Sémantique formelle de modèles d'architecture

ETR'15

Jean-Pierre Talpin

Contributeurs Loïc Besnard, Adnan Bouakaz, Etienne Borde, Pierre Dissaux, Thierry Gautier, Paul Le Guernic, Yue Ma, Huafeng Yu

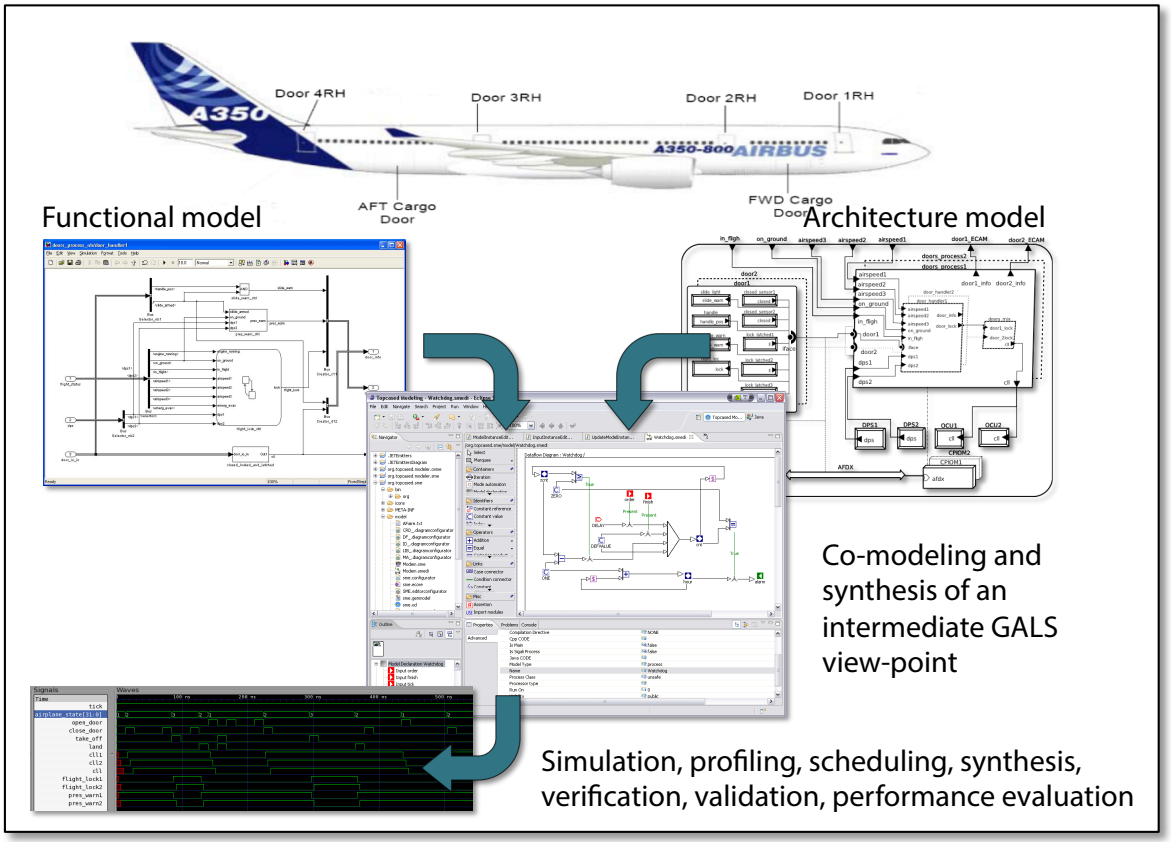
Partly supported by

Agenda

- Problem statement
- AADL is a formal specification language
- Behavioral semantics of the AADL
- Extent and applications of the semantics

Problem Statement

Use-case Virtual prototyping and verification of the Airbus A350 doors management system



(Artemisia project CESAR, 2010+)

Problem Statement

Why a formal semantics for an (architecture specification) standard, AADL ?

