The IF toolset

VERIMAG

M. Bozga, S. Graf, L. Mounier, Y. Lakhnech, Il. Ober, Iu. Ober, J. Sifakis

4th International School on
Formal Methods for the Design of
Computer, Communication and Software Systems:
Real Time

September 2004

Model-based development of real-time systems

Use of high level modeling and programming languages

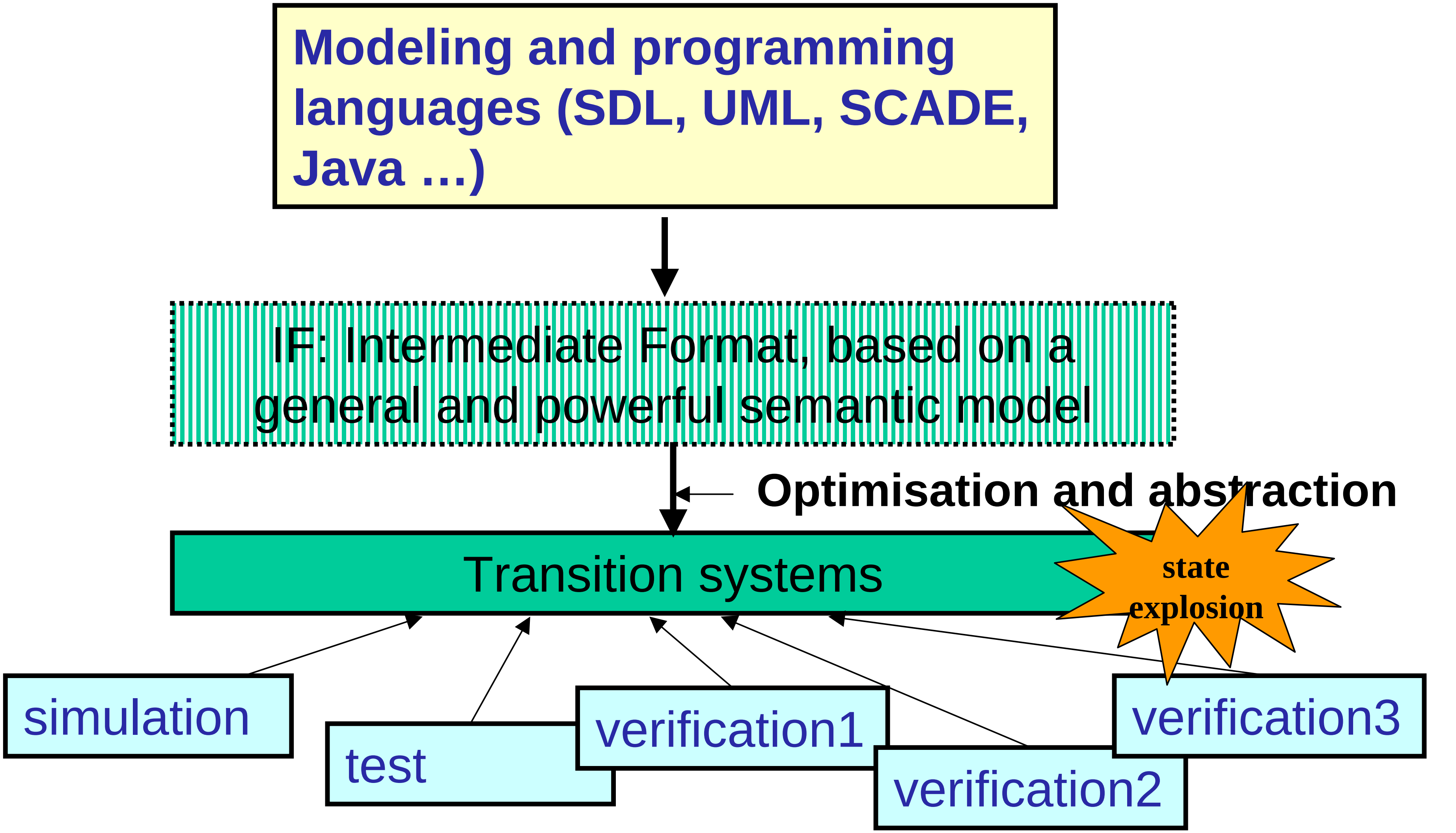
- Expressivity for faithful and natural modeling
- Cover functional and extra-functional aspects
- Openness

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 toolset: approach



The IF toolset: challenges for IF

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

Outline

Key Research issues

- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework

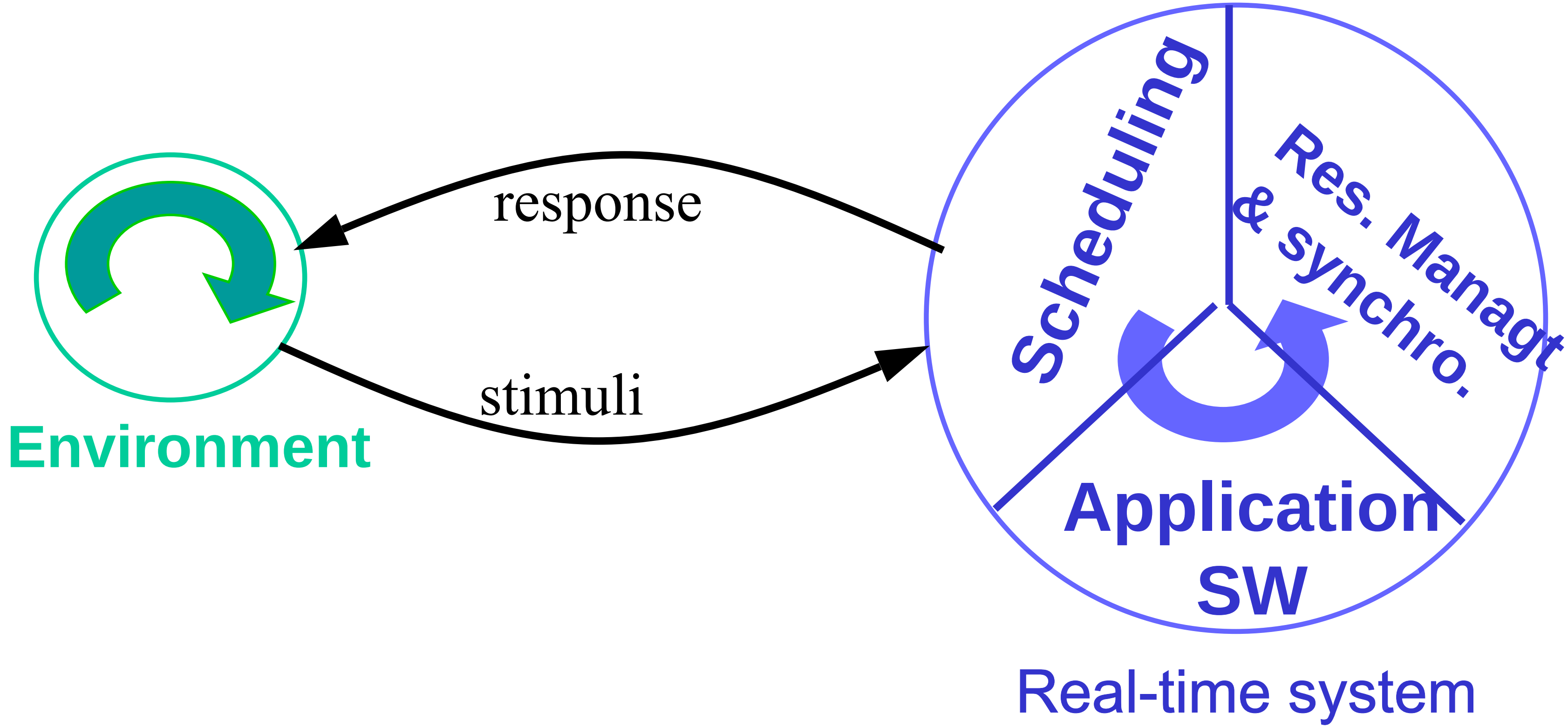
- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

The IF toolset

- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion

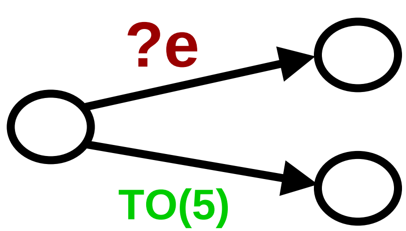
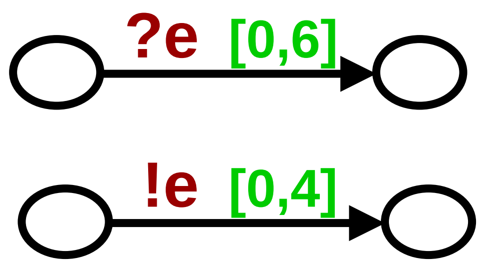
Modeling real-time systems



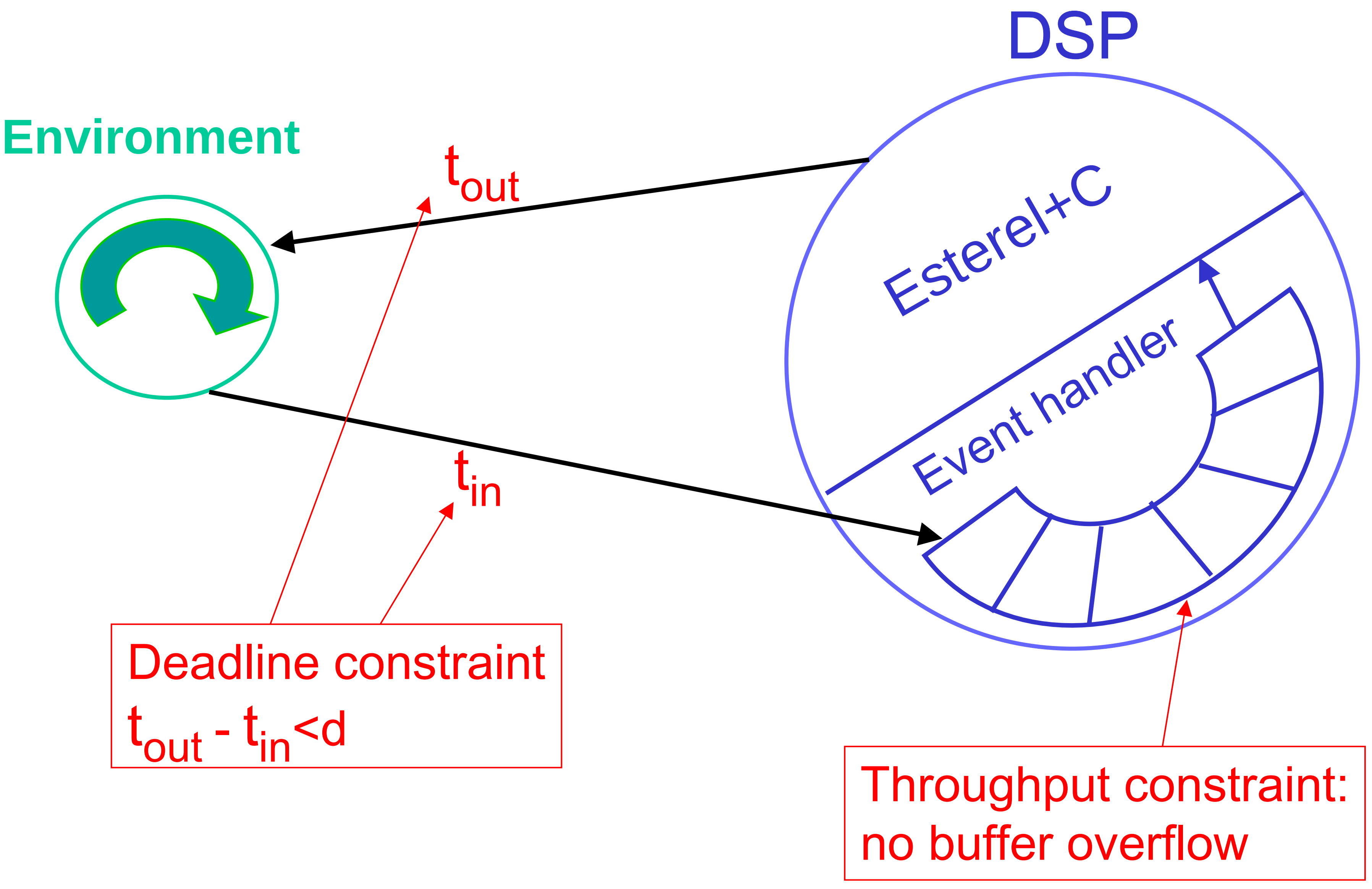
Thesis :

A Timed Model of a RT system can be obtained by “composing” its application SW with timing constraints induced by both its execution and its external environment

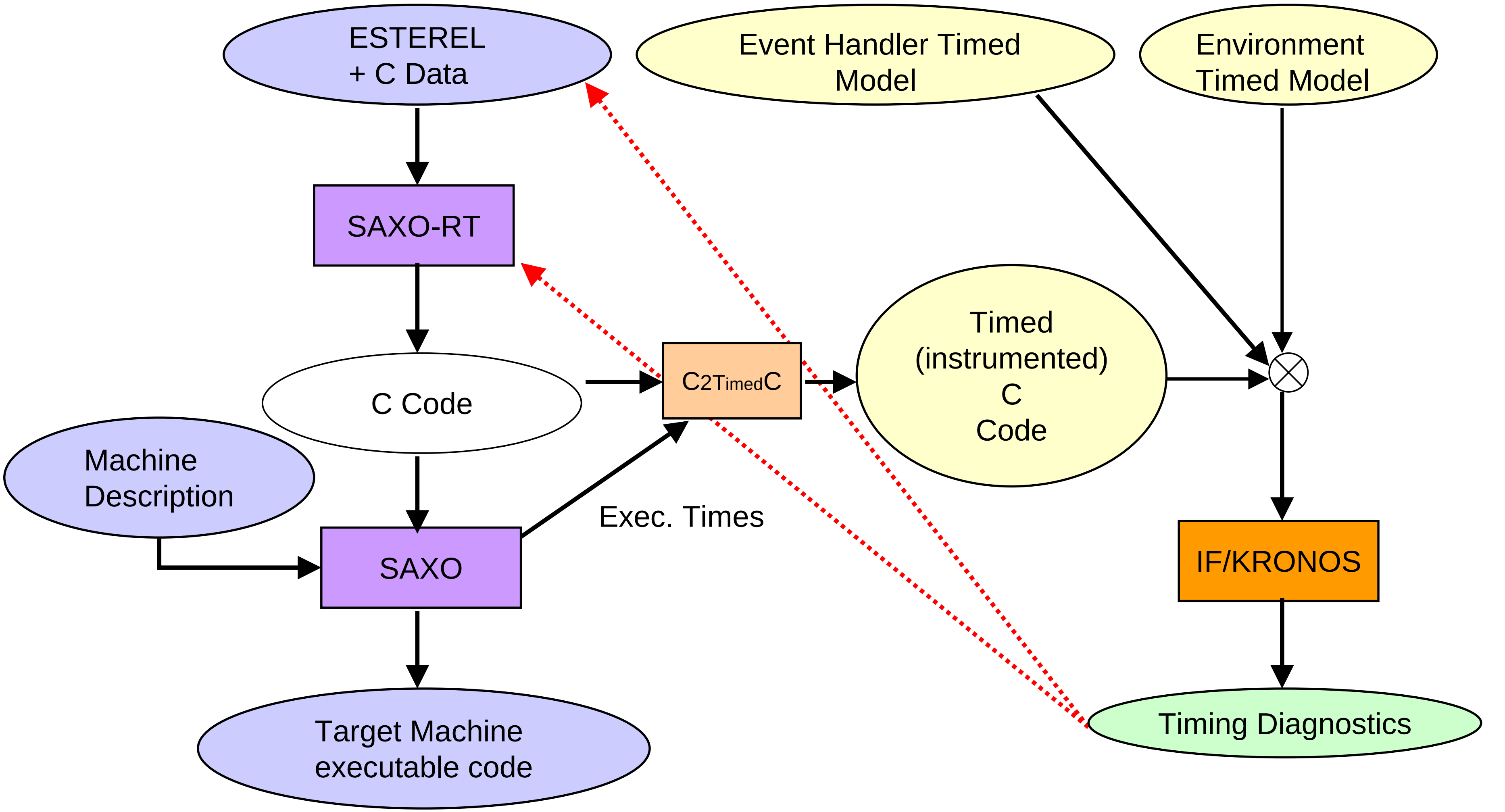
Modeling real-time systems

	Application SW	Timed model
DESCRIPTION	Reactive machine (untimed)	Reactive machine + External Environment + Execution Platform
TIME	Reference to physical (external) time	Quantitative (internal) time Consistency pbs- timelocks
TRIGGERING	Timeouts to control waiting times 	Timing constraints on interactions 
ACTIONS	No assumption about Execution Times Platform-independent	Assumptions about Execution Times Platform-dependent

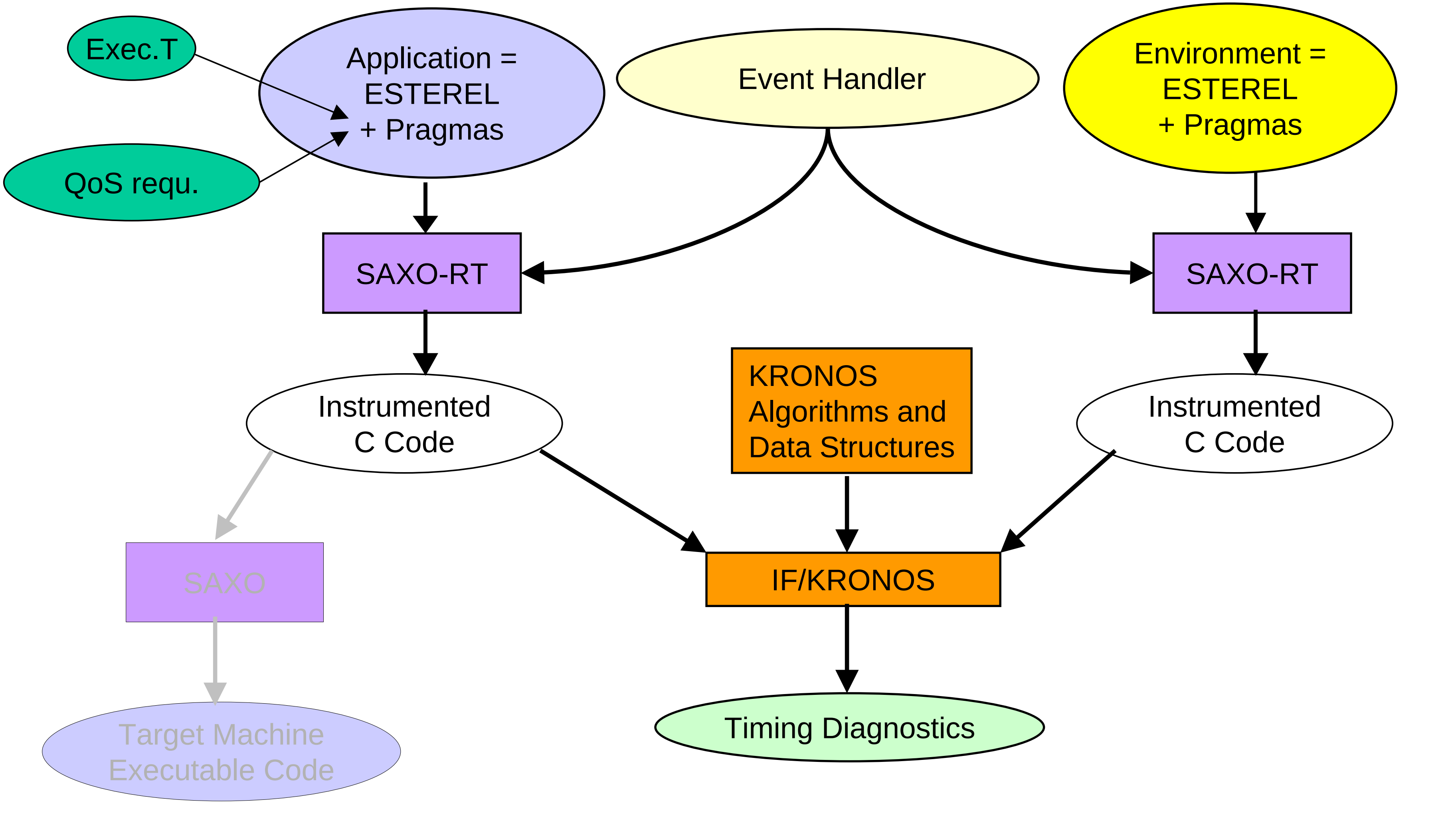
Modeling real-time systems – Taxys (1)



Modeling real-time systems – Taxys (2)



Modeling real-time systems – Taxys(3)



Outline

Key Research issues

- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework

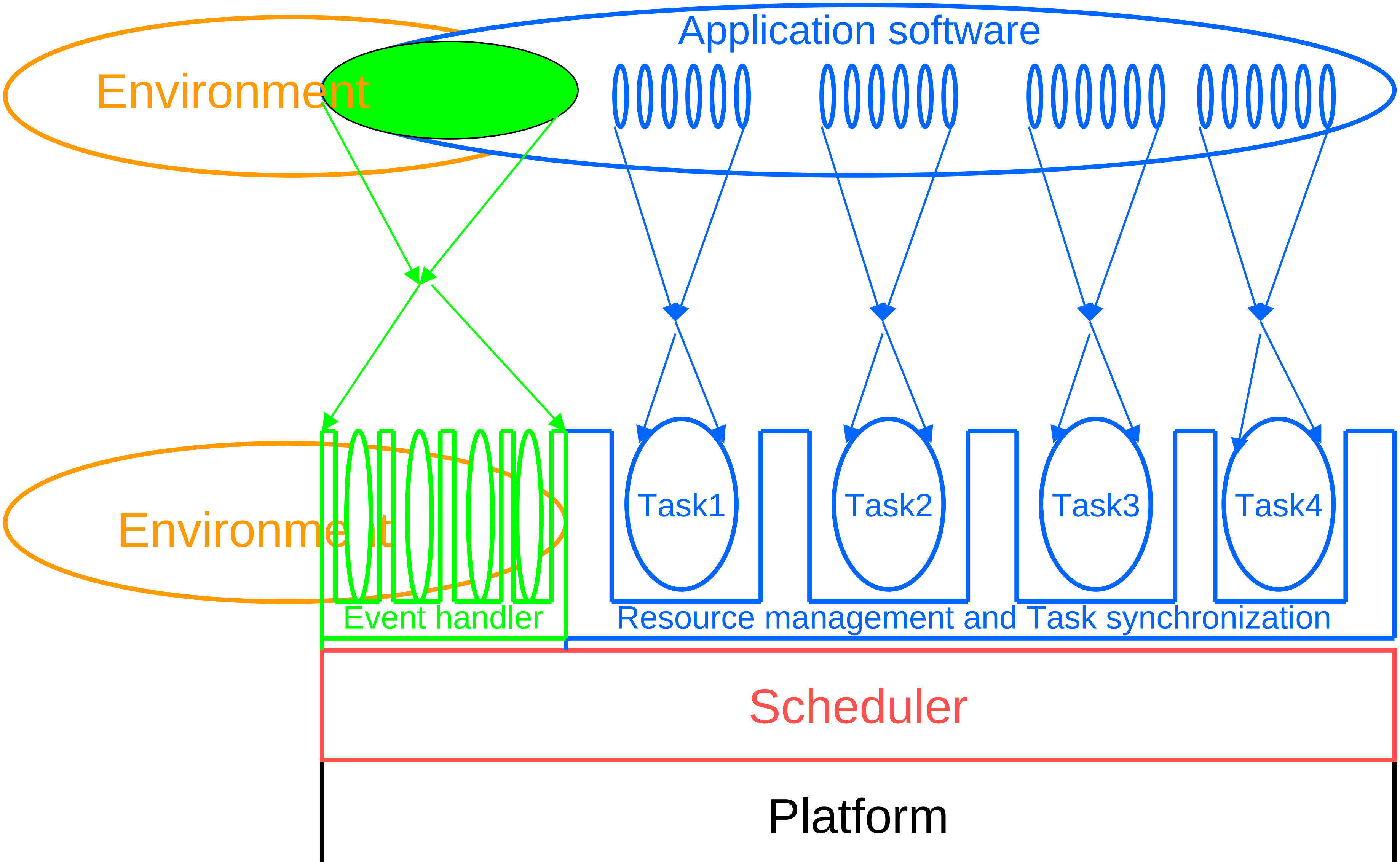
- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

