IF Tutorial

Marius BOZGA   Susanne GRAF

Iulian OBER   Laurent MOUNIER

VERIMAG

Distributed and Complex Systems Group

www-verimag.imag.fr/PEOPLE/async/IF/

model based development

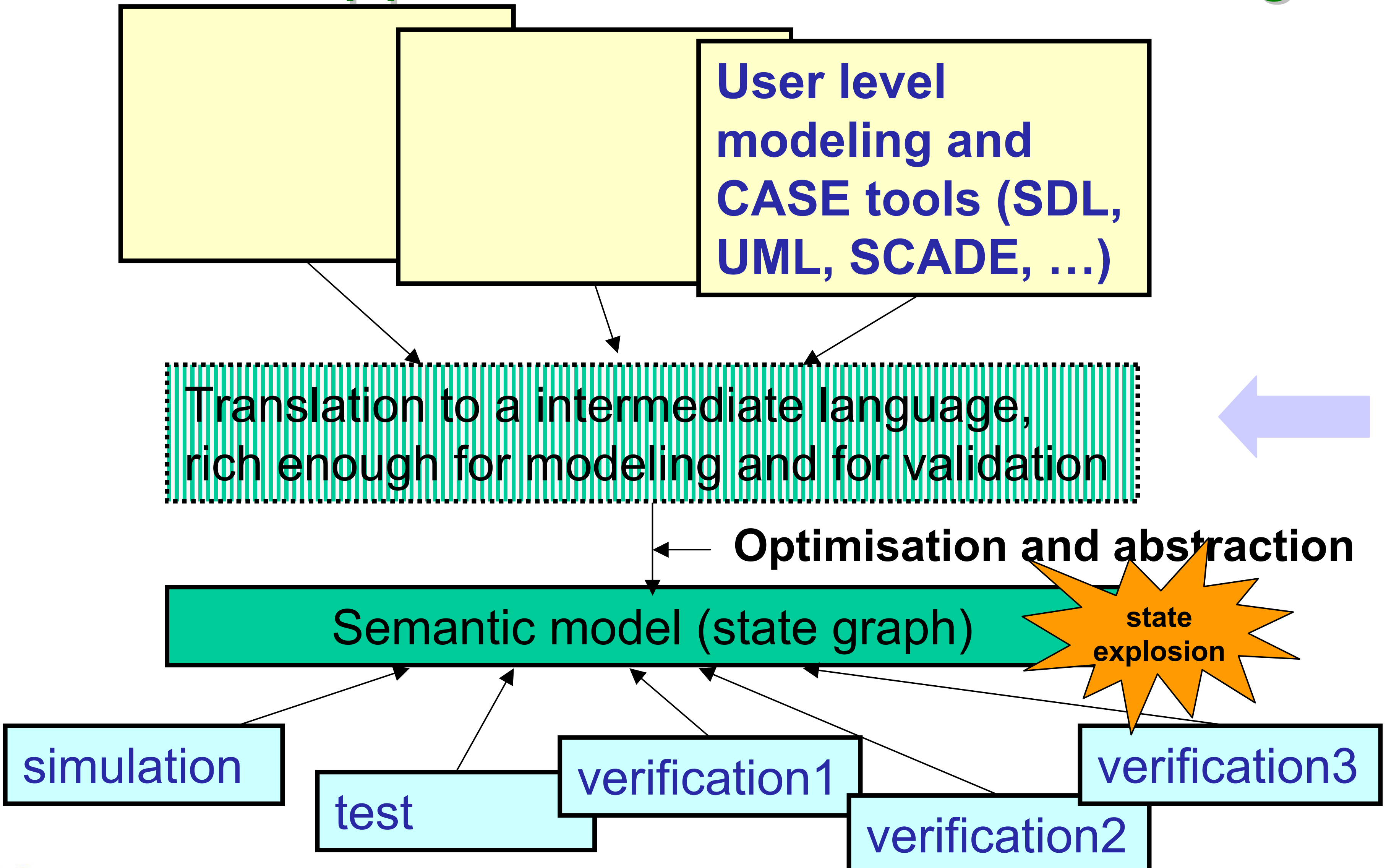
Goal

Early detection of problems, concerning functional and non functional aspects ➔ model-based simulation, testing and verification

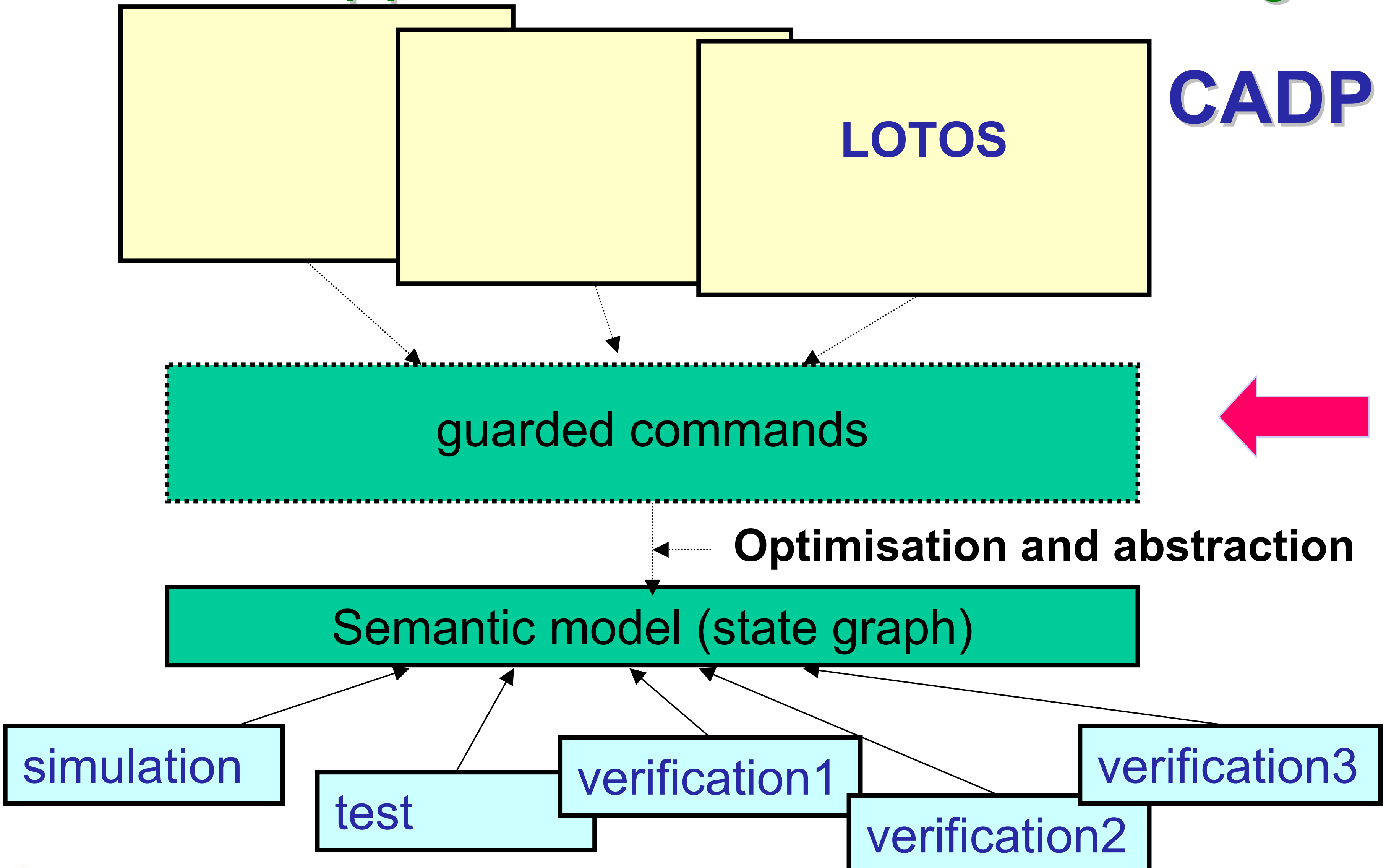
Context

Telecommunication protocols,
Real-time embedded systems,
Distributed systems,
Scheduling problems,…

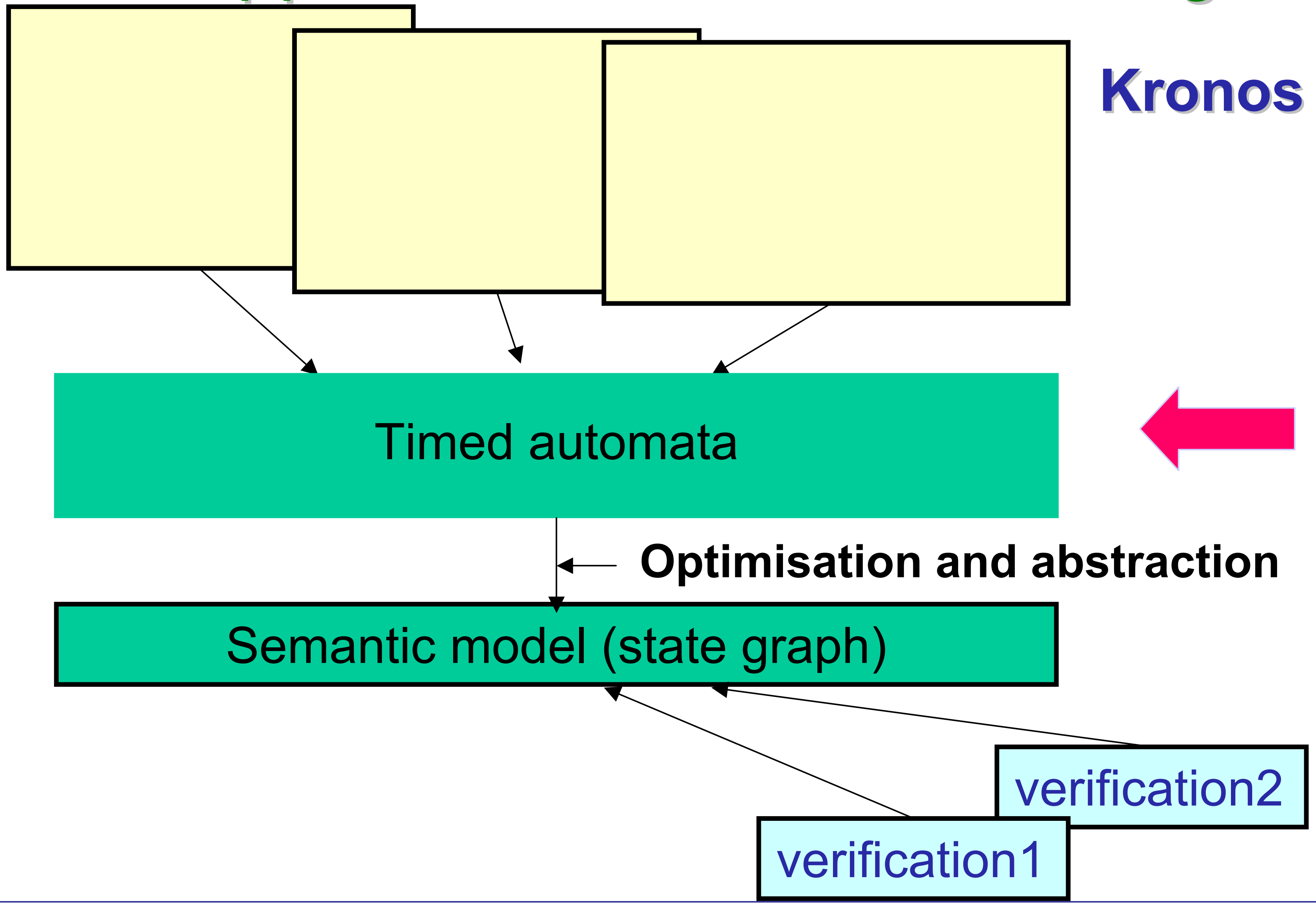
approach: build on the existing



approach: build on the existing



approach: build on the existing



challenge

A good intermediate representation

- Sufficient *expressiveness*: allows to map concepts of diverse modeling languages (asynchronous, synchronous, timing,…)
- Enough *concepts*: structured representation of
  - Concepts existing in validation tools
  - Concepts exploitable for more efficient validation
- Allows *semantic fine tuning*: allows expression of alternative options of semantic variation points: time progress, execution and interaction modes,…

overview

- Motivation and challenge
- IF: the language concepts
  - Functional aspects
  - Non-functional aspects
- IF: the toolset
  - Core components
  - Model-based validation
  - Front-end tools
- Demos
- Case studies
- Perspectives

perspectives

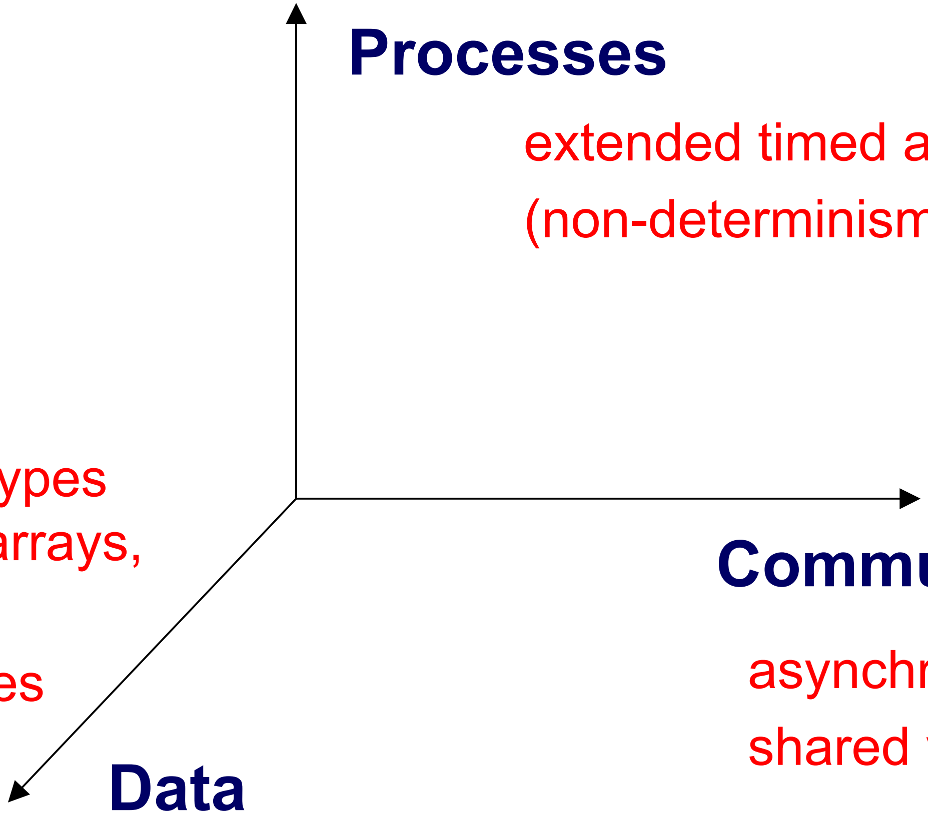
- UML-based methodology for real-time systems
  - component-based modeling
  - combination asynchronous and synchronous systems
  - relate functional and non-functional aspects
- improve verification and test generation methods
  - more static analysis, abstraction and constraint propagation
  - more compositional verification methods
  - better diagnostics facilities
- more connections
  - connections with performance evaluation tools

The IF Language

Functional Part

IF Specification

System description : 3 axes



Processes

extended timed automata

(non-determinism, dynamic creation)

IF

predefined data types
(basic types, arrays, records)

abstract data types

Communications

asynchronous channels

shared variables

Data

execution model

- A process instance:
  - executes asynchronously with other instances
  - can be dynamically created
  - owns local data (public or private)
  - owns a private FIFO buffer

- Inter-process communications:
  - asynchronous signal exchanges (directly or via signalroutes)
  - shared variables