- Its purpose
  - Share logical, quantitative, behavioral information about the system under design
  - Information has value: user and standards requirements, manual or automated analysis, …
  - Information is (should be) unambiguous
- Its uses

  All of them involve formal methods: fault analysis, error modeling, safety analysis, schedulability analysis, conformance verification, simulation, scheduler synthesis

Problem Statement
Related Works and Different Approaches

- The MARTE/CCSL profile : a formal semantic profile for the UML standard

- Integrated CPS design workbenches : Ptolemy, BIP, …

- System Modeler, Simplorer : interpreting SysML architectures in SCADE
- Project P : interpreting multiple modeling formalisms in a synchronous MoCC

- Semantics of AADL in Fiacre, Lustre, Signal, TASM, …

Problem Statement

How to define the semantics of an existing standard, AADL ?

What it is not

- Write a conference paper, a book, a user manual, a tutorial, a rationale, a self contained encyclopedia of logics, type theory, automata, trace theory
- Impose an interpretation of the standard in a specific model of concurrency and/or tune the standard to a specific tool

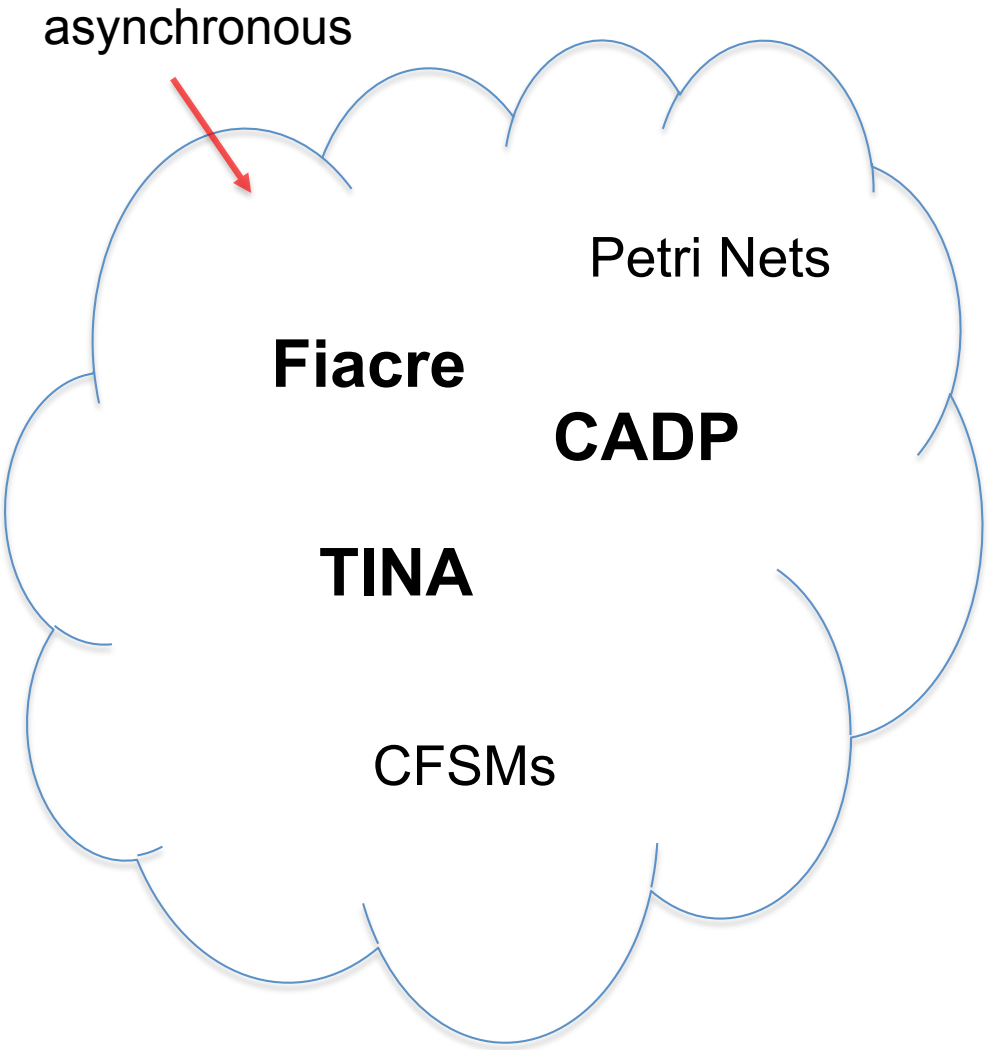
The standard is unambiguous in the first place!

