tau-confluence**, **tau-compression**, **tau-divergence**, etc.

- Several tools T

  bcg_min, **reductor**

- Tools used: BCG_MIN or REDUCTOR

Reduction: Examples

- **strong reduction of** *"test.bcg"* [**with** *bcg_min*] invokes BCG_MIN (default tool for strong bisimulation) and returns an LTS minimized for strong bisimulation
- **stochastic branching reduction of** *"test.bcg"* invokes BCG_MIN (default tool for branching bisimulation) and returns an LTS minimized for stochastic branching bisimulation
- **trace reduction of** *"test.bcg"* [**with** *reductor*] invokes BCG_OPEN/REDUCTOR and returns an LTS minimized for trace equivalence

Abstraction

- LTS generation of B_2 abstracted w.r.t. interface B_1

  B ::= **abstraction** B_1 **of** B_2

  | **user abstraction** B_1 **of** B_2

- Equivalent syntax

  B ::= B_2 -|| B_1

  | B_2 -||? B_1

  where ? has the same meaning as *user*

- Invokes PROJECTOR
- Detailed in Section on Compositional verification (later)

Other operators

- Priorities between transitions (invokes EXP.OPEN)
- Transition cutting (invokes EXP.OPEN)
- Particular automata (invokes BCG_GRAPH):
  - stop (empty automaton)
  - chaos automaton (parameterized by a set of labels)
  - FIFO or bag buffer (parameterized by a size and receive/send sets of labels)

Abstract Syntax of Statements

S ::= "*F.E*" = B_0

| "*F.E*" = *R* **comparison** B_1 [== | <= | >=] B_2

| "*F.E*" = **deadlock** [**with** *T*] **of** B_0

| "*F.E*" = **livelock** [**with** *T*] **of** B_0

| ["F_1.*E*" =] **verify** "F_2.*mcl*" **in** B_0

Assignment Statement

S ::= *"F.E"* = B_0

- *Computes* B_0 and stores it in file *"F.E"*
- Extension *E* tells the format for *"F.E"*
  (*aut*, *bcg*, *exp*, *or seq*, *but not lnt*, *lotos*, *lts*)
- Principles:
  - Format conversions are implicit (BCG_IO)

    e.g. *"spec.bcg" = "spec.aut"* is permitted
  - No implicit LTS generation

    If *E* is an explicit LTS format (i.e. all but *exp*)
    then B_0 must not denote an implicit LTS
    ⇒ **generation** must be used explicitly (otherwise a warning is issued)

Comparison of Behaviours

S ::= "*F.E*" = R **comparison** B_1 == B_2
| "*F.E*" = R **comparison** B_1 <= B_2
| "*F.E*" = R **comparison** B_1 >= B_2

- Compares B_1 and B_2 and stores the distinguishing path(s) (if any) in "*F.E*"
- Equivalence or preorders
- Several relations R
- Invokes BISIMULATOR

Deadlock and Livelock Checking

S ::= *"F.E"* = **deadlock** [**with** *T*] **of** B_0

| *"F.E"* = **livelock** [**with** *T*] **of** B_0

- Detects deadlocks or livelocks using tool *T* (*exhibitor* or *evaluator*)
- Results in a (set of) paths leading to deadlock or livelock states (if any) and stored in *"F.E"*
- Verification may be on-the-fly (EXHIBITOR or EVALUATOR with OPEN/CÆSAR)

Temporal Property Verification

S ::= ["F_1.*E*" =] **verify** "F_2.*mcl*" **in** B_0

- Checks whether B_0 satisfies the temporal logic property contained in "F_2.*mcl*"
- May generate a diagnostic and store it in "F_1.*E*" (example or counter-example which explains the resulting truth value)
- Verification may be on-the-fly (OPEN/CAESAR and EVALUATOR)

Shell Constructs in SVL Scripts

Shell commands can be inserted (%)

- Direct call to Unix commands (“echo”...)
- Setting of SVL shell variables
  - *% DEFAULT_REDUCTION_RELATION=branching*
  - *% GENERATOR_OPTIONS=-monitor*
- Enables the use of all shell control structures
  - "if-then-else" conditional
  - "for" loop
  - function definitions
  - etc.