$\Rightarrow$ semantics can be expressed by an (infinite) LTS

system structure

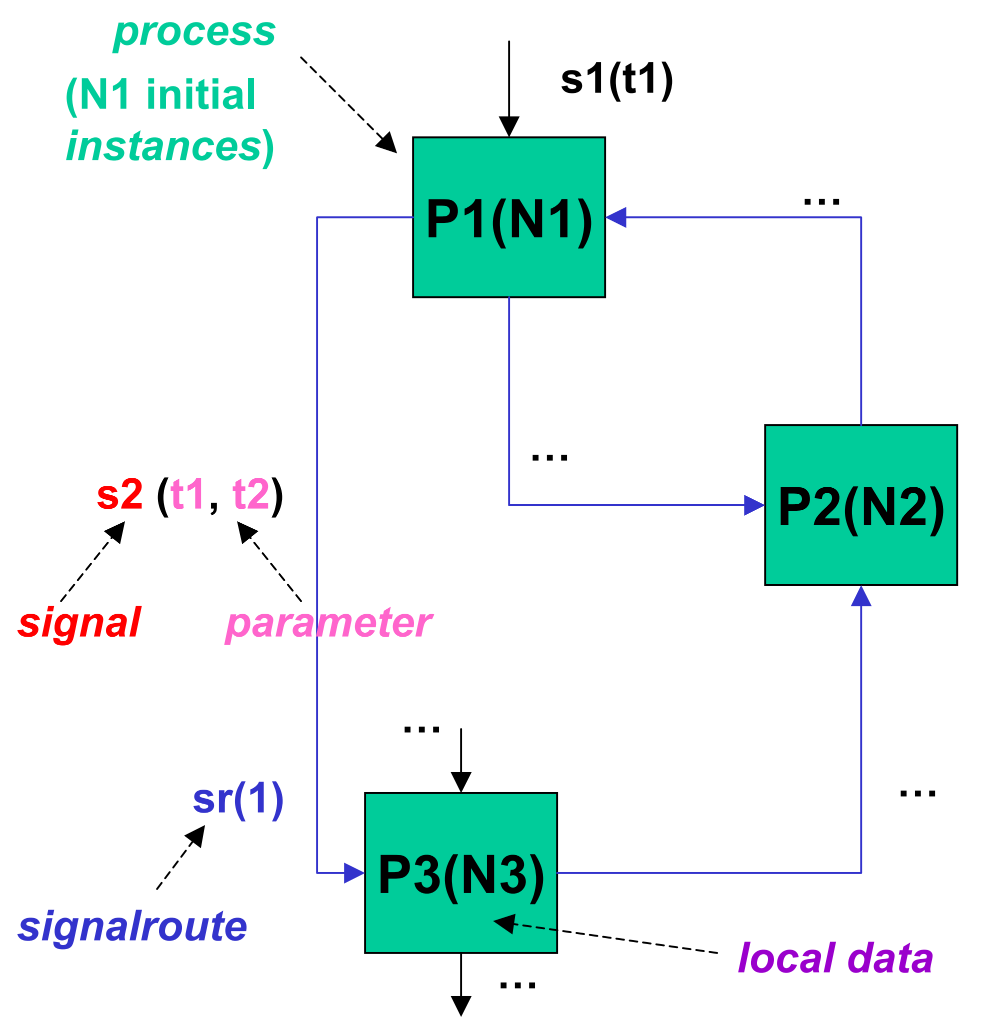
const N1 **=** ... ; // constants
type t1 **=** ... ; // types

signal s2(t1, t2), // signals

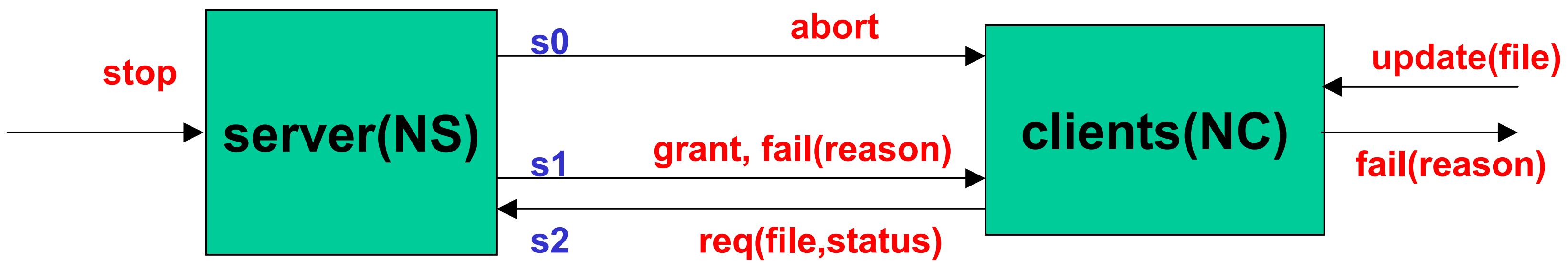
// signalroutes
signalroute sr1(1) ... // route attributes
from P1 to P3

// processes
process P1(N0)
... // data + behaviour
endprocess;

...
process P3(N3)
...
endprocess;



example



const NS= … , NC= … **;**
type file= … , status= … , reason= … ;

signal stop(), req(file, status), fail(reason), grant(), abort(), update(data);

signalroute s0(1) **#multicast**
    from server **to** clients **with** abort;
signalroute s1(1) **#unicast #lossy**
    from server **to** clients **with** grant,fail;
signalroute s2(1) **#unicast**
    from clients **to** server **with** req;

process server(NS) … **endprocess;**
process clients(NC) … **endprocess;**

process

IF processes = timed, hierarchical, finite-state automata with actions

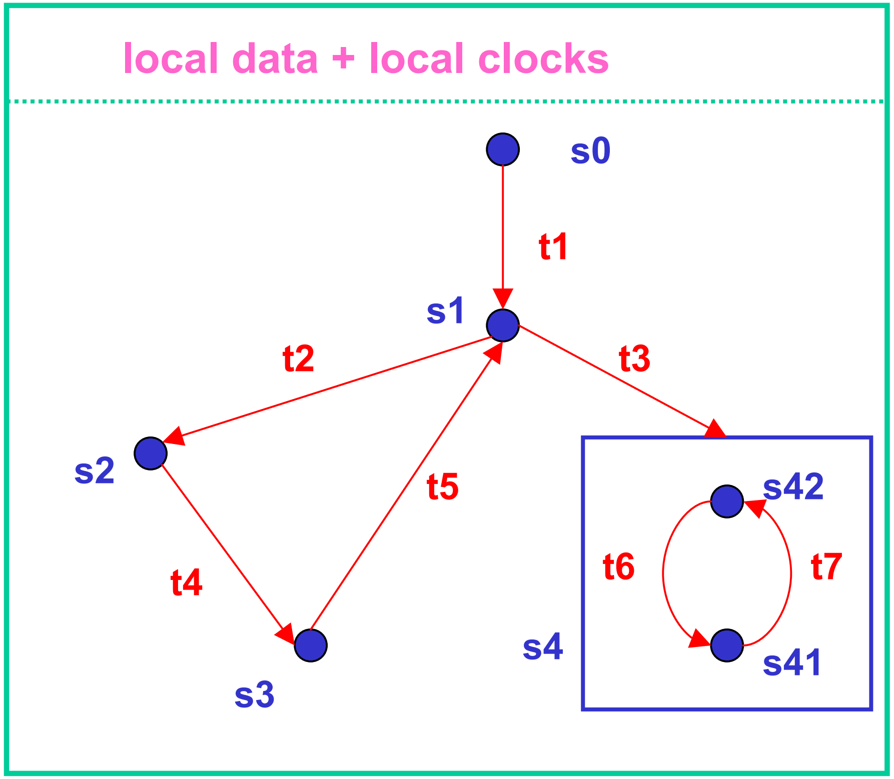
```
process P1(N1);
fpar ... ;

// types, variables, constants, procedures

state s0 ... ;
        ...  // transition t1
endstate;

state s1 ...;
        ... // transitions t2, t3
endstate;

  ...   // states s2, s3, s4
endprocess;
```

parameters

local data

state

outgoing transitions



P1(N1)

state

state id

attributes

```
state waiting #start
save abort, update;
        ...
        ...
        ...
endstate;
```

deferred signals set

outgoing transitions

nested sub-states

the consumption of these signals
Is (temporarily) postponed
(to control the queuing policy)

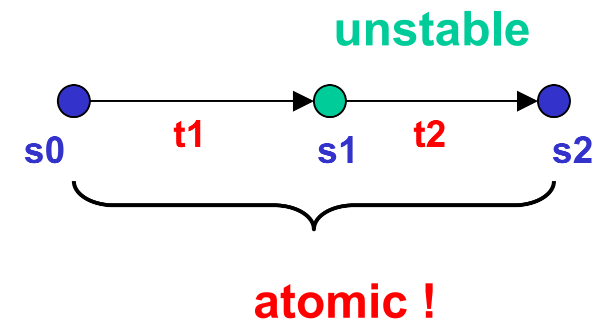
attributes:

- #start
- #stable | #unstable

interleaving between processes can happen only on #stable states
(to control transition atomicity)

transition

transition = *deadline* + optional trigger + statement list



t1

```
state s0
...

deadline eager
provided x!=10;
when c2 >= 4;
input update(m);
        ....
nextstate s1;
...
endstate;
```

- *deadline*
- discrete guard
- *timed guard*
- signal consumption from the process buffer

discrete guard + timed guard + signal consumption = **trigger**

statement list

statement = data assignment
message emission,
process or signalroute creation or destruction, …

sequential. conditional, or iterative composition

types and data

Variables:

- are ***statically typed*** (but *explicit conversions* allowed: {t1}(x))
- can be declared ***public*** (= shared), or not …

Predefined basic types: integer, boolean, float, pid, *clock*

⊇ {self, nil}

Predefined type constructors:

- (integer) interval: **type** fileno = **range** 3..9;
- enumeration: **type** status= **enum** open, close **endenum**;
- array: **type** vector= **array**[12] **of** pid
- structure: **type** file = **record** f fileno; s status **endrecord**;

Abstract Data Type definition facilities …

signal exchange

Signal emission:

signal | parameters | pid expression

to a specific process: **output** req (3, open) **to** {server}(2);

signalroute

via a signalroute: **output** req(3, open) **via** {s0}(1);

mixed: **output** token **via** {link}(1) **to** {client}(k+1)%N;

k={integer}(self) | pid expression

Signal consumption:

input req (f, s);

formal parameters

blocking if no req signal in top of the process buffer …

signal routes

signal route = process to process communication channel with ***attributes,***
can be ***dynamically*** created

route name

initial instance number

attributes

signal set

signalroute s1(1) **#unicast #lossy**

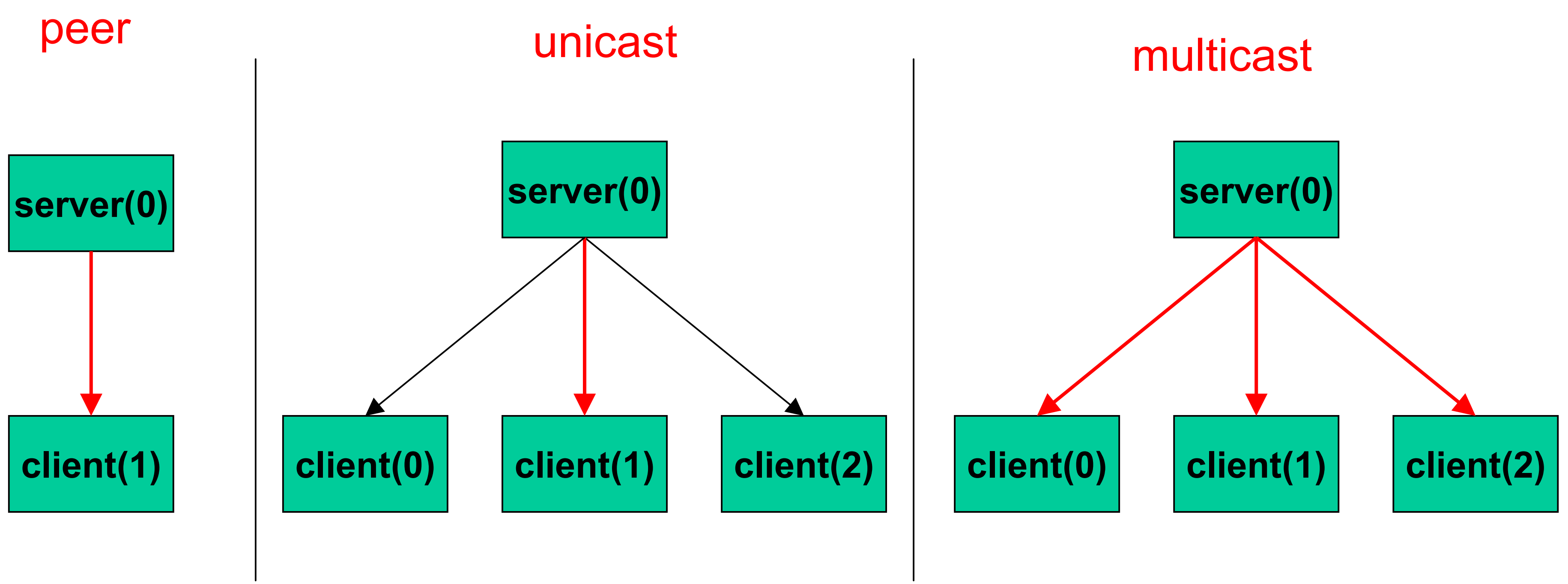
from server **to** clients **with** grant, fail;

endpoints

attributes:

- queuing policy: **fifo | multiset**
- reliability: **reliable | lossy**
- delivering policy: **peer | unicast | multicast**
- *delaying policy: urgent | delay[l,u] | rate[l,u]*

delivering policies

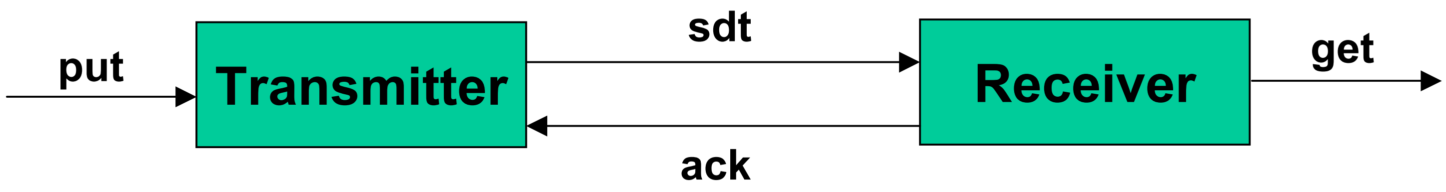


to one specific instance

to a randomly chosen instance

to all instances

example: ABP



type data = **range** 0 .. 3;

signal get(data), put(data), ack(boolean), sdt(data, boolean);

signalroute tr(1) **#unicast #lossy**
from transmitter **to** receiver **with** sdt;
signalroute rt(1) **#unicast #lossy**
from receiver **to** transmitter **with** ack;

process transmitter(1) … **endprocess**;
process receiver(1) … endprocess;

transmitter

```
process transmitter(1);

var t clock;
var b boolean;
var c boolean;
var m data;

state start #start ;
   task b := false;
    nextstate idle;
endstate;

state idle;
 input put(m);
   output sdt(m, b) via {tr}0;
   set t := 0;
    nextstate busy;
endstate;

state busy;
 input ack(c);
    nextstate q8;
 when t = 1;
   output sdt(m, b) via {tr}0;
   set t := 0;
    nextstate busy;
endstate;

state q8 #unstable ;
 provided c = b;
   task b := not b;
   reset t;
    nextstate idle;
 provided c <> b;
    nextstate busy;
endstate;
endprocess;
```

local data

initialization

message transmission

ack recepion

timeout: retransmission

incorrect ack

correct ack

receiver

```
process receiver(1);

var b boolean;
var c boolean;
var m data;

state start #start ;
  task b := false;
   nextstate idle;
endstate;
```

initialization

```
state idle;
 input sdt(m, c);
  if b = c then
   output ack(b) via {rt}0;
   output get(m);
   task b := not b;
  else
   output ack(not b) via {rt}0;
  endif
  nextstate idle;
endstate;

endprocess;
```

message reception

dynamic creation

- remote process/signalroute creation:

p := **fork** client (true)

parameters

pid of the newly created instance

process name

a new instance is created

- process/signalroute destruction:

kill {client} (2)

kill p

pid expression

the instance is destroyed, together with its buffer, and local data

- process termination:

stop

the "self" instance is destroyed, together with its buffer, and local data

example


```
const NS = …, Max = … ;
signal done(pid);
signal req ();

signalroute cs(1)
        from client to server with done;

process client(0);
 fpar parent pid;

state init #start ;
  informal "work";
  output done(self) via {cs}0 to parent;
   stop;
endstate;
endprocess;
```

a work is done

dies when work is finished

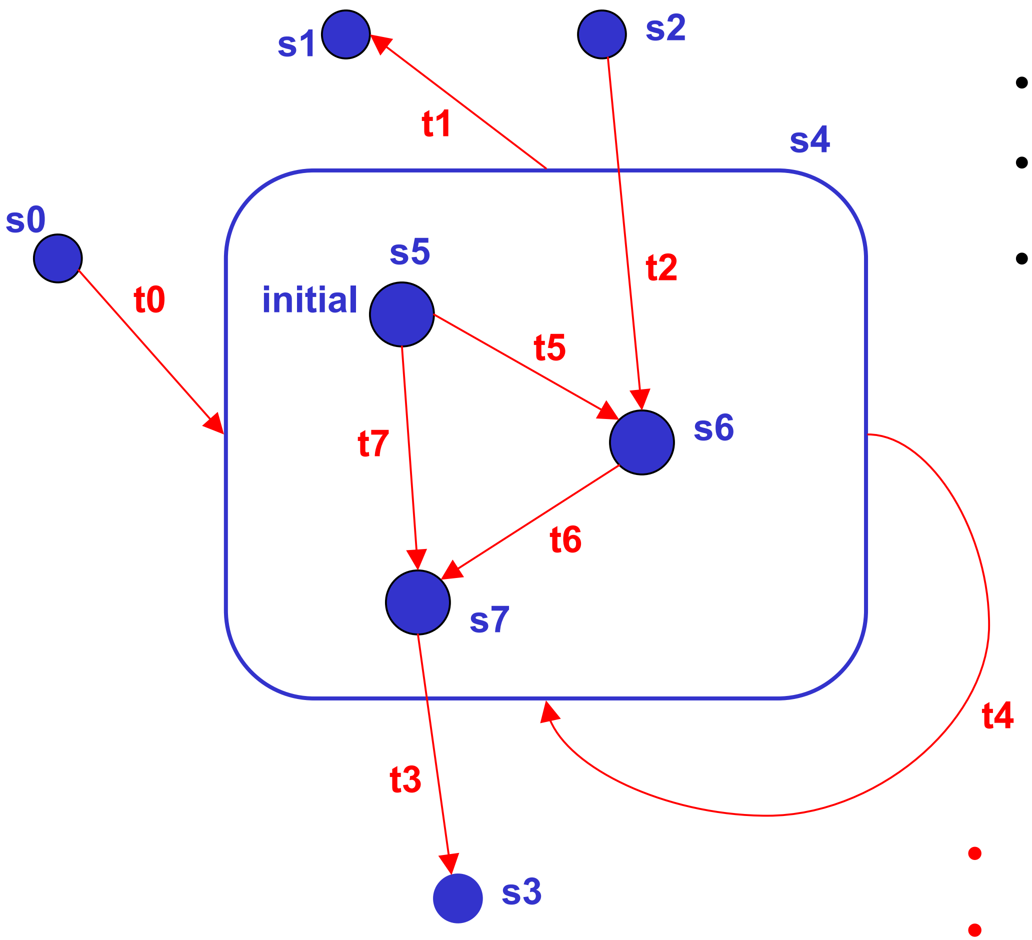
```
process server(1);
var i integer;
var x pid;

state idle #start ;
  provided (i < Max);
  input req ();
    x := fork client(self);
    task i := (i + 1);
      nextstate idle;
  input done(x);
    task i := (i - 1);
      nextstate idle;
endstate;
endprocess;
```

receives a new request

creates a new client

nested states

Several kinds of transitions …



- t0 = s0 → s5
- t1 = current_state → s1
- t4 = current_state → current_state
  or
  t4 = current_state -> s5

