IF simulation and verification tool for UML

VERIMAG

- introduction
- the IF notation
- toolbox architecture
- the UML front-end IFx
- conclusions

The IF toolbox: objectives

Model-based development of real-time systems

Use of high level modeling and programming languages

- Expressivity for faithful and natural modeling
- Cover functional and extra-functional aspects
- Openness

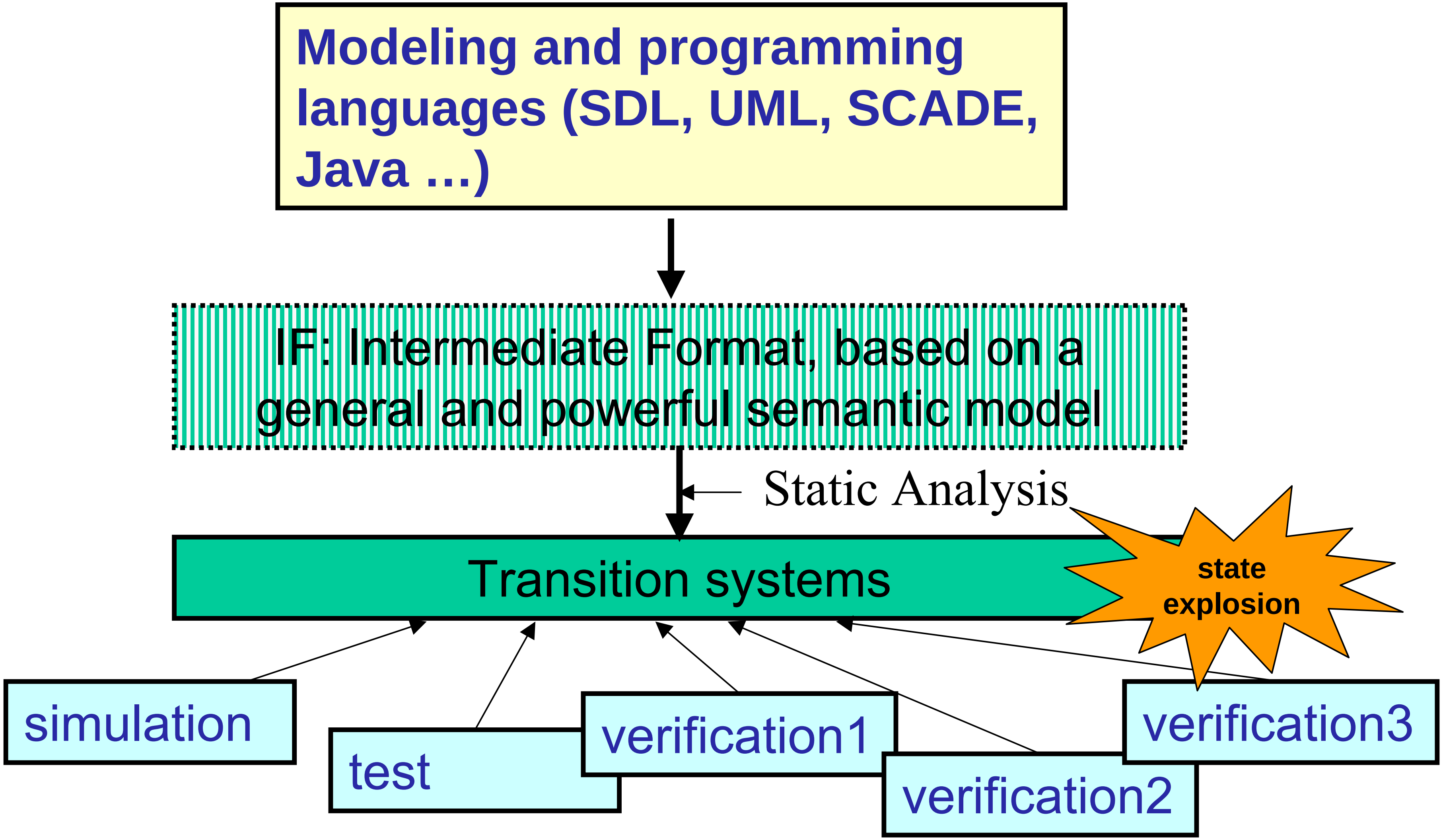
Model-based validation

- Combine static analysis and model-based validation
- Integrate verification, testing, simulation and debugging

Applications:

Protocols, Embedded systems, Asynchronous circuits, Planning and scheduling

The IF toolbox: approach



Modeling and programming languages (SDL, UML, SCADE, Java …)

IF: Intermediate Format, based on a general and powerful semantic model

Static Analysis

Transition systems

state explosion

simulation

test

verification1

verification2

verification3

The IF toolbox: challenges

Find an adequate intermediate representation

Expressiveness**:** direct mapping of concepts and primitives of high modeling and programming languages

- asynchronous, synchronous, timed execution
- buffered interaction, shared memory, method call …

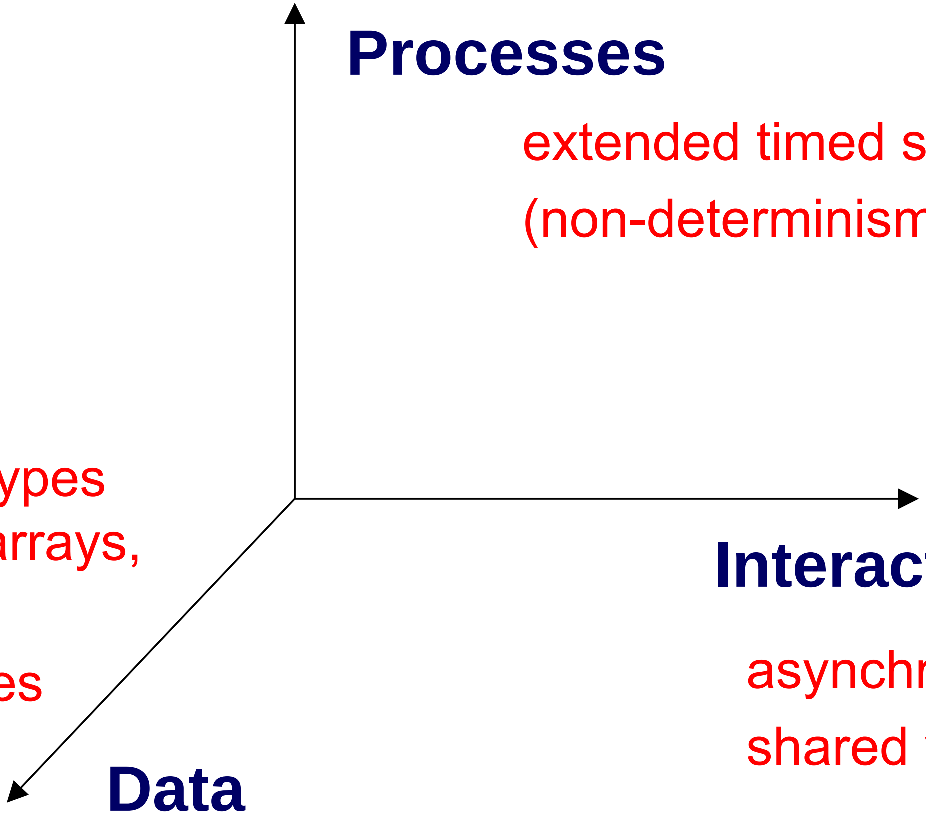
→ Use information about structure for efficient validation and traceability

Semantic tuning**:** when translating languages to express semantic variation points, such as time semantics, execution and interaction modes

overview

- introduction
- the IF notation
- the IF validation tools
- the
- UML front-end (IFx)
- conclusions

System description



Processes

extended timed systems

(non-determinism, dynamic creation)

Interactions

asynchronous channels

shared variables

Data

predefined data types
  (basic types, arrays, records)

abstract data types

System description

A set of interacting processes

- 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 interactions:
  - asynchronous signal exchanges (directly or via signalroutes)
  - shared variables