The IF toolset

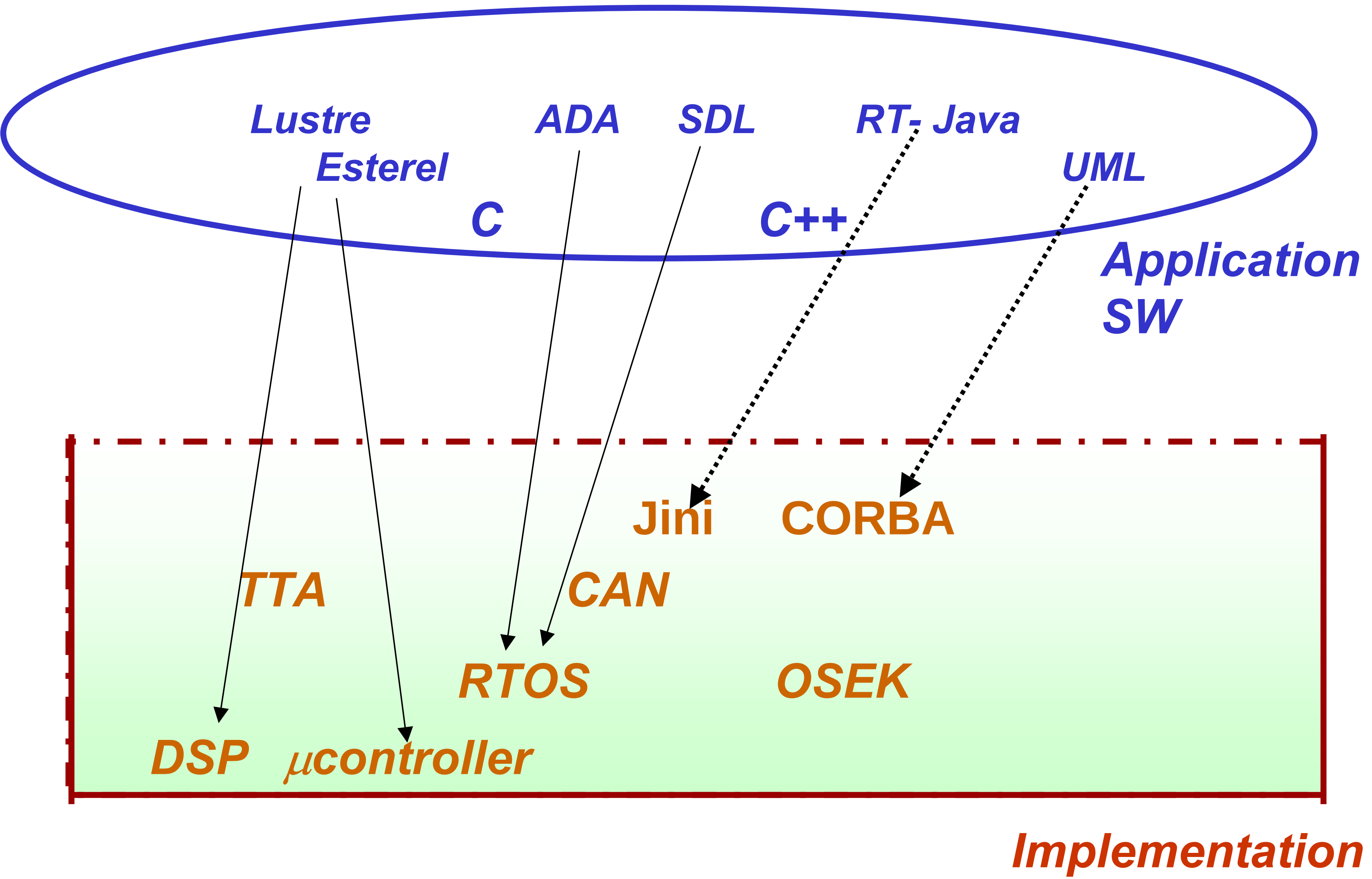
- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion

From application SW to implementations



From application SW to implementations



From application SW to implementations

Functional, Logical, Abstract time,

High level structuring constructs and primitives

Simplifying synchrony assumptions wrt environment

Application SW

abstraction

refinement

Physical, Non functional properties

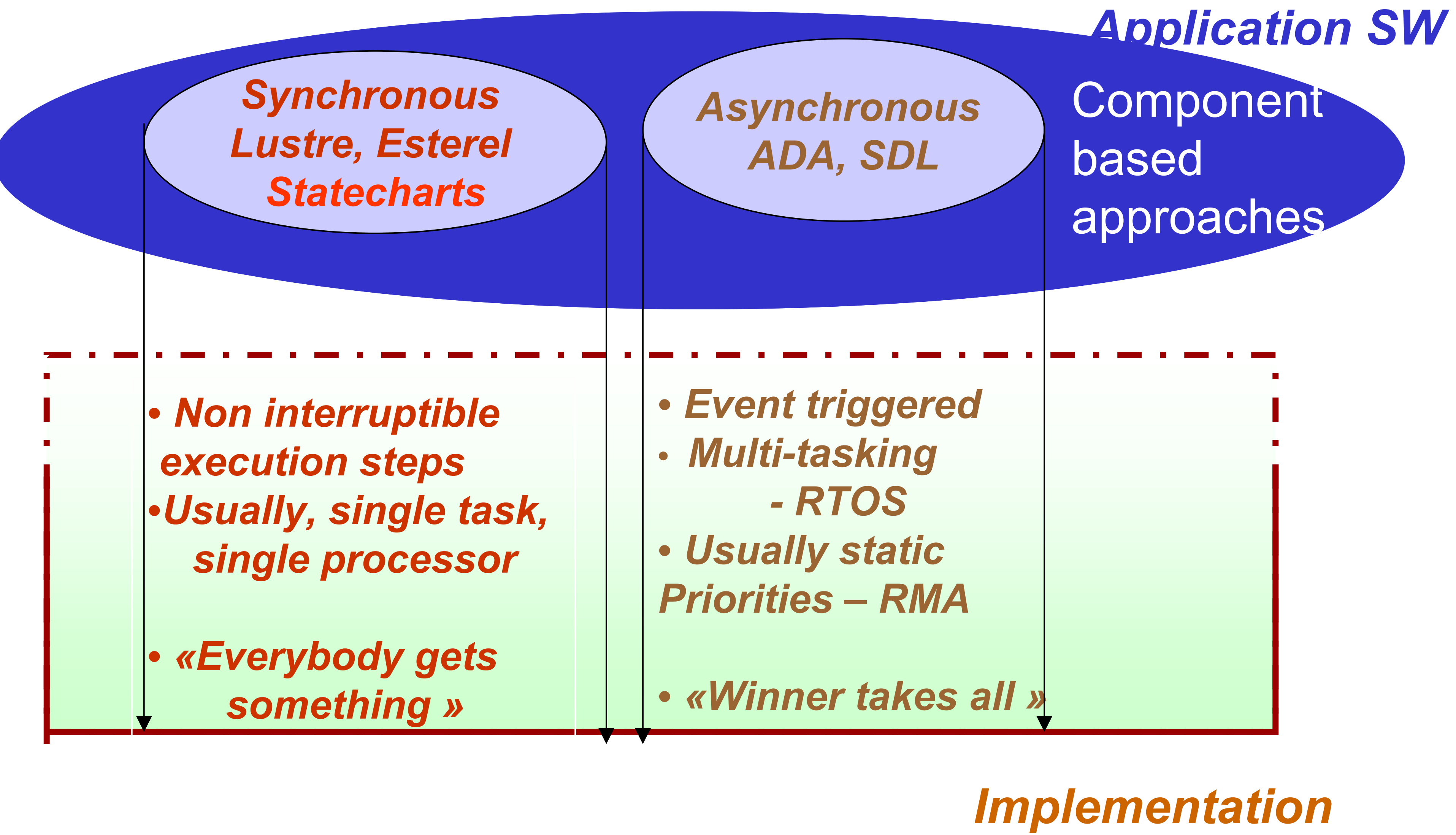
Execution times, interaction delays, latency, QoS

Mapping functional design into tasks, data, resources

Task coordination, resource management, scheduling

Implementation

From application SW to implementations – synchronous vs. asynchronous



Outline

Key Research issues

- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework

- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

The IF toolset

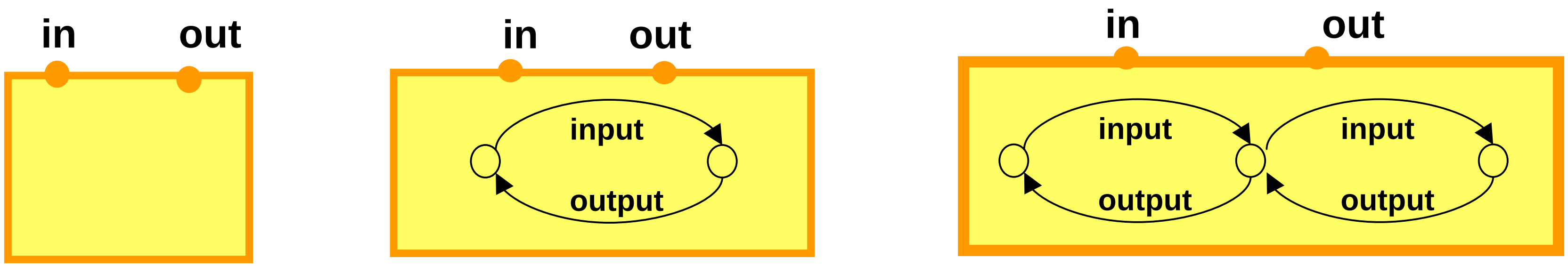
- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion

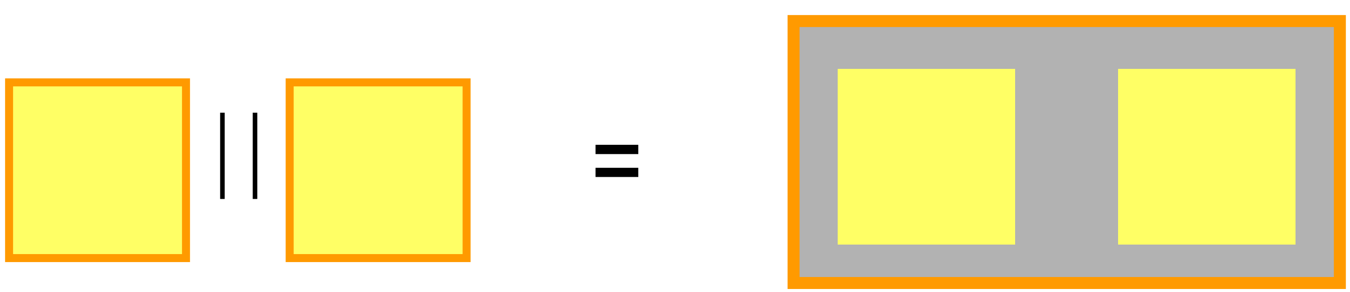
Component-based construction

Build systems by **composition of components**

Component =
Interface (set of interactions) + Behavior (transition system)



Composition operation allows building new components



Component-based construction

Construction problem:
Given a component **C** and a property **P** find **C'** and || such that **C** || **C'** satisfies **P**

C || C' **satisfies P**

Composition:

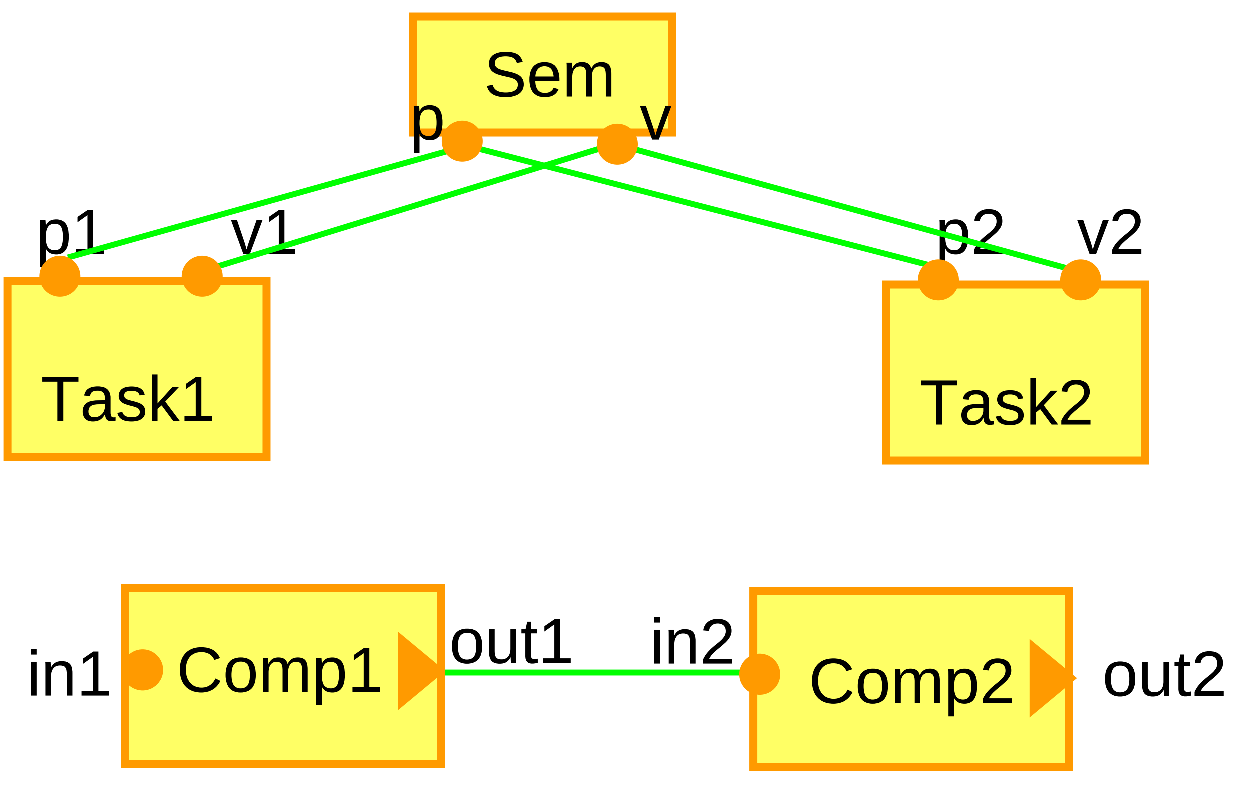
- Creates new interactions
- Restricts the behavior of the components

Key issue: Heterogeneity

Composition - interactions

Interactions are specified by connectors. They can be

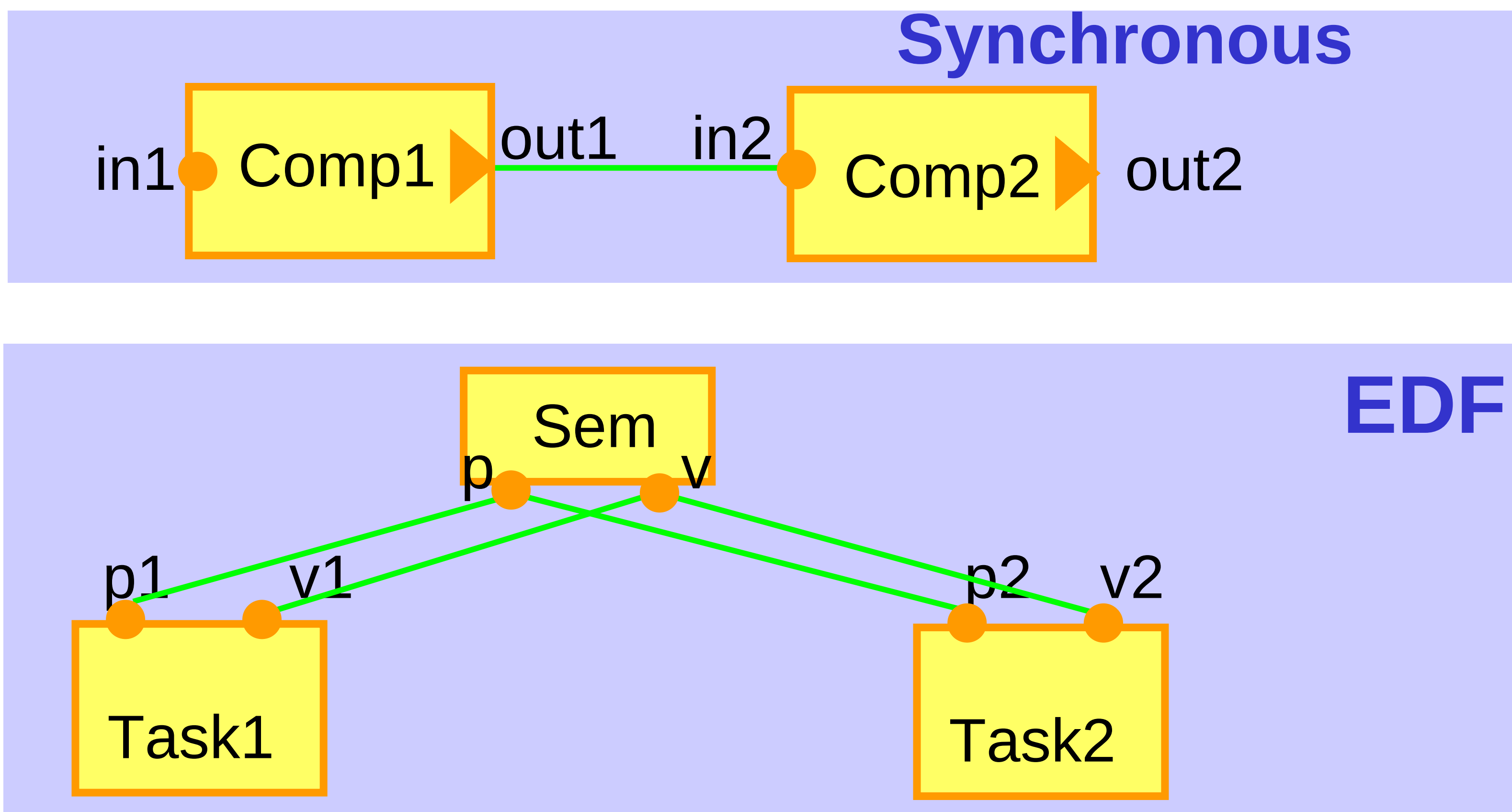
- strict (rendez-vous in CSP) or non strict (msg sending,broadcast)
- atomic (rendez-vous) or non atomic (asynchronous comm.)
- binary (point to point as in CCS, SDL) or n-ary in general



Composition - restriction

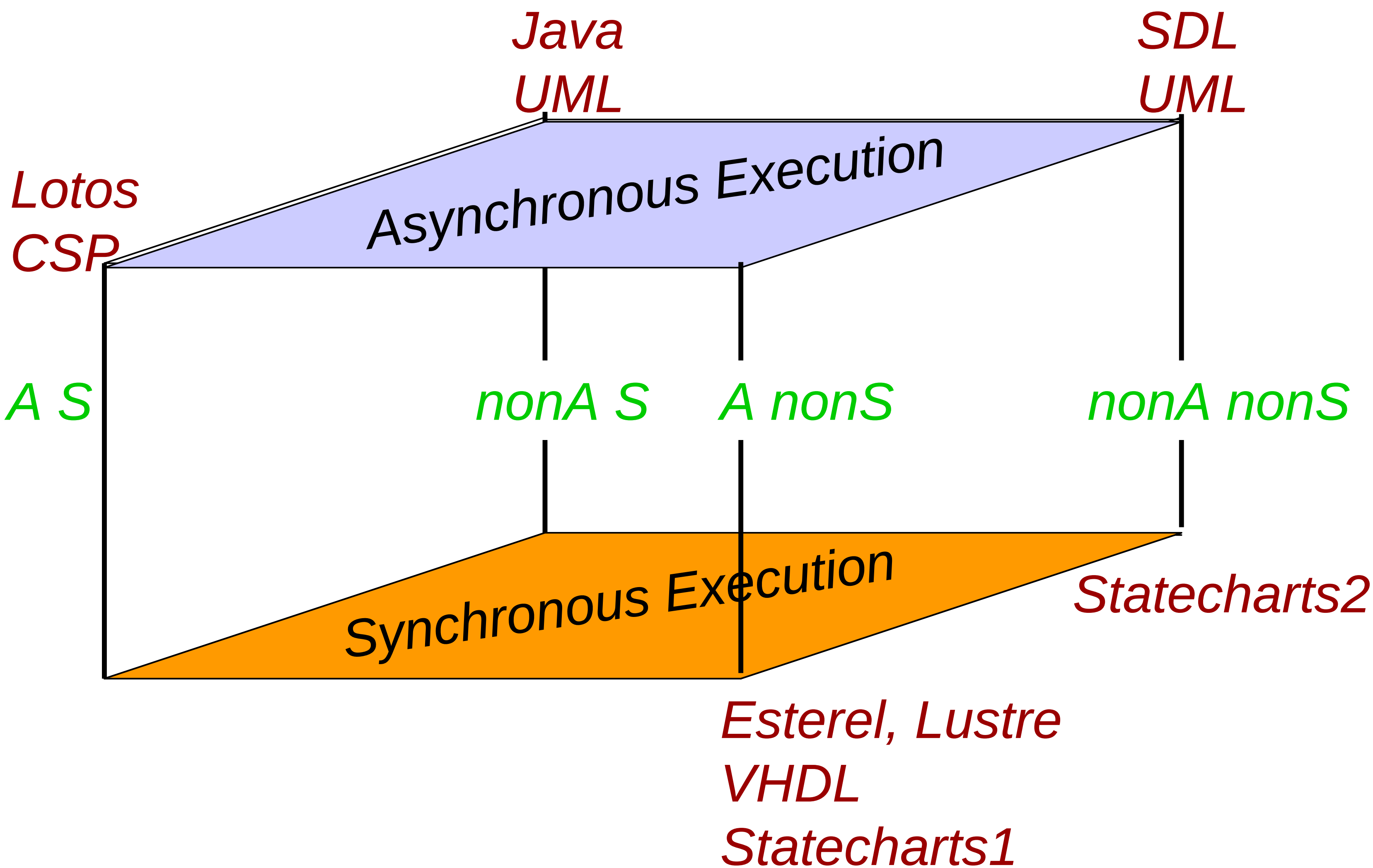
Restrictions enforce properties of execution such as synchrony, scheduling policies, run-to-completion.

Synchronous execution is a restriction of asynchronous execution



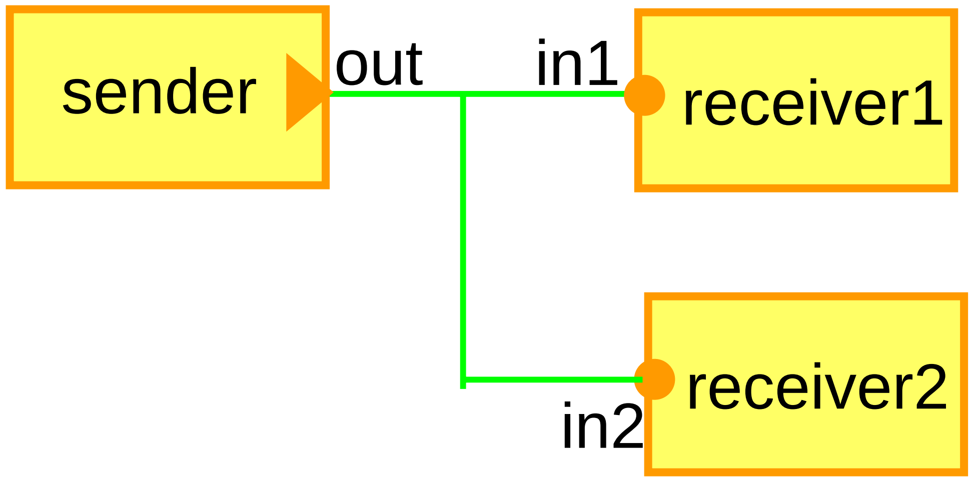
Composition - heterogeneity of interaction and execution

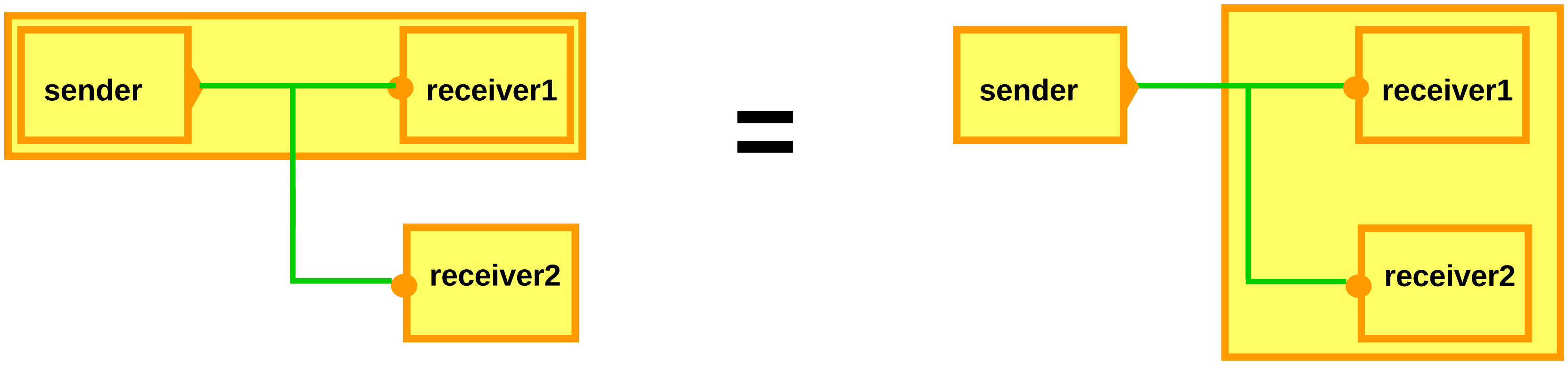
A: Atomic interaction *S: Strict interaction*



Composition: incrementality

Use a unique binary associative compositon operation (express n-ary composition by binary composition)