- no parallelism inside a state
- essentially a macro-notation

ADT

Use of Abstract Data Types:

```
type sqn = range 0.. N;
type sqnSet = abstract
  sqnSet Empty();
  sqnSet Insert(sqnSet, item);
  boolean isIn (sqnSet, item)
endabstract;
```

IF

```
#typedef unsigned if_sqn_set_type;
#define if_sqn_set_copy(x,y) (x)=(y)
#define if_sqn_set_compare(x,y) (x)-(y)
#define if_sqn_set_print(x,f) fprintf(f,"%#x",x)
#define if_sqn_set_reset(x) (x)=0

if_boolean_type if_isIn_function(if_sqn_set_type p1,if_integer_type p2)
        {return (p1 & (1 << p2)) ? if_boolean_true : if_boolean_false;}
if_sqn_set_type if_Insert_function(if_sqn_set_type p1,if_integer_type p2)
        { return p1 | (1 << p2);}
if_sqn_set_type if_Empty_function()
        { return 0;}
```

C/C++

At the IF level only the signature is required …

… but a concrete C/C++ implementation must be provided to use the simulation tools

external code

C++ procedures can be used to describe data transformations:

```
const NUSERS = 5, NFILES = 10;
type UserIdType = range 0 .. NUSERS;
type FileIdType = range 0 .. NFILES;
```

```
type SystemStatusType = array [NFILES] of FileControlBlockType;
type FileControlBlockType = array [NUSERS] of boolean;
var updating SystemStatusType;
```

```
procedure File_Available_For_Write;
fpar in f FileIdType, in u UserIdType, in systemStatus SystemStatusType;
  returns boolean;
{#              /* true if nobody is updating, maybe except u */
  int uprime, result=1 ;
for (uprime=0; uprime<if_NUSERS_constant; uprime++)
    result &= (uprime==u || ! updating[f][uprime]);
  return result;
#}
endprocedure;
```

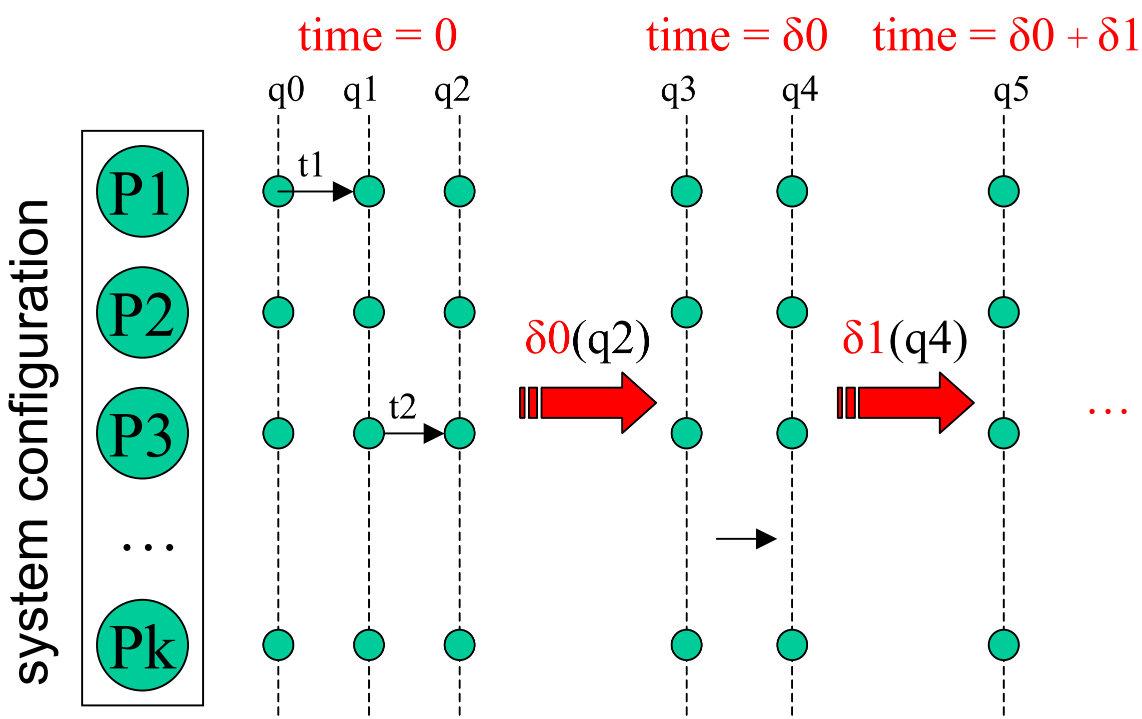
Checks if for all u'.
updating[u'][f]
implies u'<>u

The IF Language

Non-functional Part

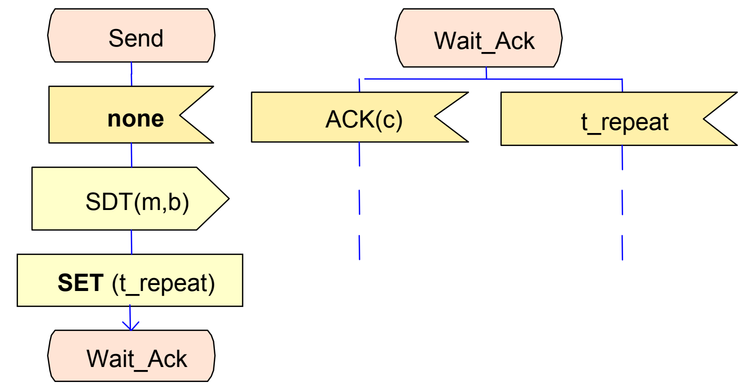
time in system execution

- the model of time [timed automata with urgency]
  - centralized → same clock speed in all processes
  - passes in stable states → transitions are instantaneous
  - depends on the system state → precisely timed behavior



specifying timed behavior

- real-valued clocks
  - operations : set, reset (deactivate)
- timed guards
  - comparison of a clock to an integer
  - comparison of a difference of two clocks to an integer




```
state send;
  output sdt(self,m,b) to {receiver}0;
  set t_repeat := - dt_repeat;
  nextstate wait_ack;
endstate;
state wait_ack;
  input ack(sender,c);
  …
  when t_repeat = 0;
  …
endstate;
```

linking time and system progress

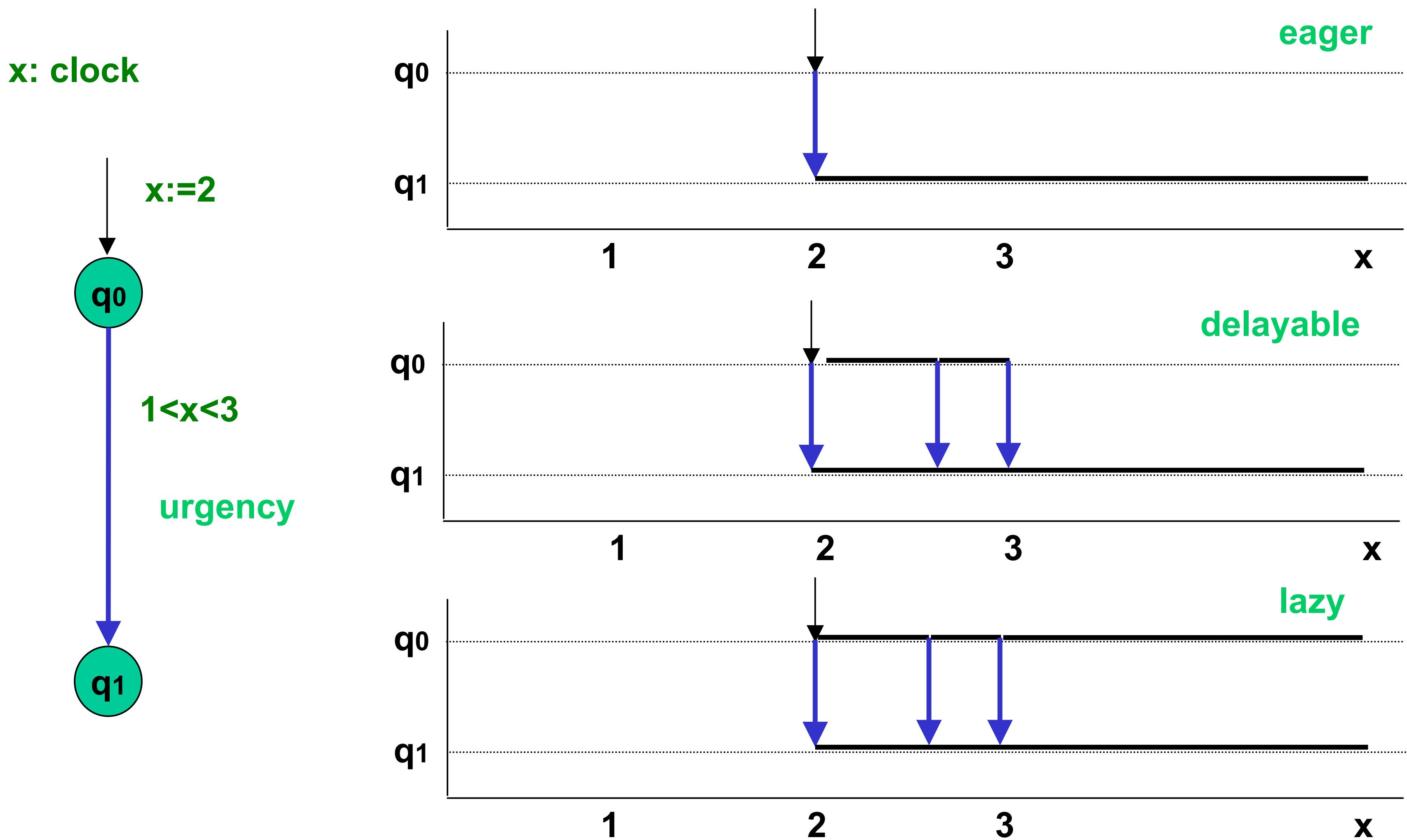
- 3 types of urgency for time-guarded transitions
  - eager transitions : urgent as soon as they are enabled
    block time progress
  - lazy transitions : never urgent
    always allow time progress
  - delayable transitions : urgent when about to be disabled by time progress
    allow time progress otherwise

```
state wait_ack;
  …
  deadline eager;
  when t_repeat = 0;
  …
endstate;
```

```
state idle;
  …
  deadline lazy;
  input PUT(p) // from ENV;
  …
endstate;
```

```
process channel;
state get;
  input SDT(p,q,r);
    nextstate forward;
endstate;
state forward;
  deadline delayable;
  when x >= a and x <= b;
    output SDT(p,q,r)
    nextstate get;
endstate;
```

semantics of urgency



resources

- mutually exclusive access to a physical or logical resource by concurrent IF processes
  - acquisition: precondition to a transition – models passive wait
  - release: action – executed when resource is not needed any more

```
resource buffer;
…
state waiting_update;
  deadline eager;
  acquire buffer;
    informal "upd²ate";
    set x_m := 0;
    nextstate updating;
endstate;
```

```
state updating;
  deadline delayable;
  when x_m = 10;
    informal "updated";
    reset x_m;
    release buffer;
    task ({data}0).updated := true;
    nextstate sleeping_m;
endstate;
…
```

dynamic priorities

- partial priority order between processes based on global state

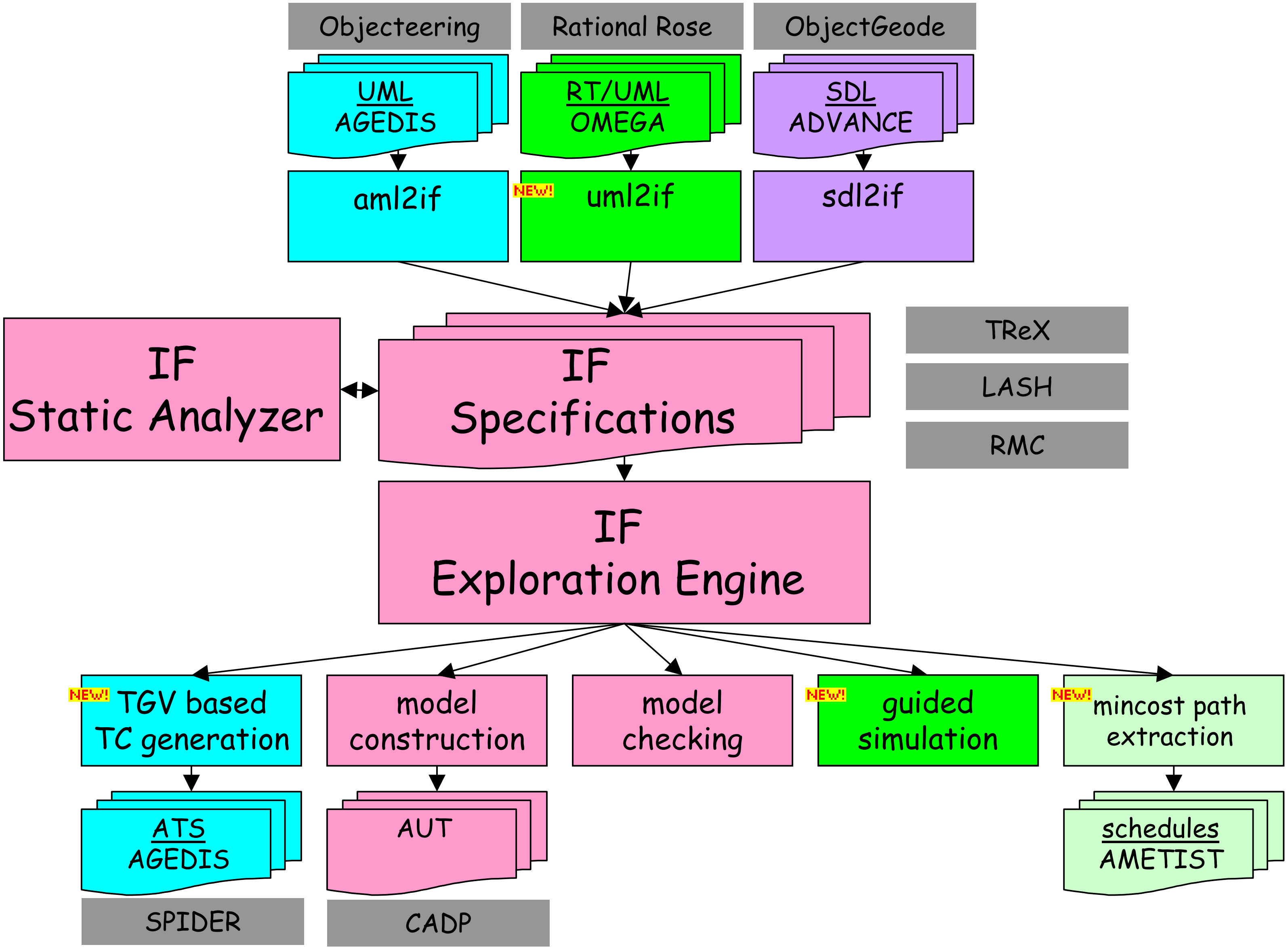
  priority_rule : p1 < p2 **if** *condition*(p1,p2)

- p1, p2 are free variables ranging over the active process set
- semantics:

  among enabled processes, only maximal elements execute

- applications: scheduling policies
  - fixed priority: p1 < p2 if p1 instanceof Tx and p2 instanceof Rx
  - run-to-completion: p1 < p2 if p2 = ({manager}0).running
  - EDF: p1 < p2 if ({task}p2).deadline < ({task}p1).deadline
  - ...

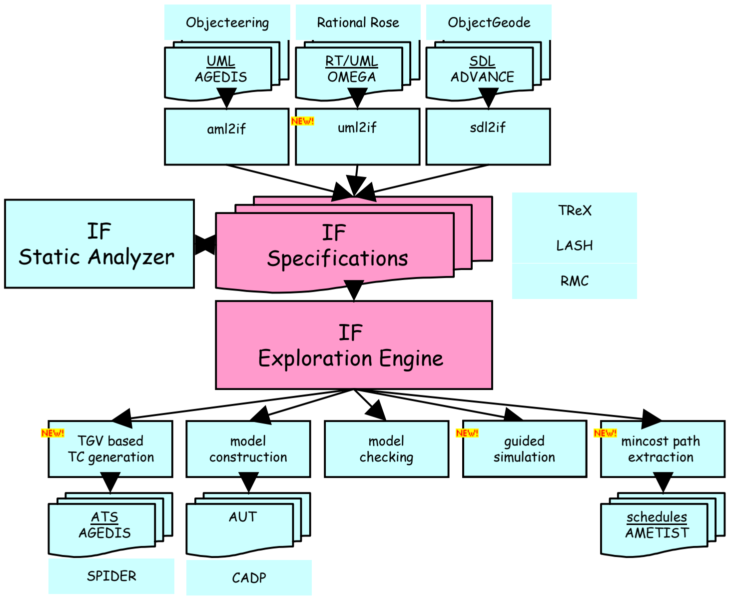
IF Toolset



Objecteering
Rational Rose
ObjectGeode
UML
AGEDIS
RT/UML
OMEGA
SDL
ADVANCE
aml2if
NEW!
uml2if
sdl2if
IF
Static Analyzer
IF
Specifications
TReX
LASH
RMC
IF
Exploration Engine
NEW!
TGV based
TC generation
model
construction
model
checking
NEW!
guided
simulation
NEW!
mincost path
extraction
ATS
AGEDIS
AUT
schedules
AMETIST
SPIDER
CADP