System description

// processes

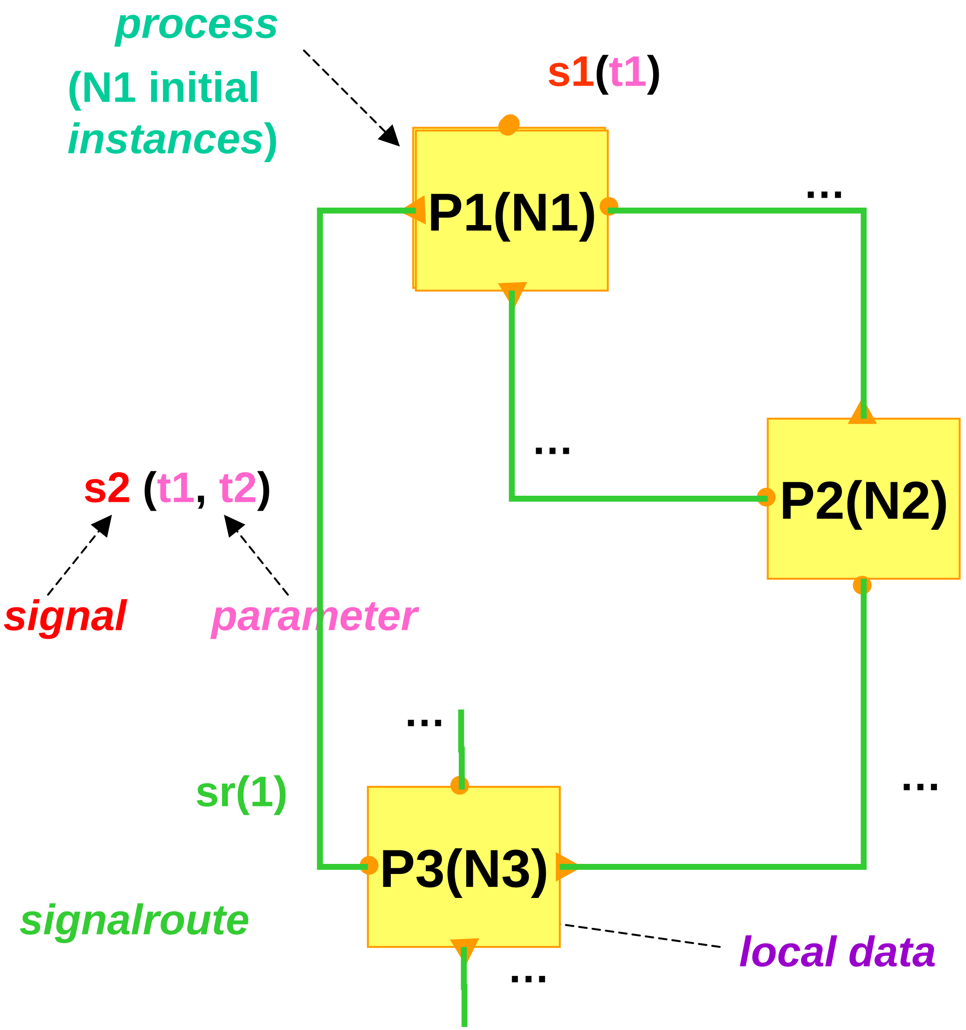
```
process P1(N1)
        ...
endprocess;
...
process P3(N3)
        ...
endprocess;
```

// signalroutes

```
signalroute sr1(1) ...
  from P1 to P3 ;
```

// signals

```
signal s1(t1)
signal s2(t1, t2),
```



Process description

Process = hierarchical timed automaton

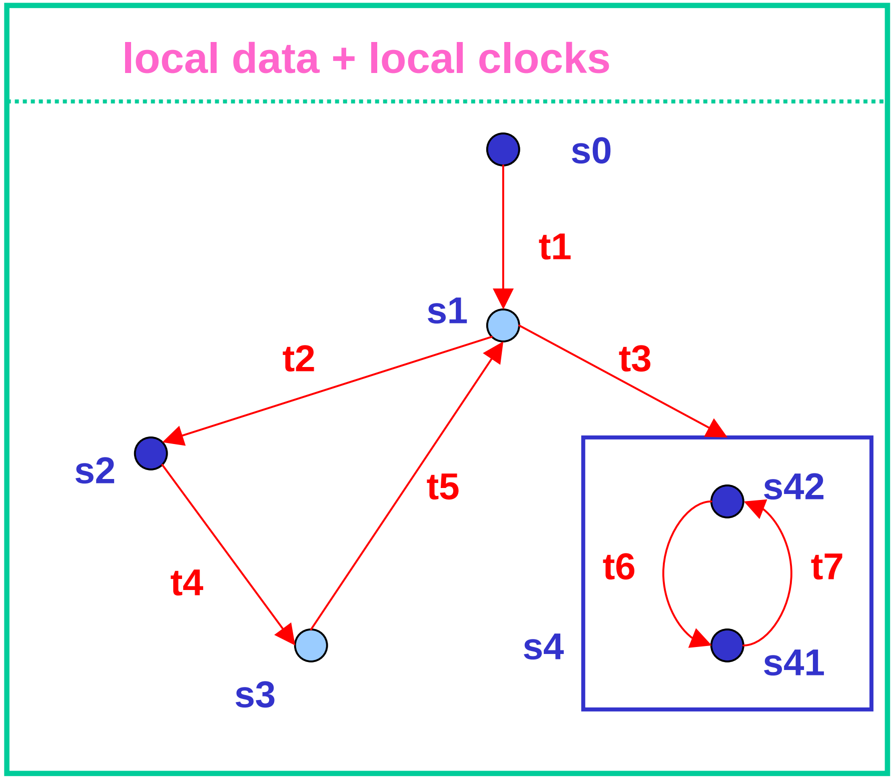
```
process P1(N1);
fpar ... ;

// types, variables, constants,
procedures

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

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

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

parameters

local data

state

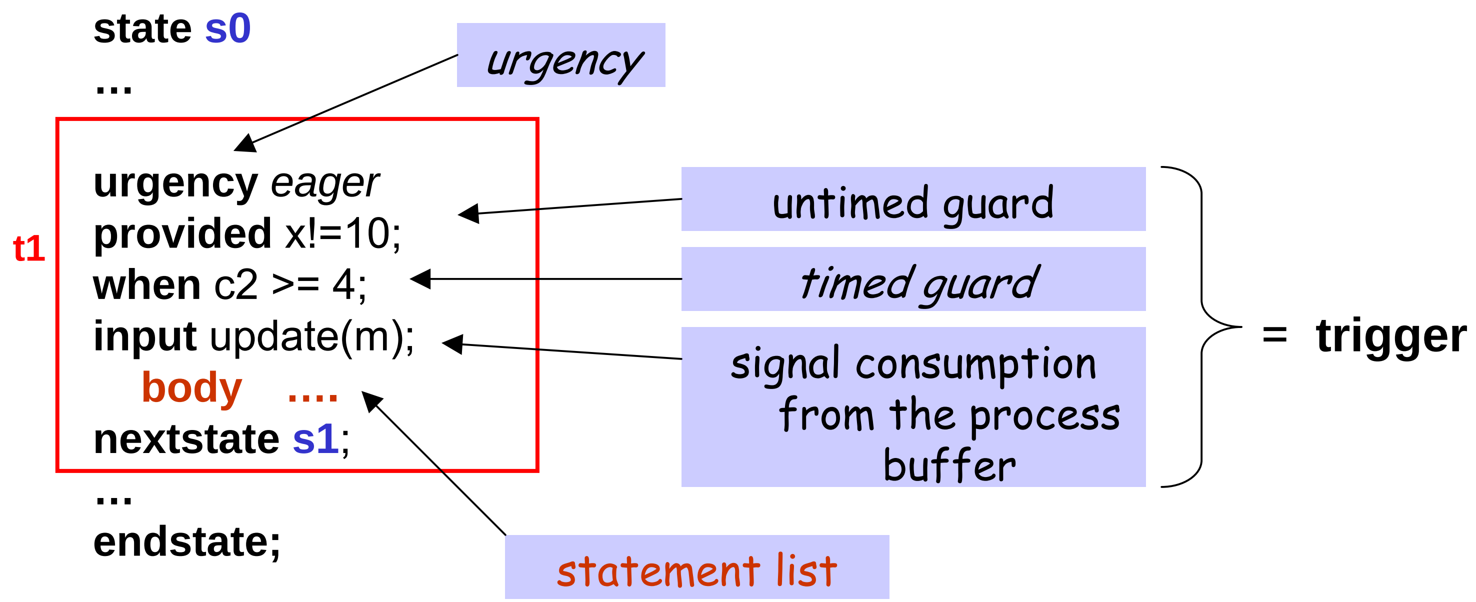
outgoing transitions



P1(N1)

Transitions

transition = *urgency* + trigger + body



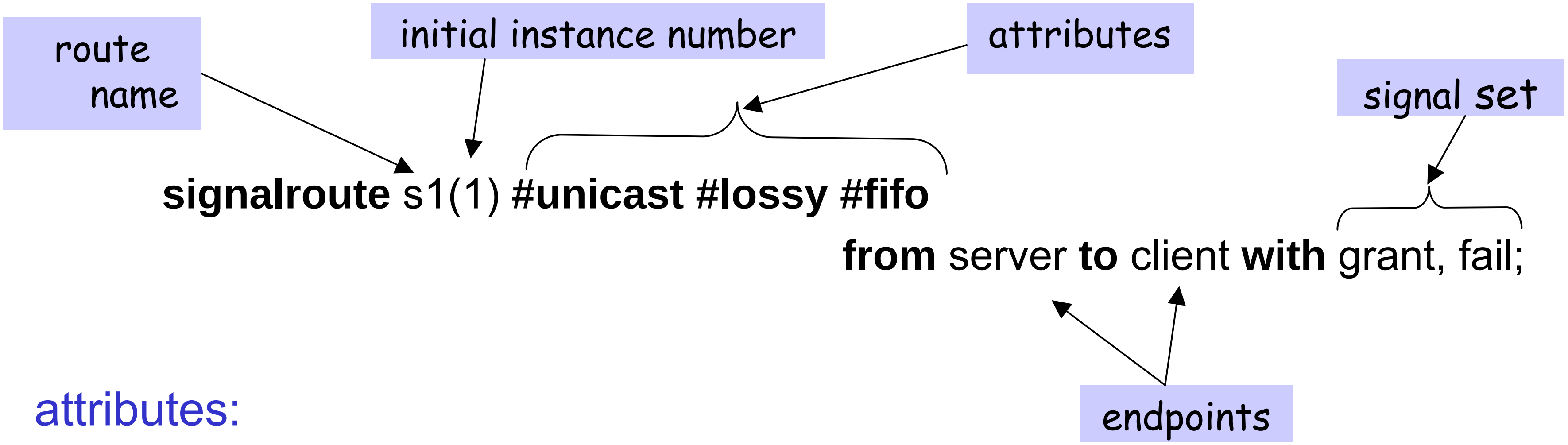
```
state s0
…
t1
urgency eager
provided x!=10;
when c2 >= 4;
input update(m);
   body   ….
nextstate s1;
…
endstate;
```

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

sequential. conditional, or iterative composition

Signal routes

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



route name

initial instance number

attributes

signal set

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

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

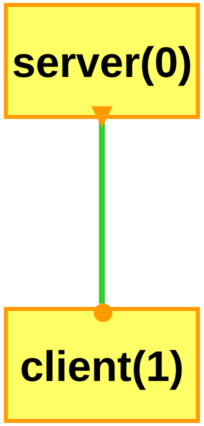
endpoints

attributes:

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

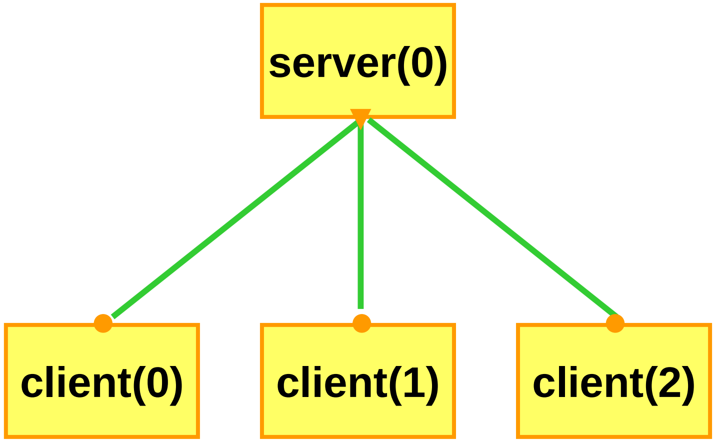
Delivery policies

peer



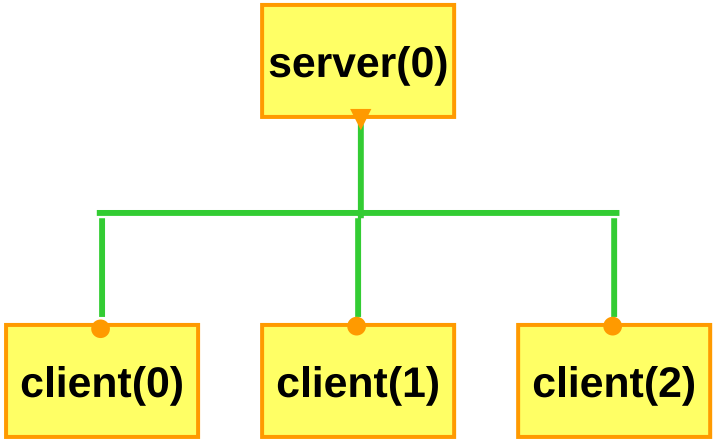
to one
specific
instance

unicast



to a randomly
chosen
instance

multicast

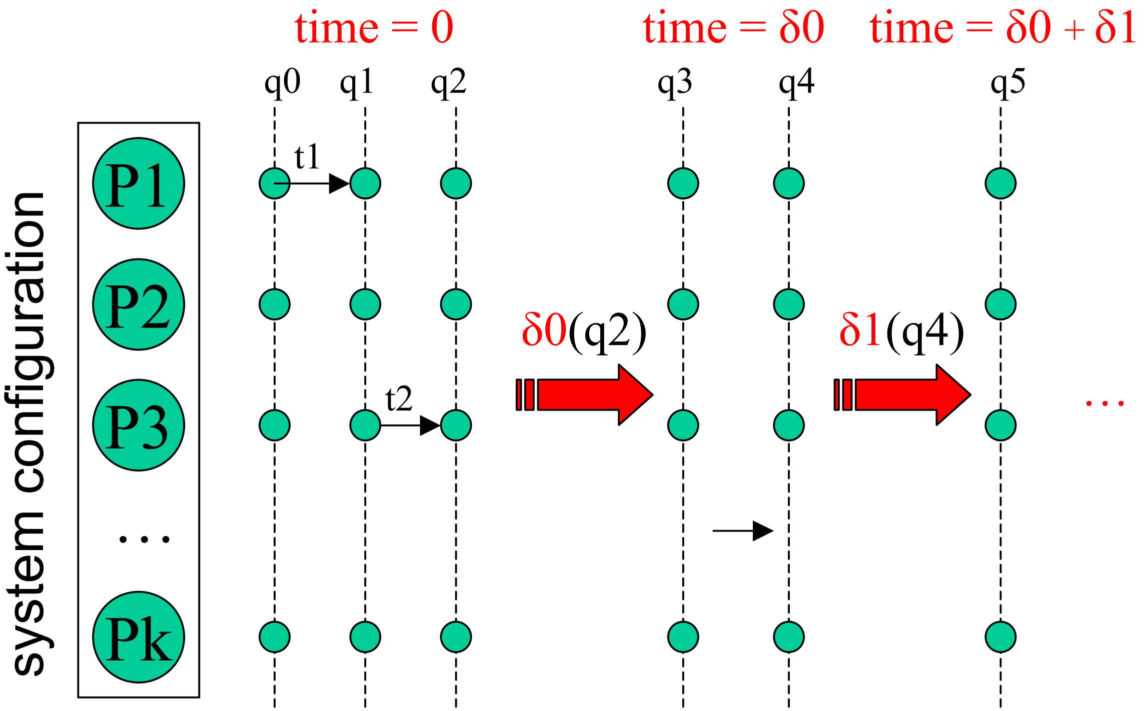


to all instances

Timed behavior

The model of time [timed automata]

- global time → same clock speed in all processes
- time progress in stable states only → transitions are instantaneous



Timed behavior

- operations on clocks
  - set to value
  - deactivate
  - read the value into a variable
- 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:=  10;
  nextstate wait_ack;
endstate;

state wait_ack;
  input ack(sender,c);
  …
  when 10 <t<20 ;
  …
endstate;
```