Outline

Key Research issues

- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework

- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

The IF toolset

- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion

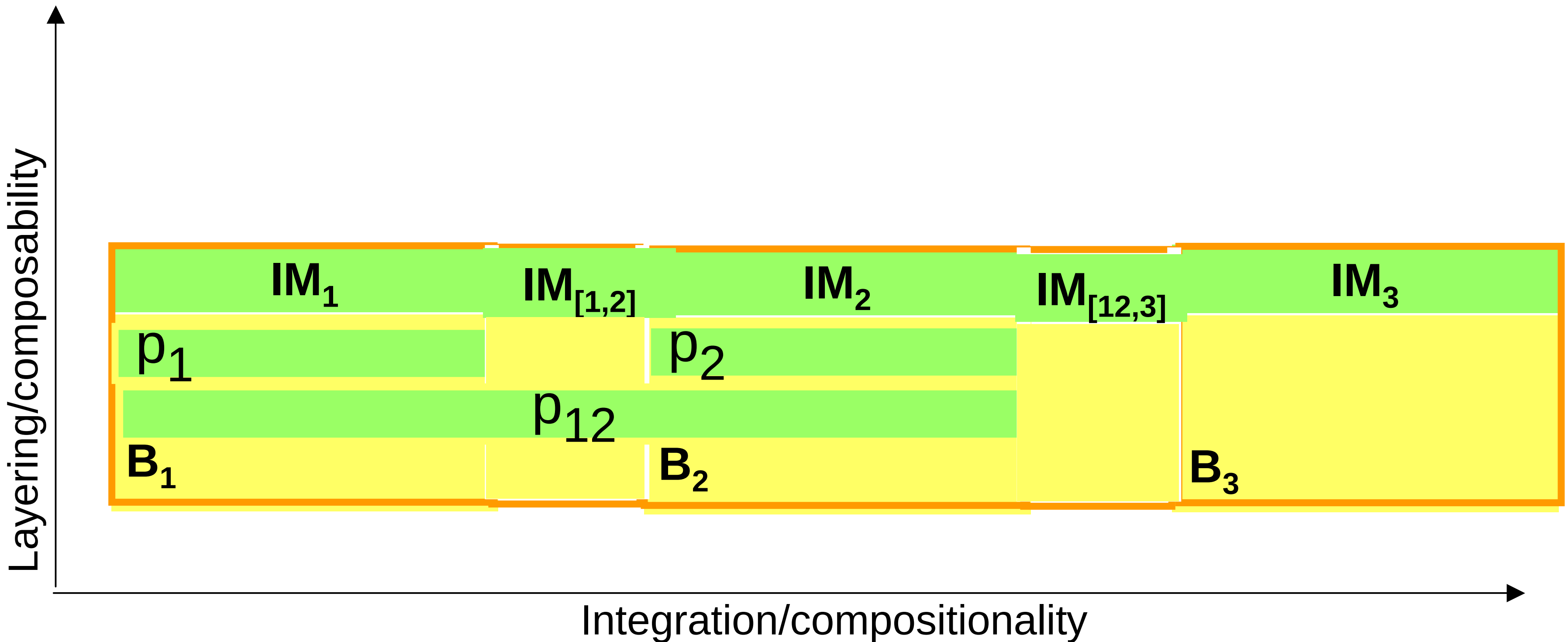
Layered system construction

A **component** is a pair **(B,IM)** where

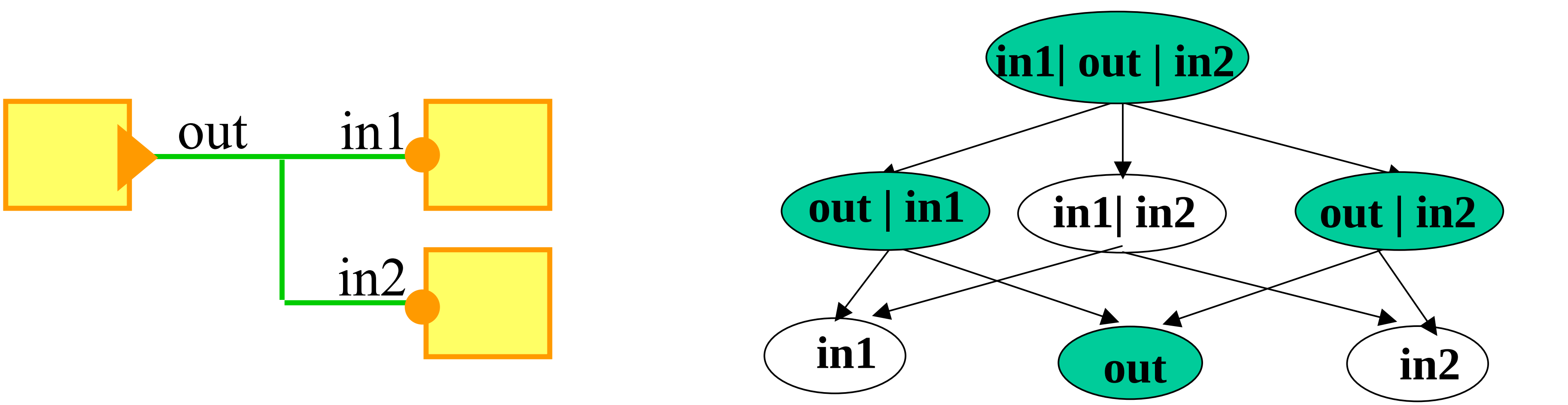
- **B** *is a* ***transition system***
- **IM** *an* ***interaction model***

Composition operators:

- ***Parallel composition*** *:* $(B_1, IM_1) \parallel_{IM[1,2]} (B_2, IM_2) = (B, IM)$
- ***Restriction*** *to enforce a property* ***p*** *:* $(B, IM) \longrightarrow (B/p, IM)$



Parallel composition: Interaction models - examples



NB : Only complete or maximal incomplete interactions are legal!

Parallel composition: Interaction models - definition

Let **K** is a set of component names with disjoint action vocabularies **A**i for i$\in$**K.**

A **connector c** of **K** is a non empty subset of $\cup_{i \in K}$ Ai such that $|c \cap Ai| \leq 1$

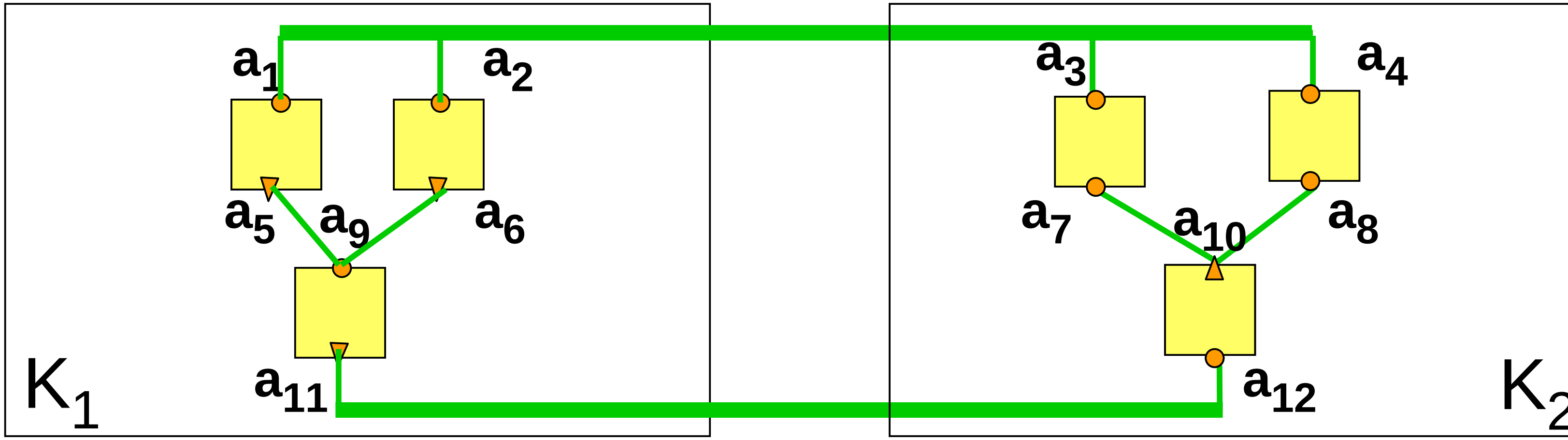
The set of the interactions of a connector c, I(c), is the set of of all the non empty subsets of c.

An interaction model IM is a pair IM=(C, I(C)+)

• A set of **connectors C** or equivalently the set of the interactions of C, $I(C) = \cup_{\mathbf{c} \in \mathbf{C}} I(c)$

• A set of the **complete** interactions I(C)+, I(C)+ $\subseteq$ I(C) such that
$a \in I(C)+ \quad a \subseteq a' \quad$ implies $\quad a' \in I(C)+$

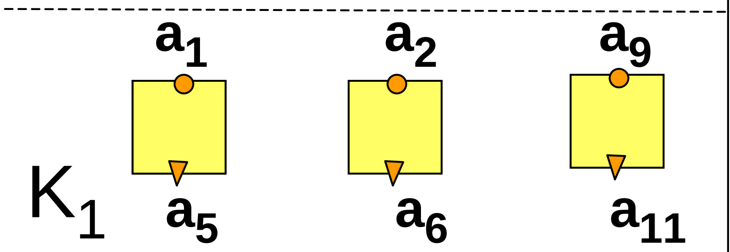
Parallel composition: Interaction models - composition



IM[K_1,K_2]:
C[K_1,K_2] = {{a_1, a_2, a_3, a_4}, {a_{11}, a_{12}}}
IC[K_1,K_2]$^+$ = {$a_1|a_2|a_3|a_4$, a_{11}, $a_{11}|a_{12}$}

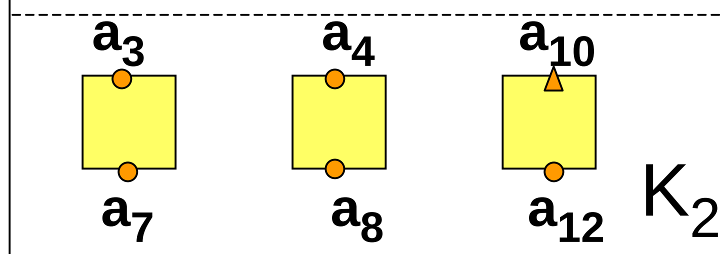
=

IM[K_1]:
C[K_1] = {{a_1, a_2}, {a_5, a_9},{a_6, a_9}}
IC[K_1]$^+$ = {a_5, a_6, a_{11}, $a_5|a_9$, $a_6|a_9$}



||

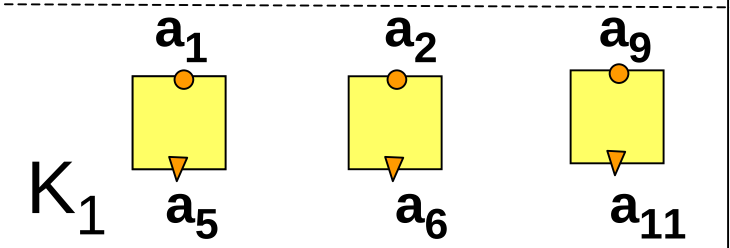
IM[K_2]:
C[K_2] = {{a_3, a_4}, {a_7, a_{10}}, {a_8, a_{10}}}
IC[K_2]$^+$ = {a_{10}, $a_7|a_{10}$, $a_8|a_{10}$}



Parallel composition: Interaction models – composition (2)

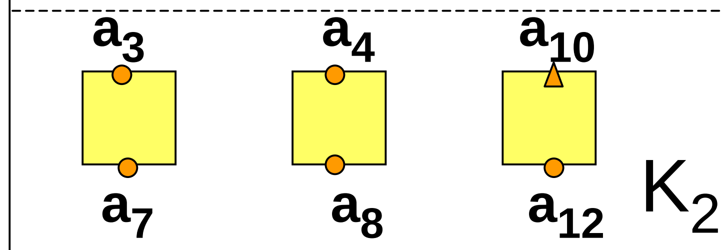
IM[K_1,K_2]:
C[K_1,K_2] = $\{\{a_1, a_2, a_3, a_4\}, \{a_{11}, a_{12}\}\}$
IC[K_1,K_2]$^+$ = $\{a_1|a_2|a_3|a_4, a_{11}, a_{11}|a_{12}\}$

IM[K_1]:
C[K_1] = $\{\{a_1, a_2\}, \{a_5, a_9\},\{a_6, a_9\}\}$
IC[K_1]$^+$ = $\{a_5, a_6, a_{11}, a_5|a_9, a_6|a_9\}$



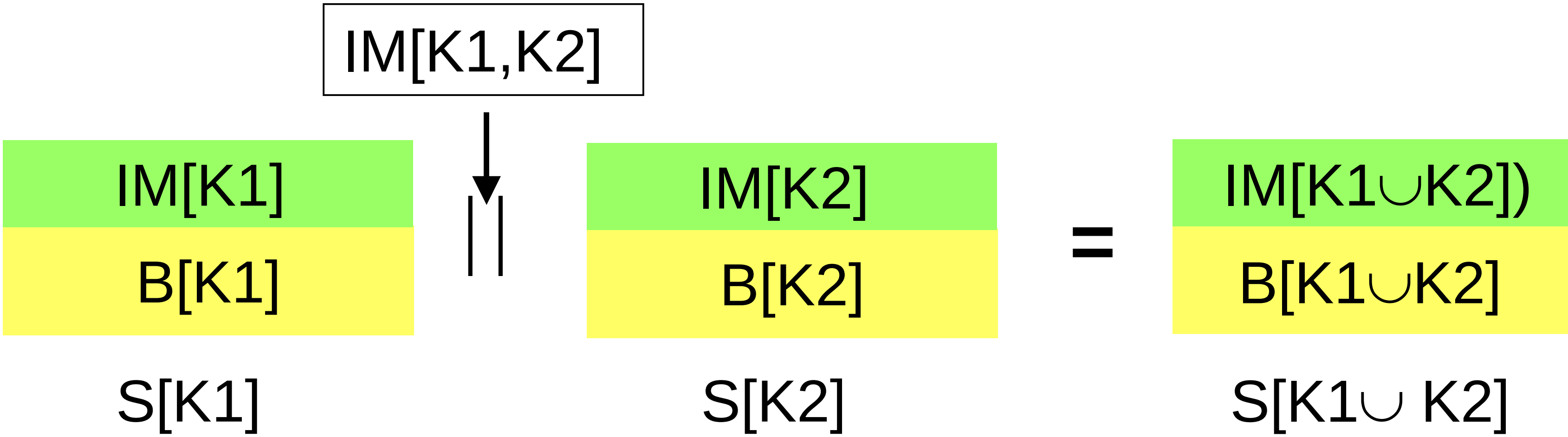
||

IM[K_2]:
C[K_2] = $\{\{a_3, a_4\}, \{a_7, a_{10}\}, \{a_8, a_{10}\}\}$
IC[K_2]$^+$ = $\{a_{10}, a_7|a_{10}, a_8|a_{10}\}$



IM[$K_1 \cup K_2$]:
C[$K_1 \cup K_2$] = C[K_1] $\cup$ C[K_2] $\cup$ C[K_1, K_2]
IC[$K_1 \cup K_2$]$^+$ = IC[K_1]$^+$ $\cup$ IC[K_2]$^+$ $\cup$ IC[K_1, K_2]$^+$

Parallel composition: General definition



$$S[K1] \;||\; SK[2] = (B[K1], IM[K1]) \;||\; (B[K2], IM[K2])$$

$$= (B[K1] \times B[K2], IM[K1] \cup IM[K2] \cup IM[K1,K2])$$

$$= S[K1 \cup K2]$$

where $\times$ is an associative and commutative operation such that

$$B[K1] \times B[K2] = B[K1 \cup K2]$$

Composition is associative and commutative

Flexible parallel composition : transition systems with priorities

Behavior : transition systems
Interaction model : priority relation on interactions

A **transition system with priorities** is a pair **(B,** $\langle$ **)** where,

- **B** is a labeled transition system with labels from a set of interactions **A**
- $\langle$ is a strict partial order on **A** that restricts **B** :

Semantics of (B, $\langle$ **)** :

q–a1→ q'∈ **(B,** $\langle$ **)** if q–a1→ q'∈ **B**

and there is no q–a2→ q''∈ **B,** a1 $\langle$ a2

The **sum** $\langle^{1} \oplus \langle^{2}$ of two priority orders $\langle^{1}, \langle^{2}$ is the least priority order (if it exists) such that $\langle^{1} \cup \langle^{2} \subseteq \langle^{1} \oplus \langle^{2}$

Remark : $\oplus$ is a (partial) associative and commutative operation

Flexible parallel composition - definition

$\langle^{12}$

$\langle^{1}$ B[K1] $\|$ $\langle^{2}$ B[K2] $=$ $\langle^{1} \oplus \langle^{2} \oplus \langle^{12}$ B[K1$\cup$K2]

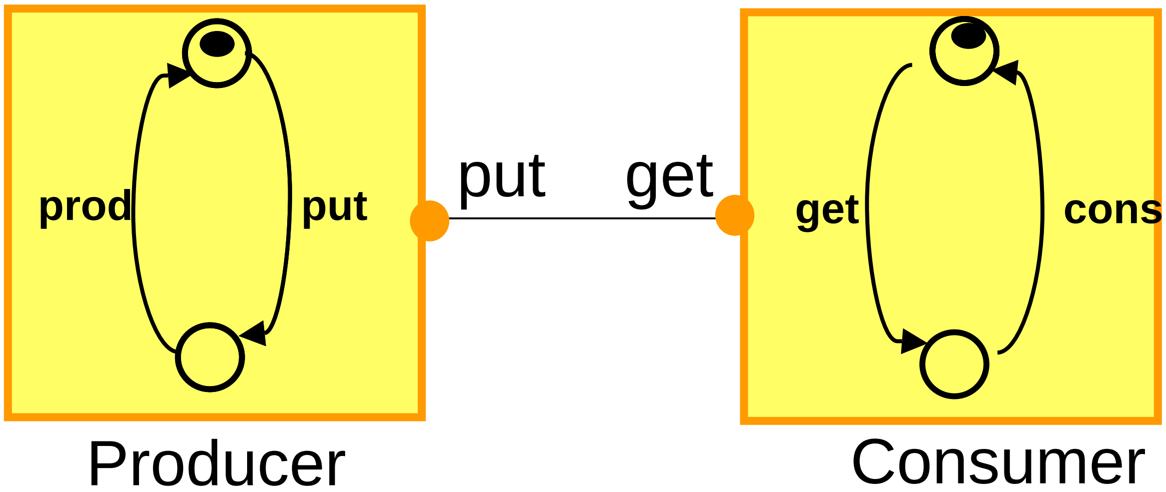
Composition of behaviors:

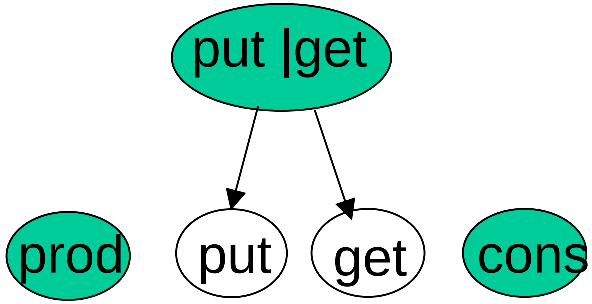
$$\left.\begin{array}{l} q1 - a1 \rightarrow q1' \\ q2 - a2 \rightarrow q2' \end{array}\right\} \text{ implies } \left\{\begin{array}{l} (q1,q2) - a1 \rightarrow (q1',q2) \\ (q1,q2) - a2 \rightarrow (q1,q2') \\ (q1,q2) - a1 \mid a2 \rightarrow (q1',q2') \quad \text{if } a1 \mid a2 \in IC[K1 \cup K2] \end{array}\right.$$

$\langle^{12}$ is defined by the rules :

- Maximal progress : $a1 \langle^{12} a1 \mid a2$, if $a1 \mid a2 \in IC[K1 \cup K2]$
- Completeness : $a1 \langle^{12} a2$, if a1 is incomplete and non maximal
  a2 is complete in $IC[K1 \cup K2]$

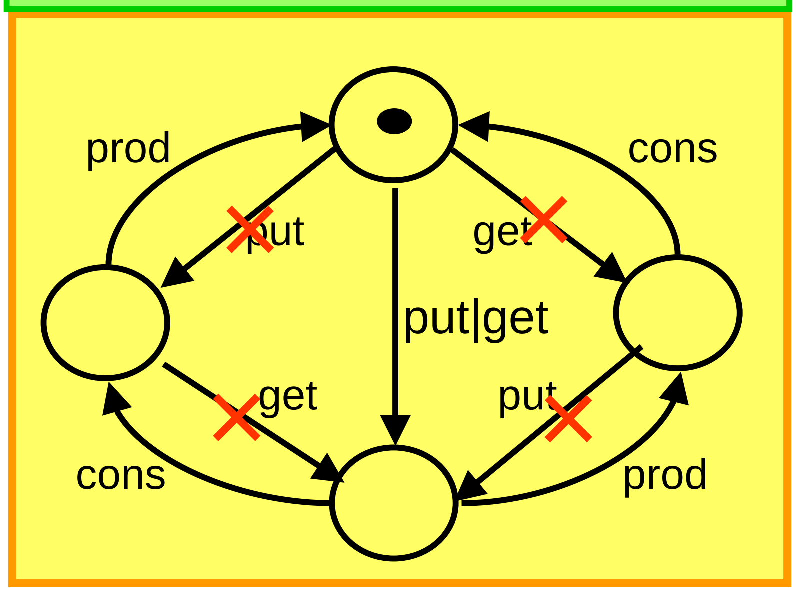
Flexible parallel composition : producer-consumer





put 〈 put | get, get 〈 put | get

put 〈 cons, get 〈 prod



Producer || Consumer

Flexible parallel composition : deadlock-fredom by construction

$(B^1, \langle^{1}) \,||\, (B^2, \langle^{2}) = (B^1 \times B^2, \langle^{1} \oplus \langle^{2} \oplus \langle^{12})$

is an associative total operation on components if no incomplete interaction dominates a complete interaction in the components

$(B, \langle)$ is deadlock-free if B is deadlock-free

$(B^1, \langle^{1}) \,||\, (B^2, \langle^{2})$ is deadlock-free if B^1, B^2 are deadlock-free

*! Check that after composition the resulting component can**not** execute incomplete interactions which are not maximal*

Outline

Key Research issues

- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework

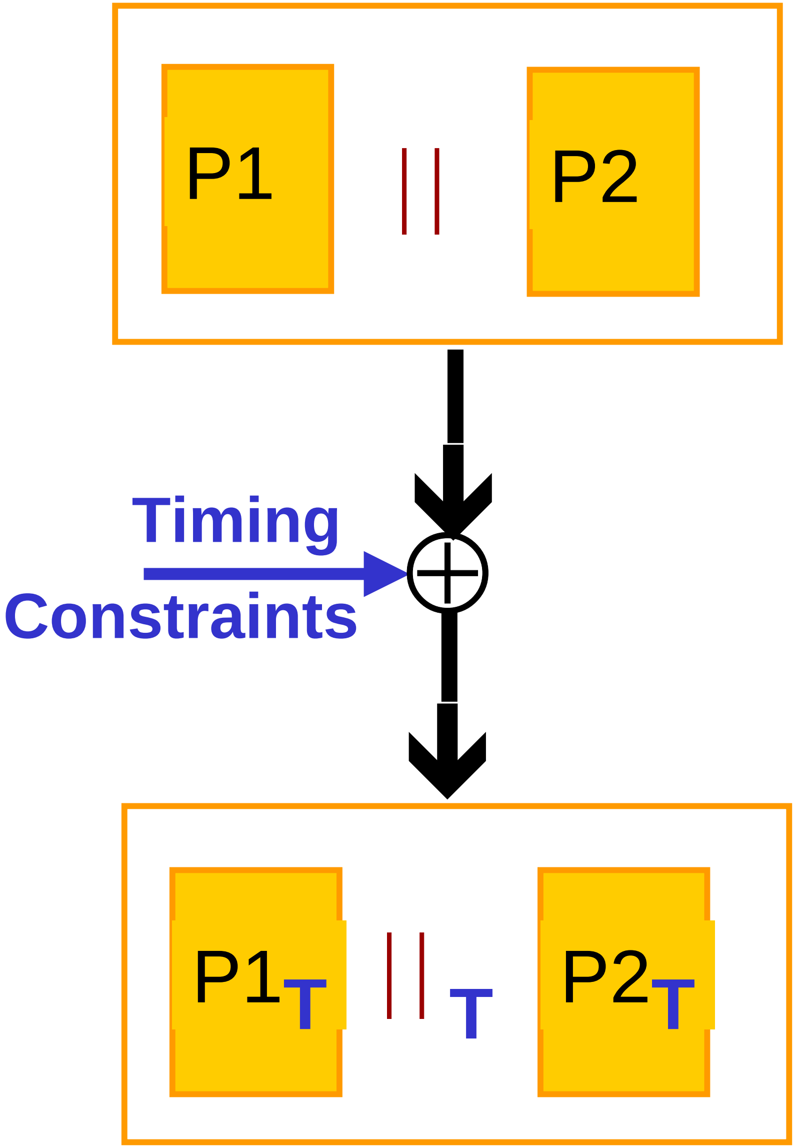
- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

The IF toolset

- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion

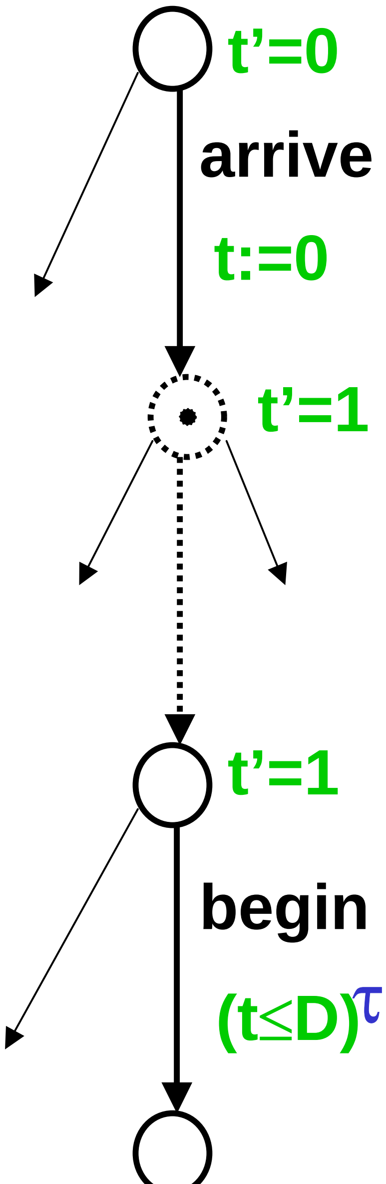
Adding timing constraints



- there exist different timed extensions for $||_T$ corresponding to different assumptions about idling before interaction

- compositionality: define $||_T$ so as to preserve properties such as well-timedness, deadlock-freedom, liveness.