Compositional Verification (key features)

- Support for basic compositional verification
  Example: alternating bit protocol
- Script Simplification using meta-operations
- Support for refined compositional verification
  Example: rel/REL protocol
- Support for smart heuristics
- Compositional Performance Evaluation
  Examples: SCSI-2 and Mutual Exclusion Protocols

Meta-operations

B ::= **leaf** R **reduction** [**with** T] **of** B_0
| **root leaf** R **reduction** [**with** T] **of** B_0
| **node** R **reduction** [**with** T] **of** B_0

- Three "static" compositional verification strategies:
  - Reduction of LTSs at the leaves of parallel compositions in B_0
  - Reduction of LTSs at the leaves of parallel composition in B_0 and then reduction of the whole behaviour
  - Reduction at every node of B_0
- Meta-operations expand to basic SVL behaviours

The Abstraction Behaviour

- Implements refined compositional verification
- The LTS of a behaviour *B* may be larger than the LTS of a behaviour containing *B* because of *context constraints*
- Example

```
par G in
        par in "User1.bcg" || "User2.bcg" end par
||      "Medium.bcg"
end par
```

  "Medium.bcg" may constrain the interleaving
- Restrict the interleaving using abstraction:

```
par in "User1.bcg" || "User2.bcg" end par
-|[G]| "Medium.bcg"
```

Smart heuristics

B ::= **smart** *R* **reduction** [**with** *T*] **of** B_0

- Compositional verification strategy determined by a metric on B_0
- Incrementally select the subset of concurrent processes to compose and minimize, that:
  - yield as much internal transitions as possible (likely eliminated by reduction) and
  - are as tightly coupled as possible (less interleaving)
- Necessarily approximate
  - the heuristics consider both reachable and unreachable transitions
- Most often: good results, especially on large networks

SVL example: verification of MCS

```
% DEFAULT_PROCESS_FILE="mcs.lnt"
% DEFAULT_SMART_LIMIT=7

"mcs.bcg" = smart branching reduction of
  hide all but CS_ENTER, CS_LEAVE in
    par M, L in
        par in P1 || P2 || P3 || P4 || P5 end par
    ||
        par in Lock || Memory end par
    end par;

"mcs_diag_branching.bcg" = branching comparison
    "mcs.bcg" == Service;
```

VIII. CONCLUSION

Further features of CADP

- Cosimulation and rapid prototyping (EXEC/CÆSAR framework)
- Test generation (TGV)
- XTL query language on BCG graphs
- Distributed BES resolution (work in progress)

Distribution of CADP

- Commercial license for industrial users
- Free distribution to academic users
  - Until July 2011:
    - signed paper contract with the academic organization
    - one license per machine
  - Since July 2011:
    - personal license for each CADP user, authenticated by valid academic email address and academic web page
    - license terms available in French and in English
- http://cadp.inria.fr/registration

Some figures about CADP

- Wide dissemination
  - ≥ 441 academic license contracts
  - CADP installed on 613 machines in 2011
  - ≥ 139 published case studies using CADP since 1990 (http://cadp.inria.fr/case-studies)
  - ≥ 57 third-party tools connected to CADP since 1996 (http://cadp.inria.fr/software)
  - ≥ 196 users and ≥ 1300 messages in the CADP forum since 2007 (http://cadp.inria.fr/forum.html)
- Various supported architectures
  - processors: Itanium, PowerPC, Sparc, x86, x64
  - operating systems: Linux, MacOS X, Solaris, Windows
  - C compilers: gcc3, gcc4, Intel, Sun
- Significant testing effort (Contributor tool)

A promising future

- Ubiquitous concurrency
  - Hardware: multi-/many-core CPUs, clusters, grids, clouds
  - Software: concurrency required to exploit new hardware
- Industry awareness
  - Increasing need for hardware and software reliability
  - Models (even non-formal) become standard practice
- "Applied concurrency" starts being effective

For more information...

- CADP Web site:

  http://cadp.inria.fr

- CADP forum:

  http://cadp.inria.fr/forum.html
  http://cadp.forumotion.com

- CADP on-line manual pages:

  http://cadp.inria.fr/man