Dynamic priorities

- priority order between process instances p1, p2 (free variables ranging over the active process set)

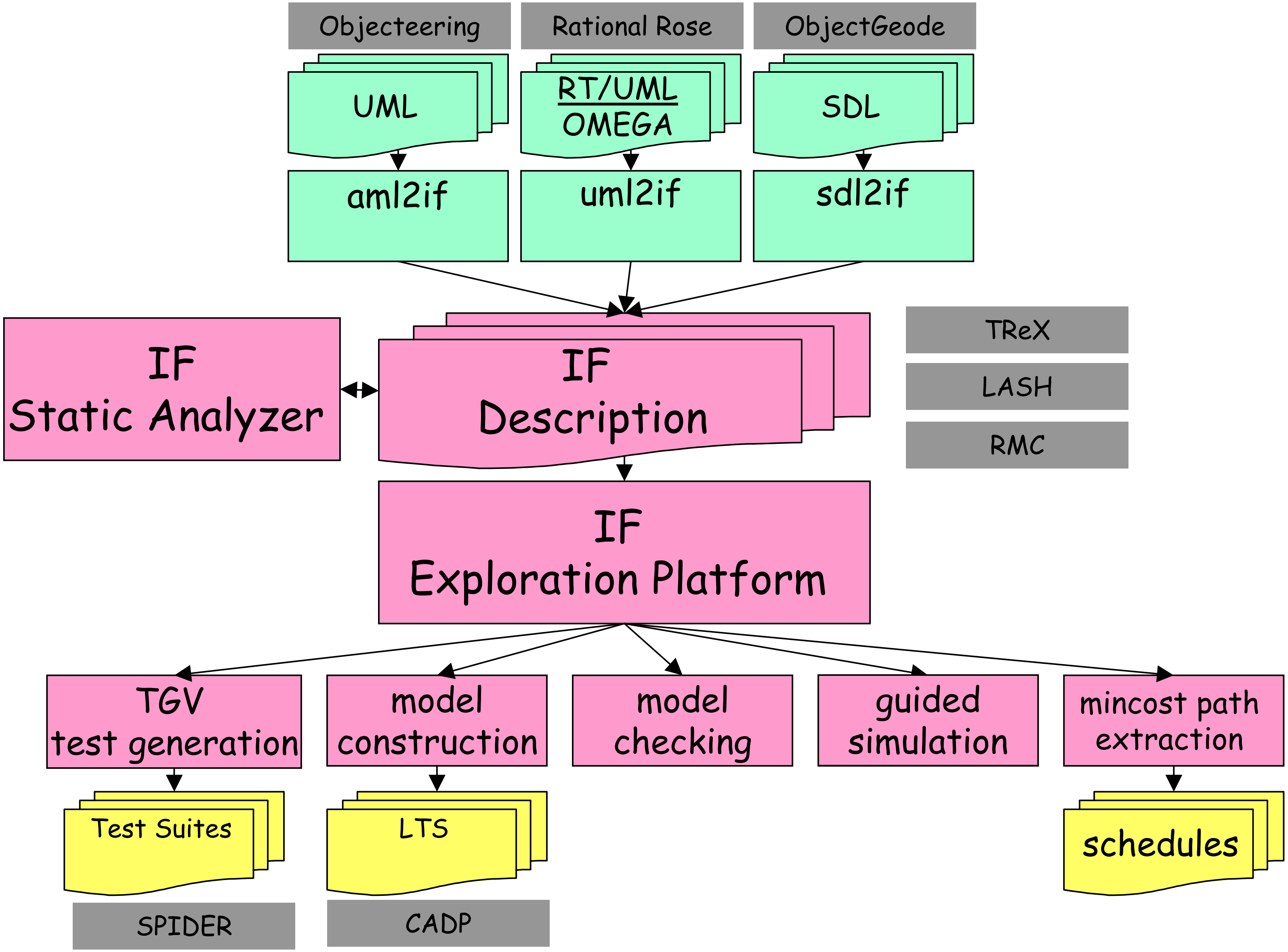
priority_rule_name : p1 < p2 **if** *condition*(p1,p2)