Adding timing constraints: Timed systems



Automata: labeled transition relations on a set of actions

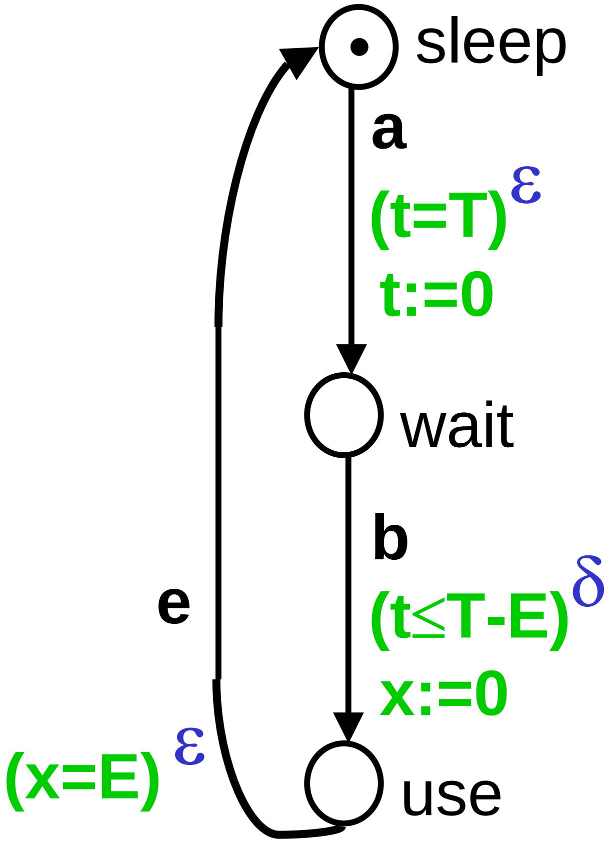
Timers: real-valued variables that can
- be reset and tested at transitions
- increase (derivative =1) or remain unchanged at states (derivative =0)

Types of urgency τ associated with guards express priority over time progress at states

ε	(eager) :	if enabled then must fire asap
λ	(lazy) :	if enabled then may fire
δ	(delayable) :	if enabled must fire before it becomes disabled

Adding timing constraints : example

A periodic process of period **T** and execution time **E**



Actions

a: arrive

b: begin

e: end

$t'=x'=1$ at all states

Adding timing constraints

Three different kinds of timing constraints:

- *from the execution platform e.g. execution times, latency times*

- *from the external environment about arrival times of triggering events e.g. periodic tasks*

- *user requirements e.g. QoS, which are timing constraints relating events of the real-time system and events of its environment e.g. deadlines, jitter*

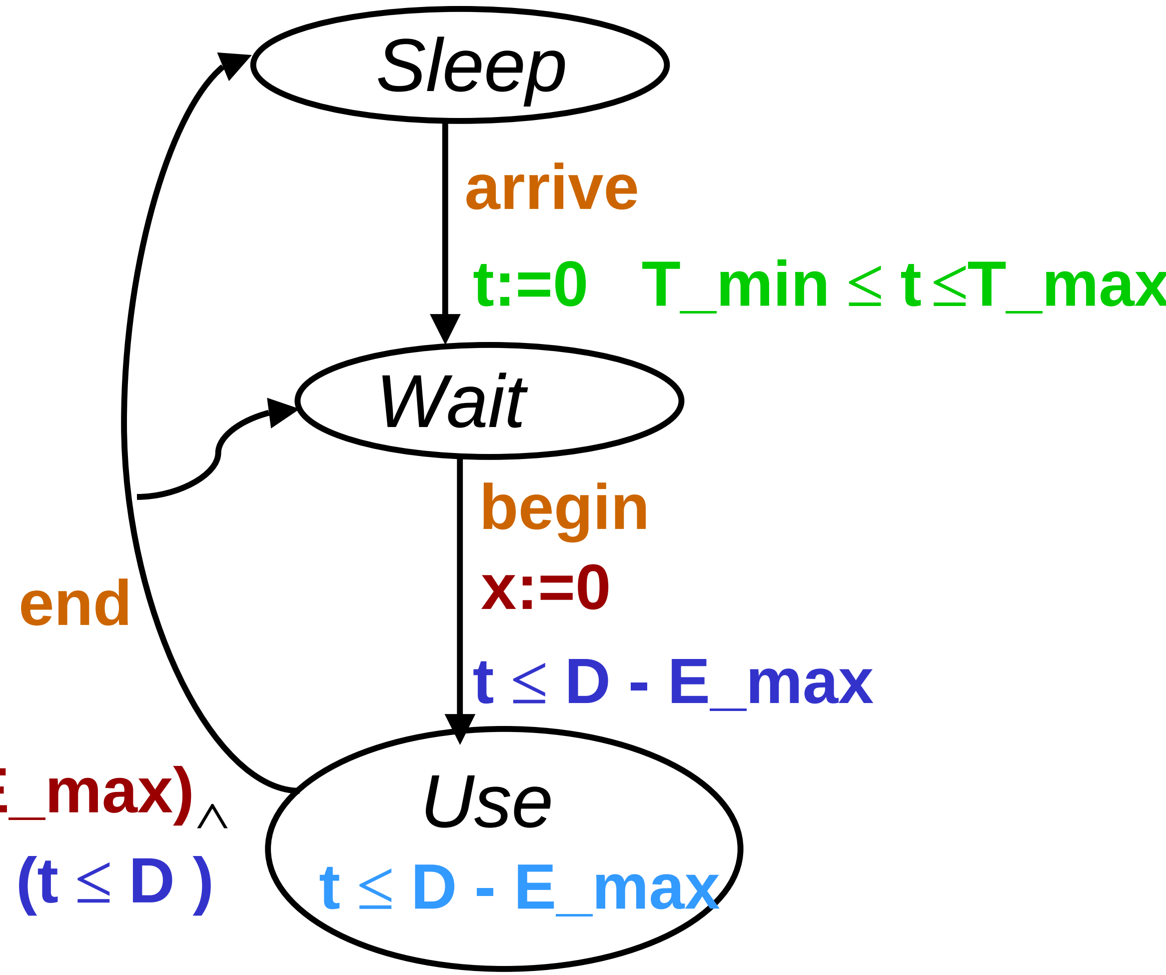
Adding timing constraints

Each shared resource induces a partition on the control states of a process { Sleep, Wait, Use}.

Arrival times (t)

Execution times (x)

Deadline D



Outline

Key Research issues

- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework

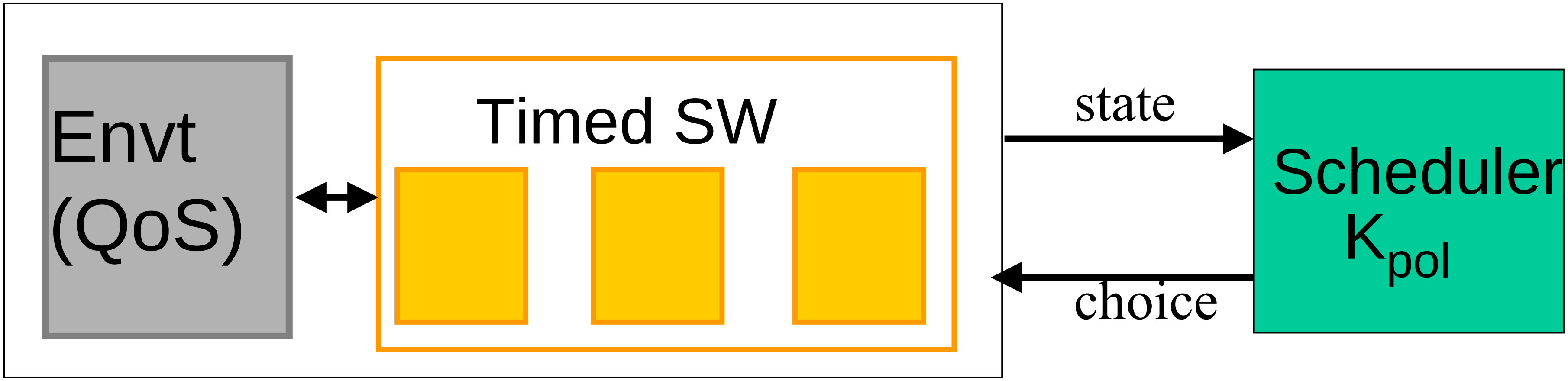
- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

The IF toolset

- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion

Scheduler modeling



A scheduler is a controller which restricts *access to resources* so as to meet the timing constraints (deadlock-free behavior) by applying a scheduling policy K_{pol} :

$$K_{pol} = \wedge_{r \in R} \quad K_{r\_pol}$$

$$K_{r\_pol} = K_{r\_res} \wedge K_{r\_adm}$$

K_{r_res} says how conflicts for the acquisition of resource r are resolved e.g. EDF, RMS, LLF

K_{r_adm} says which requests for r are considered by the scheduler at a state e.g. masking

Scheduler modeling

Example : K_{pol} for the Priority Ceiling Protocol

Admission control: *"Process P is eligible for resource r if the current priority of P is higher than the ceiling priority of any resource allocated to a process other than P"*

Conflict resolution: " *The CPU is allocated to the process with the highest current priority"*

Result : *Any feasible scheduling policy K_{pol} induces a restriction that can be described by dynamic priorities*

Outline

Key Research issues

- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework

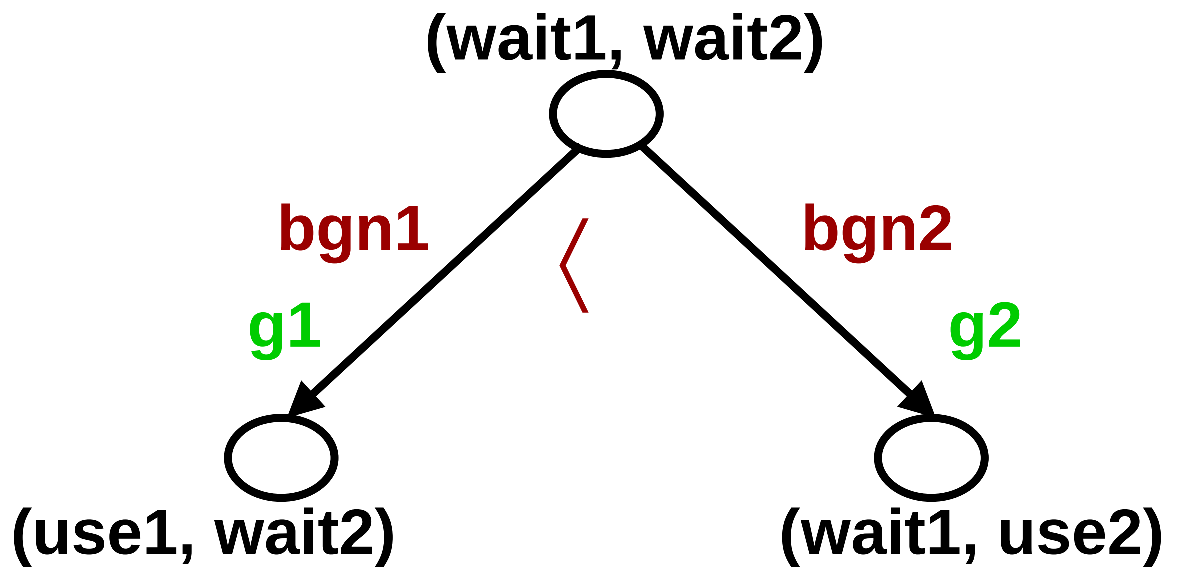
- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

The IF toolset

- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion

Timed Systems with priorities

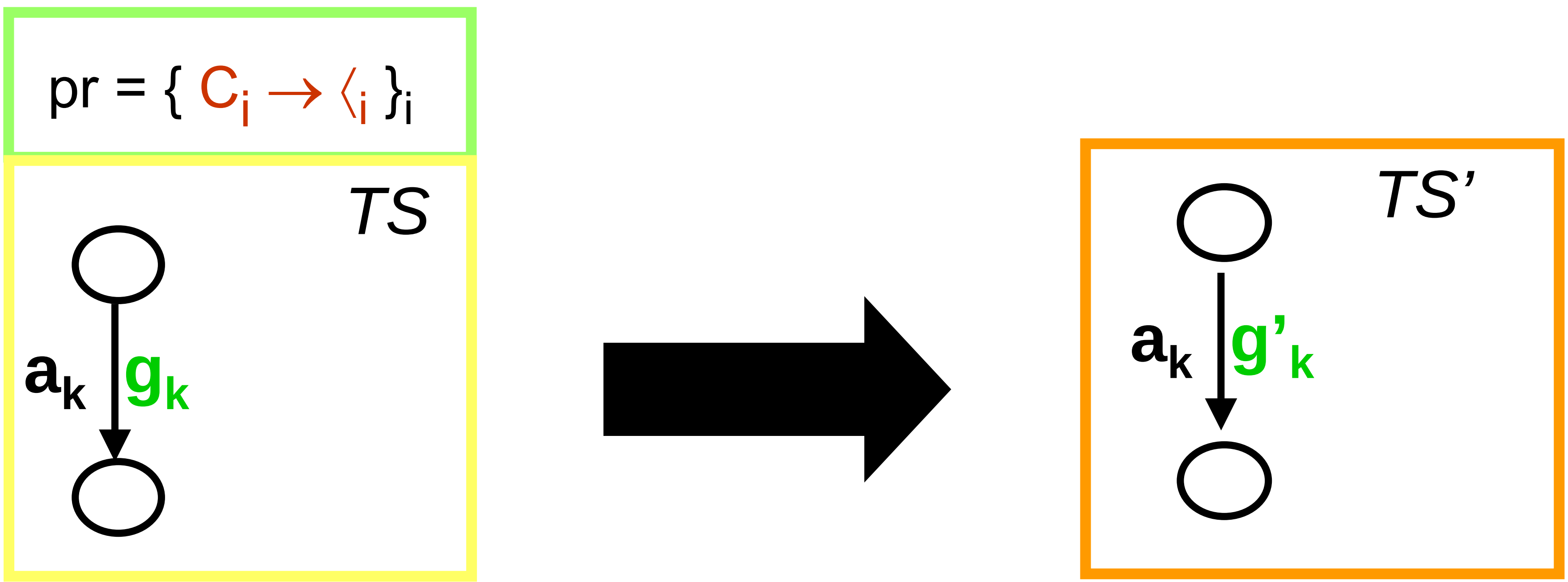


Priority rule	Strengthened guard of bgn1
true $\rightarrow$ bgn1 $\langle$ bgn2	$g1' = g1 \wedge \neg g2$
C $\rightarrow$ bgn1 $\langle$ bgn2	$g1' = g1 \wedge \neg(C \wedge g2)$

Timed Systems with priorities

A priority order is a strict partial order, $\langle \subseteq A \times A$

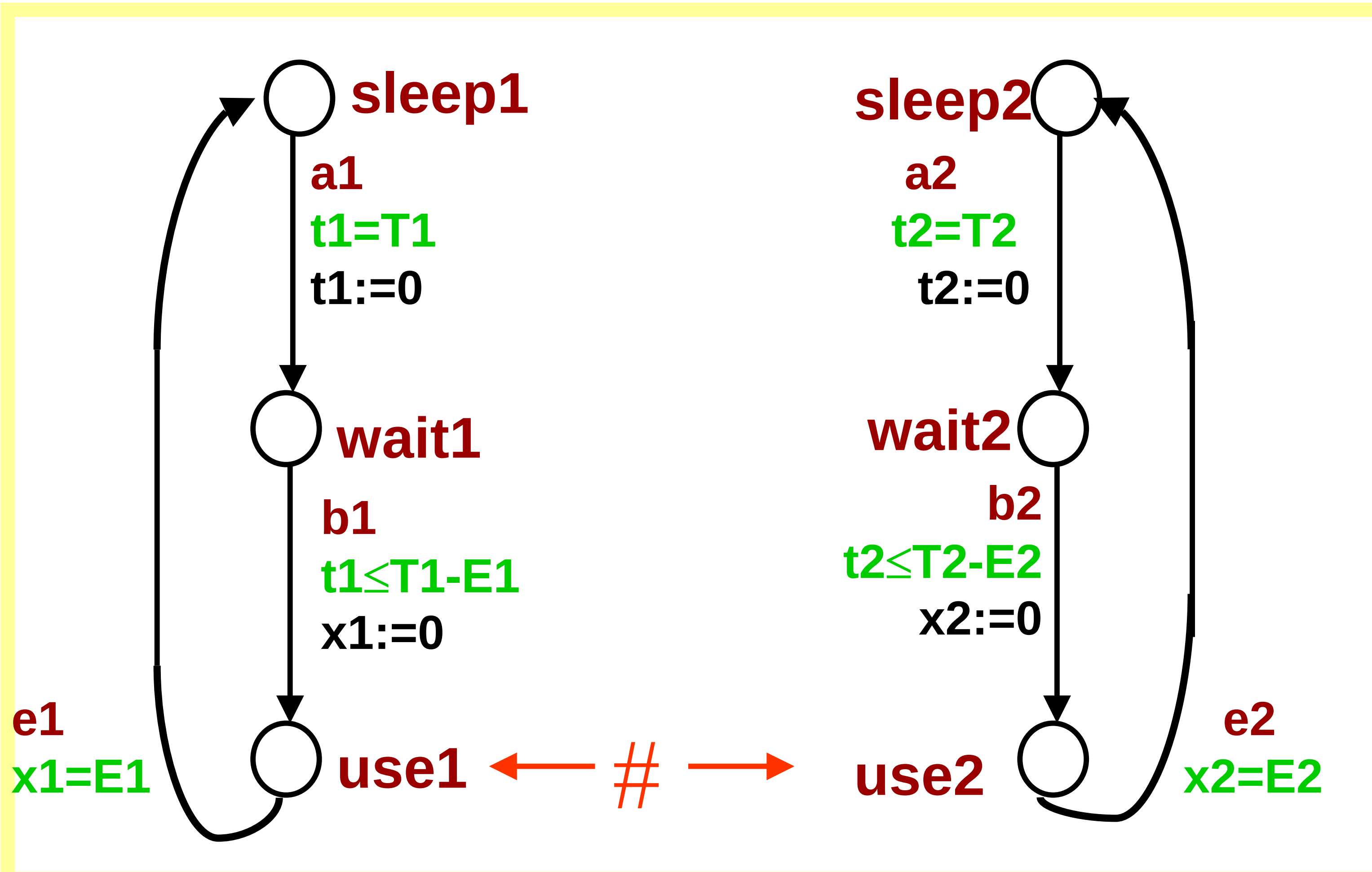
A set of priority rules, $pr = \{ C_i \rightarrow \langle_i \}_i$ where $\{C_i\}_i$ is a set of disjoint state predicates



$$g'_k = g_k \wedge \bigwedge_{C \rightarrow \langle \, \in pr} \left(C \Rightarrow \bigwedge_{ak \,\langle\, ai} \neg \, g_i \right)$$

Timed Systems with priorities: FIFO policy

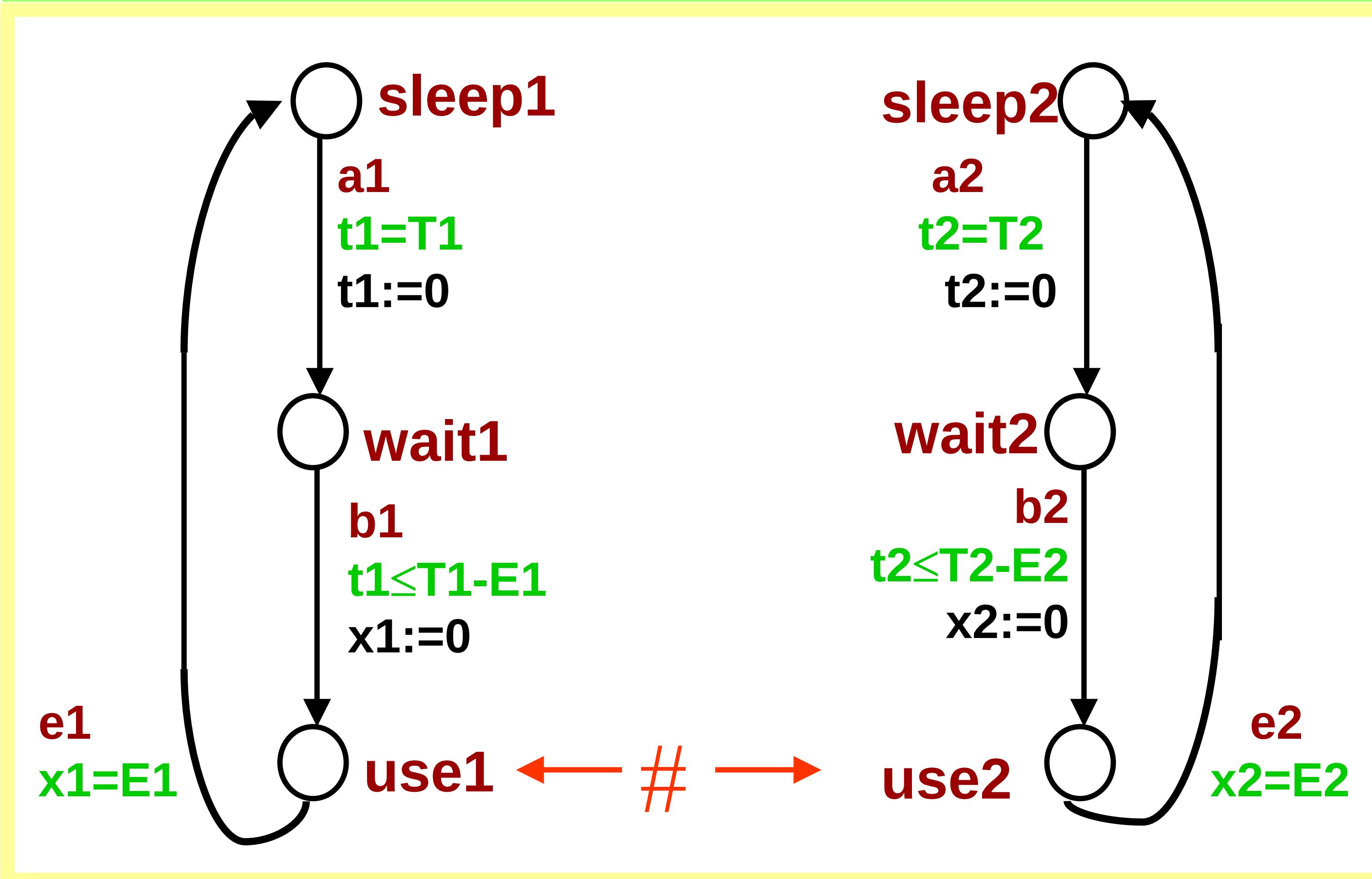
$t1 \leq t2 \rightarrow b1 \langle b2$ $t2 \leq t1 \rightarrow b2 \langle b1$