Formal definitions of the standard's constituents: purpose, structure, meaning, relations, extent, and uses ?

➔ Its semantics has to reflect its present and intended uses

Rationale

How to define the semantics of an existing standard, AADL ?



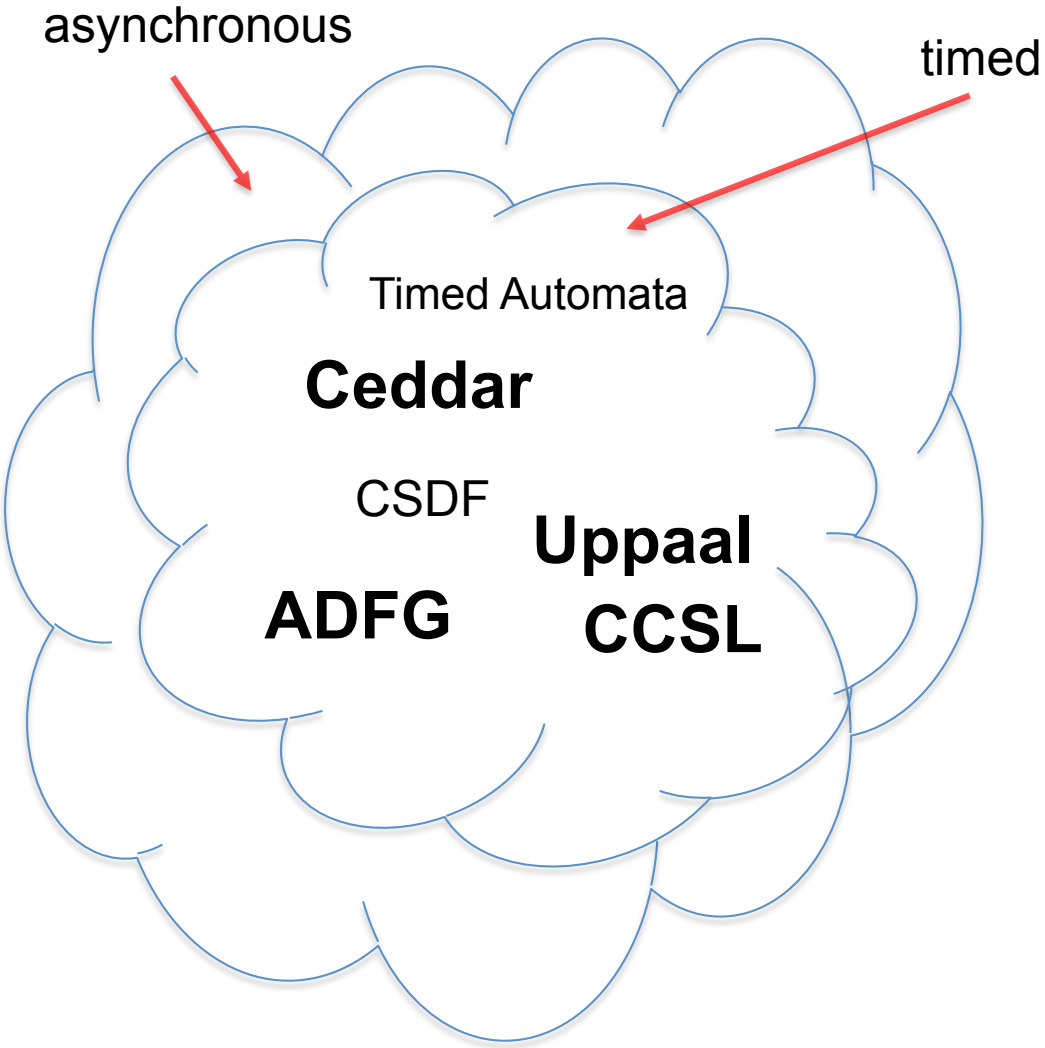
The standard is unambiguous in the first place!

Formal definitions of the standard's constituents: purpose, structure, meaning, relations, extent, and uses ?

➔ Its semantics has to reflect its present and intended uses

Rationale

How to define the semantics of an existing standard, AADL ?