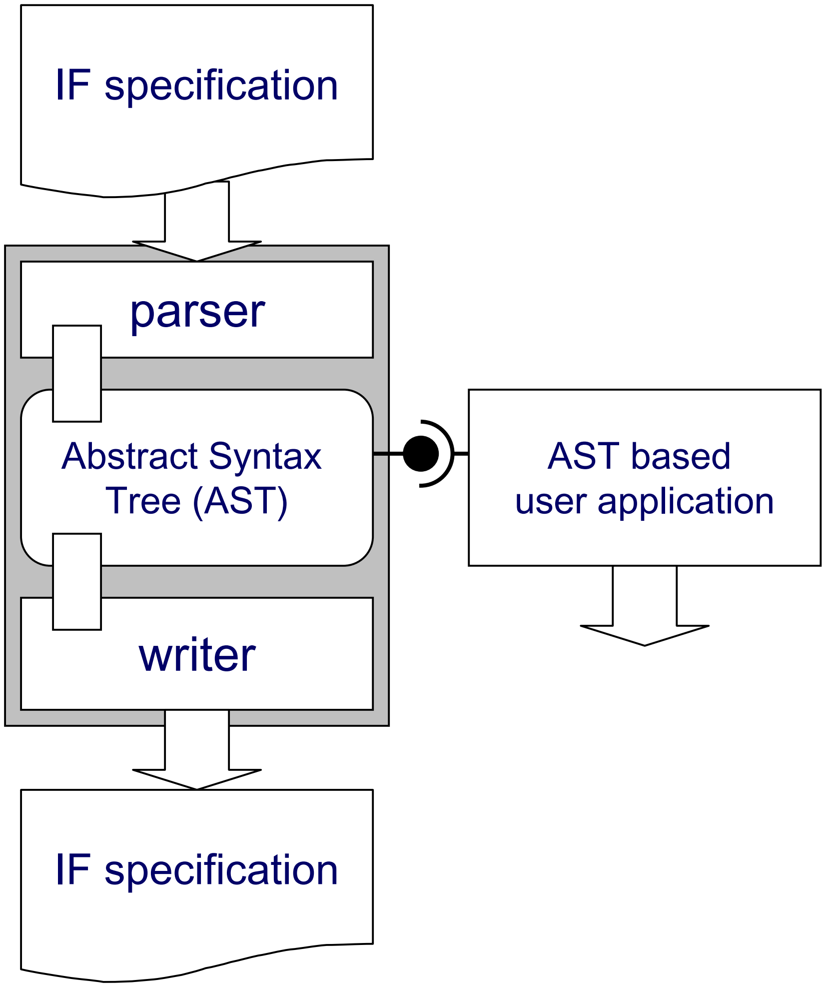
IF Toolset

Core Components
- language API
- exploration API
- simulator design

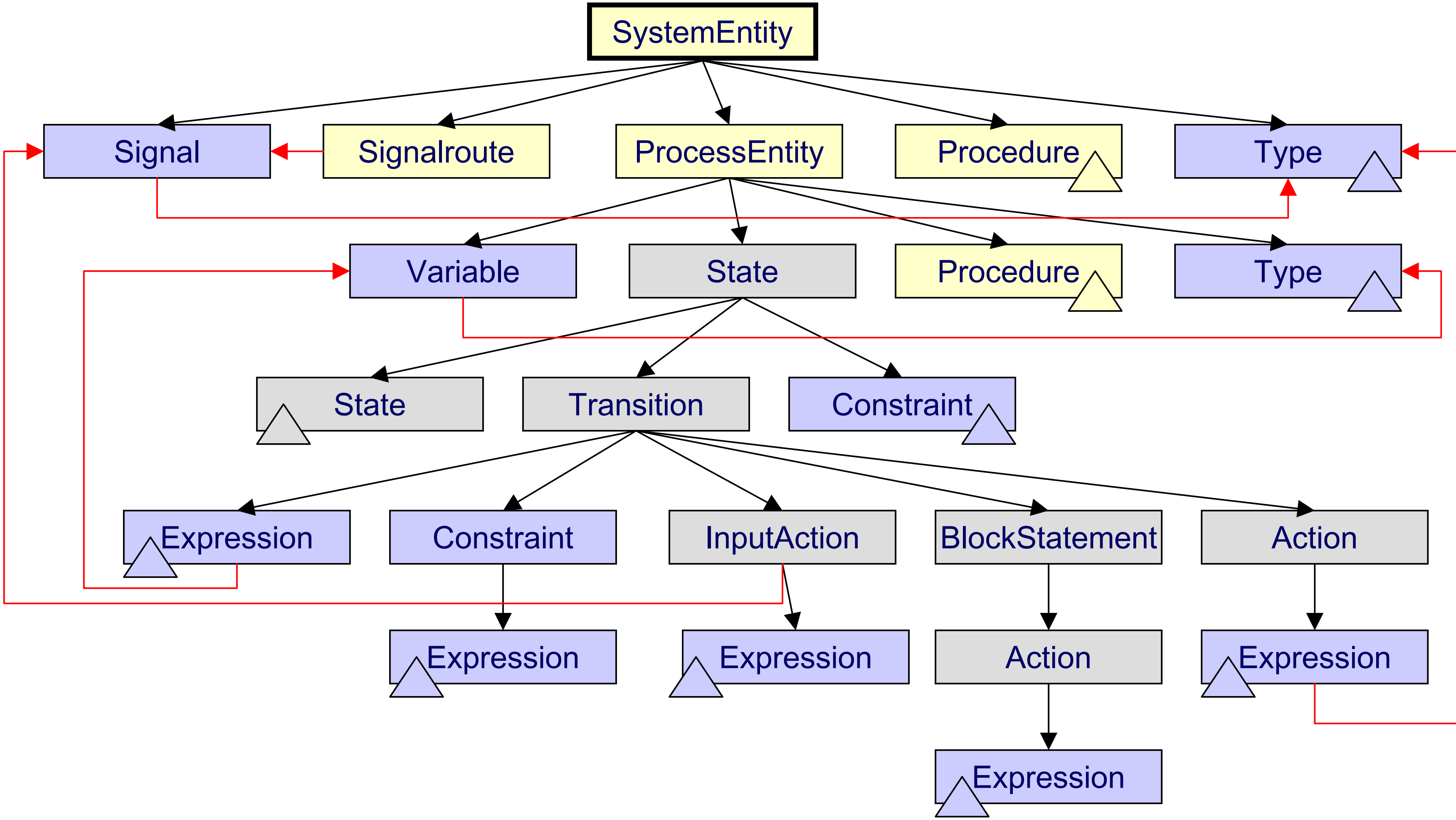


language API

- gives programming access to the AST of an IF specification
- AST represented as a collection of C++ objects



AST overview



an example: used variables

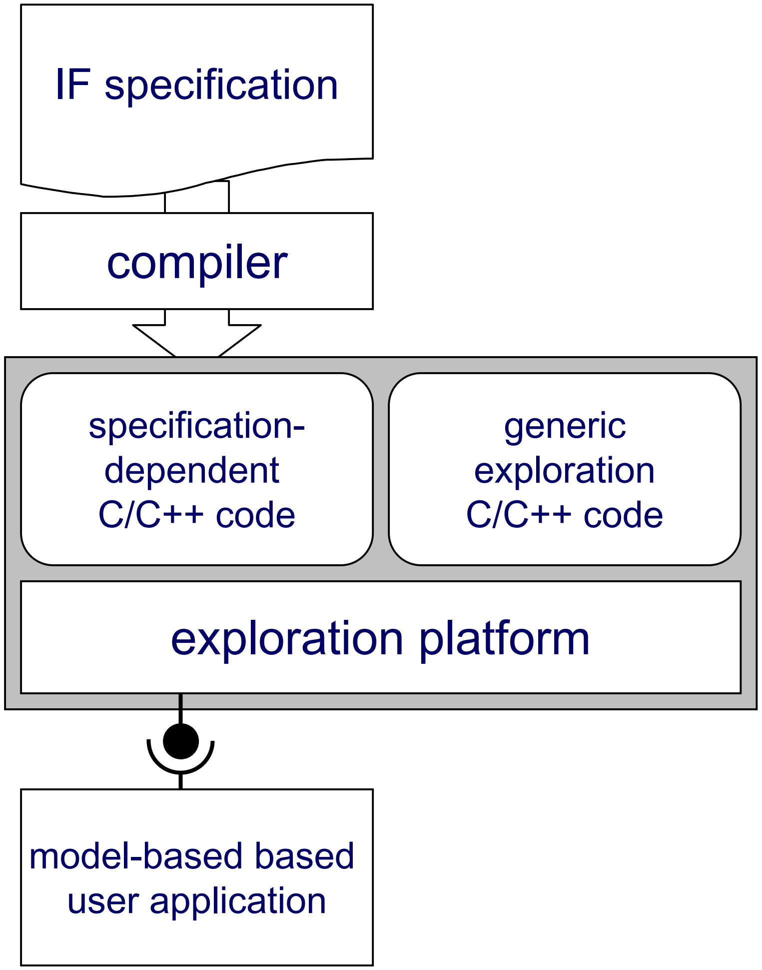
```
#include "model.h"

void main() {
  IfObject::Initialize();
  // parse the input
  IfSystemEntity* sys = Load(stdin);
  if (sys != NULL)
    sys->Compile();
  // for each process...
  for(int i = 0; i < sys->GetProcesses()->GetCount(); i++) {
    IfProcessEntity* proc = sys->GetProcesses()->GetAt(i);
    printf("\n%s:", proc->GetName());
    // for each local variable...
    for(int j = 0; j < proc->GetVariables()->GetCount(); j++) {
      IfVariable* var = proc->GetVariables()->GetAt(j);
      // find if the variable is used in some state
      int used = 0;
      for(int k = 0; k < proc->GetStates()->GetCount(); k++) {
        IfState* state = proc->GetStates()->GetAt(k);
        used |= state->Use(var);
      }
      if (! used)
        printf("%s ", var->GetName());
    }
  }
}
```

applications

- static analysis
  - live variables, slicing, dead code
- code generation
  - simulation code, application code
- translation
  - if2pml (by Eindhoven TU)
- pretty printing
  - if2if, if2dot, if2html

exploration API

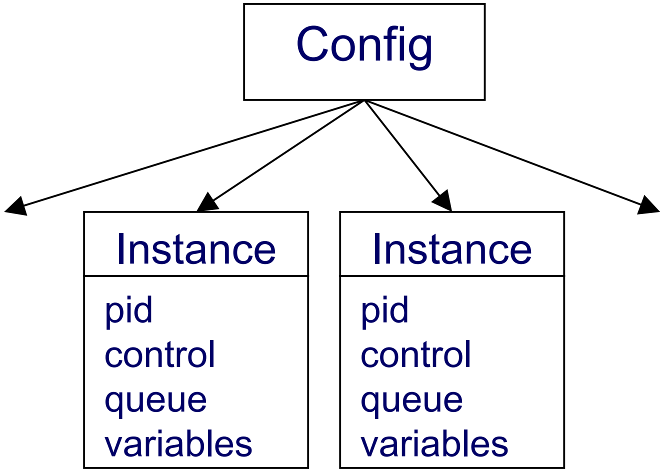
- gives programming access to the underlying labeled transition system of an IF specification
- the API provides
  - state, label representation
    - type definition
    - access primitives
  - forward traversal primitives
    - initial state function (*init*)
    - successor function (*post*)
- on-the-fly, forward, explicit, enumerative



LTS representation

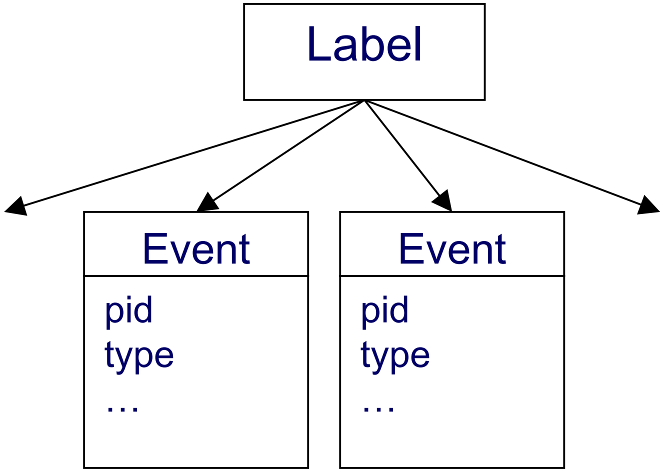
states are global (system) configurations

- gray-box structural representation as set of local (process) configurations (instances)
- the content of each process configurations can be accessed
  - process identifier (pid)
  - control state pointer
  - queue of pending input signals
  - local variables and parameters

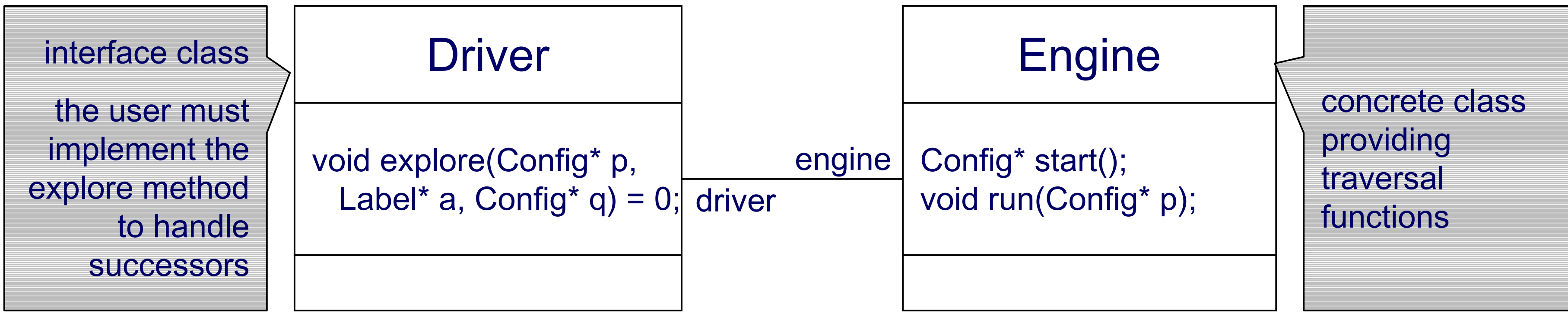


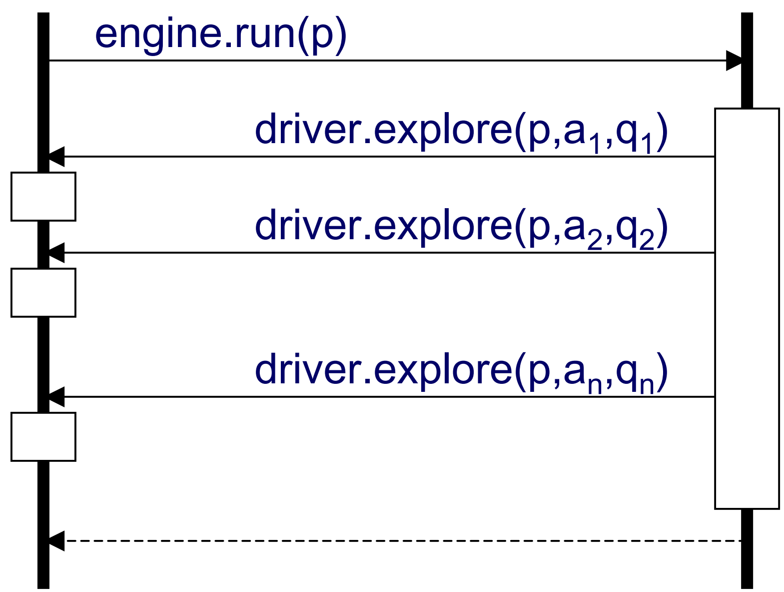
labels record observable events occurring on transitions

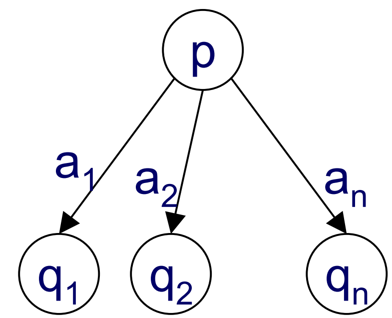
- structural representation as a list of events
- each event can be accessed
  - issuing process
  - event type (INPUT, OUTPUT, FORK, etc.)
  - type dependent auxiliary information



LTS traversal







an example: bfs search

```
#include "simulator.h"

class BfsExplorer : public IfDriver {
  static const int REACHED = 1; // reachable state marking
  Queue m_queue; // the queue of unexplored states

public:
  // successor handler: append target state to the queue, if not yet reached
  void explore(IfConfig* source, IfLabel* label, IfConfig* target) {
    if (! (target->getMark() & REACHED) )
      { target->setMark(REACHED); m_queue.put(target); }
  }
  // visit one state i.e, print it on the screen
  void visit(IfConfig* state) {
    state->print(stdout);
  }
  // visit all states, main bfs loop
  void visitAll() {
    IfConfig* start = m_engine->start();
    start->setMark(REACHED); m_queue.put(start);
    while (! m_queue.isEmpty()) {
      IfConfig* state = m_queue.get();
      visit(state);
      m_engine->run(state);
    }
  }
};
```

applications

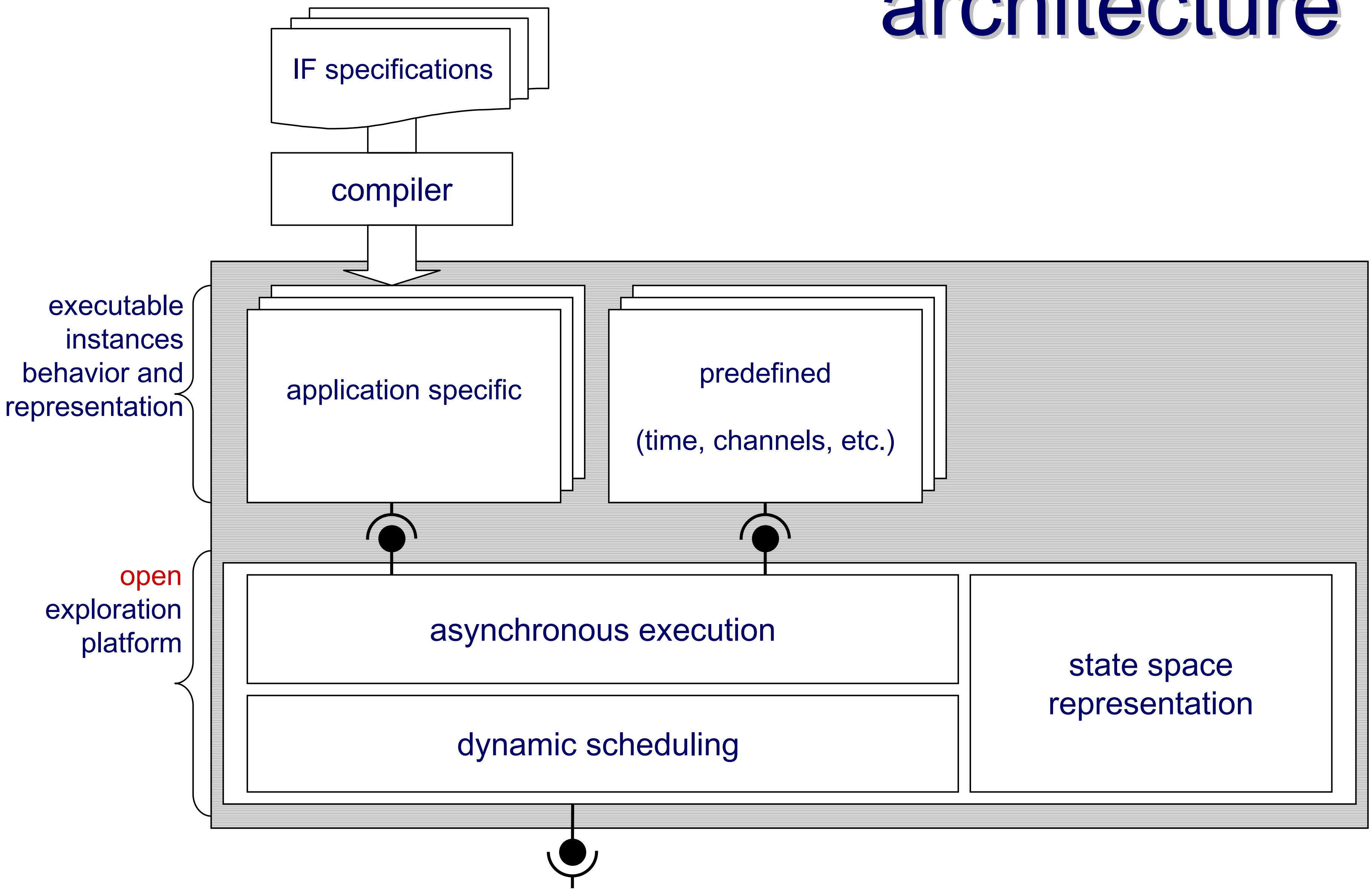
- Debugging
  - interactive, random simulation
- Model-checking
  - exhaustive model generation
  - on-the-fly μ-calculus evaluation
  - model exploration with observers
- Testing
  - test case generation
  - on-the-fly timed testing
- Optimization
  - shortest path computation

simulator design

- goal: offer primitives to explore the state space of IF specifications in an exhaustive manner