- semantics: *only maximal enabled processes can execute*

- scheduling policies
  - fixed priority: p1 < p2 if p1 instanceof T and p2 instanceof R
  - run-to-completion: p1 < p2 if p2 = manager(0).running
  - EDF: p1 < p2 if Task(p2).timer < Task(p1).timer

overview

- introduction
- the IF notation
- the IF validation tools
- the UML front-end (IFx)
- conclusions



Objecteering

Rational Rose

ObjectGeode

UML

RT/UML OMEGA

SDL

aml2if

uml2if

sdl2if

IF Static Analyzer

IF Description

TReX

LASH

RMC

IF Exploration Platform

TGV test generation

model construction

model checking

guided simulation

mincost path extraction

Test Suites

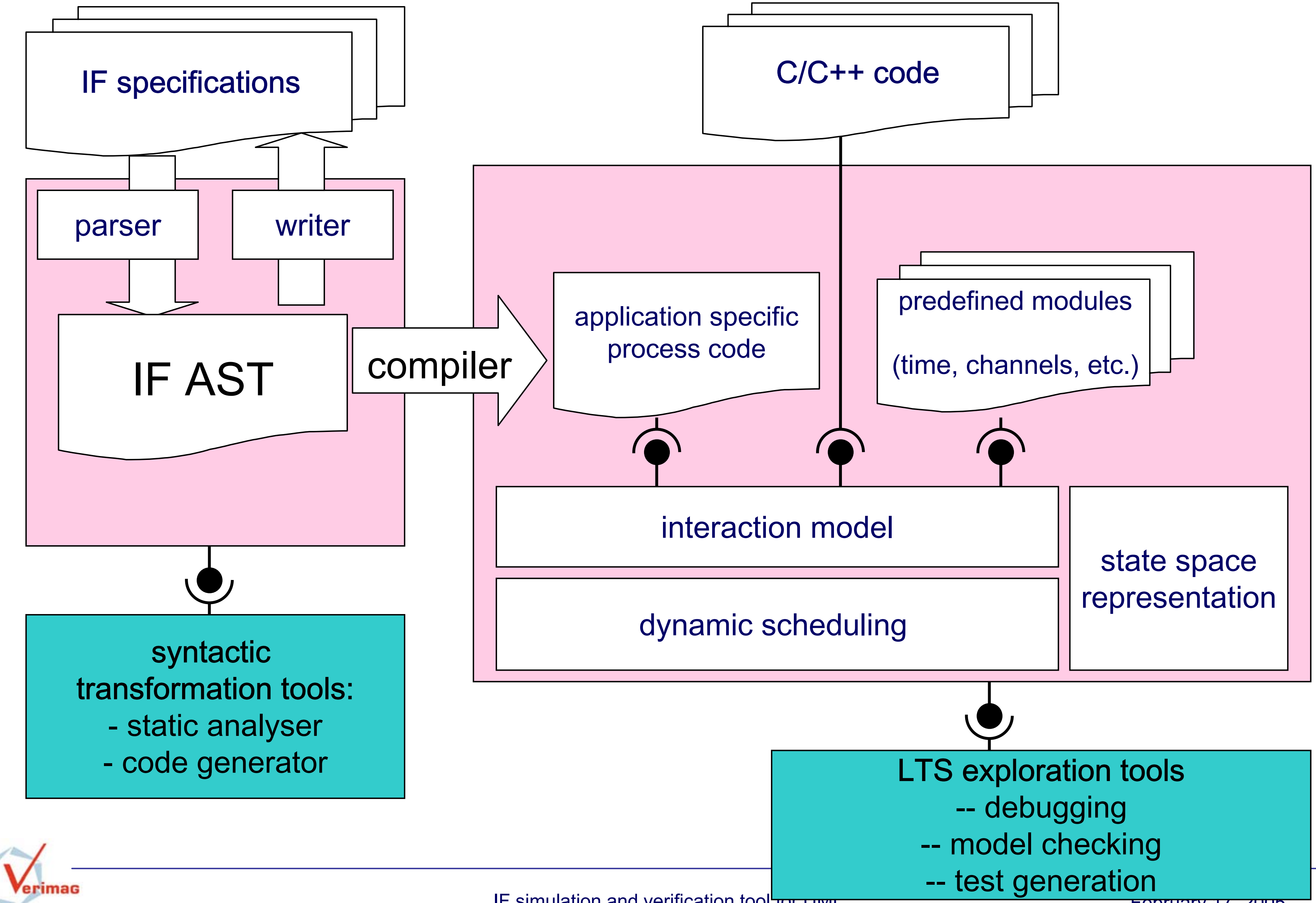
LTS

schedules

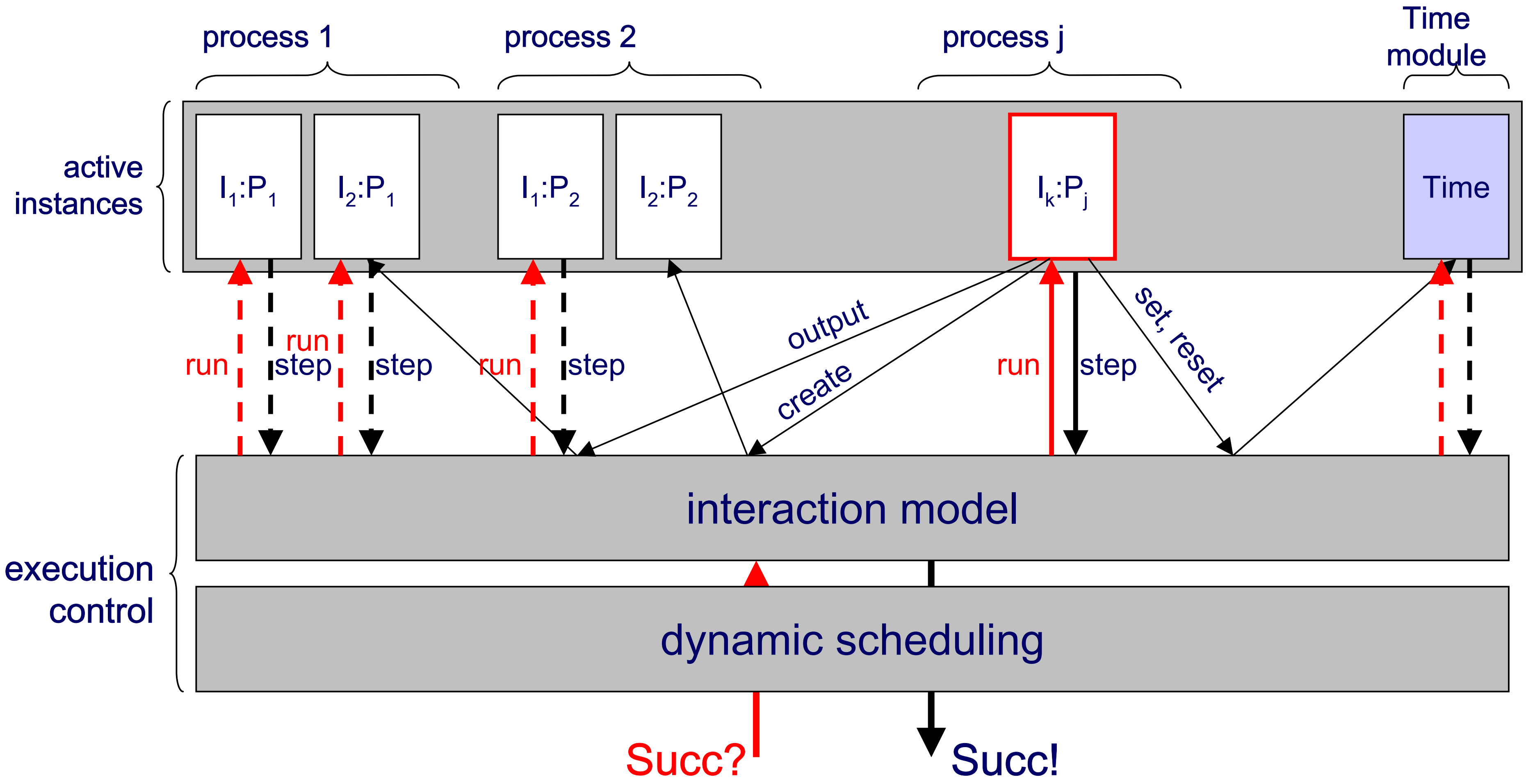
SPIDER

CADP

Core components



Exploration platform



Dealing with Time

Dedicated module

- including clock variables
- handling dynamic clock allocation (set, reset)
- checking timing constraints (timed guards)
- computing time progress conditions w.r.t. actual deadlines and
- fires timed transitions, if enabled

Two implementations for discrete and continuous time (others can be easily added)

i) discrete time

- clock valuations represented as varying size integer vectors
- time progress is explicit and computed w.r.t. the next enabled deadline