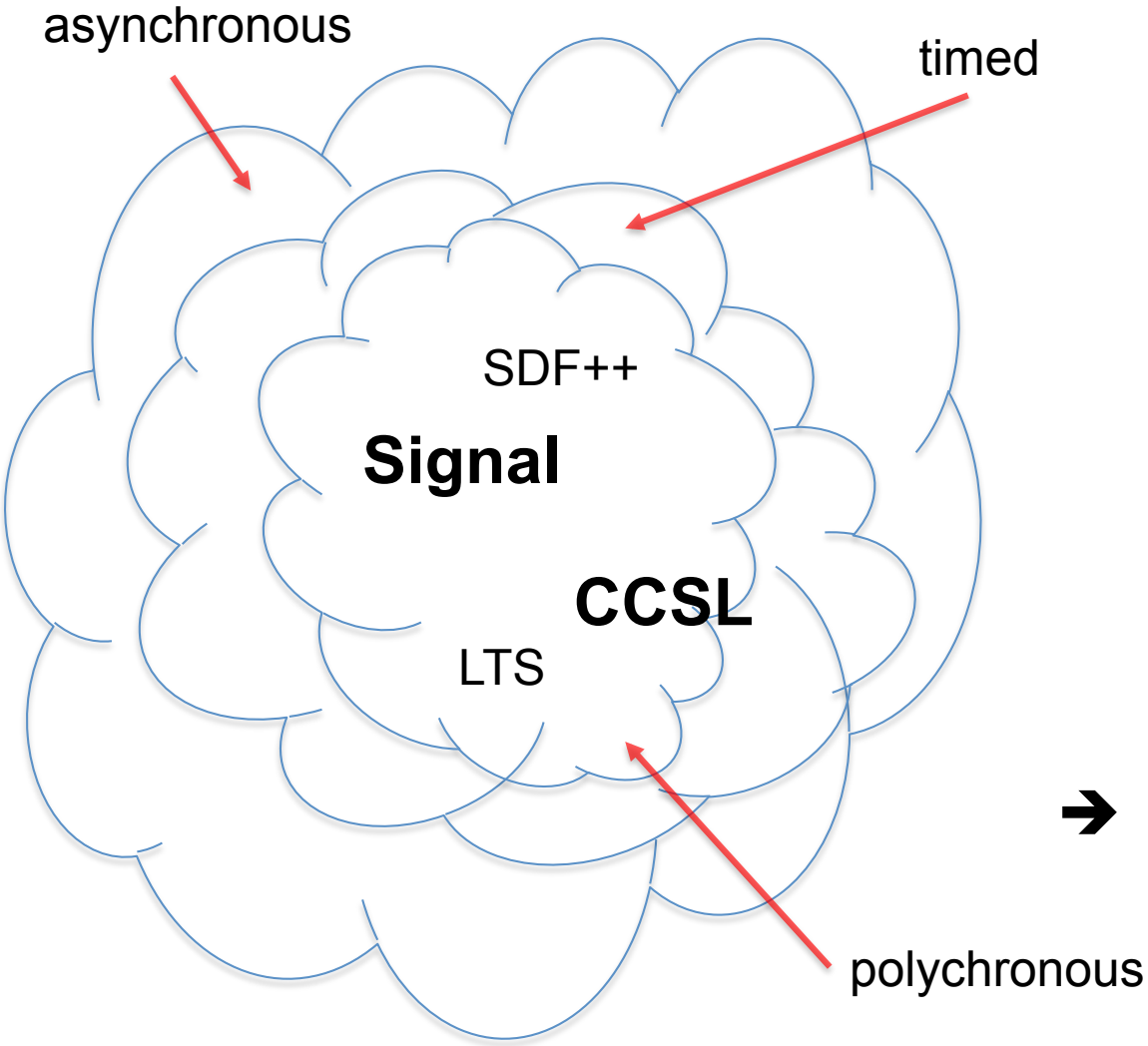
The standard is unambiguous in the first place!

Formal definitions of the standard's constituents: purpose, structure, meaning, relations, extent, and uses ?

➔ Its semantics has to reflect its present and intended uses

Rationale

How to define the semantics of an existing standard, AADL ?