- main functionalities
  - simulate the process execution
    - inter-process communication
    - process creation / destruction
    - control of simulation time
  - handle non-determinism
    - asynchronous execution
    - internal non-deterministic choices
    - open environment
  - state space representation

architecture



execution control

1st layer : emulate asynchronous parallel execution

- ask in turn each instance to execute its enabled transitions
  - ensures atomicity at level of instance transitions
- when an instance is executing provides
  - message delivery, shared variable update
  - global time constraints check and clocks update
  - dynamic instance creation and destruction
  - record generated observable events
- get informed when a local step is finished and
  - take a snapshot of the global configuration and store it
  - send the successor to the 2nd layer (dynamic scheduler)

obtain global (system) steps from local (process) steps

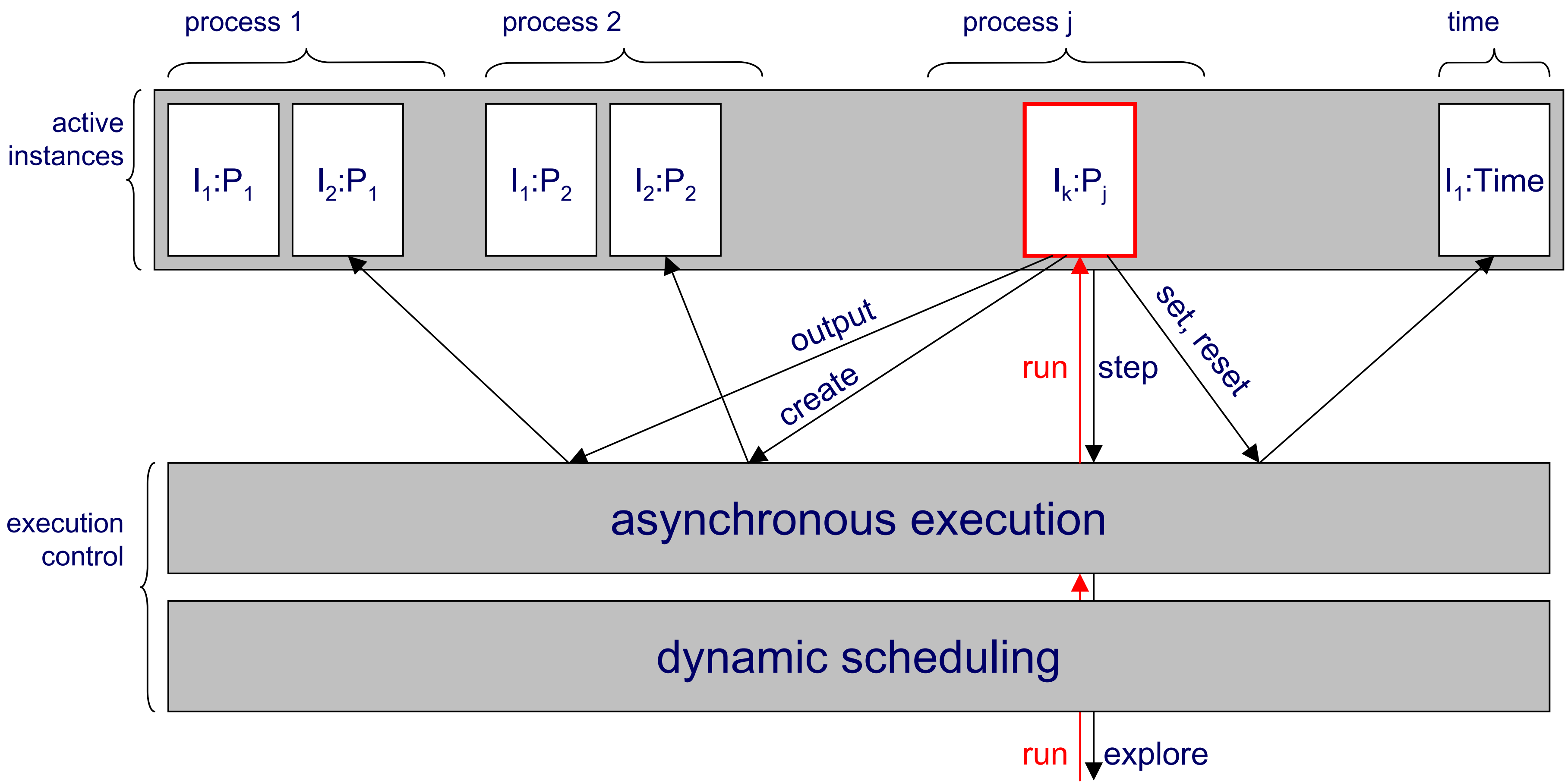
execution control

2^{nd} layer: dynamic scheduling

- collect all potential global successors
- filter them accordingly to dynamic priorities
  - evaluate each priority constraint
  - if applicable on current state

    remove successors produced by the low priority instance
- deliver the remaining set to the user application through the exploration API

execution control



simulation time

at simulation, time is a dedicated process instance handling

- dynamic clock allocation (set, reset)
- represent clock valuations
- check time constraints (timed guards)
- compute time progress conditions w.r.t. actual deadlines and
- fire time transitions, if enabled

two concrete implementations are available (other can be easily added)

i) discrete time

clock valuations represented as varying size integer vectors

time elapse is explicit and computed w.r.t. the next enabled deadline

ii) dbm time

clock valuations represented using varying size difference bound matrices (DBMs)

time elapse is symbolic

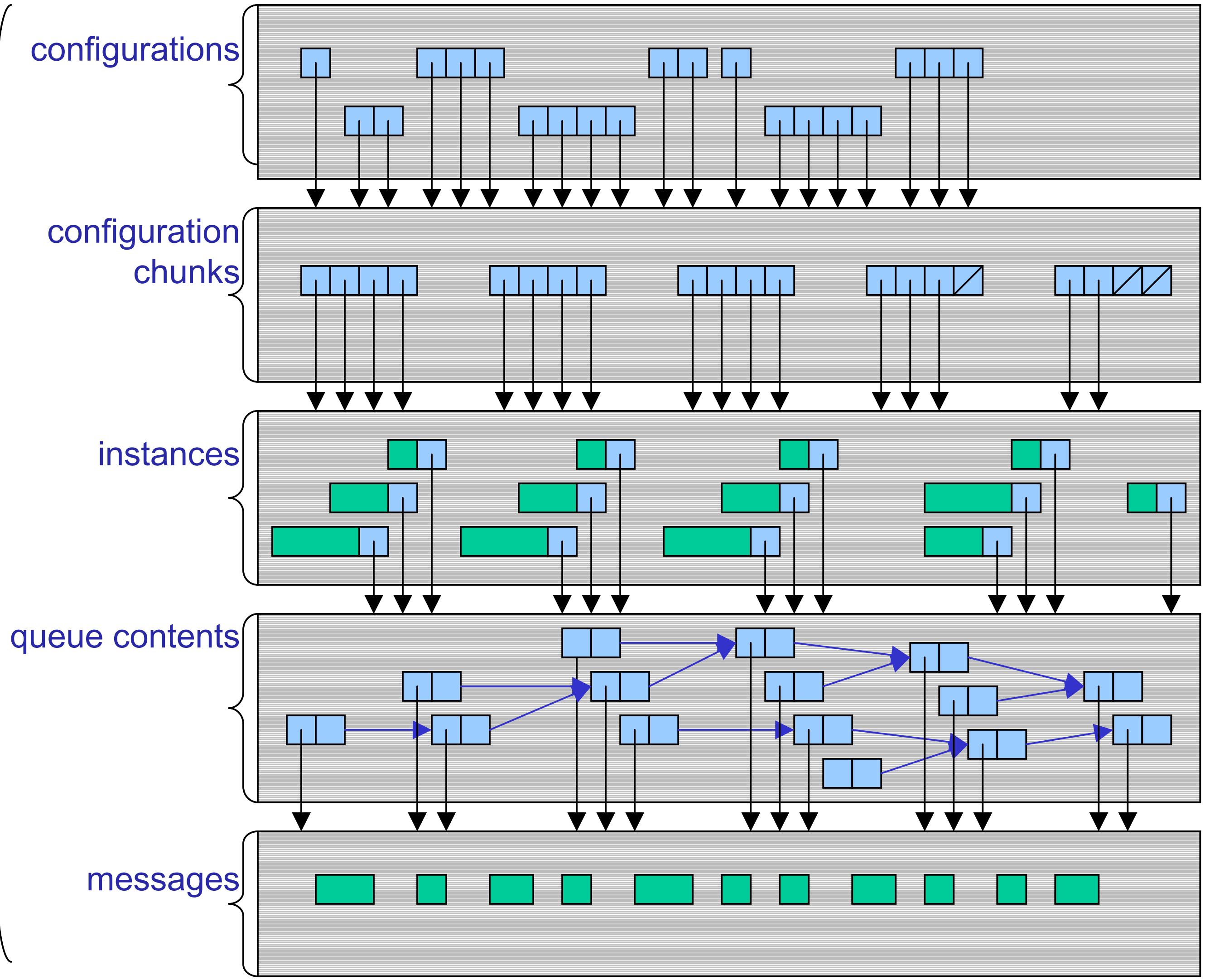
non-convex time zones may arise because of deadlines: they are represented implicitly as unions of DBMs

state representation

state storage is completely done by the simulator

structural representation of configurations offering maximal sharing

unique tables implemented as hash tables with collision or search trees (splay trees or 2-3 trees)

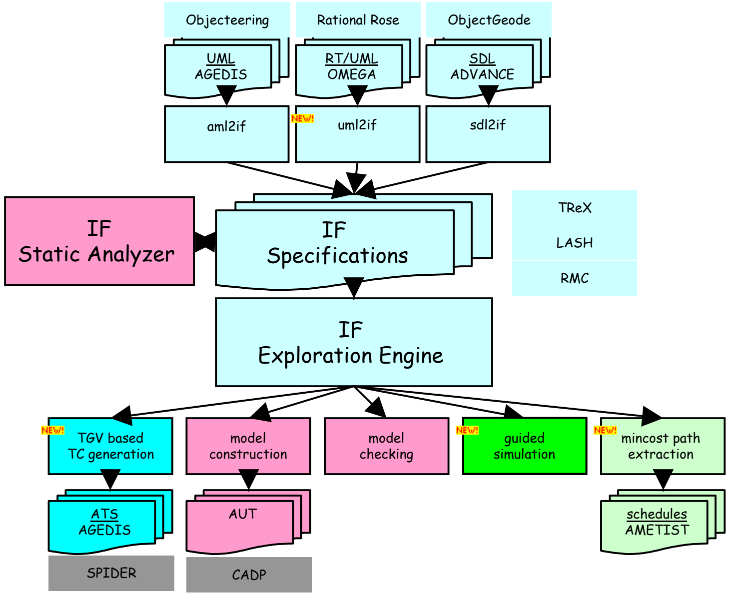


credits

Open/Cæsar	exploration API i.e, labeled transition system interface
System C	simulator architecture i.e, open platform + running objects
Kronos, Uppaal	symbolic time representation and operations using DBMs
BDDs	state space representation

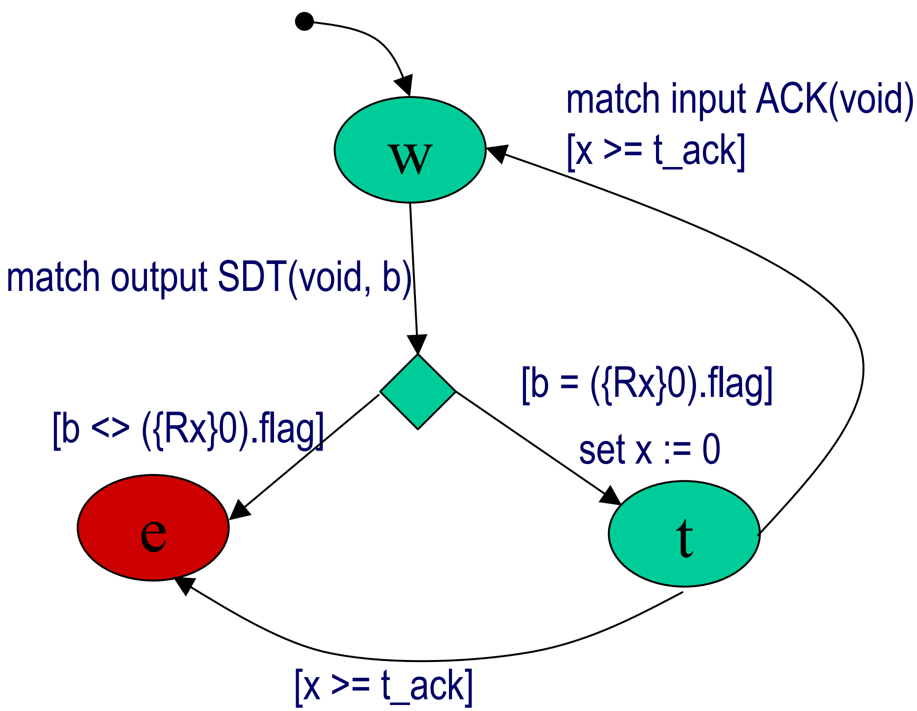
IF Toolset

Model-Based Validation
- model checking
- test generation
- optimization
- static analysis



using observers

- specify system properties in an operational way
- observes
  - events
  - system state
  - time
- states
  - normal / error / success
- properties
  - linear, timed
  - safety/liveness
- semantics
  - weakly synchronized composition (i.e. greater priority than the system)



observation and actions

- state observation
  - variables, queues, process-in-state
- event observation
  - event types : INPUT, OUTPUT, FORK, KILL, DELIVER, …
  - retrieve data related to event
    - signal parameters
    - created process' pid…
- actions
  - internal : local variables, etc.
  - control system simulation/exploration
    - cut the exploration
    - inject signals, mutate variables

Verification : reachability (safety)

μ-calculus evaluation

- alternating-free fragment

$$\varphi ::= T \mid X \mid \langle a \rangle \varphi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mu X.\varphi(X)$$

where a denotes a regular expression on labels

- macros available to describe complex formula e.g,

$$\text{all } \varphi \equiv \upsilon X.\ \varphi \wedge [*]X$$

$$\text{pot } \varphi \equiv \mu X.\ \varphi \vee \langle * \rangle X$$

$$\text{inev } \varphi \equiv \mu X.\ \varphi \vee \langle * \rangle T \wedge [*]X$$

- IF toolset includes an on-the-fly local model-checker
- diagnostics can be extracted either as sequences (if the property is “linear”) or sub-graphs (if the property is “branching”)

behavioral relations

- LTS comparison:
  - equivalence relations (“behavior equality”):

    System $\approx$ Specification
  - preorder relations (“behavior inclusion”):

    System $\leq$ Specification
- LTS minimization:
  - quotient w.r.t an equivalence relation:

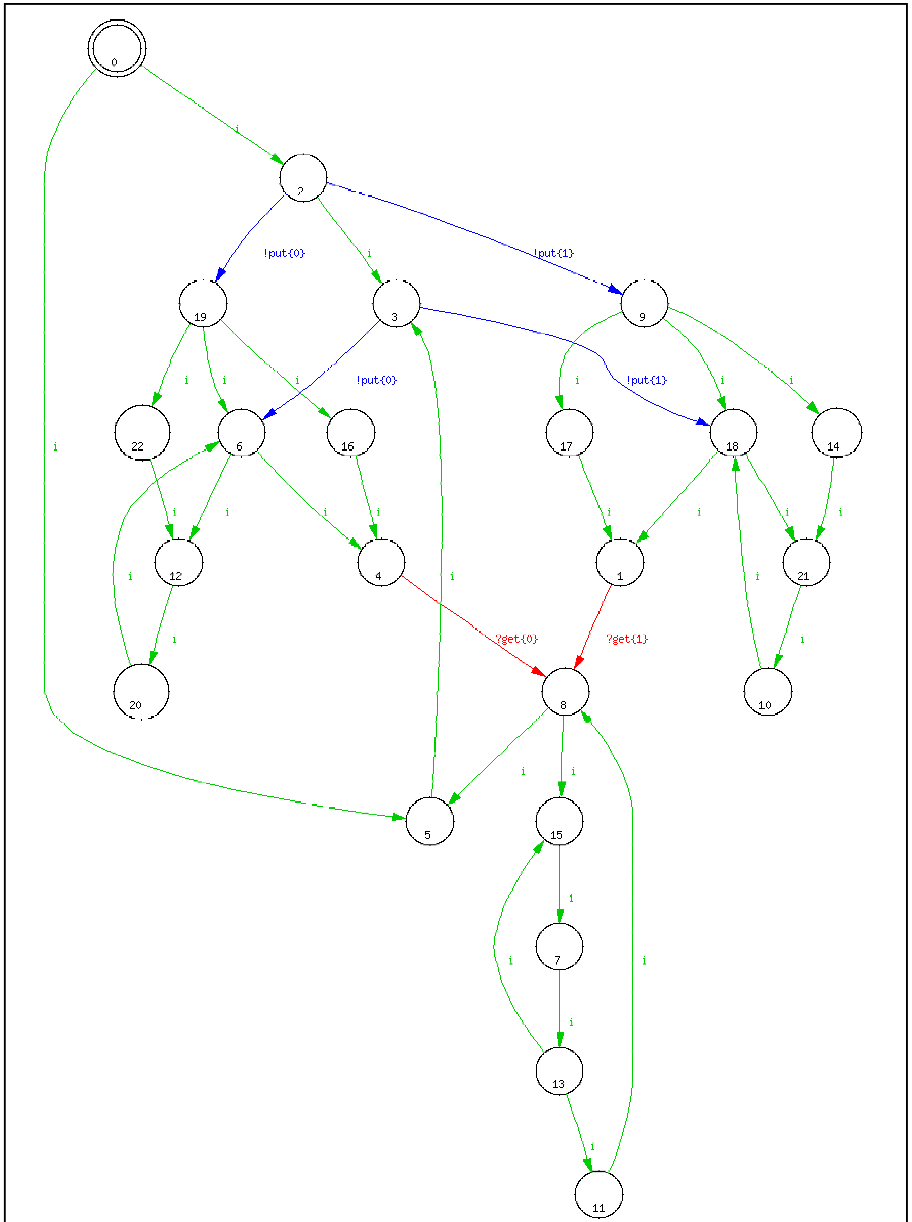
    (System / $\approx$)
- several relations available:

  weak/strong bisimulation, branching, safety, trace equivalence

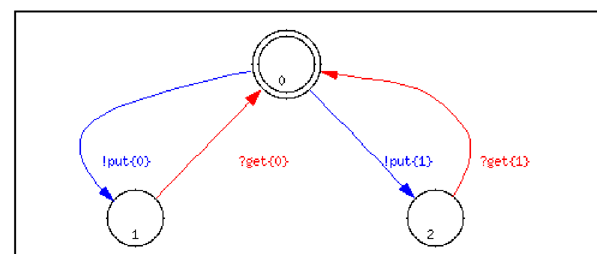
- use of CADP as back-end:

  aldebaran, bcg_min

example

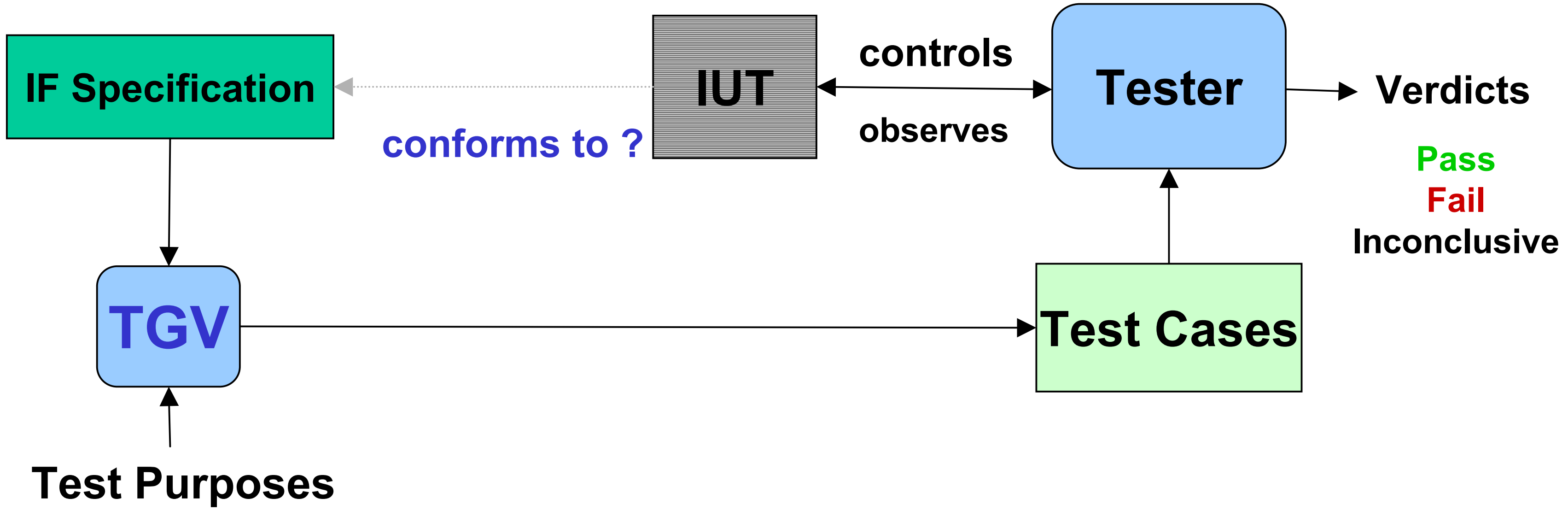


reduction w.r.t.
branching bisimulation



the TGV test generation tool

Conformance testing for distributed applications

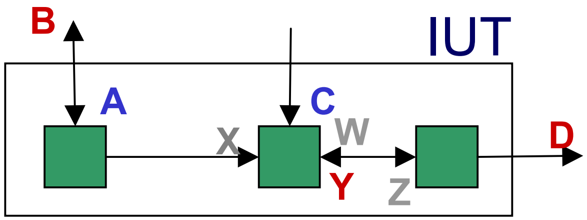


Two implementations:

- TGV (Irisa/Verimag) for Lotos, SDL, UML and IF
- TestComposer (Telelogic), inside ObjectGeode

principle of TGV

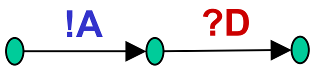
- System architecture:



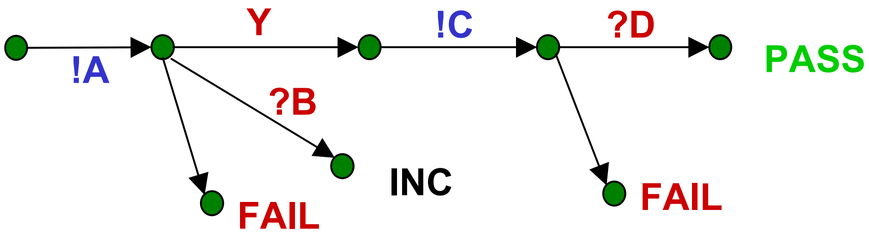
A, **C**: controlable
B, **D**, **Y**: observable
W, **X**, **Z**: internal

- Specification (IF,…) → Exhaustive system behaviour (in terms of **A**,**B**,**C**,**D**,**W**,**X**,**Y**,**Z**)

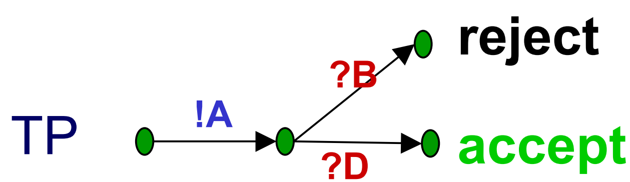
- Test purpose: property

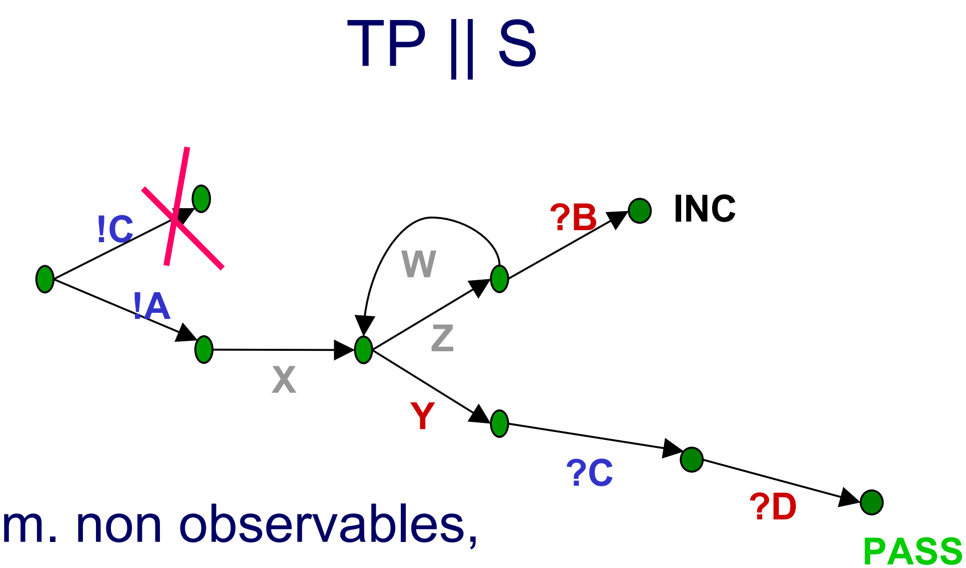


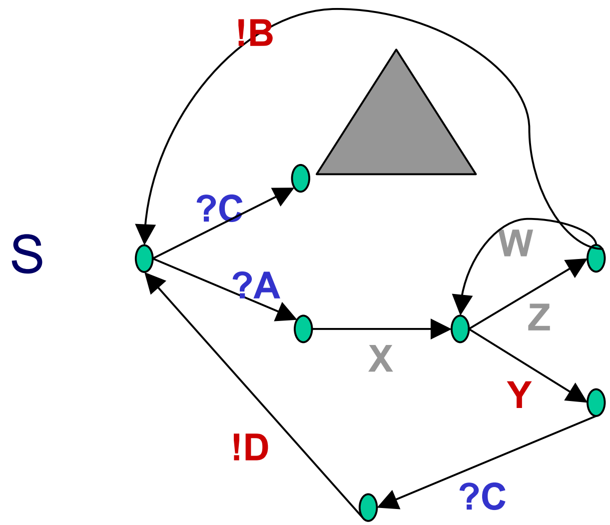
⇒ TGV computes test cases:



test case generation in TGV



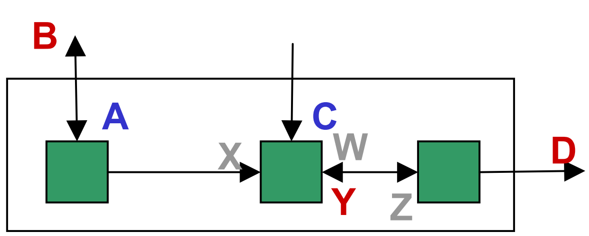


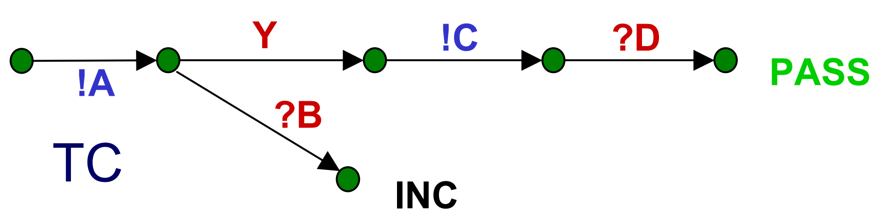


elim. non observables,

determinize,

eliminate conflicts



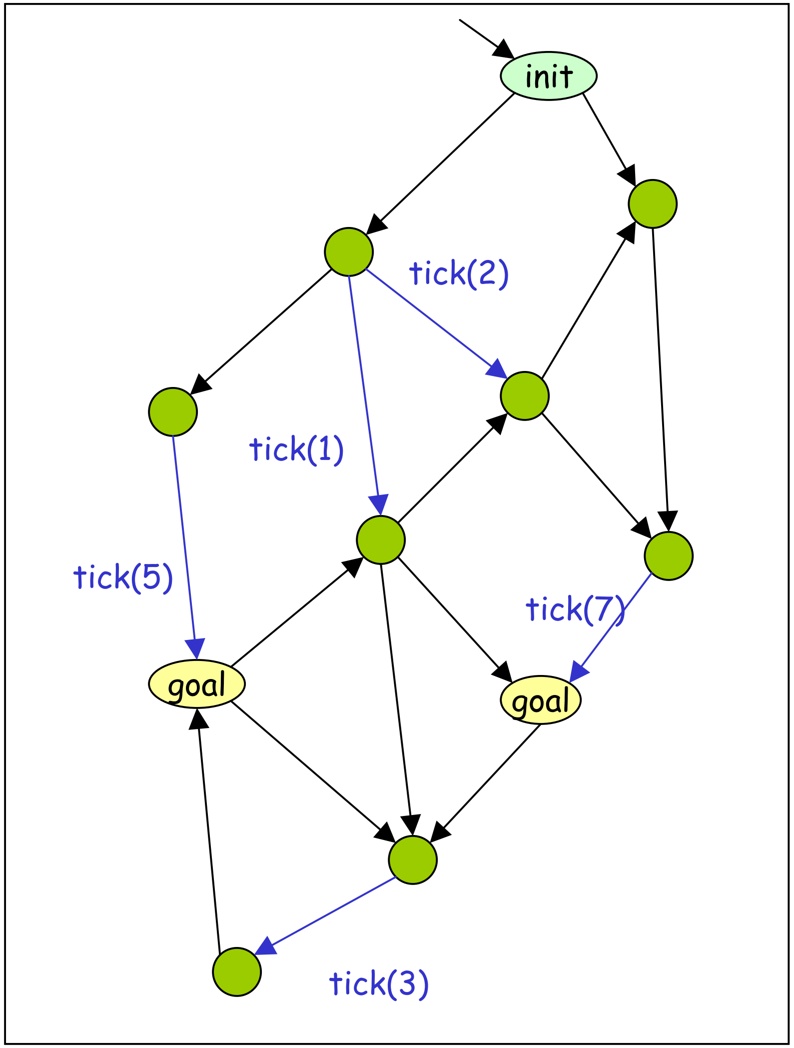


TGV results

- advantages of automatic test case generation:
  - less error prone
  - less time consuming
  - applicable to real systems
- problems of automatic test case generation:
  - manual tests are symbolic -> less test cases
  - detailed formal specification is needed
- AGEDIS IST project (integration of IF/TGV inside a complete testing framework):
  - model specification in UML, translation to IF
  - test generation with TGV
  - test execution on Java programs with Spider (IBM)

optimization

- there are (user defined) costs associated to transitions of the semantic model of IF specifications e.g, waiting times
- problem: find the min-cost execution path leading from the initial state to some goal state
- three algorithms implemented:
  - Dijkstra algorithm (best first)
  - A* algorithm (best first + estimation)
  - branch and bound (depth-first)
- applications: job-shop scheduling (find the makespan), asynchronous circuit analysis (find the maximal stabilization time)



static analysis

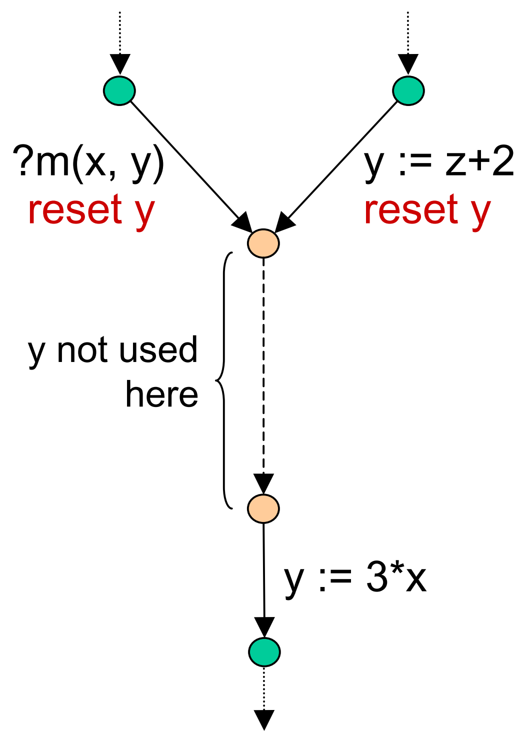
- philosophy
  - source code transformations for model reduction
  - code optimization methods

- techniques implemented so far
  - live variable analysis: remove dead variables and/or reset variables when useless in a control state
  - dead-code elimination: remove unreachable code w.r.t. assumptions about the environment
  - variable abstraction: extract the relevant part after removing some variables

- usually, impressive state space reduction

live variables

a variable is dead in a control point if its value is not used before being redefined on any path starting at that point



find live variables

usual backward dataflow analysis extended to IF communication primitives

- asynchronous communication via queues
- parameter passing at process creation

live variables are propagated both intra and inter processes !