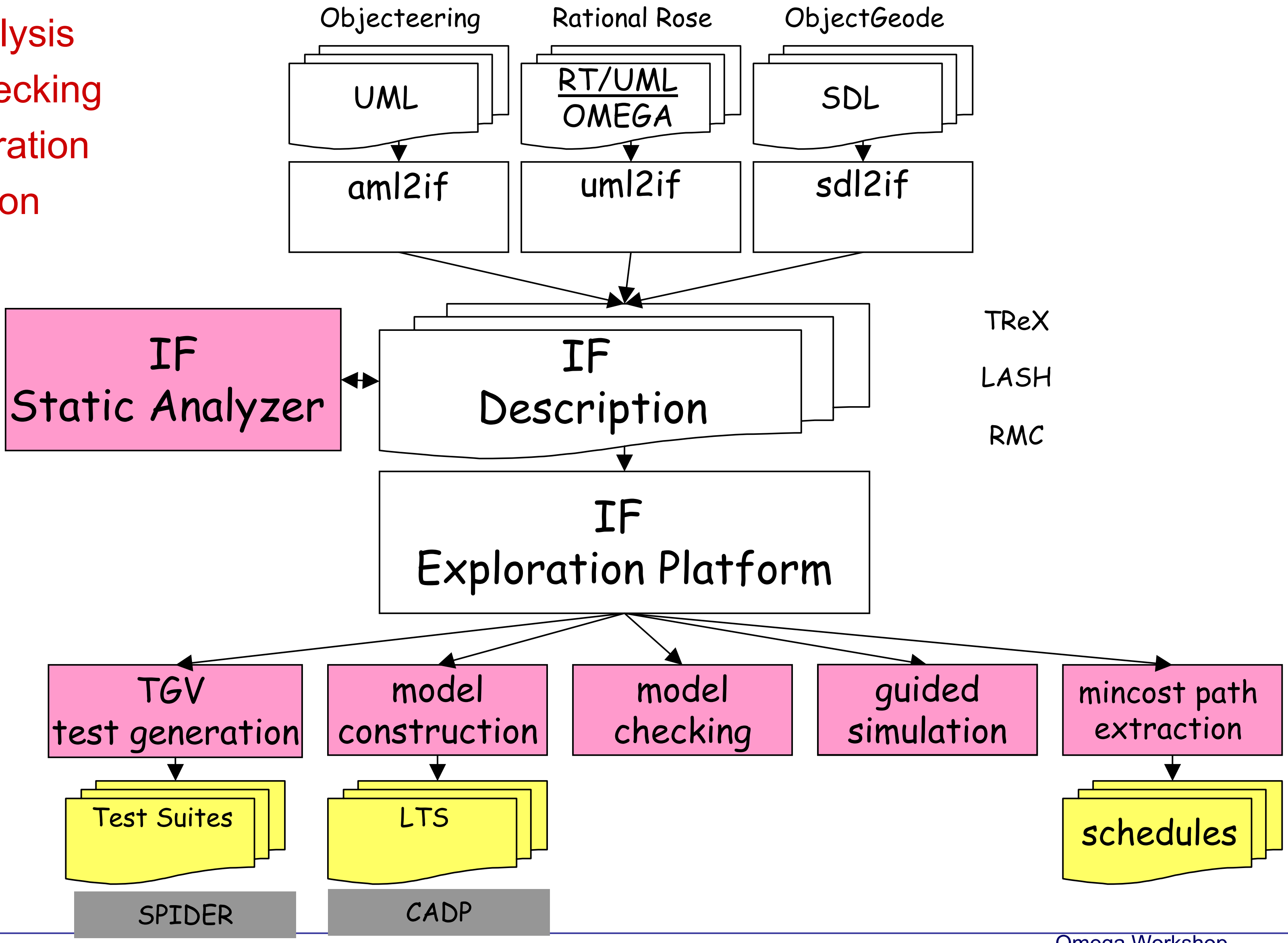
ii) continuous time

- clock valuations represented using varying size difference bound matrices (DBMs)
- time progress represented symbolically
- non-convex time zones may arise because of deadlines: they are represented implicitly as unions of DBMs

Validation tools

Model-Based Validation

- static analysis
- model checking
- test generation
- optimization



Objecteering

Rational Rose

ObjectGeode

UML

RT/UML
OMEGA

SDL

aml2if

uml2if

sdl2if

IF
Static Analyzer

IF
Description

TReX

LASH

RMC

IF
Exploration Platform

TGV
test generation

model
construction

model
checking

guided
simulation

mincost path
extraction

Test Suites

LTS

schedules

SPIDER

CADP

Static analysis

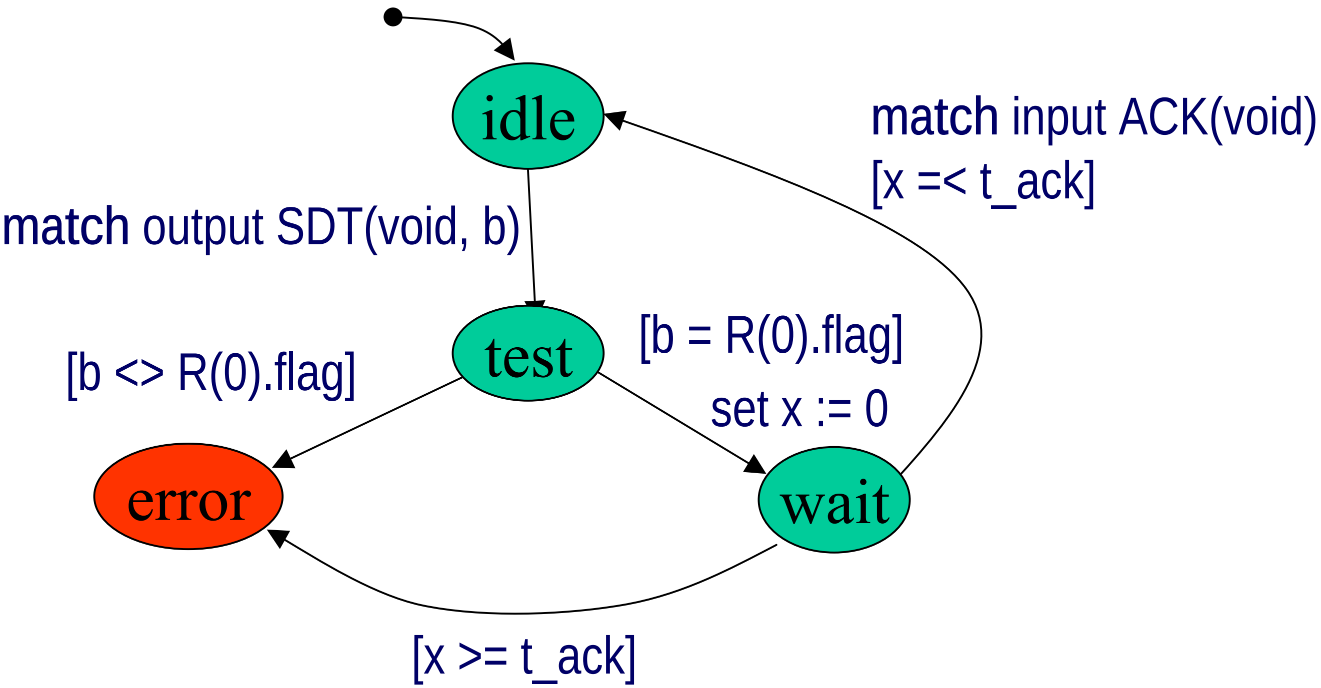
- approach
  - source code transformations for model reduction
  - code optimization methods

- 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

Model-checking using observers

- **Observers** are used to specify safety properties in an operational way
- They are described as the processes – specific command for monitoring events, system state, elapsed time
- 3 types of states : normal / error / success
- **Semantics:** Transitions triggered by monitored events and executed with highest priority



Behavioral equivalence checking

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

    System $\approx$ Requirements

  - preorder relations (“behavior inclusion”):

    System $\leq$ Requirements

- LTS minimization:
  - quotient w.r.t an equivalence relation:

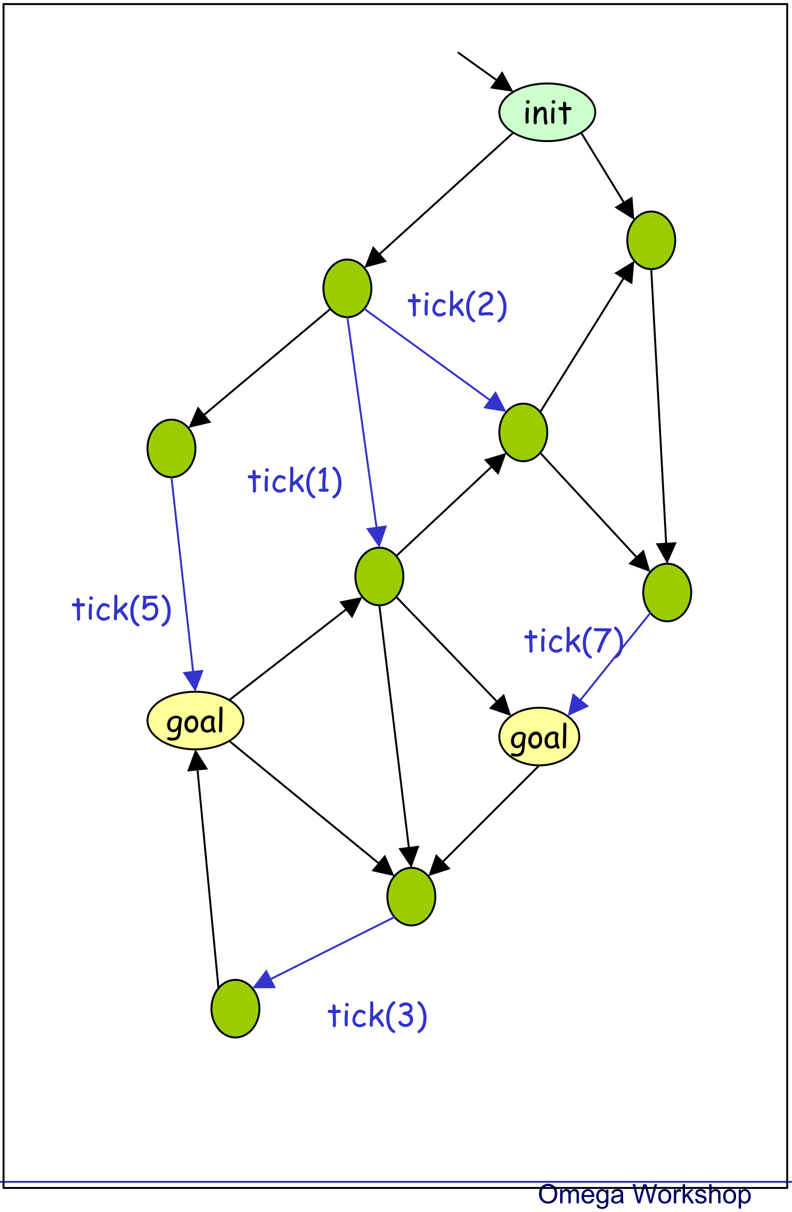
    (System / $\approx$)