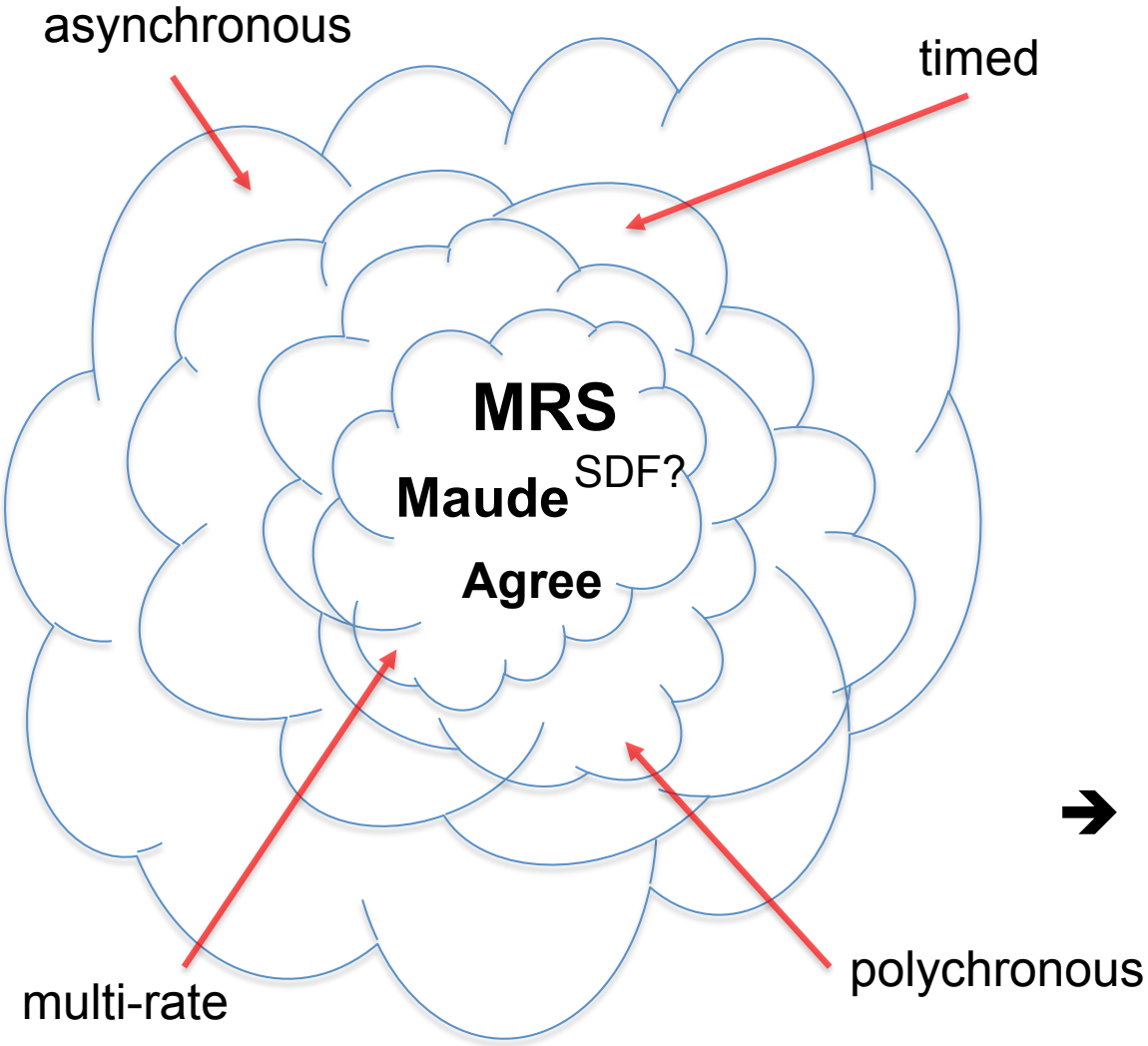
The standard is unambiguous in the first place!

Formal definitions of the standard's constituents: purpose, structure, meaning, relations, extent, and uses ?

➔ Its semantics has to reflect its present and intended uses

Rationale

How to define the semantics of an existing standard, AADL ?