exploit live variables

transform IF specification by

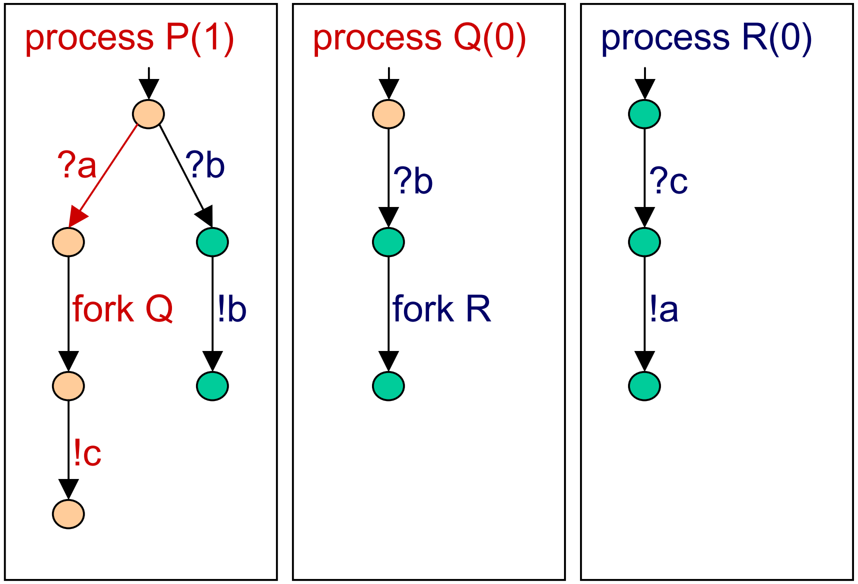
- removing completely dead variables and signal / process parameters
- resetting partially dead variables

the gains are multiple:

- drastically reduce the size of the model (orders of magnitude on realistic examples)
- strongly preserve the initial behaviour

dead-code elimination

a part of code is dead if it will never been entered, for any execution



provides only "a" signals to the process P

find dead code

algorithm for static accessibility of control states and control transitions given user assumptions about the environment

accessibility propagated both intra- and inter processes

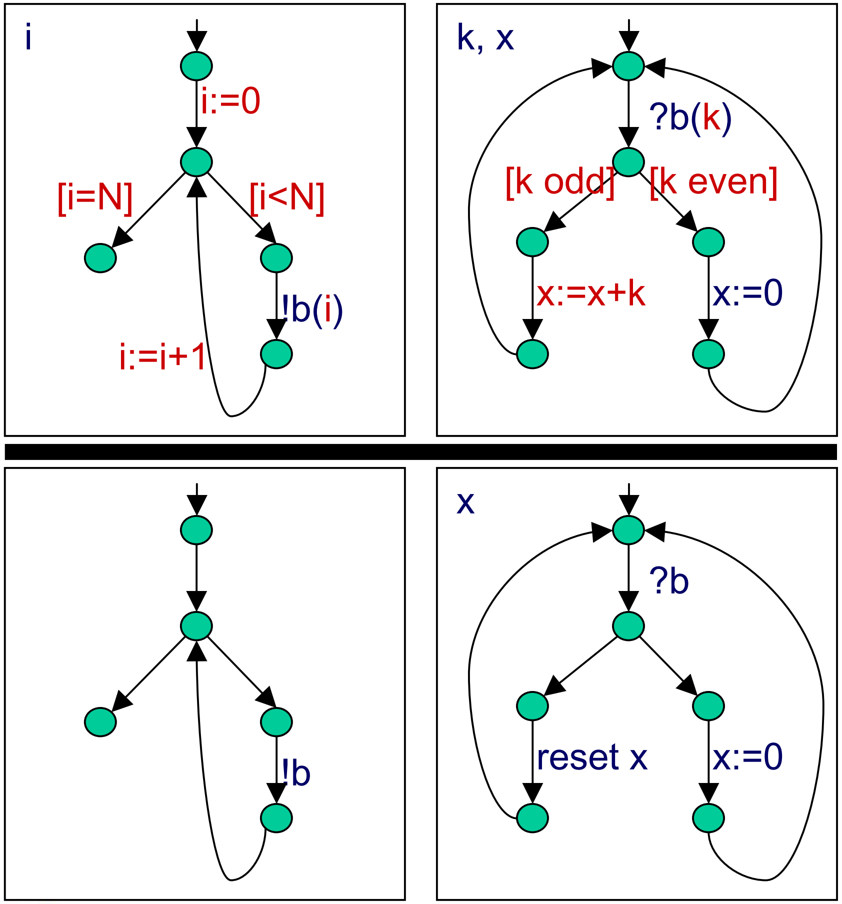
exploit dead code

transform IF specifications by
- removing processes never created
- removing signals never sent
- removing unreachable control states and control transitions

the gains are
- reduce the size of the specification
- enable more reduction by live analysis
- strongly preserve the initial behavior, under the given assumptions

variable elimination

abstraction w.r.t. a set of variables (to eliminate) provided by the user



find undefined variables

forward dataflow analysis propagating the influence of removing variables

- local undefined-ness of variables
- global undefined-ness of signal and process parameters

the propagation is performed both intra- and inter-processes

exploit undefined variables

transform IF specifications by

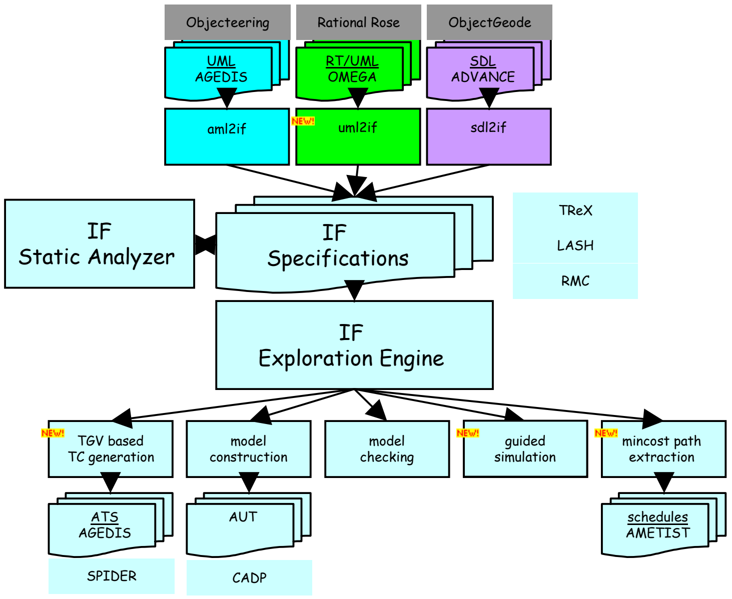
- removing assignments to undefined variables
- removing undefined signal and process parameters
- relaxing guards involving undefined variables

obtain a conservative abstraction of the initial specification i.e, including all the behaviors of the initial one

IF Toolset

Front-Ends

- sdl2if
- uml2if



SDL overview

Specification and Description Language

- formal specification language for distributed systems
  - concurrent processes (Extended FSM)
  - asynchronous buffered communication
- widely accepted in telecommunication area
  - ITU standard, revised every 4 years ('88 – '00)
  - development methodologies
  - commercial tool support

SDL concepts

- hierarchical structuring mechanism
  - system, blocks, processes, services (agents)
- high level process description language
  - nested states, structured transitions
  - various elementary triggers and actions
  - procedures
- dynamical features
  - process creation and destruction
- timing aspects
  - timer concept, global time (now)
- object-oriented features
  - parameterization, inheritance

- formal semantics defined in terms of Abstract State Machines (ASM)

SDL translation

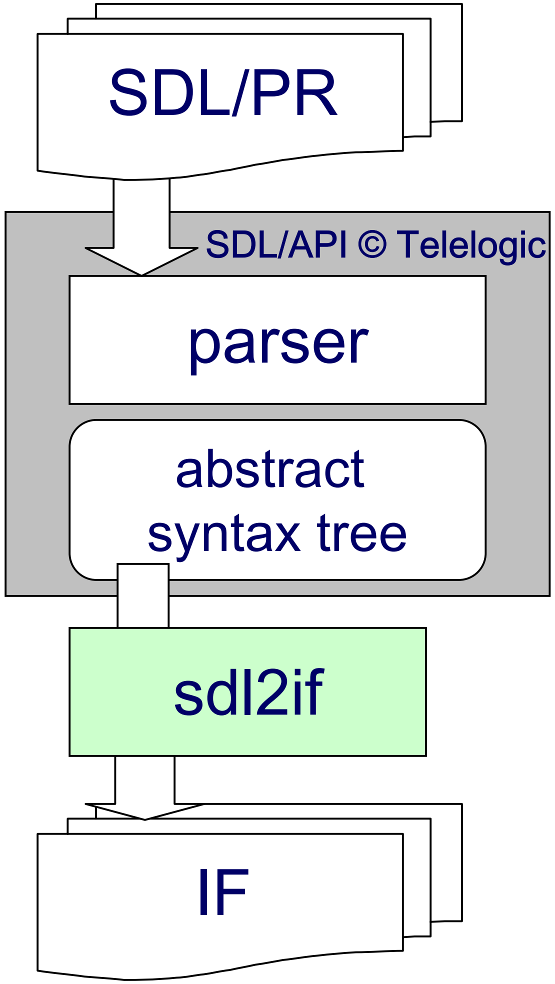
- translation of SDL into IF is straightforward
  - direct mapping of SDL elements into IF ones
  - at origin, IF was an intermediate representation for SDL

- but there exists some limitations
  - hierarchical system decomposition
  - procedures and procedures calls
  - complex data types
  - arbitrary use of now in expressions

sdl2if

sdl2if relies on a full SDL parser provided by Telelogic AB

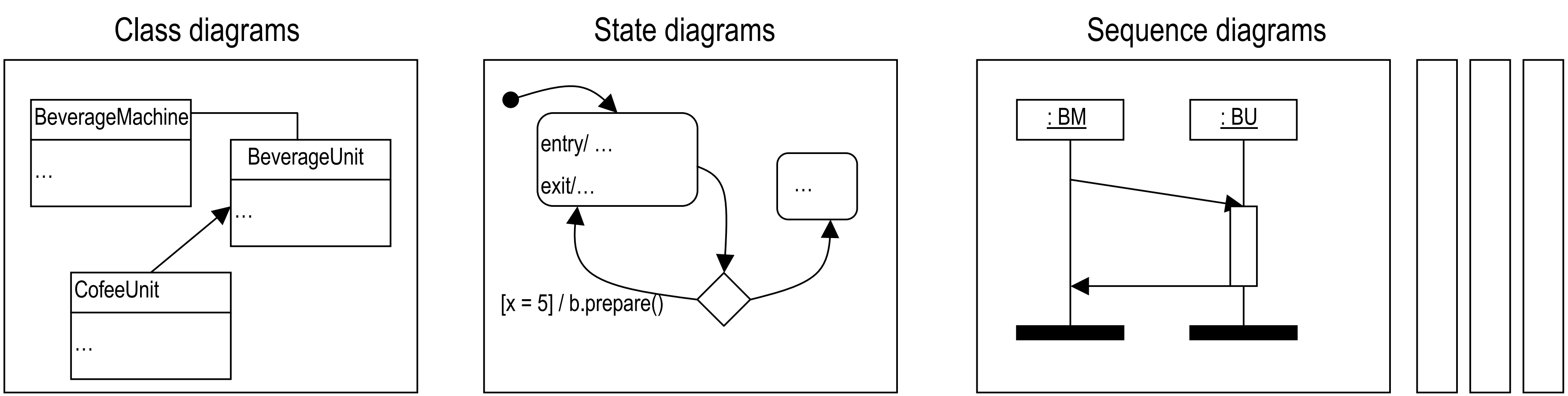
several transformations are applied on the SDL/AST prior to its translation (i.e, SDL'xx reduced to SDL'88)



an overview of UML

OMG's standard modeling language

- developed since 1998, current versions: 1.4, 2.0
- widely accepted in industry, wide tool support
- complex (10 types of diagrams, ≈150 types of concepts)
  - mixes declarative / imperative, OO, synchronous/asynchronous, aspect oriented, …
  - for requirements / design
- informal semantics



language choices

our focus : real-time and embedded systems (OMEGA)

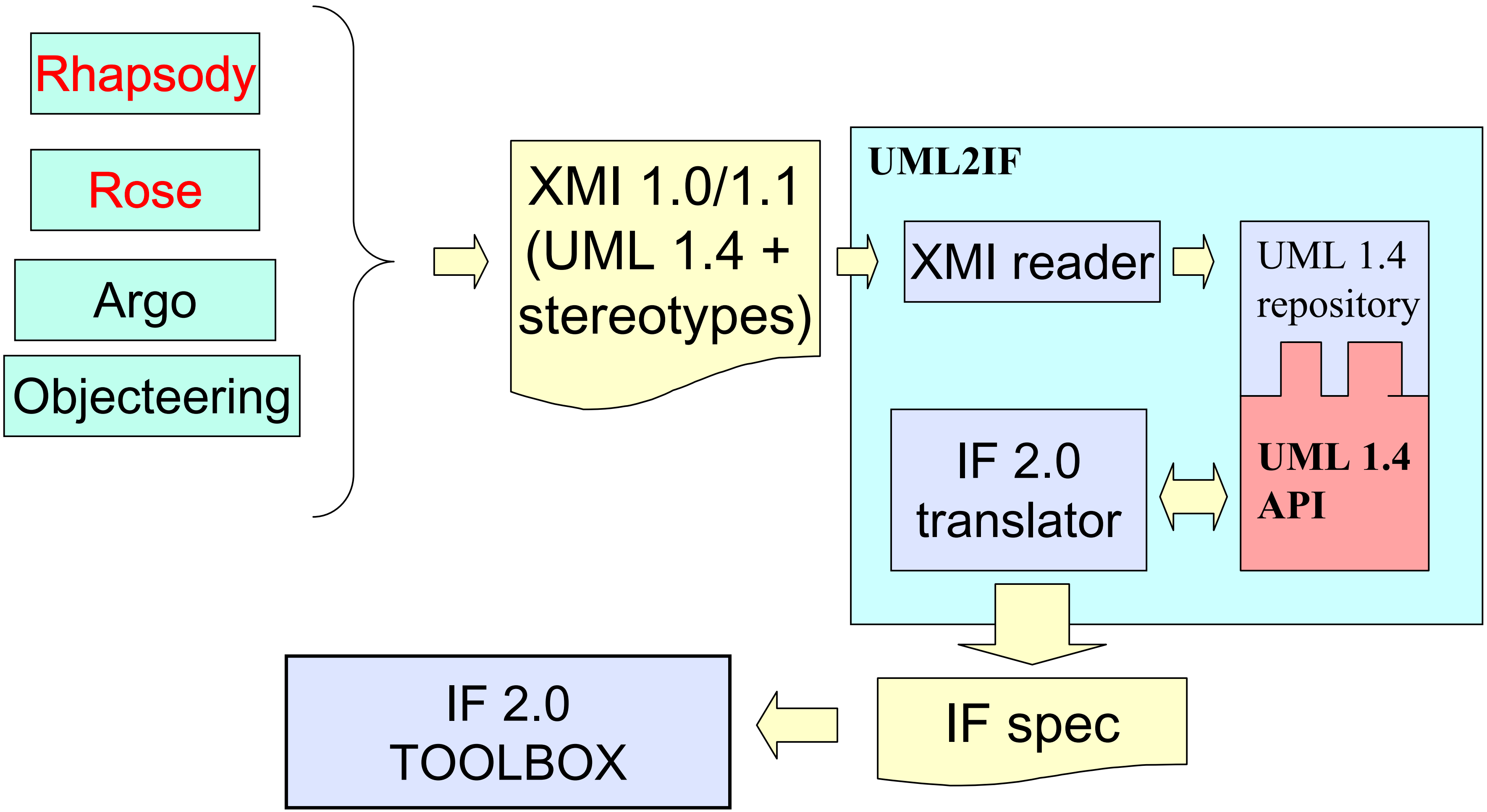
- cover operational specifications
  - classes with operations, attributes, associations, generalization, statecharts; basic data types
- define a particular execution model
  - a notion of active class
  - active objects define activity groups
  - run-to-completion, group stability
- communication and behavior
  - primitive operations – procedural, stacked
  - triggered operations – embedded in state machine, queued
  - asynchronous signals
- define an **Action Language**

translation to IF

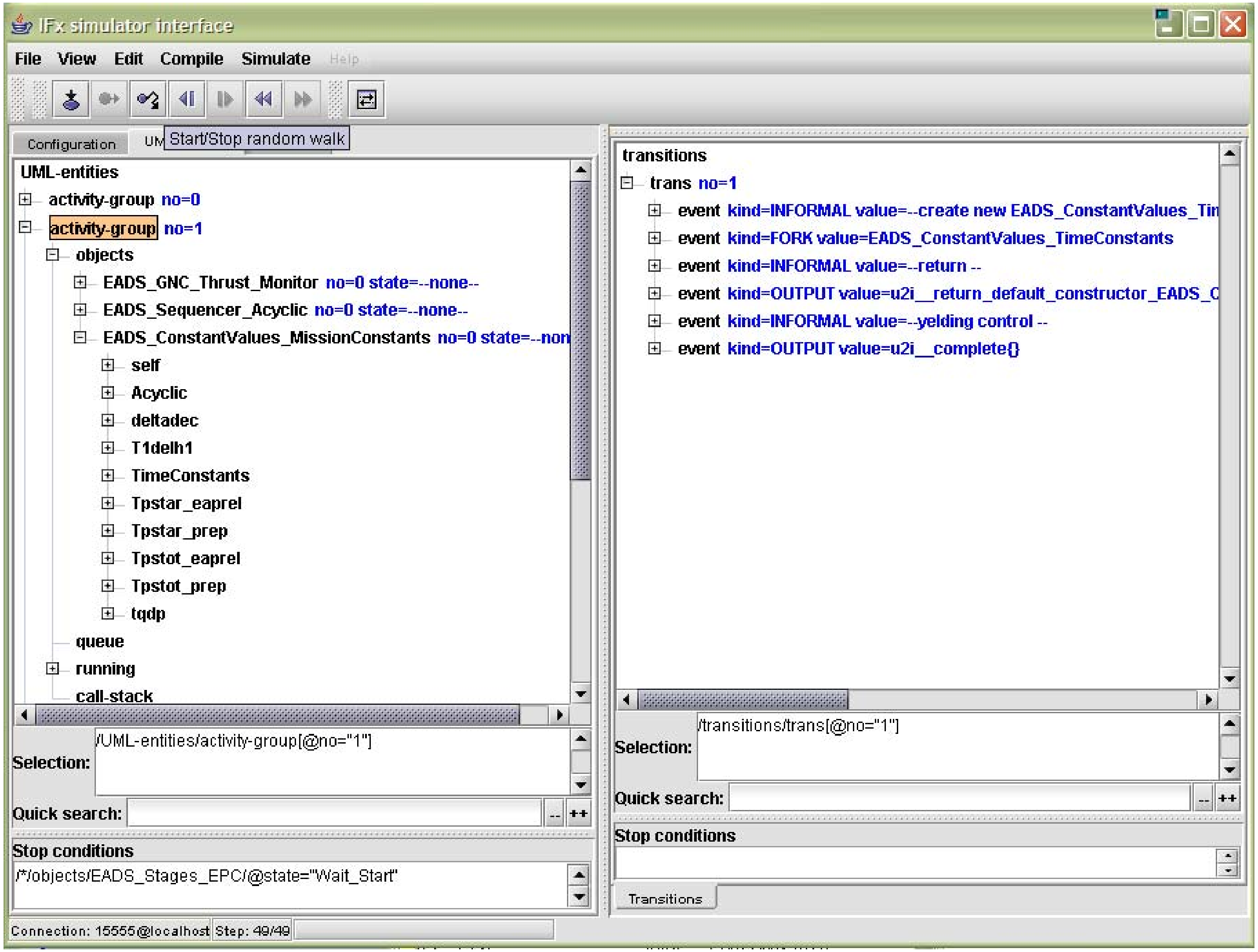
a mapping of OO concepts to (extended) automata

- structure
  - class → process type
  - attributes & associations → variables
  - inheritance → replication of features
  - signals, basic data types → direct mapping
- behavior
  - state machines (with restrictions) → IF hierarchical automata
  - action language → IF actions, automaton encoding
  - operations:
    - operation call/return → signal exchange
    - procedure activations → process creation
    - polymorphism → untyped PIDs
    - dynamic binding → destination object automaton determines the executed procedure

tool architecture



simulation / verification interface



- user friendly simulation
- system state exploration…
- customizable presentation of results for UML users

Case Studies

telecommunication protocols

embedded and distributed software

manufacturing problems

asynchronous circuits

protocols

SSCOP

Service Specific Connection Oriented Protocol

M. Bozga et al. **Verification and test generation for the SSCOP Protocol**. In *Journal of Science of Computer Programming - Special Issue on Formal Methods in Industry*. Vol. 36, number 1, January 2000.