- CADP can be used to check the following relations :

  weak/strong bisimulation, branching, safety, trace equivalence

Optimization

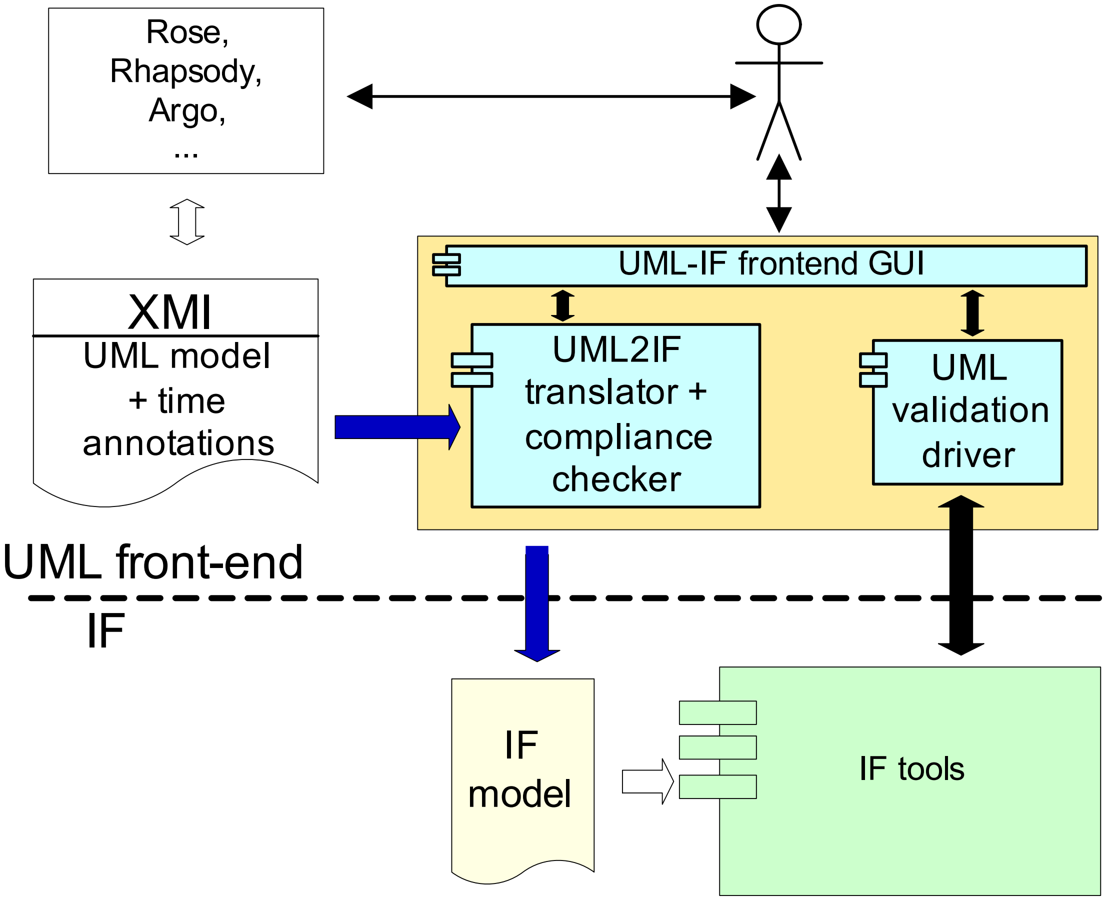
- User defined costs associated to transitions of IF descriptions e.g, execution times
- problem: find the min-cost execution path leading from the initial state to some goal state
- three algorithms implemented:
  - Dijkstra algorithm (best first)
  - A* algorithm (best first + estimation)
  - branch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



overview

- introduction
- the IF notation
- the IF validation tools
- the UML front-end (IFx)
- conclusions

the UML front-end (IFx)



Rose,
Rhapsody,
Argo,
...

XMI

UML model
+ time
annotations

UML-IF frontend GUI

UML2IF translator + compliance checker

UML validation driver

UML front-end

IF

IF model

IF tools

UML-to-IF compiler

Coverage of the Omega-UML profile

- fully OO models : classes with operations, attributes, associations, generalization, statecharts; basic data types; the Omega Action Language (compatible to UML1.4 A.S.)
- UML observers for specifying requirements
- timing constraints

Tool connection

- XMI 1.0 or 1.1 for UML 1.4 : Rational Rose, I-Logix Rhapsody and others.

compilation of UML elements

- structure
  - UML object → IF process
  - attributes & associations → variables
  - inheritance : replication of structural features
- behavior
  - state machines, actions → syntactic translation (almost)
  - operations X::m(x,y,…)

    ⇒ one IF process for every invocation of X::m

    process X::m(x, y, …)

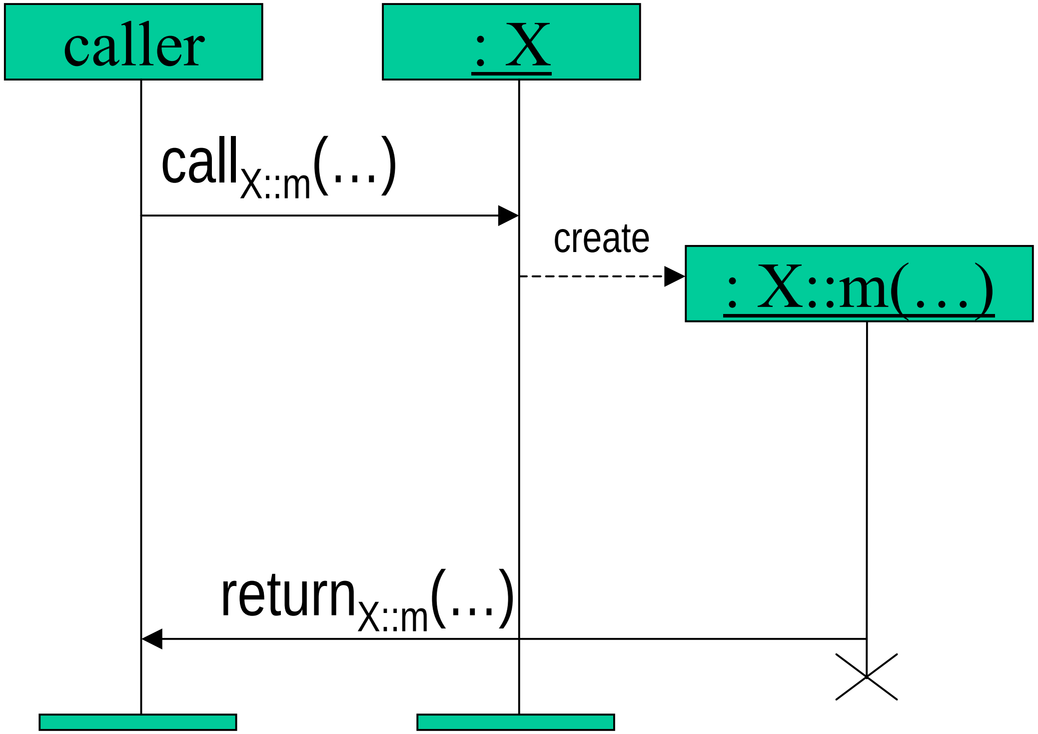
    - lives the period of activation, implements behavior
    - encapsulates the "stack frame" variables