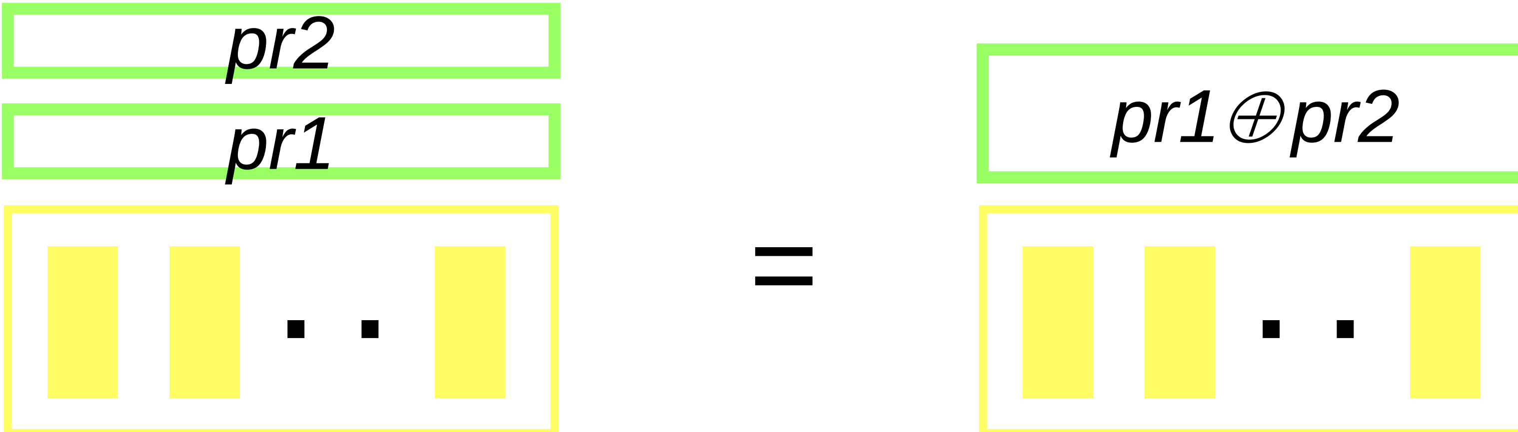
Timed Systems with priorities : Least Laxity First policy

$$L1 \leq L2 \rightarrow b2 \langle b1 \qquad L2 \leq L1 \rightarrow b1 \langle b2$$

where $L_i = T_i - E_i - t_i$ is the laxity of process i



Timed Systems with priorities: composition of priorities



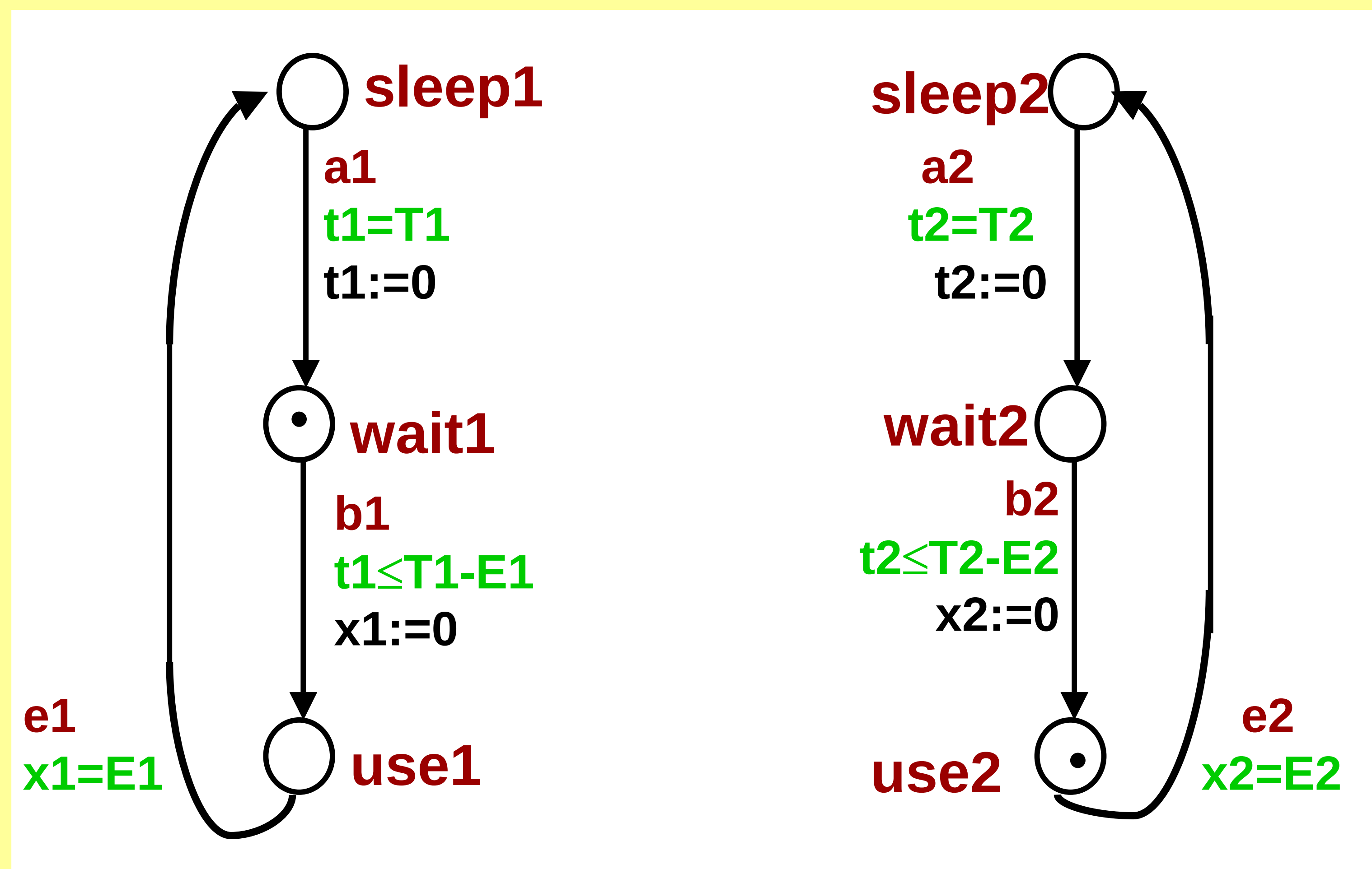
(pr1⊕ pr2)(q) is the least priority order containing pr1(q) ∪pr2(q)

Results :

- The operation ⊕ is partial, associative and commutative
- Sufficient conditions for deadlock-freedom and liveness

Timed Systems with priorities: mutual exclusion + FIFO

t1≤ t2 → b1〈 b2	t2≤ t1 → b2〈 b1
true → b1〈 e2	true → b2〈 e1



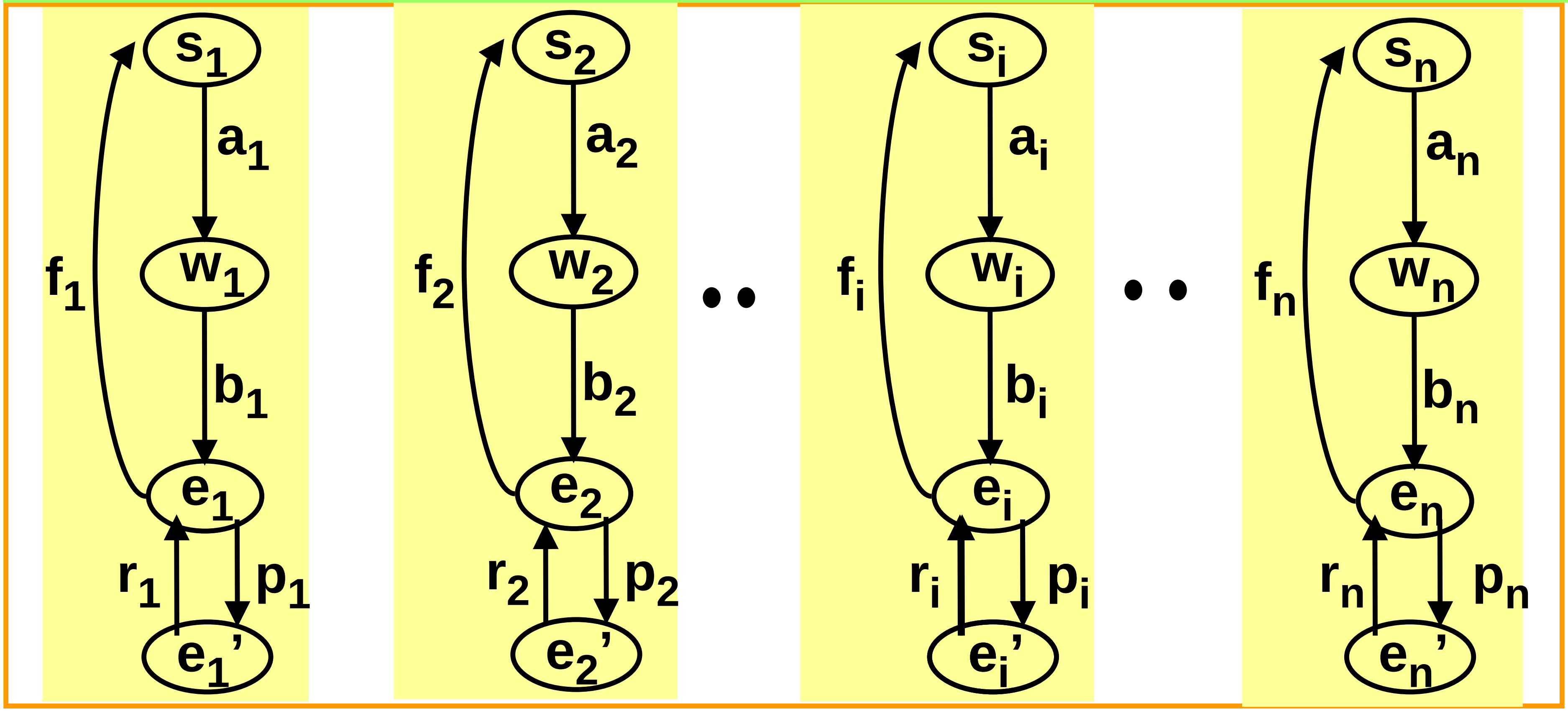
Systems with priorities : Fixed priority preemptive scheduling

Scheduling policy

$\mathbf{b_i} \langle\ \mathbf{b_j}$, $\mathbf{b_i\,|p_k} \langle\ \mathbf{b_j\,|p_k}$, $\mathbf{f_i\,|r_k} \langle\ \mathbf{f_j\,|r_k}$ for $n \geq l > j \geq 1$

Interaction model

$\mathbf{b_j|p_i, f_j|r_i} \in \mathbf{IC}$, for $n \geq i,j \geq 1$ $\mathbf{a_i, f_i, b_i} \in \mathbf{IC^+}$, for $n \geq i \geq 1$



Outline

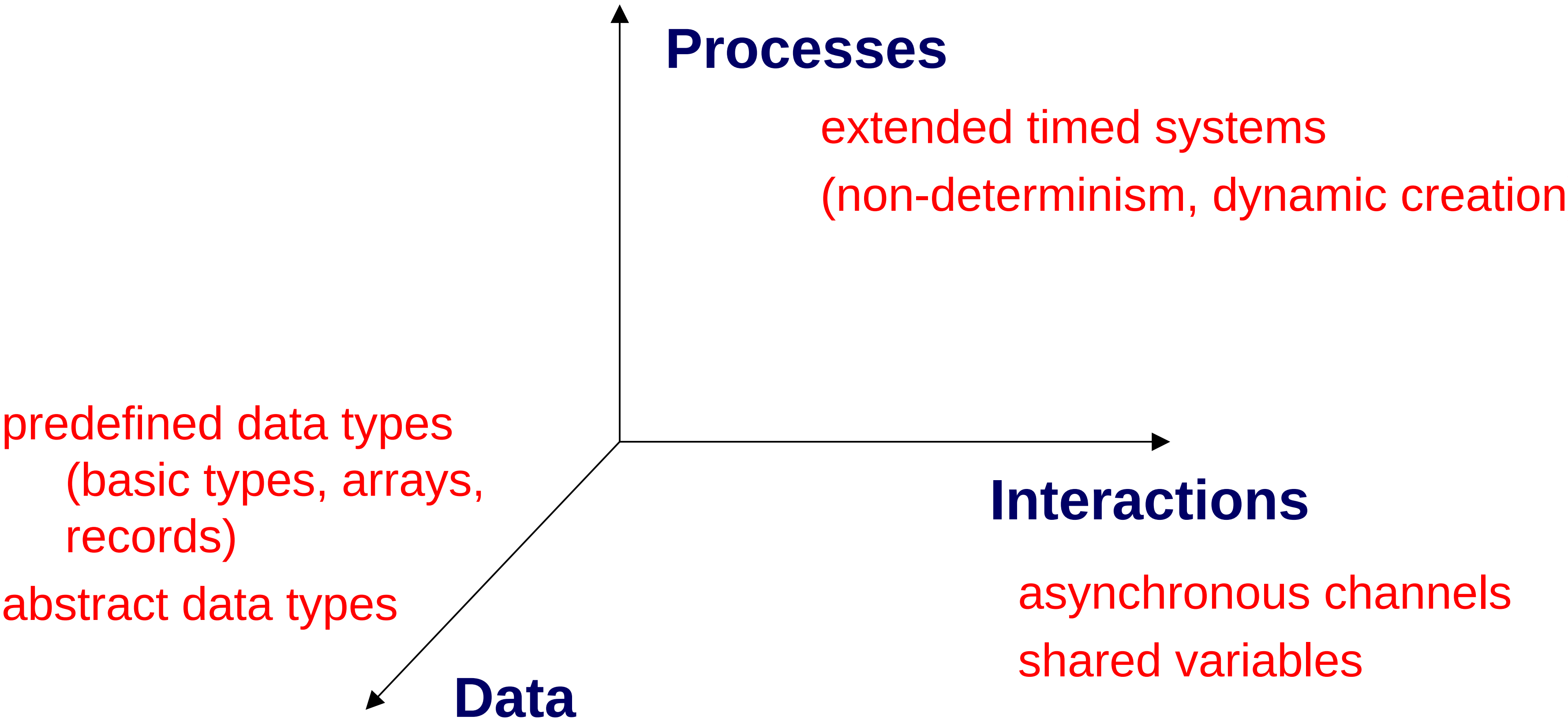
Key Research issues
- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework
- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

The IF toolset
- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion

IF notation: System description



IF notation: System description

- 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

IF notation: System description

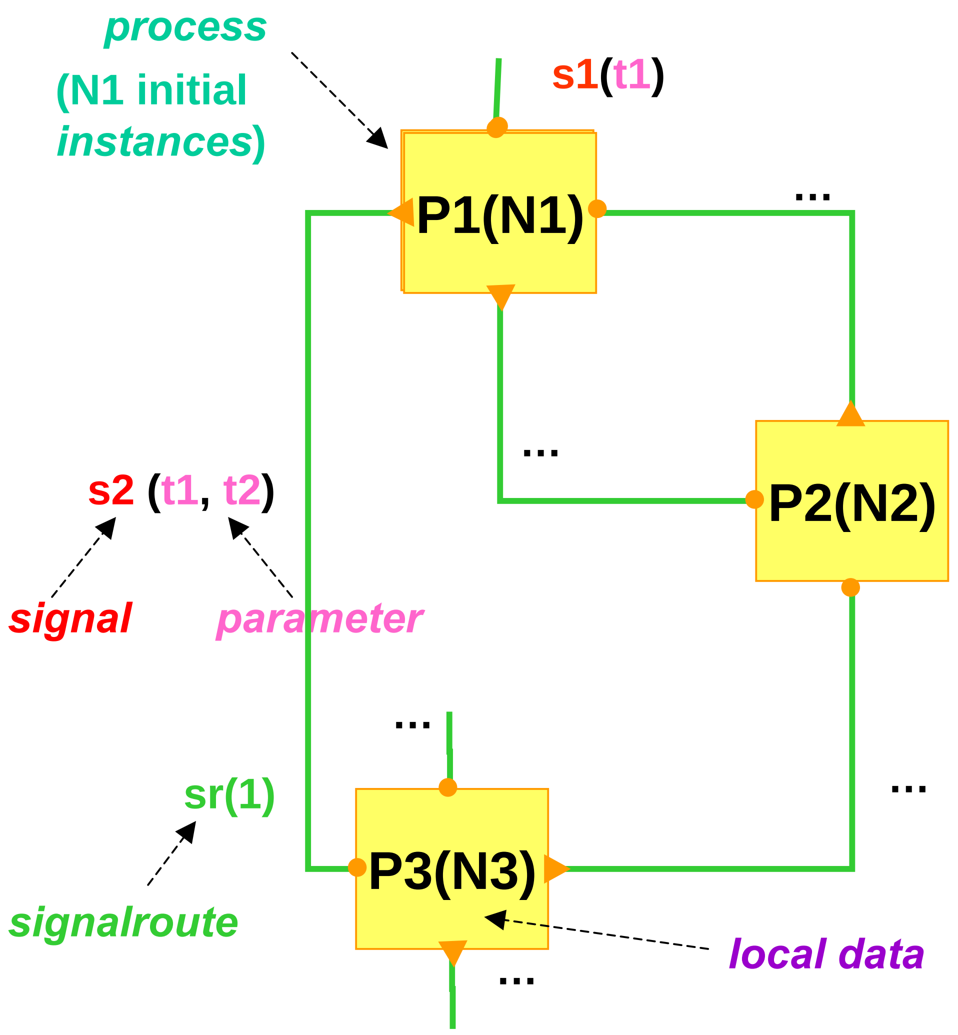
const N1 = ... ; // constants
type t1 = ... ; // types

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;



IF notation: Process description

Process = hierarchical, timed systems with actions

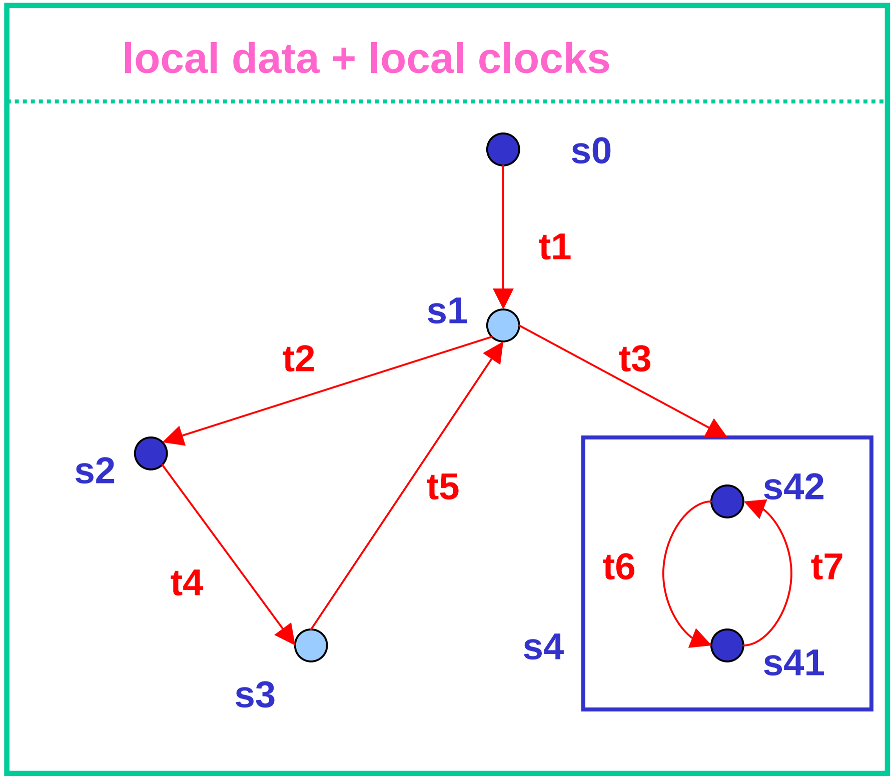
```
process P1(N1);
fpar ... ;

// 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)

IF notation: dynamic creation

- process creation:

parameters

pid of the newly created instance

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

process name

a new instance is created

- process destruction:

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

kill client(2)

pid expression

kill p

- process termination:

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

stop

IF notation: Process description-transition

transition = *urgency* + trigger + body

state s0
…

t1

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

urgency

untimed guard

timed guard

signal consumption from the process buffer

= **trigger**

statement list

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

sequential. conditional, or iterative composition

IF notation: Data and types

Variables:

- are ***statically typed*** (but *explicit conversions* allowed)
- can be declared ***public*** (= shared)

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

$\supseteq$ {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 …

IF notation: interactions - 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;

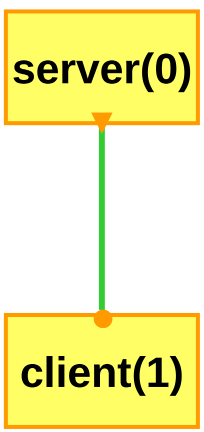
endpoints

attributes:

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

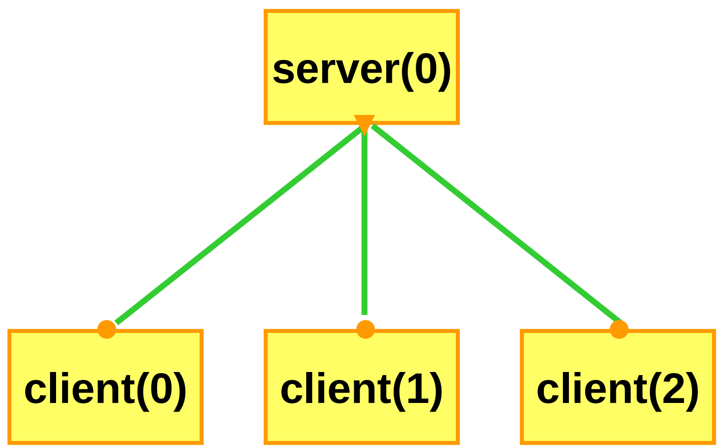
IF notation: interactions - delivery policies

peer



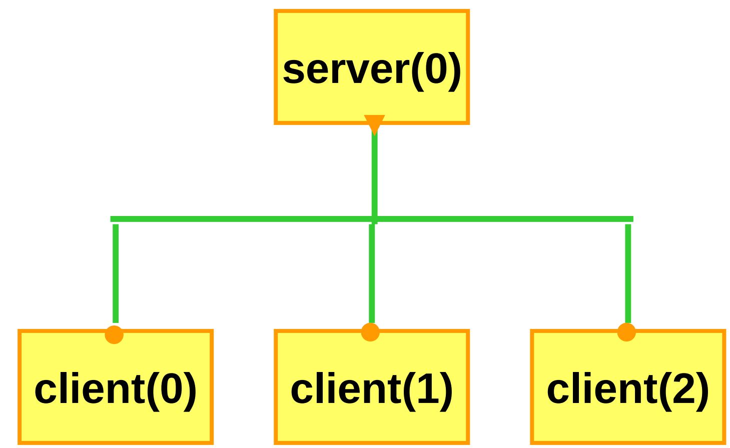
to one specific instance

unicast



to a randomly chosen instance

multicast



to all instances

IF notation: interactions - signal exchange

Signal emission (non blocking):

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;

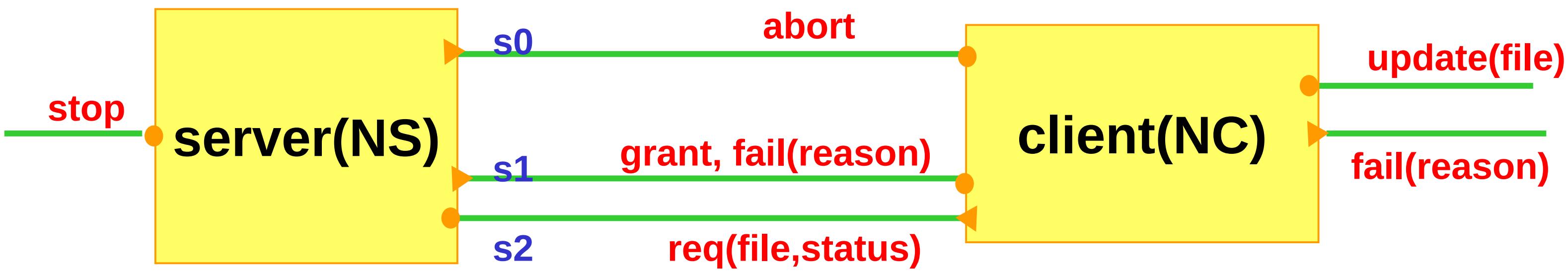
k=integer(self)   pid expression

Signal consumption (blocking):

formal parameters

input req (f, s);

IF notation: System description - example

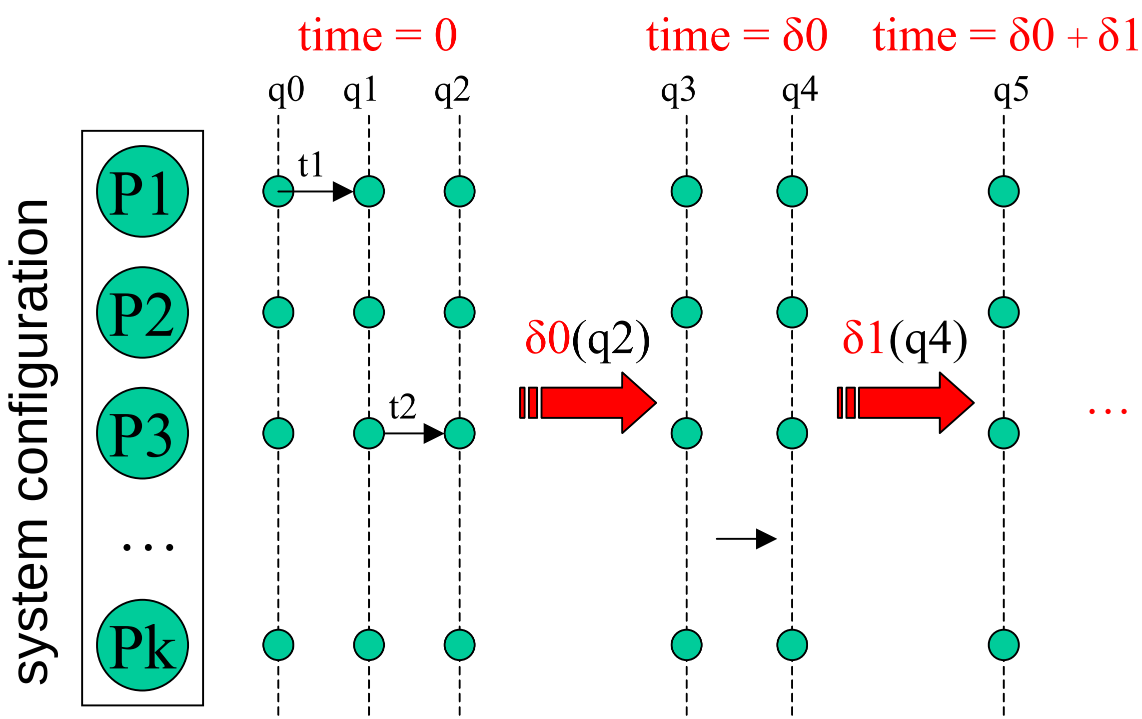



```
const NS= … , NC= … ;
type file= … , status= … , reason= … ;

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

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

process server(NS) … endprocess;
process client(NC) … endprocess;
```

IF notation: timed behavior

The model of time [timed systems]

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



IF notation: 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;

  nextstate wait_ack;
endstate;

state wait_ack;
  input ack(sender,c);
   ...

   ...
endstate;
```

IF notation: 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:
  - run-to-completion:
  - EDF:

Outline

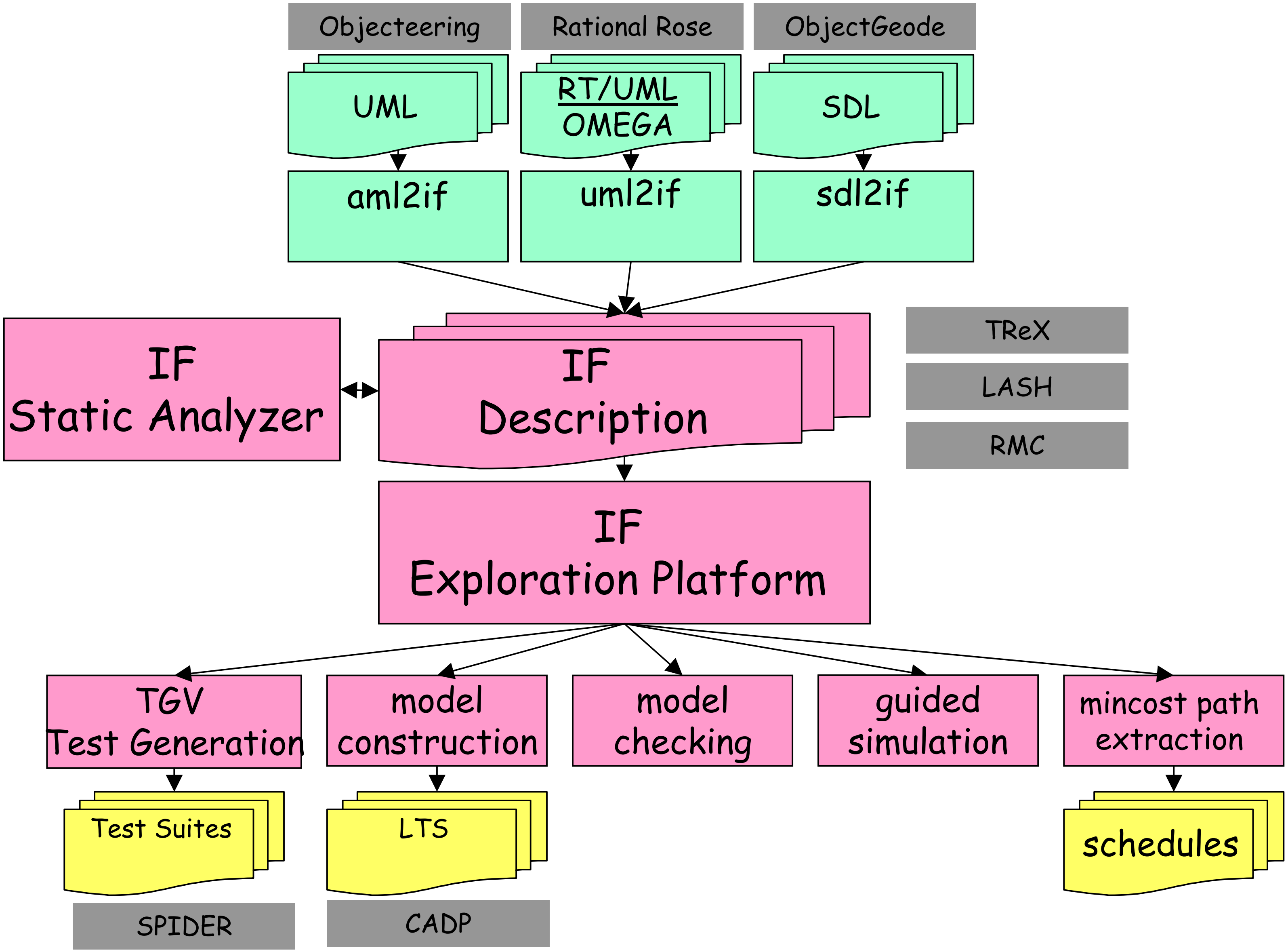
Key Research issues
- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework
- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

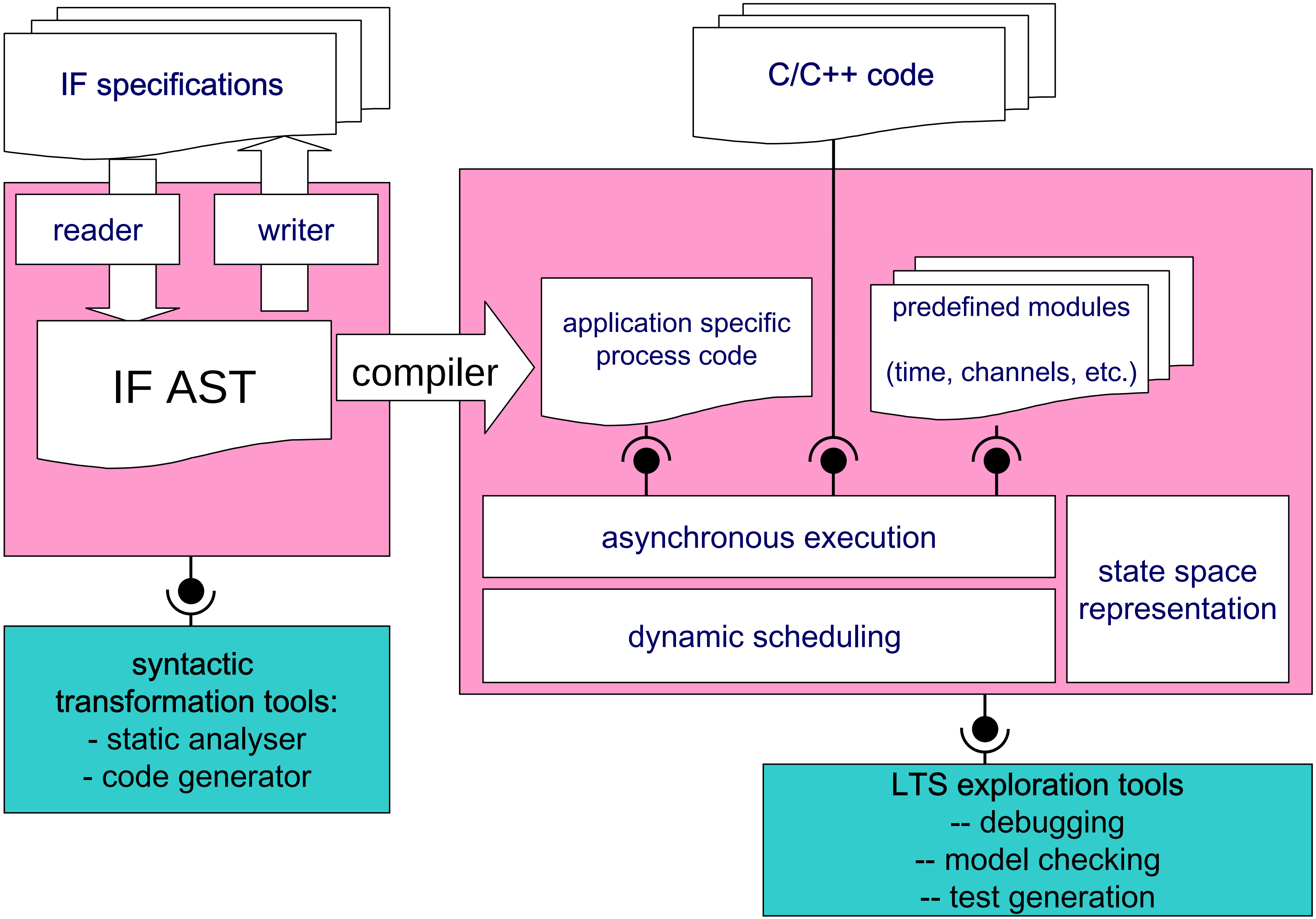
The IF toolset
- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion

IF toolset: overall architecture

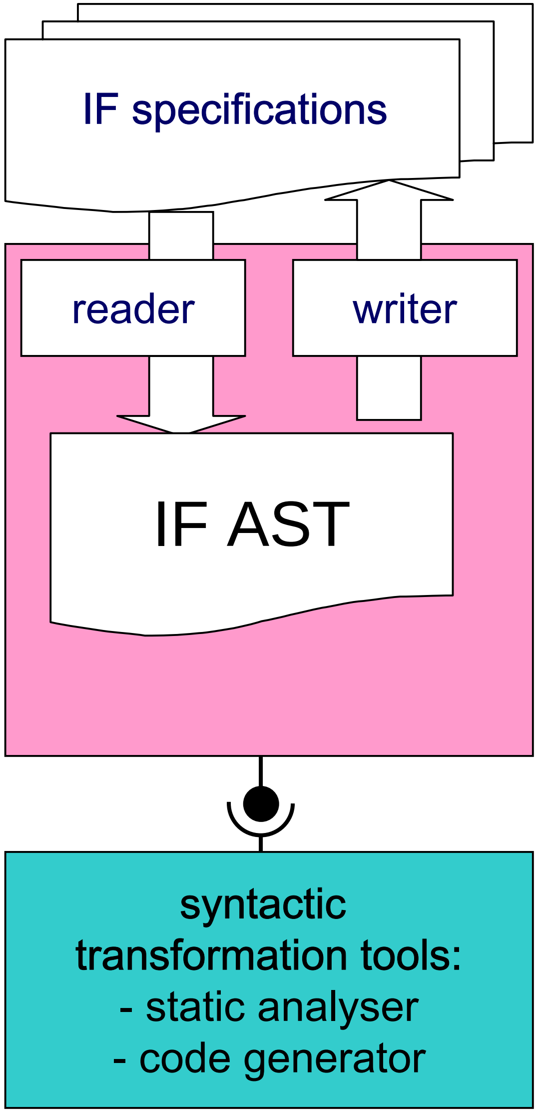


Core components



Core components: syntactic transformations

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

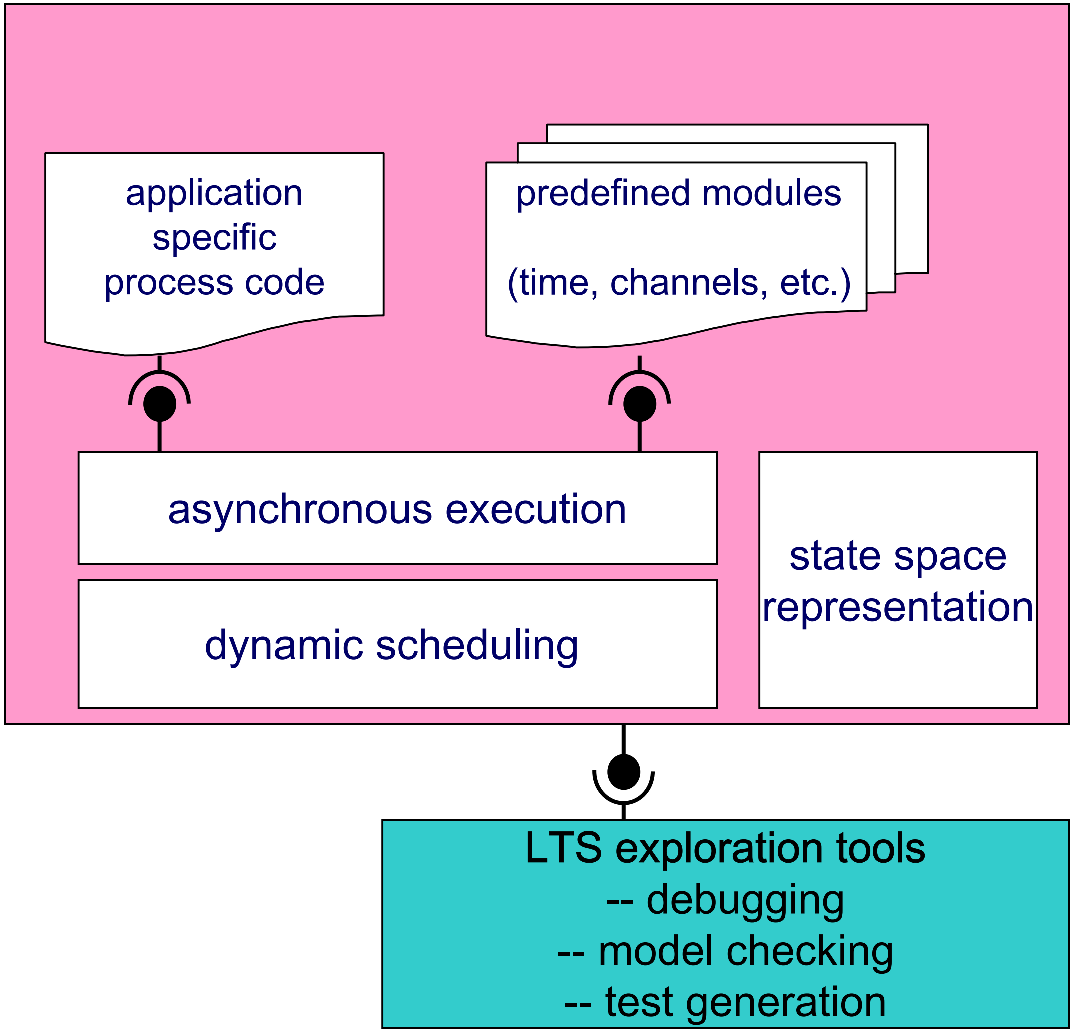


Core components : exploration platform - API

- gives programming access to the underlying labeled transition system of an IF description

- 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

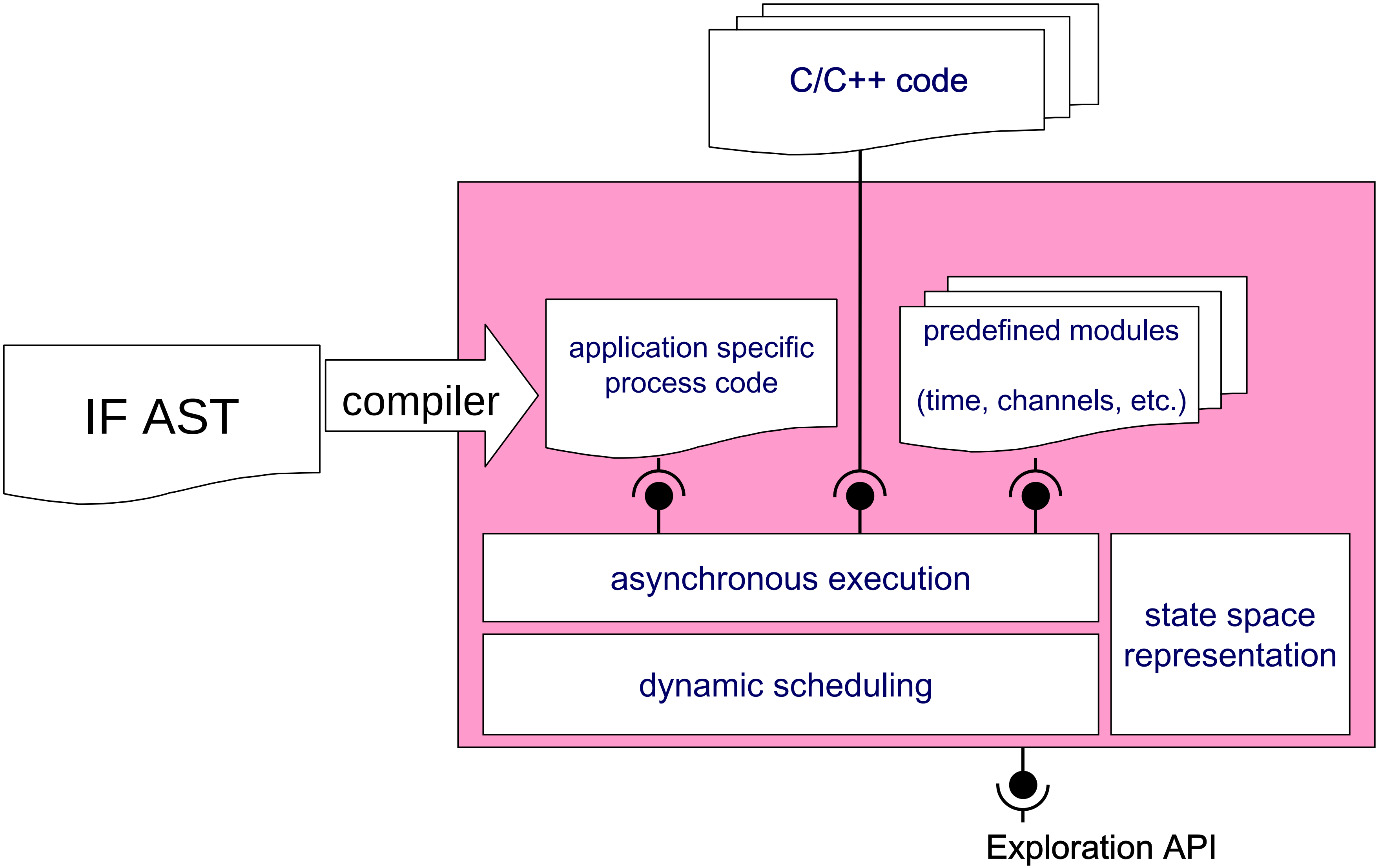


Offers primitives for exhaustive state space exploration

Main features

- process execution simulation
  - inter-process interaction
  - process creation / destruction
  - control of simulation time
- non-determinism handling
  - asynchronous execution
  - internal non-deterministic choices
  - open environment
- state space representation

Core components: exploration platform - architecture



Core components: exploration platform – execution

1st layer: emulates asynchronous parallel execution to obtain globa (system) steps from local (process) steps

- it asks successively, each process instance to execute its enabled transitions
- during the execution of a transition by a process instance,
  - it ensures message delivery and shared variable update
  - it manages dynamic instance creation and destruction
  - it records generated observable events
- when a local step is finished,
  - It takes a snapshot of the global configuration and stores it
  - It sends the successor to the 2nd layer (dynamic scheduler)
- It manages time progress and clocks updates

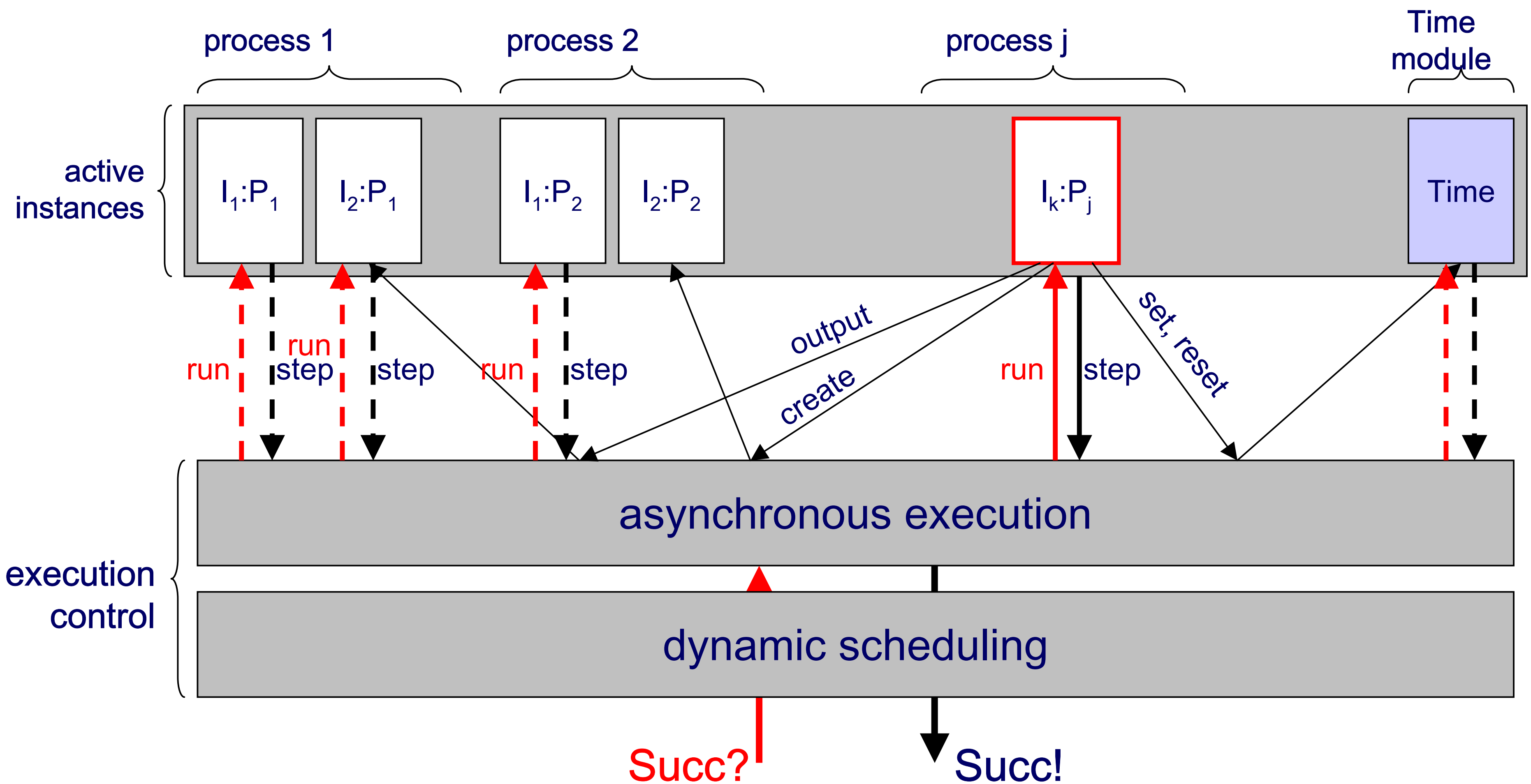
Core components: exploration platform – execution

2^{nd} layer: dynamic scheduling (priorities)

- collects all potential global successors
- filters them according to dynamic priorities
  - evaluates each priority constraint
  - if applicable on current state, it

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

Core components: exploration platform – execution



Core components: exploration platform – 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

Outline

Key Research issues

- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework

- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

The IF toolset

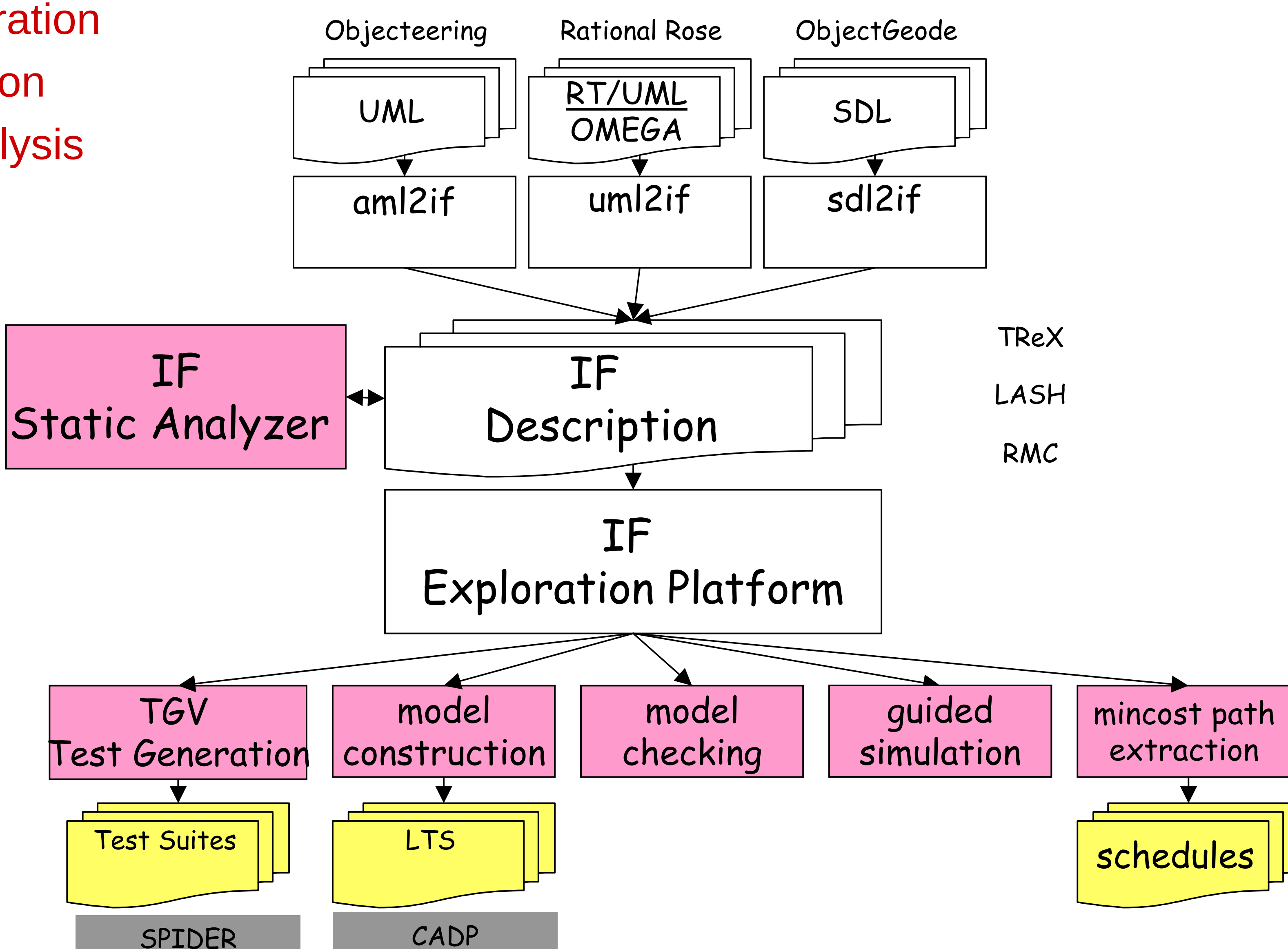
- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion

Validation

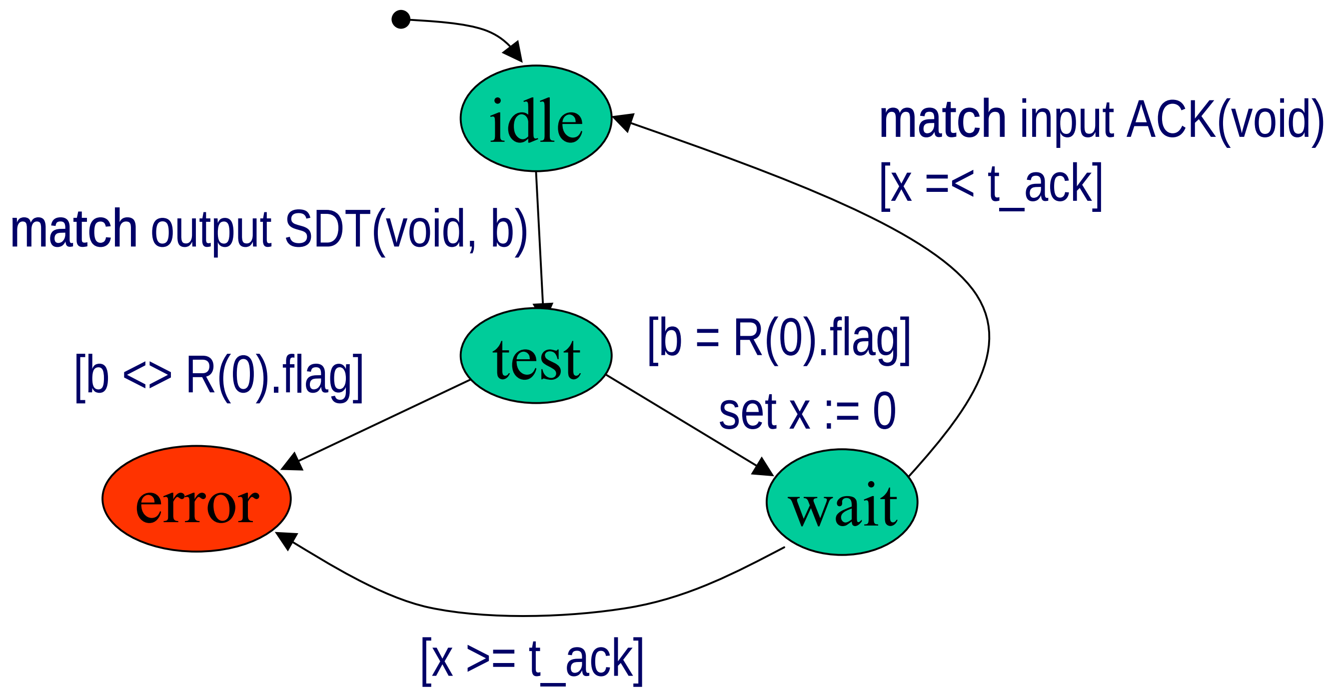
Model-Based Validation

- model checking
- test generation
- optimization
- static analysis



Validation: 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



Validation: requirements - using μ-calculus

- 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,

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

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

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

- 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”)

Validation: behavioral equivalence checking

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

    System $\approx$ Requirements

  - preorder relations (“behavior inclusion”):

    System $\leq$ Requirements

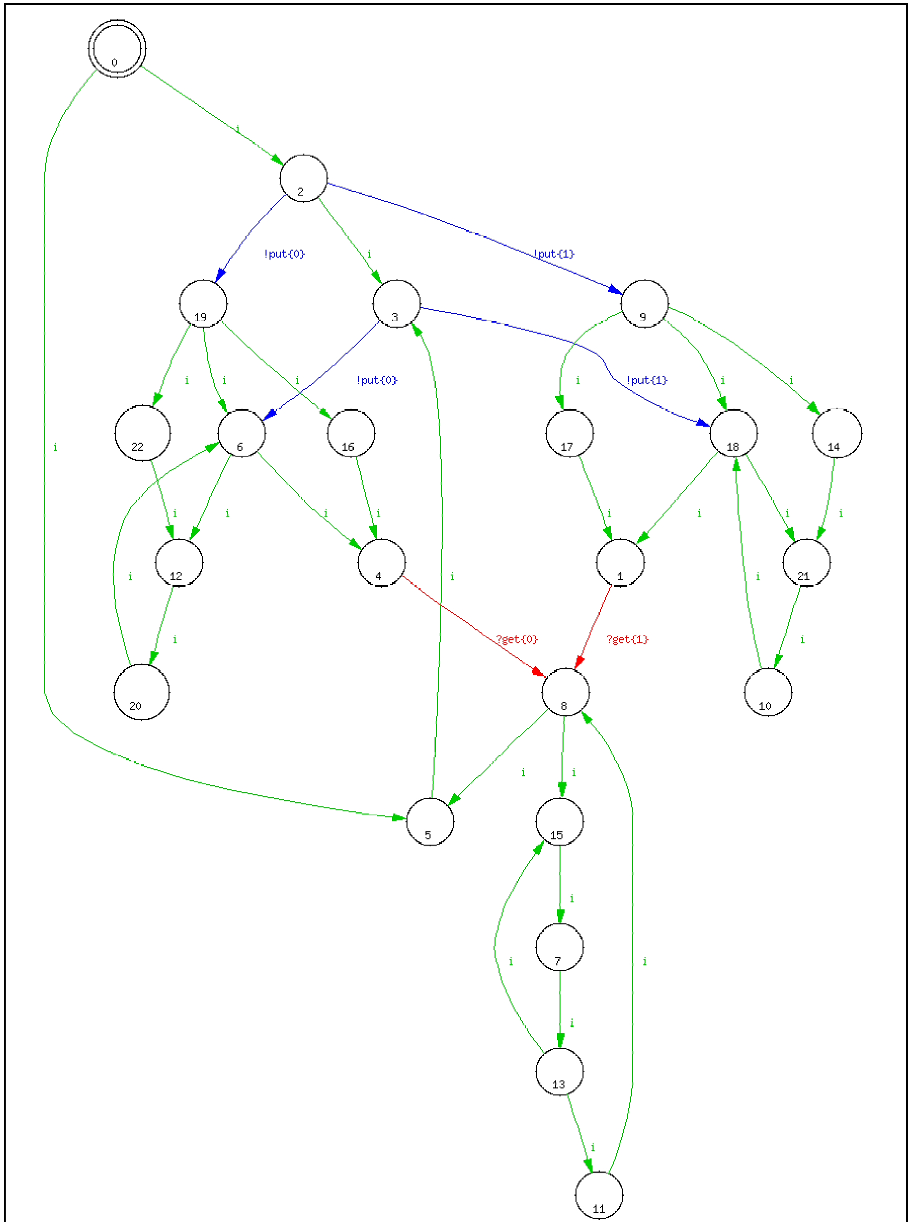
- LTS minimization:
  - quotient w.r.t an equivalence relation:

    (System / $\approx$)

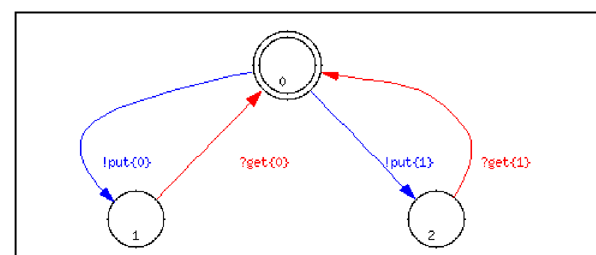
- CADP can be used to check the following relations :

  weak/strong bisimulation, branching, safety, trace equivalence

Validation: behavioral equivalence checking



reduction w.r.t.
branching bisimulation



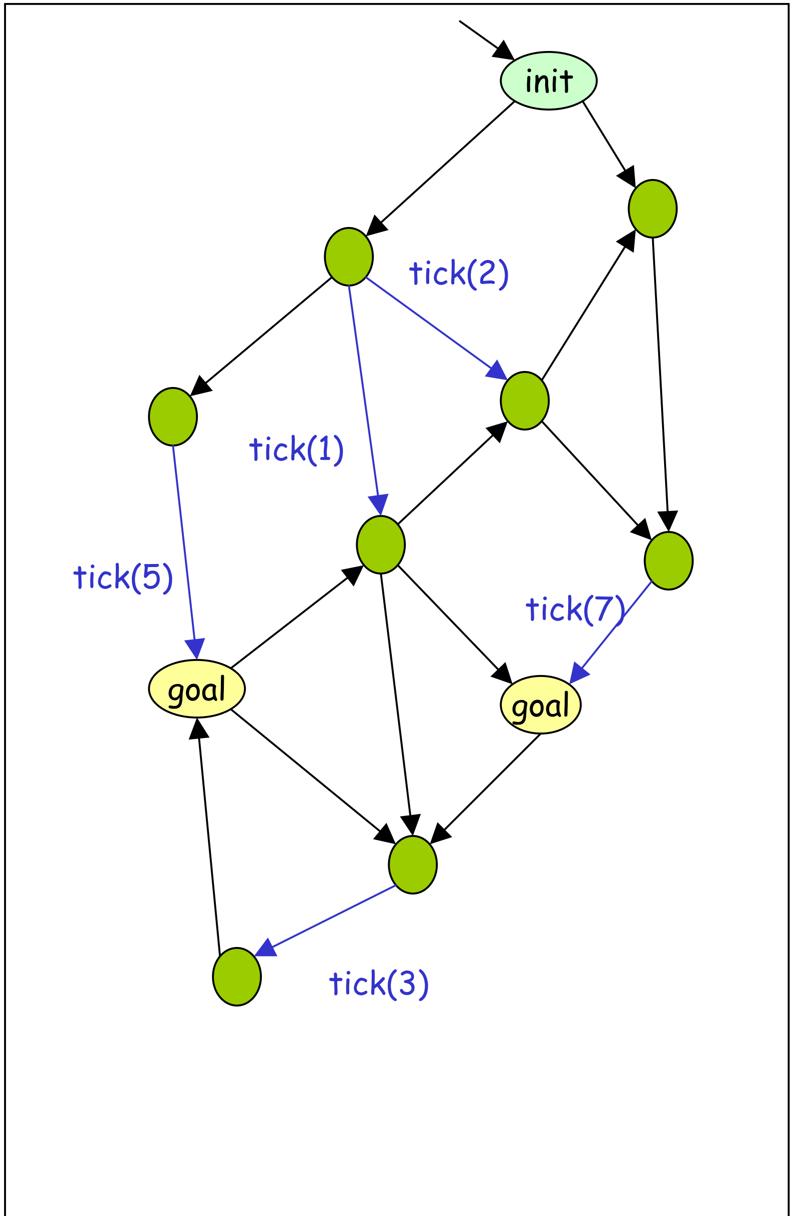
Validation: optimization

- User defined costs associated to transitions of IF descriptions e.g, execution times

- problem: find the min-cost execution path leading from some 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)



Validation: static analysis

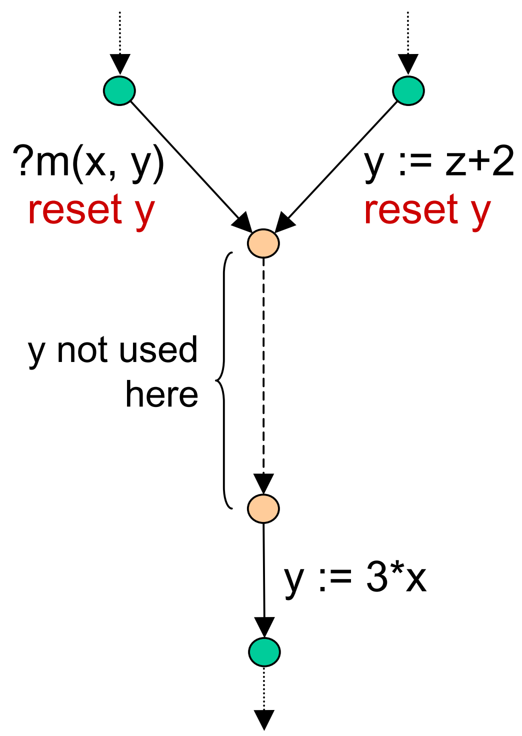
- approach
  - source code transformations for model reduction
  - code optimization methods

- 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

Validation: static analysis – live variables

a variable is **dead** at 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 interaction primitives

- asynchronous interaction via queues
- parameter passing at process creation

live variables are propagated both intra and inter processes !

exploit live variables

transform IF description by

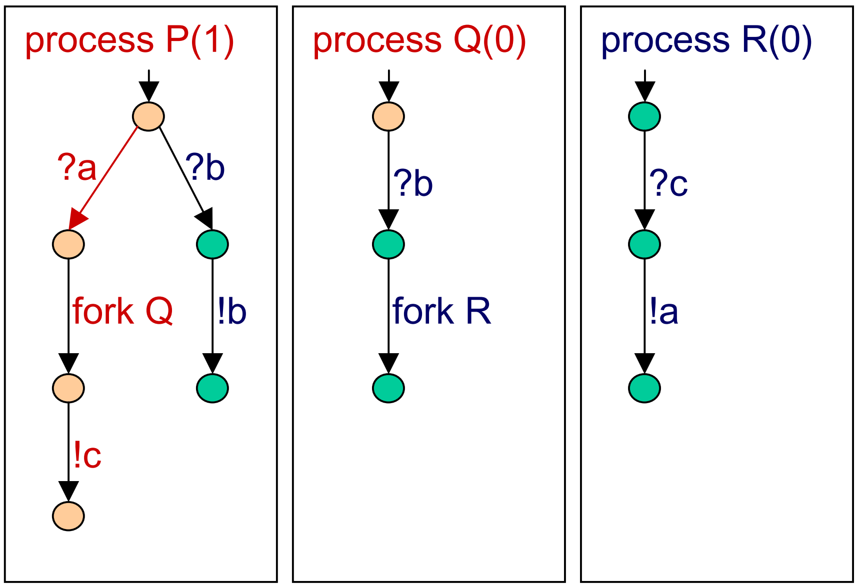
- removing completely dead variables and signal / process parameters
- resetting partially dead variables

the gains are multiple:

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

Validation: static analysis – 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 description by

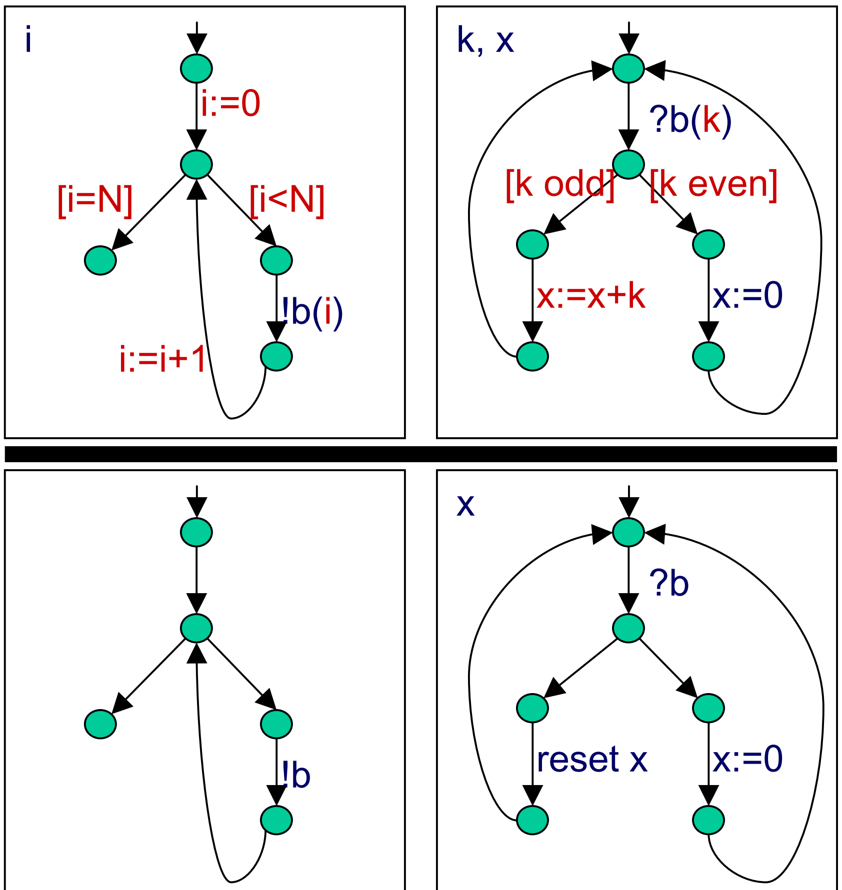
- removing processes never created
- removing signals never sent
- removing unreachable control states and control transitions

the gains are

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

Validation: static analysis – 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 descriptions by

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

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

Outline

Key Research issues

- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework

- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

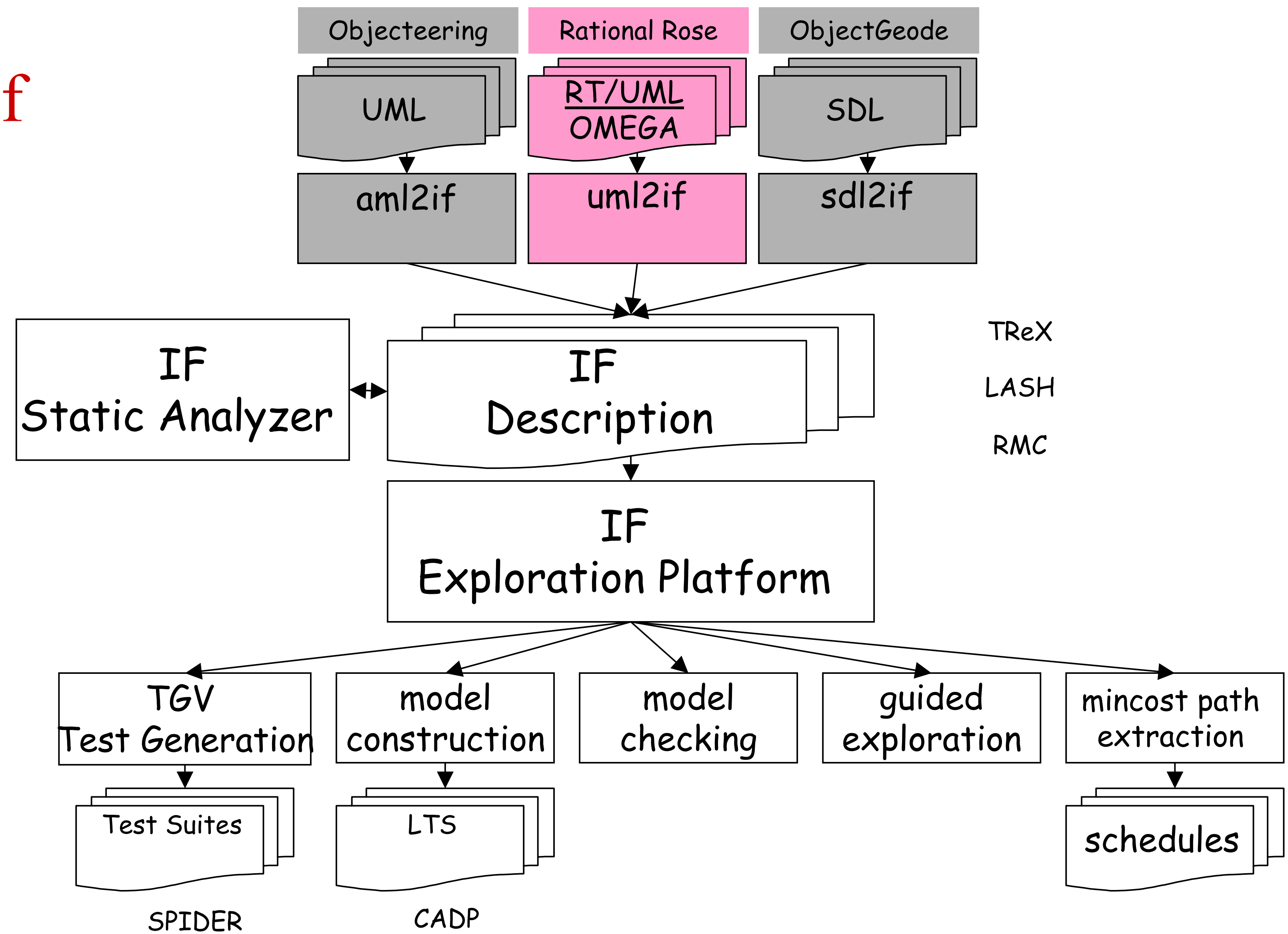
The IF toolset

- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion

Front-Ends

- sdl2if
- uml2if



Front ends: UML2IF – Omega UML

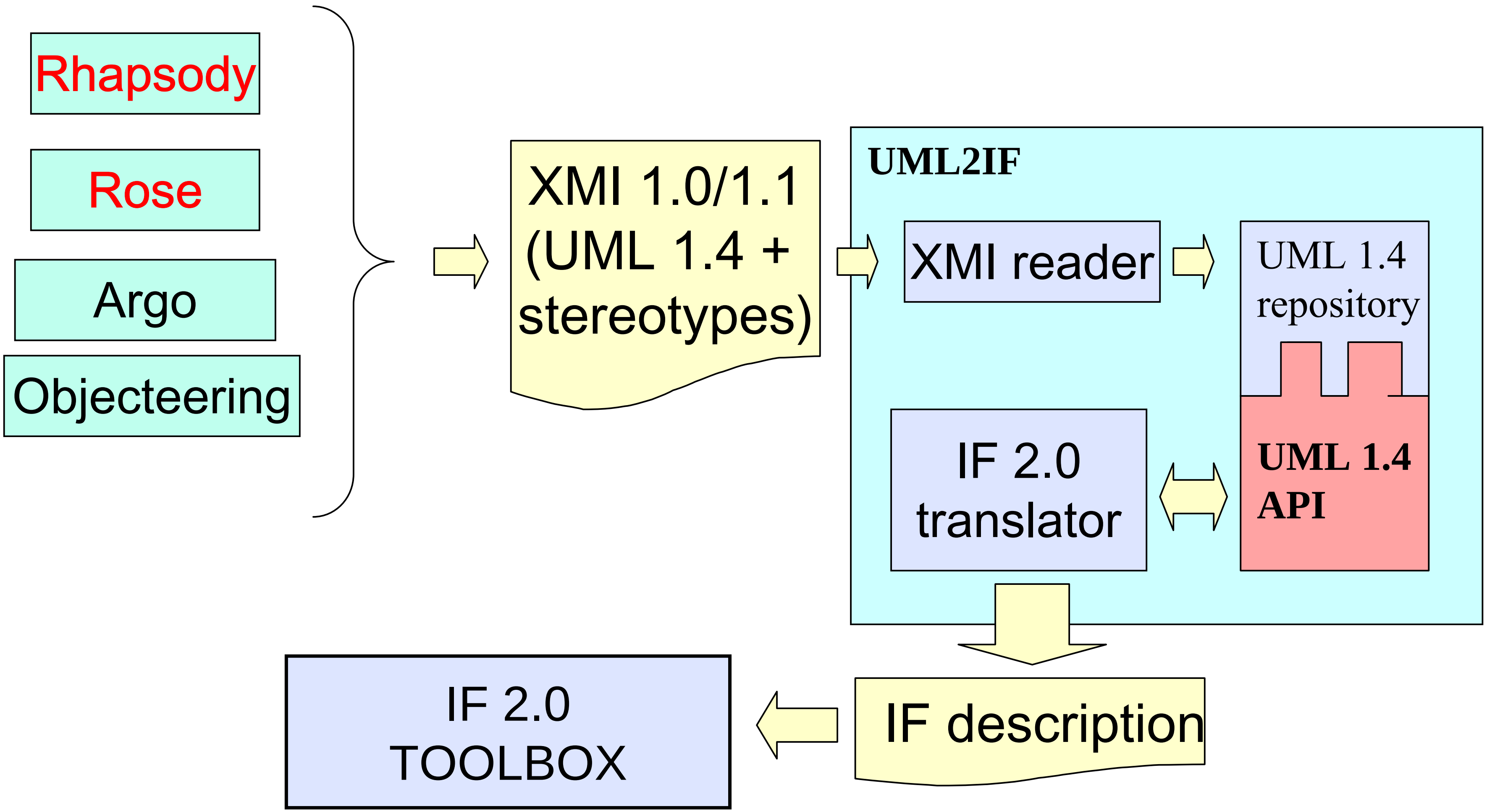
UML for real-time and embedded systems (OMEGA IST project)

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

Front ends: UML2IF – translation principle

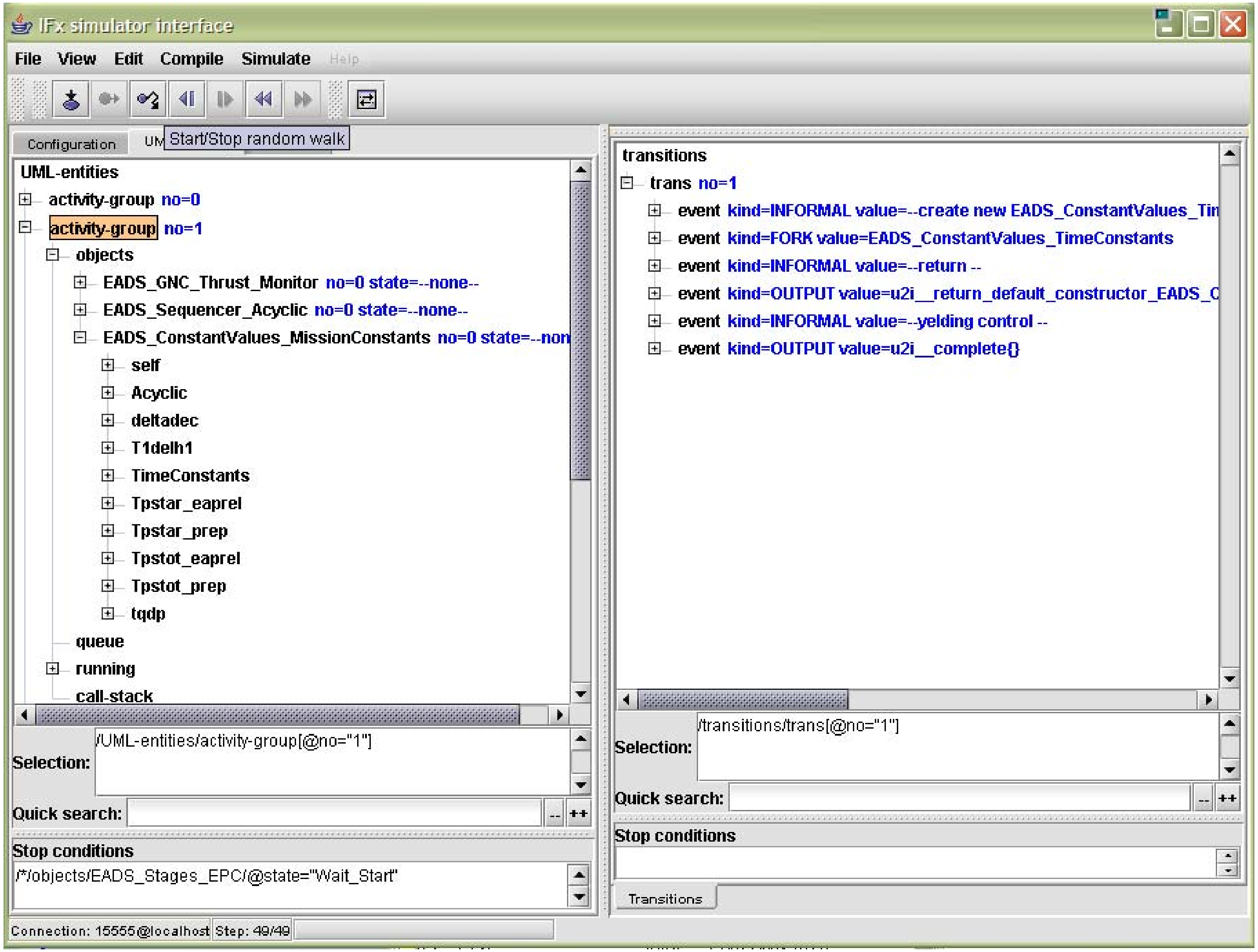
- structure
  - class → process type
  - attributes & associations → variables
  - inheritance → replication of features
  - signals, basic data types → direct mapping
- behavior
  - state machines (with restrictions) → IF hierarchical automata
  - action language → IF actions, automaton encoding
  -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

Front ends: UML2IF – architecture



Front ends: UML2IF – simulation interface

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



Outline

Key Research issues

- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework

- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

The IF toolset

- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion

Case studies: 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

Case studies: 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

Case studies: manufacturing

Job-shop Scheduling

Axxom Lacquer Production

case study proposed in AMETIST IST-2001-35304

Case studies: 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 specifications using IF/CADP**. In *Proceedings of IFIP Intl. Conference on VLSI, Darmstadt, Germany*

Case studies: 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*.