MASCARA

Mobile Access Scheme based on Contention and Reservation for ATM

case study proposed in VIRES ESPRIT LTR

S. Graf and G. Jia. **Verification Experiments on the Mascara Protocol**. In M.B. Dwyer (Ed.) *Proceedings of SPIN Workshop 2001, Toronto, Canada*. LNCS 2057.

PGM

Pragmatic General Multicast

case study proposed in ADVANCE IST-1999-29082

pragmatic general multicast

protocol specification

Key features

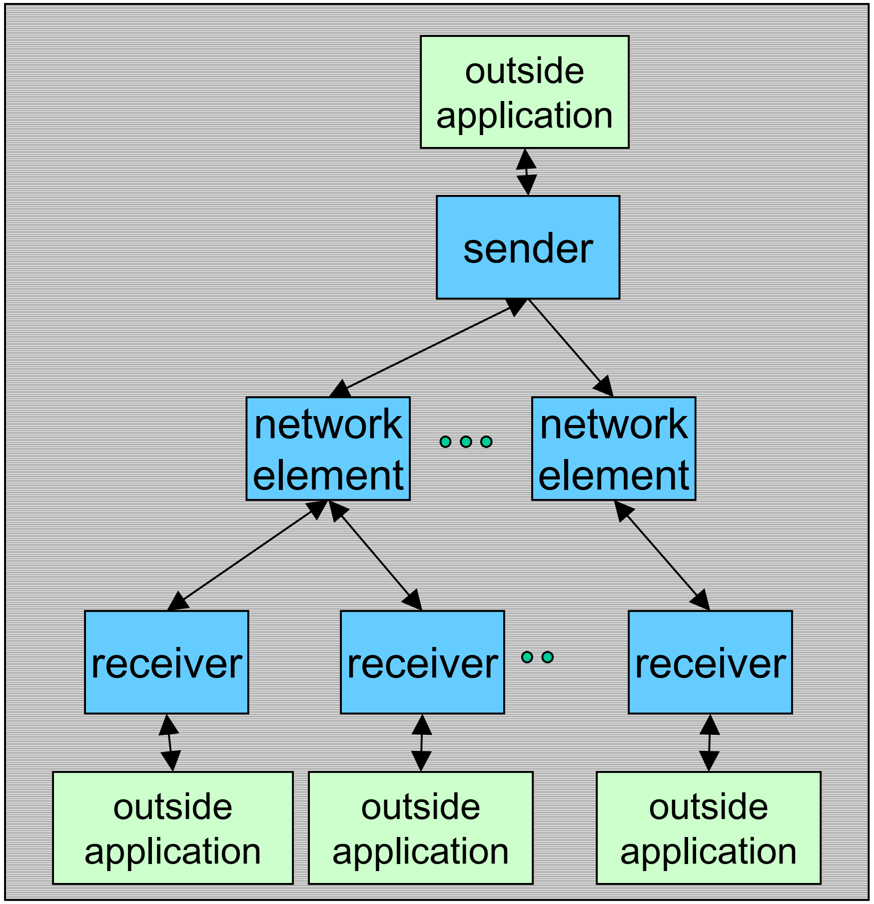
real-time data transmission for multimedia
multicast using tree architecture
generalised sliding window for error recovery
negative acknowledgment
important timing constraints
many parameters (buffer lengths, delays)

SDL specification (~3500 lines)

formalize the IETF draft
developed by France Telecom
translated completely using sdl2if

protocol requirement

any receiver either receives all data packets from transmissions and repairs or is able to detect unrecoverable data loss



pragmatic general multicast

model checking

initial model

limited by the size of state space i.e,

the configuration with 1 sender, 1 network element, 2 receivers, 2 messages sent, arbitrary loss, has more than 200000 states, 800000 transitions

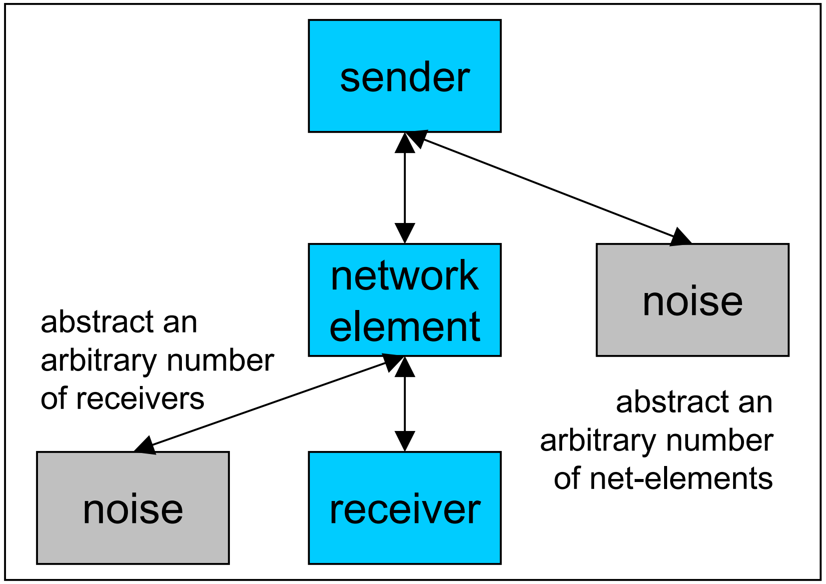
abstract model

abstract the multicast tree as a linear structure + noise processes

scenarios with up to 12 messages sent and arbitrary losses have been considered

safety properties have been verified on the fully generated state space

an error detected w.r.t. to the transmission and recovery of the last packet in a sequence



model exchange

PGM models developed in IF have been exchanged among ADVANCE partners

many other techniques applied on PGM: symbolic reachability, regular model checking, parameter synthesis

embedded software

Ariane 5 Flight Program

joint work with EADS Lauchers

M. Bozga, D. Lesens, L. Mounier. **Model-checking Ariane 5 Flight Program**. In *Proceedings of FMICS 2001, Paris, France*.

K9 Rover Executive

S.Tripakis et al. **Testing conformance of real-time software by automatic generation of observers**. In *Proceedings of Workshop on Runtime Verification, RV'04, Barcelona, Spain*.

ariane 5 flight program

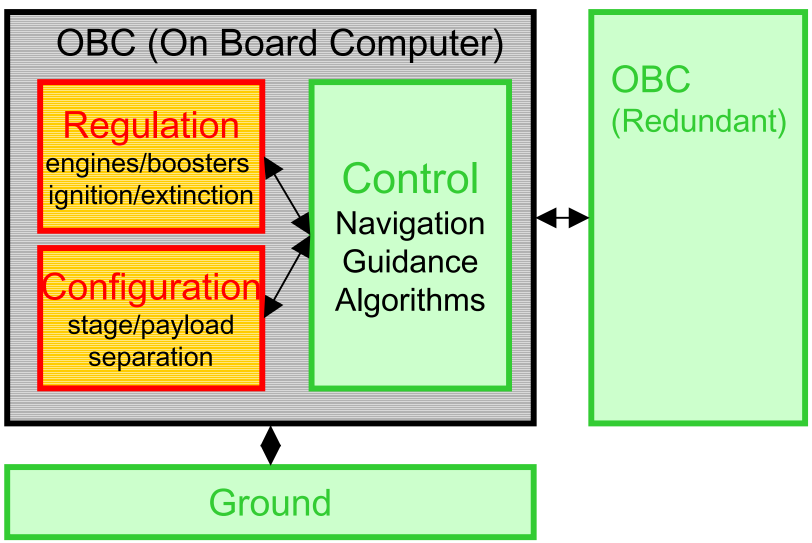
Joint work initiated by EADS-LV to evaluate verification techniques and tools through a specific case-study

fligth program specification

built by reverse engineering by EADS

high level, non-deterministic, abstracts the whole program as communicating extended finite-state machines

~3500 lines of SDL code



fligth program requirements

general requirements

- no deadlock, no timelock
- no implicit signal consumption

overall system requirements

- flight phase order
- stop sequence order

local component requirements

- activation signals arrive eventually in some predefined time intervals

ariane 5 flight program

translation

the SDL specification has been translated completely into IF using sdl2if

urgency of transitions has been explicitely defined to achieve the intended behavior

model exploration

random or guided simulation

several inconsistencies found

static analysis

clock reduction

1st version: 143 clocks reduced to 41 clocks

2nd version : 55 clocks, no more reduction

live variable analysis

20% of all variables are dead in each state

slicing

eliminate passive processes, without outputs

model generation

partial order reduction needed

31 interleaved processes

the full state space can be constructed

195718 states, 278263 transitions

model checking

evaluation of μ-calculus formula

property: the stop sequence no. 3 could happen
only in a flight phase
$\neg \mu$ X. <EPC!Stop3>True$\wedge$ <EAP!Fire>X

build bisimulation reduced models

property: whenever a problem is detected
during the ignition of the Vulcan engine,
then the whole ignition is aborted,
otherwise the launcher eventually lifts off

distributed applications

TCP/ECN Transit Computerization Project

case study proposed in AGEDIS IST-1999-20218

MQ Series Integration Broker

case study proposed in AGEDIS IST-1999-20218

mq series integration broker

specification

lightweight publish/subscribe protocol for integrating devices with WebSphere Integration Broker ™

modeling

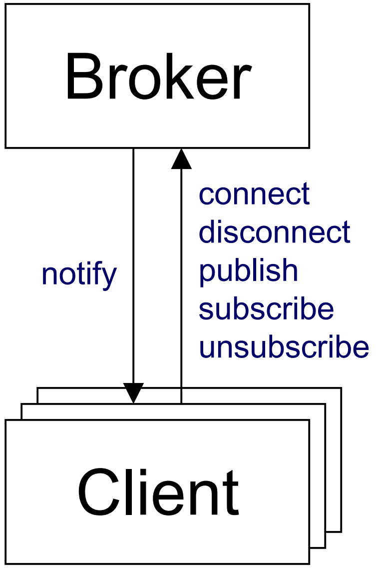
the protocol has been modeled using the AGEDIS Modeling Language AML – an UML profile for testing

IF is an intermediate representation for AML

test generation

several tests have been extracted successfully using TGV / AGEDIS

test directives combines functional goals (e.g, connection establishment, publishing, notifications) and coverage criteria (e.g, return values for methods)



test execution

generated tests have been applied on concrete implementations using SPIDER, the AGEDIS Test Execution Engine

(injected) errors have been discovered

manufacturing

Job-shop Scheduling

Axxom Lacquer Production

case study proposed in AMETIST IST-2001-35304

axxom lacquer production

chemical industry problem

there are 29 lacquers to be produced, each one in some predefined time interval [earliest-start date, due date]

lacquers are of 3 different types, each type has a specific production flow, characterized by the resources involved, processing times, flow constraints, etc.

Problem: find an optimal schedule i.e., with minimal delays for the production of 29 lacquers

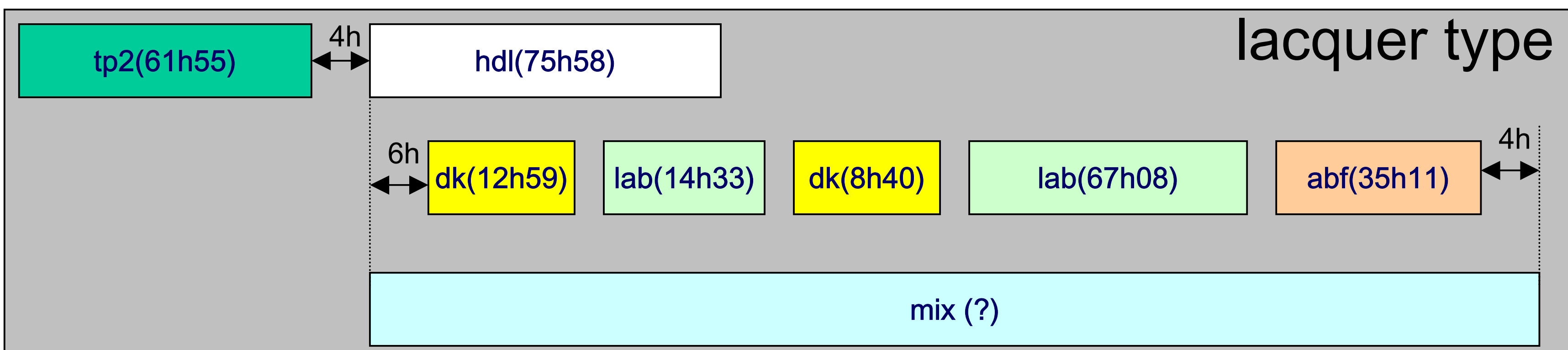
IF-based solution

reduce the scheduling problem to a minimal path cost extraction problem:

model each lacquer as an IF process encoding resource allocation/deallocation order, basic task duration, additional flow constraints

model the production plan as the parallel composition of lacquers automata + resources

The optimal schedule correspond to the minimal cost path leading from the initial state to a state where all lacquers have completed successfully



axxom lacquer production

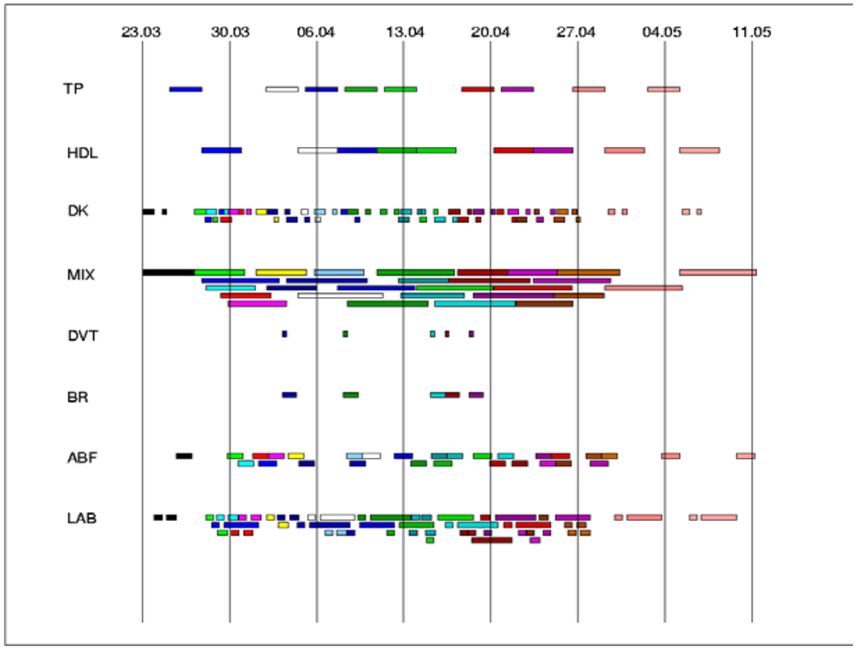
finding an optimal path

the search space is huge because of the interleaving of 29 processes using more than 73 clocks !

several heuristics have been applied at source level to reduce the search:

- avoid lazy runs i.e, remove useless waiting from schedules
- avoid phase overtaking between jobs (lacquers) of the same type i.e, ensure a pipelined execution
- enforce minimal separation time between jobs of the same type

It take 15'' to find that an optimal 0-delay schedule exists on the model an to extract it using the IF optimizer



IF outperform standard MILP (Mixed Integer Linear) approaches on the same case study

but still not all the difficulties of the real case study have been considered e.g,

- batch splitting, operating hours, sequence depending costs, performance factors

asynchronous circuits

timing analysis

O. Maler et al. **On timing analysis of combinational circuits**. In *Proceedings of the 1st workshop on formal modeling and analysis of timed systems, FORMATS'03, Marseille, France*.

functional validation

D. Borrione et al. **Validation of asynchronous circuit verification using IF/CADP**. In *Proceedings of IFIP Intl. Conference on VLSI, Darmstadt, Germany*.

timing analysis

asynchronous circuit problem

knowing individual gate latencies, find the maximal stabilization time of the circuit, for an arbitrary change of inputs

IF-based solution

model each gate as a timed automaton and the circuit as the product of gates

the maximal stabilization time correspond to the maximal delay path leading from the initial state to some next stable state

this method is exact, and therefore more accurate than usual methods which ignore the data part (no false paths !)

nevertheless, we are limited by the size of the circuit (number of gates)