    ⇒ predefined signals

    $\text{call}_{X::m}$, $\text{return}_{X::m}$

    - flexible : adaptable to different call semantics (async, with futures…)

compilation of method calls



polymorphism, concurrency…

polymorphism ⇒ dynamic binding resolved with signals

- the object state machine decides the method implementation which is executed in response to $call_{X::m}$

concurrency ⇒ activity group management

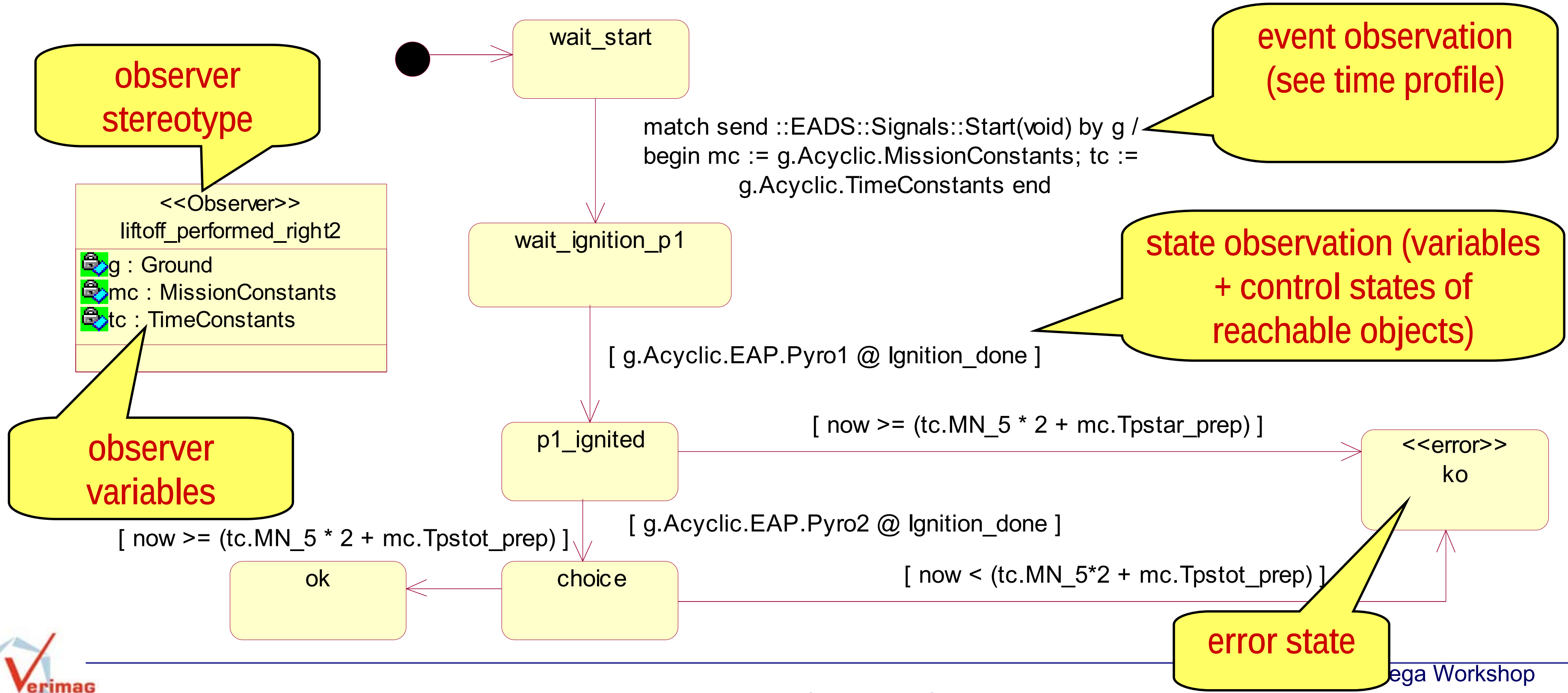
- each active object has an associated group manager
- it handles/dispatches external calls for objects of the group
- keeps track of the running object

run-to-completion

- implemented with dynamic priority rules
  e.g. : $\forall x,y.\ (x.manager = y) \Rightarrow x < y$

formalizing requirements : UML observers

- special objects monitoring the system state / events
- example (Ariane-5) : ***If the Pyro1 object enters state “Ignition_done”, then the Pyro2 object shall enter the state “Ignition_done” after the time TimeConstants.MN_5*2 + Tpstot_prep and before the time TimeConstants.MN_5*2 + Tpstar_prep.***



observer stereotype

<<Observer>>
liftoff_performed_right2

g : Ground
mc : MissionConstants
tc : TimeConstants

observer variables

wait_start

match send ::EADS::Signals::Start(void) by g / begin mc := g.Acyclic.MissionConstants; tc := g.Acyclic.TimeConstants end

event observation (see time profile)

wait_ignition_p1

[g.Acyclic.EAP.Pyro1 @ Ignition_done]

state observation (variables + control states of reachable objects)

p1_ignited

[now >= (tc.MN_5 * 2 + mc.Tpstar_prep)]

<<error>>
ko

[g.Acyclic.EAP.Pyro2 @ Ignition_done]

choice

[now >= (tc.MN_5 * 2 + mc.Tpstot_prep)]

ok

[now < (tc.MN_5*2 + mc.Tpstot_prep)]

error state

the Omega timing framework

time is global, external to the system

- imperative constructs : describe *time dependent behavior*
  - time-related primitive types Time, Duration, 0-ary operator *now : Time*
  - mechanisms for measuring durations: timers, clocks

}⇒TA

- declarative constructs : timed events and constraints
  - orthogonal to the functional specification

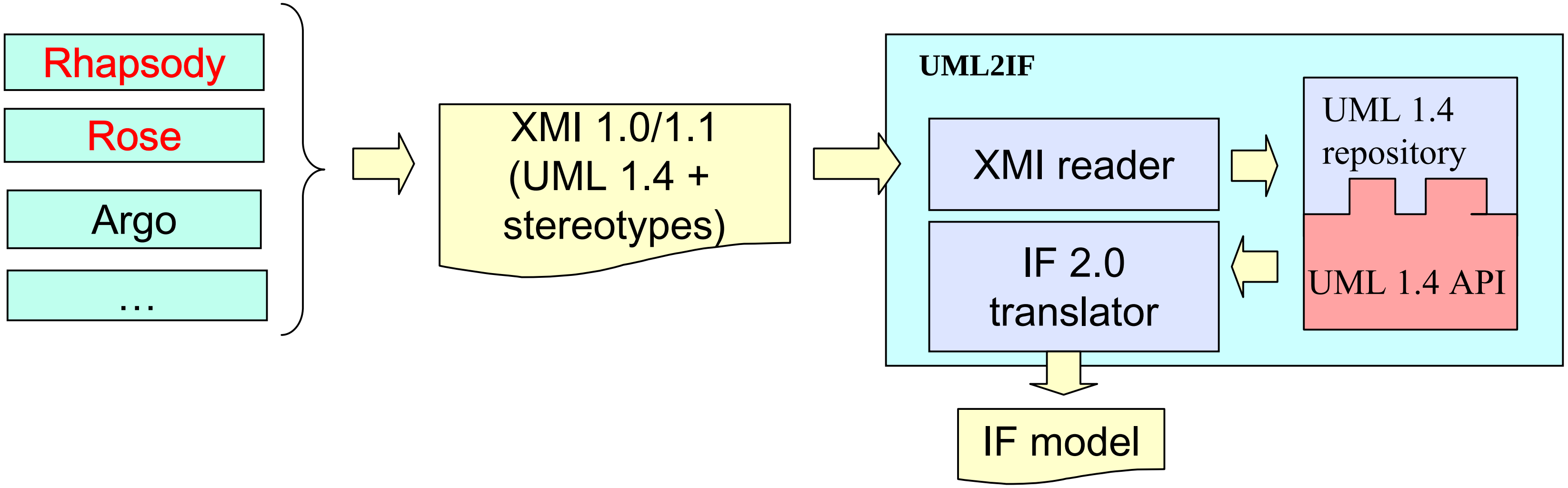
    ⇒separation of modeling concerns

    ⇒flexible semantics : independent from semantics of the functional part*

  - timed events: history of occurrence times of identified state changes
    - *Sending, receiving, consuming* a signal, *Calling, receiving, accepting, answering* an operation
    - *Executing* an action / a state machine transition, *Entering/exiting* a state, *Creating* an object,…
  - constraints on duration between event occurrences
    - Assumptions (taken as hypotheses)
    - Requirements (to be verified)