Akhavan et al. **Experiment on Verification of a Planetary Rover Controller**. In *Proceedings of 4th International Workshop on Planning and Scheduling for Space, IWPSS'04, Darmstadt, Germany.*

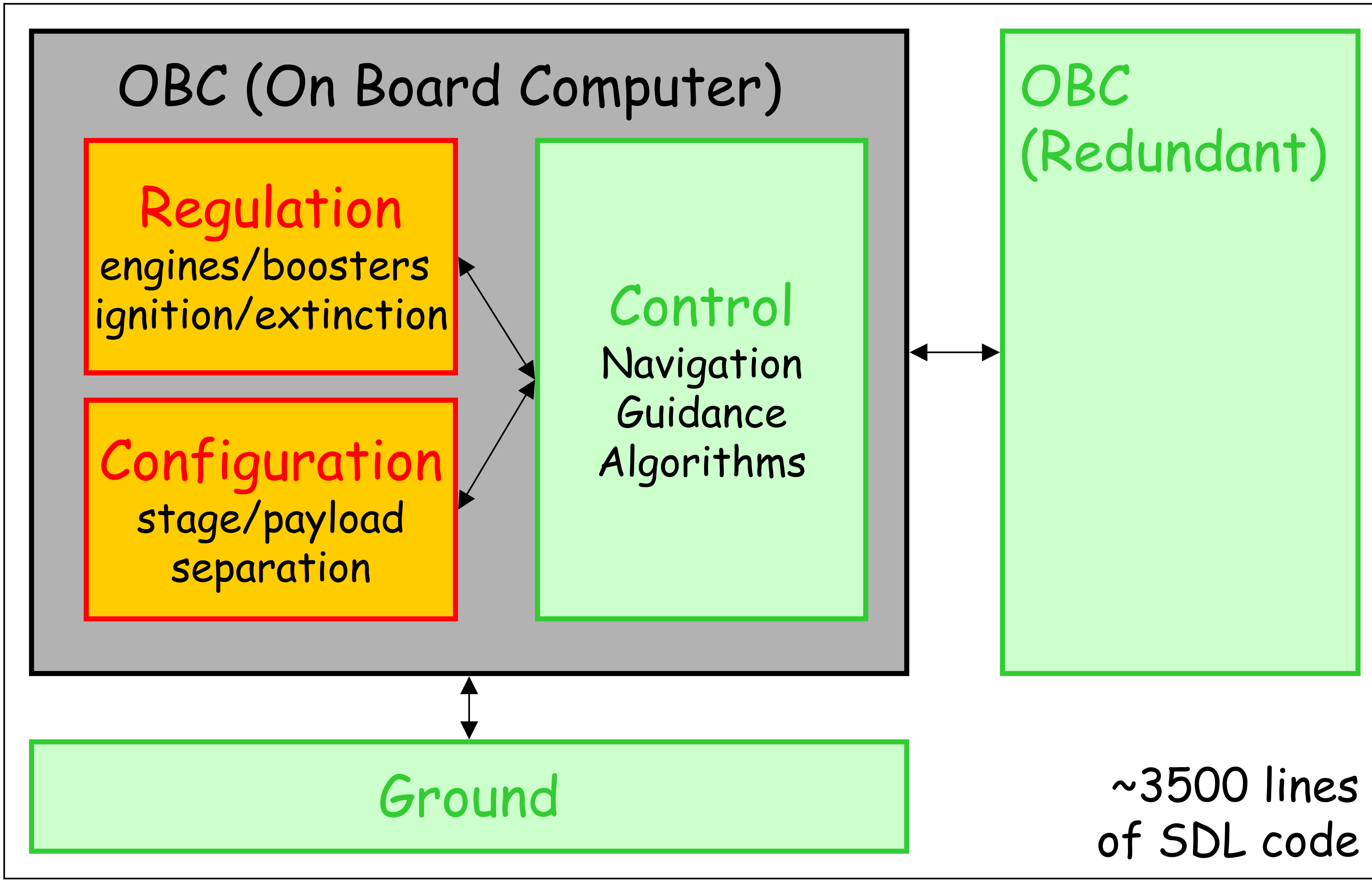
Ariane-5 flight program

Flight program specification

- built by reverse engineering by EADS-LV
- two independent views
  1. asynchronous
     – high level, non-deterministic, abstracts the whole program as communicating extended finite-state machines
  2. synchronous
     – low level, deterministic, focus on specific components …

– we focus on the asynchronous view

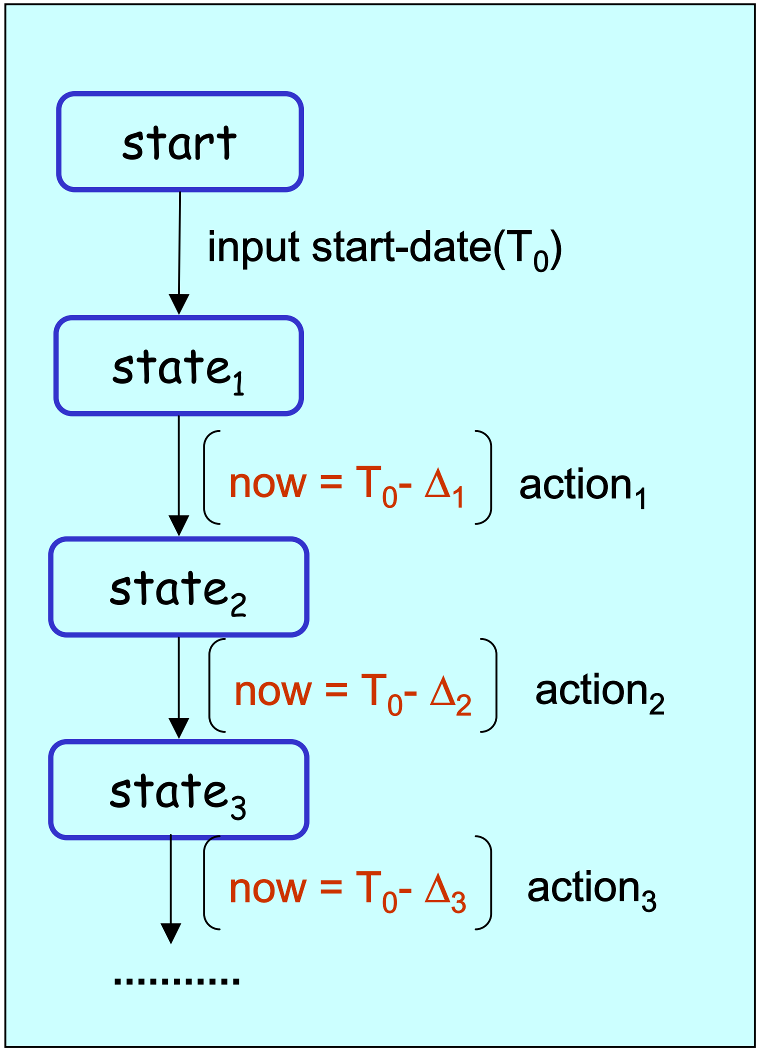
Flight program architecture



Regulation components

- initiate sequences of “regulation” commands at right moments in time :
  - at T_0 - Δ_1 execute $action_1$
  - at T_0 - Δ_2 execute $action_2$

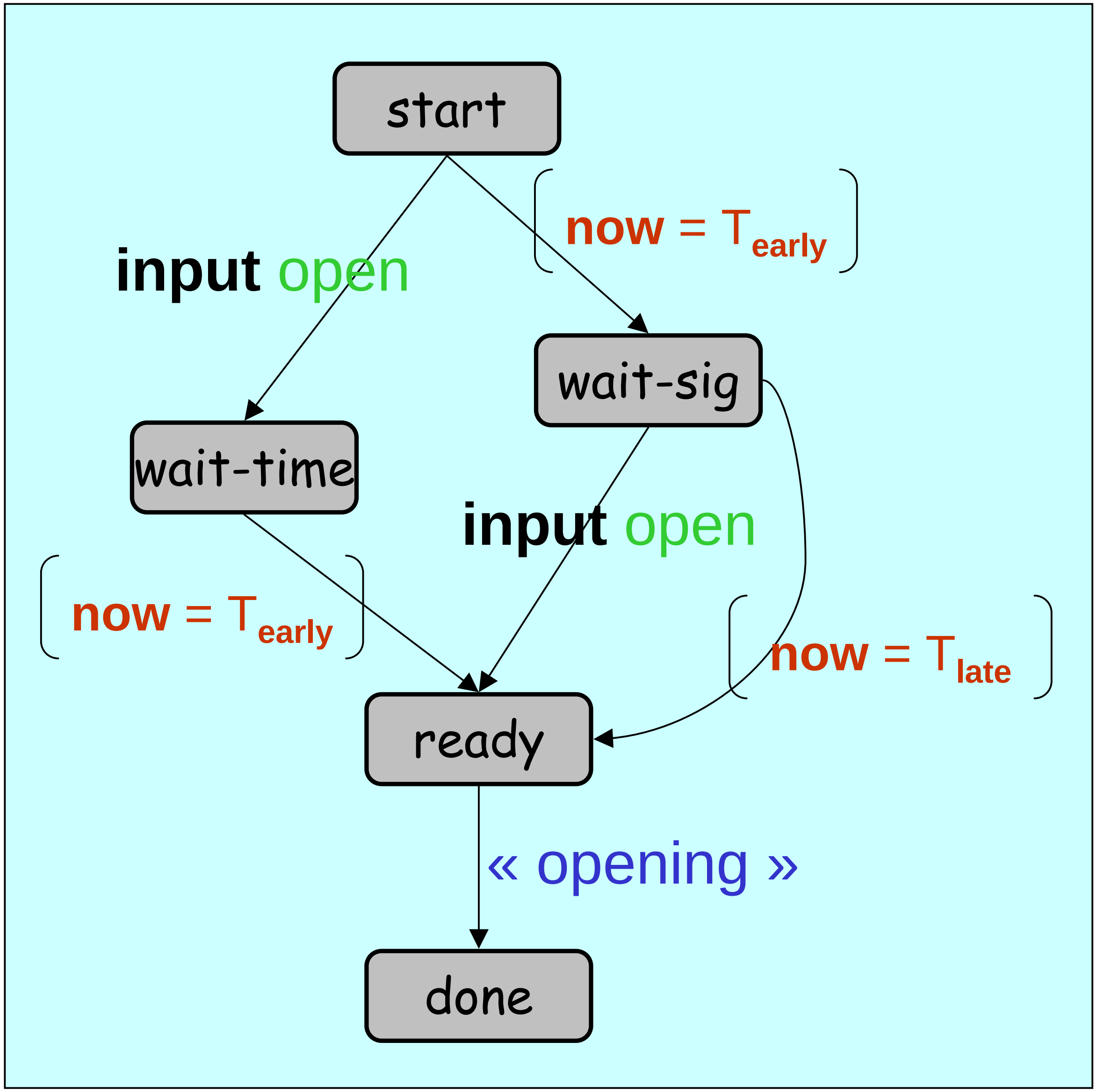
    …
  - at T_0 - Δ_n execute $action_n$
- if necessary, stopped at any moment
- described as “sequential” processes, moving on specific, precise times



Configuration components

- initiate “configuration” changes depending on :
  - flight phase : ground, launch, orbit, …
  - control information: reception of some signal, ...
  - time : eventually done in $[T_0+L, T_0+U]$
- described as processes combining signal and timeout-driven transitions

Configuration component: example



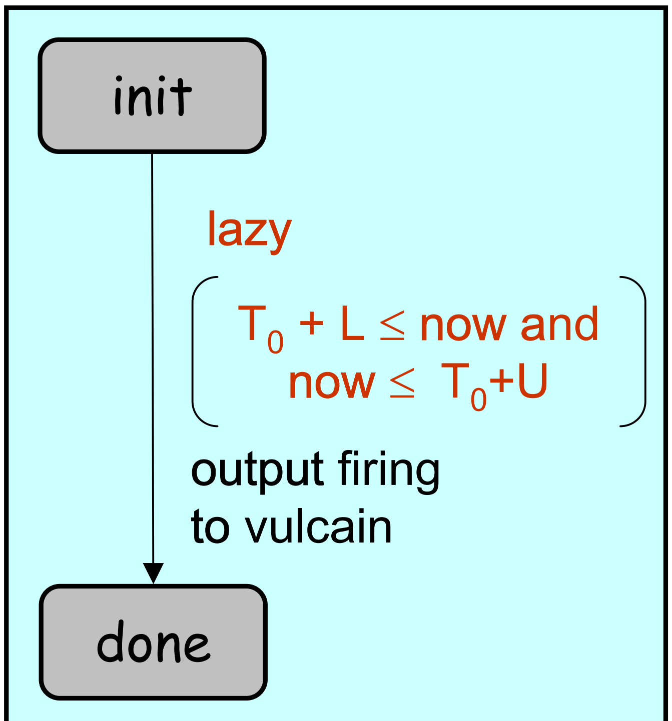
the opening action eventually happens between T_{early} and T_{late} moments, if possible, on the reception on the open signal.

Control components

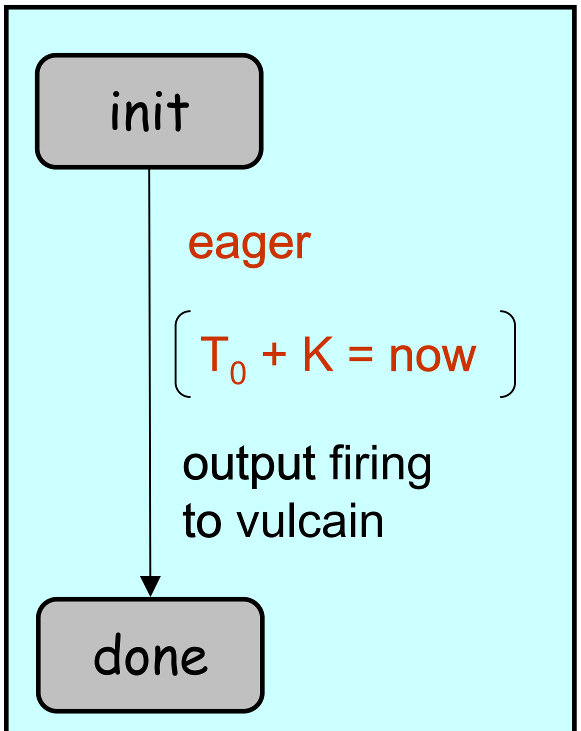
- compute the flight commands depending on the current flight evolution
  - guidance, navigation and control algorithms
- abstracted over-simplified processes
  - send flight commands with some temporal uncertainty

Control components: example

time non-deterministic:
the firing signal can be sent between $T_0 + L$ and $T_0 + U$



time deterministic:
the firing signal is sent exactly at $T_0 + K$



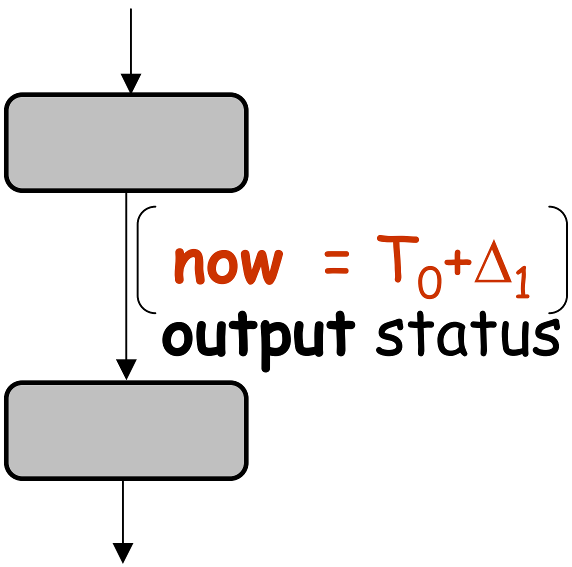
Flight program requirements

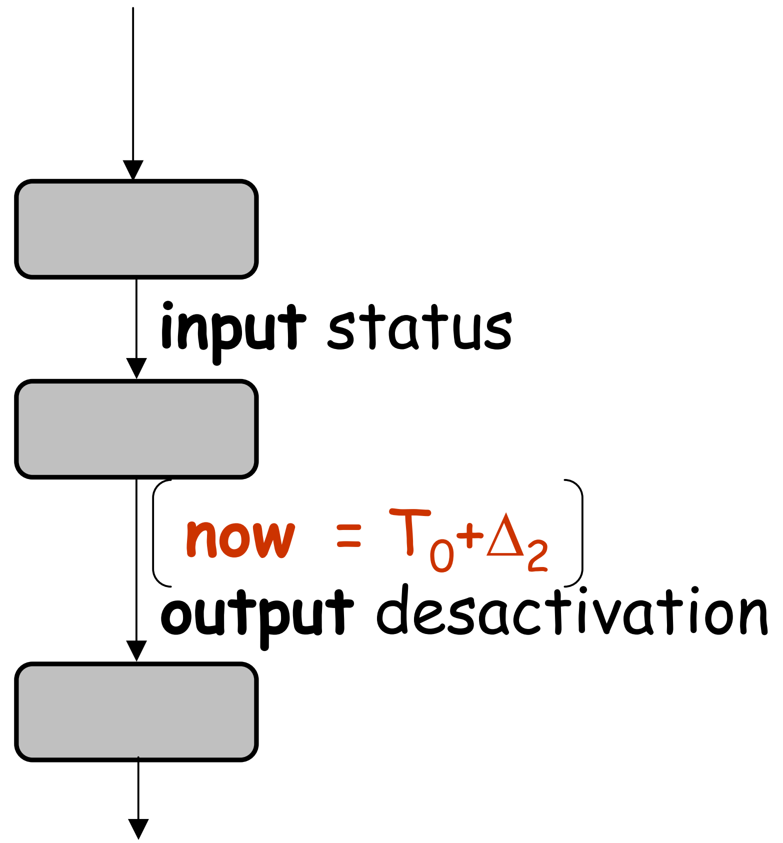
- general requirements
  - e.g, no deadlock, no timelock
- overall system requirements
  - e.g, flight phase order
  - e.g, stop sequence order
- local component requirements
  - e.g, activation signals arrive eventually in some predefined time intervals

Validation: model exploration

- test simple properties by random or guided simulation
- several inconsistencies found

e.g, deadline lost because of $\Delta_1 > \Delta_2$





Validation: 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

Validation: model generation

Some results (31 processes)

	time deterministic	**time non-deterministic**
- live reduction - partial order	**n.a.**	**n.a.**
+ live reduction - partial order	**2201760 st. 18796871 tr.**	**n.a.**
+ live reduction + partial order	**1604 st. 1642 tr.**	**195718 st. 278263 tr.**

Validation: model-checking

- evaluation of μ-calculus formula

  Property: “the stop sequence no. 3 could happen only in a flight phase”

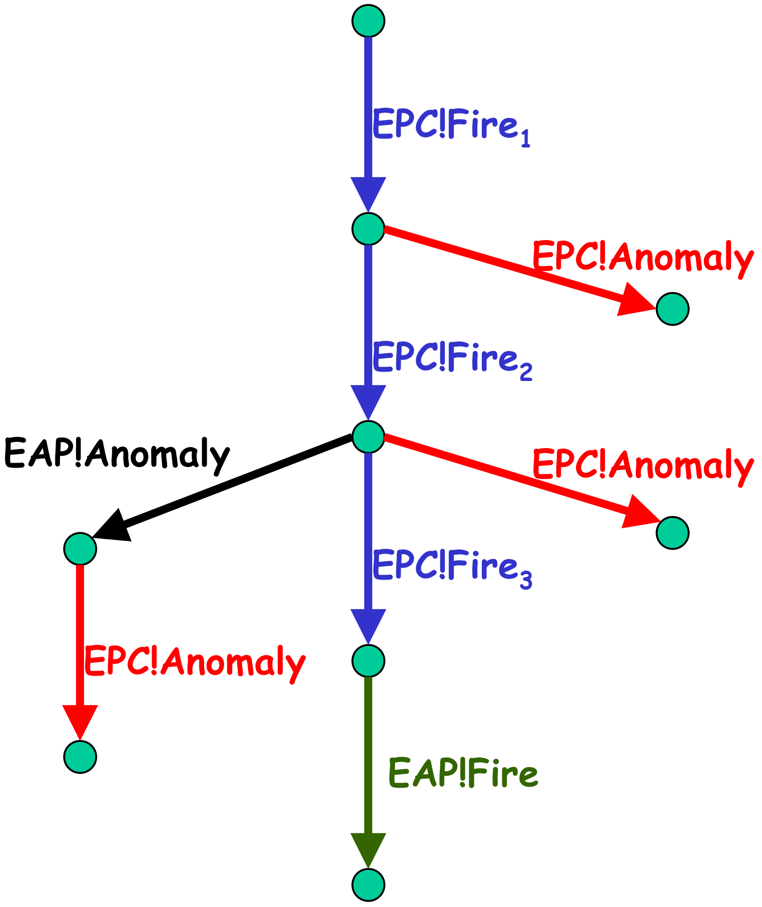
  $$\neg\, \mu X.\ \langle \mathrm{EPC!Stop3} \rangle \mathbf{True} \vee \langle \overline{\mathrm{EAP!Fire}} \rangle X$$

- construction and visualisation of bisimulation reduced models

Validation: model-checking

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

Graph obtained by weak bisimulation minimisation



Outline

Key Research issues

- Modeling Real-time systems
- From application SW to implementations
- Component-based construction

The modeling framework

- Parallel composition
- Adding timing constraints
- Scheduler modeling
- Timed systems with priorities

The IF toolset

- IF notation
- Core components
- Validation
- Front ends
- Case studies

Discussion

Discussion : Modeling – the framework

Specific and tractable construction methodology

- Rely on a minimal set of constructs and principles e.g. combines parallel composition and restriction by priorities

- Avoid declarative formalisms such as temporal logic, LSC

- Focus on specific construction principles and rules to ensure correctness constructively, especially for safety and deadlock-freedom

Discussion : Modeling - combining behavior and priorities

Priorities prove to be a very powerful modeling tool

- they can advantageously replace static restriction
- they allow straightforward modeling of urgency and of scheduling policies
- run to completion and synchronous execution can be modeled by assigning priorities to threads
- Layered description => separation of concerns => incremental description

The IF notation is expressive enough to map compositionally most UML constructs and concepts e.g. Classes, state machines, activity groups

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