} ⇒ observers

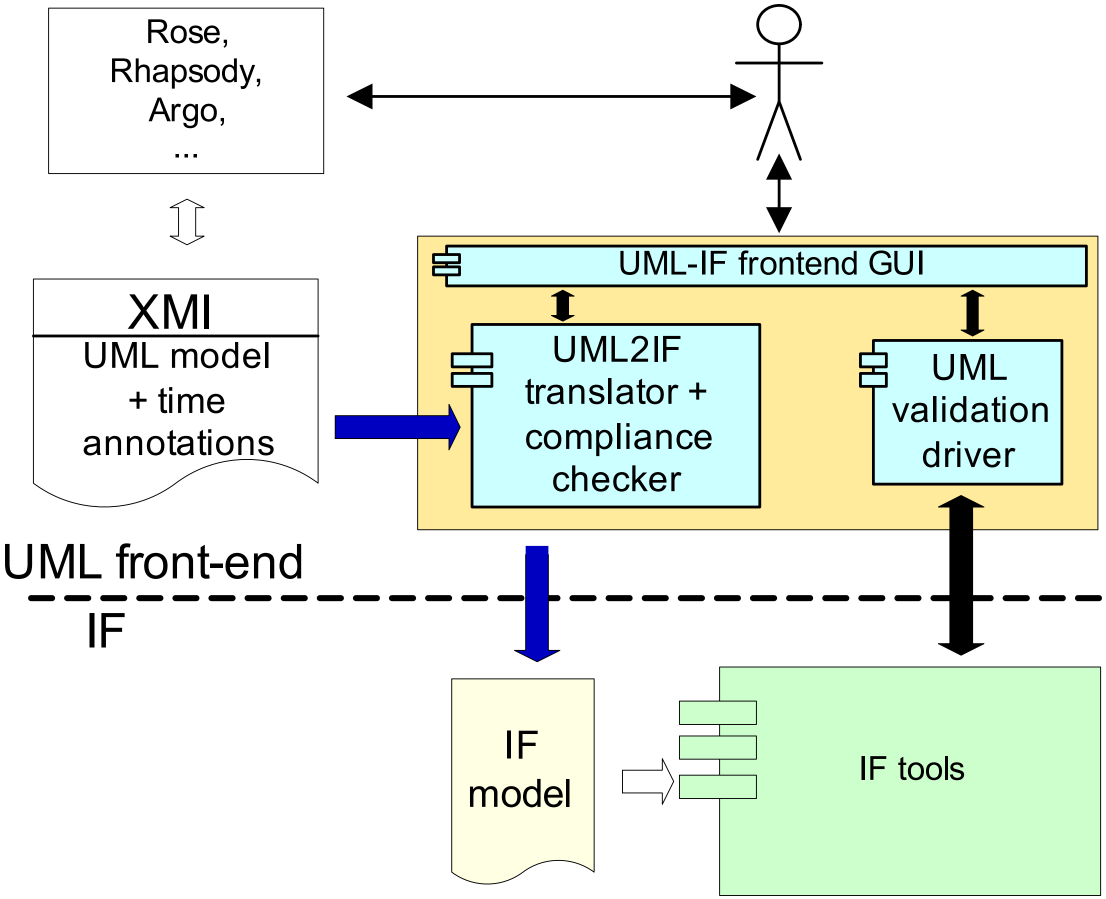
UML-to-IF compiler



Options

- ignore parts of the model
- eager / lazy semantics
- ………………………………

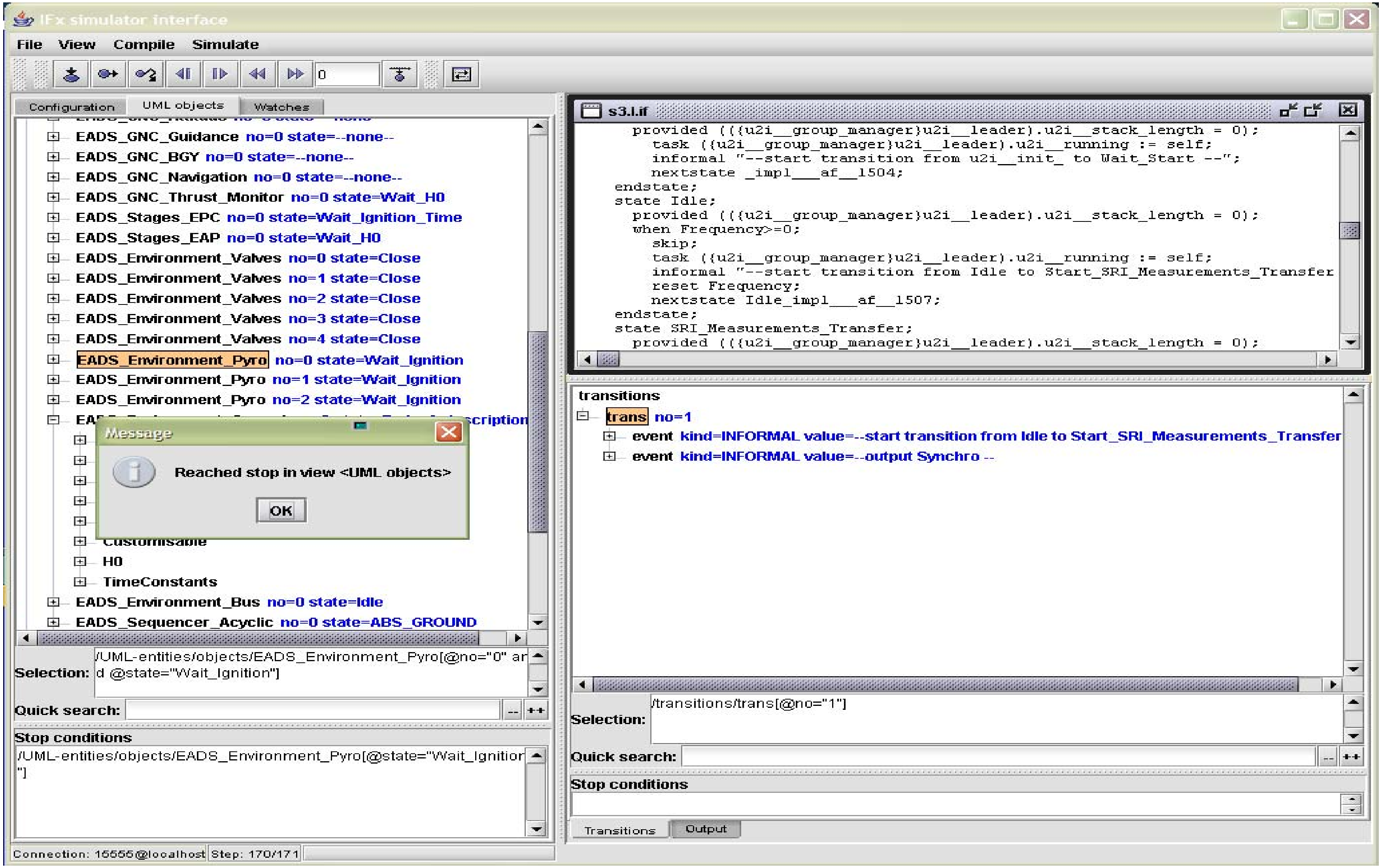
the UML front-end (IFx)



the front-end GUI

- Wrapper of IF tools → (partly) hidden from the UML user
- Functionality :
  - interactive simulation / diagnostics analysis
  - scenario rewind / reply / load /saves
  - source tracing
  - conditional breakpoints
  - UML and other customized views (XSLT stylesheets)
  - batch tool launcher (compilers, verification)

front-end



```
EADS_GNC_Guidance no=0 state=--none--
EADS_GNC_BGY no=0 state=--none--
EADS_GNC_Navigation no=0 state=--none--
EADS_GNC_Thrust_Monitor no=0 state=Wait_H0
EADS_Stages_EPC no=0 state=Wait_Ignition_Time
EADS_Stages_EAP no=0 state=Wait_H0
EADS_Environment_Valves no=0 state=Close
EADS_Environment_Valves no=1 state=Close
EADS_Environment_Valves no=2 state=Close
EADS_Environment_Valves no=3 state=Close
EADS_Environment_Valves no=4 state=Close
EADS_Environment_Pyro no=0 state=Wait_Ignition
EADS_Environment_Pyro no=1 state=Wait_Ignition
EADS_Environment_Pyro no=2 state=Wait_Ignition
    Customisable
    H0
    TimeConstants
EADS_Environment_Bus no=0 state=Idle
EADS_Sequencer_Acyclic no=0 state=ABS_GROUND
```

Reached stop in view <UML objects>

Selection: /UML-entities/objects/EADS_Environment_Pyro[@no="0" and @state="Wait_Ignition"]

Quick search:

Stop conditions

/UML-entities/objects/EADS_Environment_Pyro[@state="Wait_Ignition"]

Connection: 15555@localhost Step: 170/171

```
    provided (({u2i__group_manager}u2i__leader).u2i__stack_length = 0);
      task ({u2i__group_manager}u2i__leader).u2i__running := self;
      informal "--start transition from u2i__init_ to Wait_Start --";
      nextstate _impl___af__1504;
  endstate;
  state Idle;
    provided (({u2i__group_manager}u2i__leader).u2i__stack_length = 0);
    when Frequency>=0;
      skip;
      task ({u2i__group_manager}u2i__leader).u2i__running := self;
      informal "--start transition from Idle to Start_SRI_Measurements_Transfer
      reset Frequency;
      nextstate Idle_impl___af__1507;
  endstate;
  state SRI_Measurements_Transfer;
    provided (({u2i__group_manager}u2i__leader).u2i__stack_length = 0);
```

```
transitions
  trans no=1
    event kind=INFORMAL value=--start transition from Idle to Start_SRI_Measurements_Transfer
    event kind=INFORMAL value=--output Synchro --
```

Selection: /transitions/trans[@no="1"]

Quick search:

Stop conditions

case studies in Omega

Ariane-5 case study (EADS) – developed in Rational Rose

- statically validate the well formedness of the model wrt the Omega profile,
- proved 9 safety properties of the flight regulation and configuration components,
- analyzed the schedulability of the cyclic / acyclic components under the assumption of fixed priority preemptive scheduling policy,
- proved a safety property concerning bus read/write coherence under this policy

MARS case study (NLR) – developed in I-Logix Rhapsody

- static validation
- proved 4 safety properties concerning the correctness of the MessageReceiver,
- discover reactivity limits of the MessageReceiver and to fine-tune its behavior in order to improve reactivity.

Sensor Voting an Monitoring case study (IAI)– developed in Rational Rose

- static validation
- proved 4 safety properties concerning the timing of data acquiring by the three Sensors: end-to-end duration, duration between consecutive reads, etc.

Conclusions

- IF is an unique platform:
  - relates functional and non-functional aspects:
    - distribution, communication, external C/C++ code, dynamic creation,
    - real-time, deadlines, resources, dynamic priorities
  - integrates state of the art validation techniques and tools
  - provides front-end to SDL/UML and related tools
- one step further, component-based modeling and validation
  - combination of synchronous and asynchronous systems (e.g, Lustre/Esterel/StateCharts and SDL/UML)
  - composability of models and compositional reasoning

Discussion : validation

Combination of static analysis and validation techniques proves to be crucial for coping with complexity and broadens the scope of application of the tool e.g.,

- use static analysis for data intensive applications
- use partial order reduction techniques for control intensive applications

The use of high level languages incurs additional costs wrt low level modeling languages

- There is a price to pay for enhanced expressivity and faithful modeling
- Abstraction and simplification can be carried out automatically by static analysis

Observers are a powerful formalisms for safety requirements

- Easy to use by practitioners
- Limitation to safety properties is not a serious one, especially for RT systems