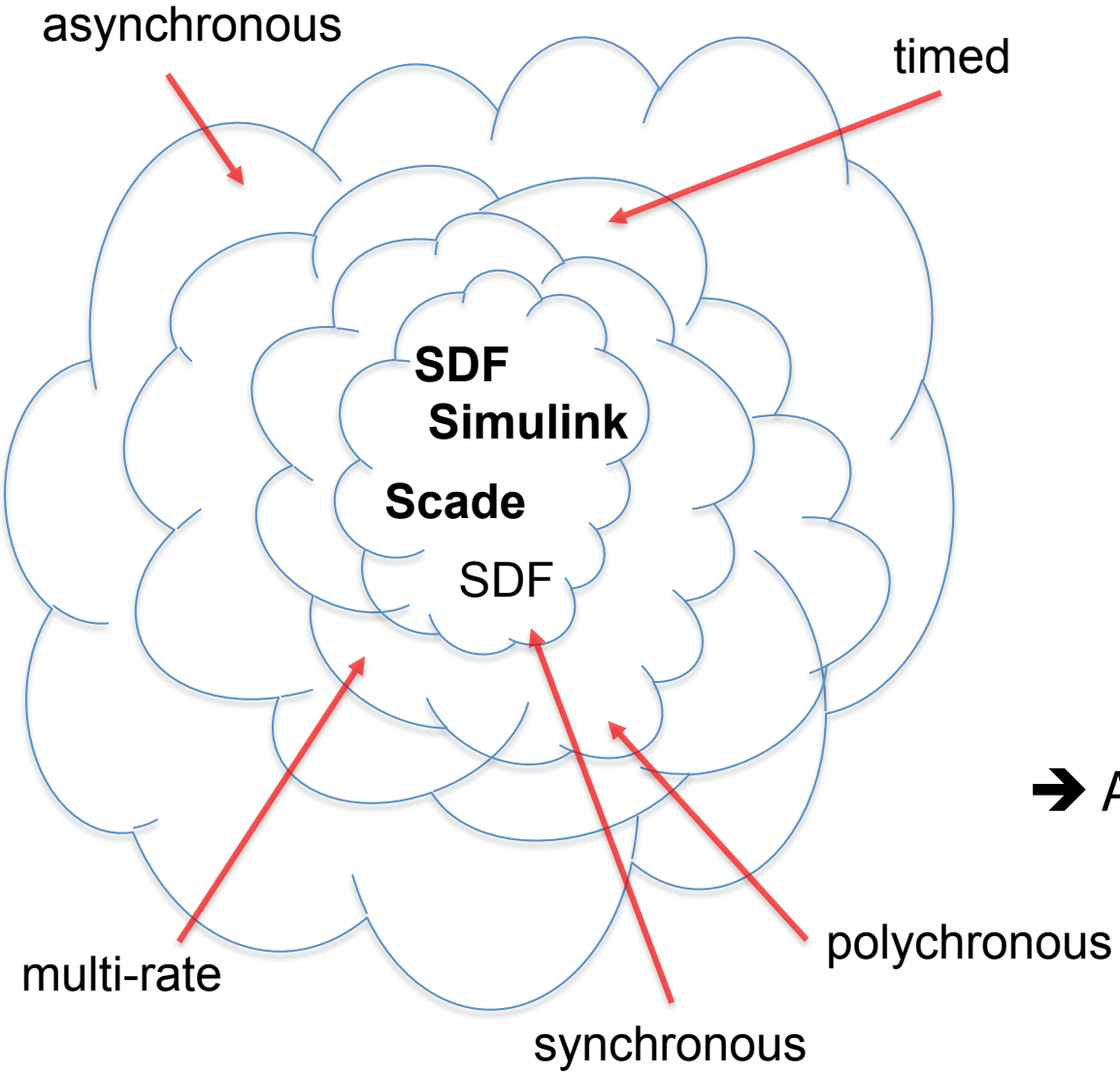
The standard is unambiguous in the first place!

Formal definitions of the standard's constituents: purpose, structure, meaning, relations, extent, and uses ?

➔ Its semantics has to reflect its present and intended uses

Rationale

How to define the semantics of an existing standard, AADL ?



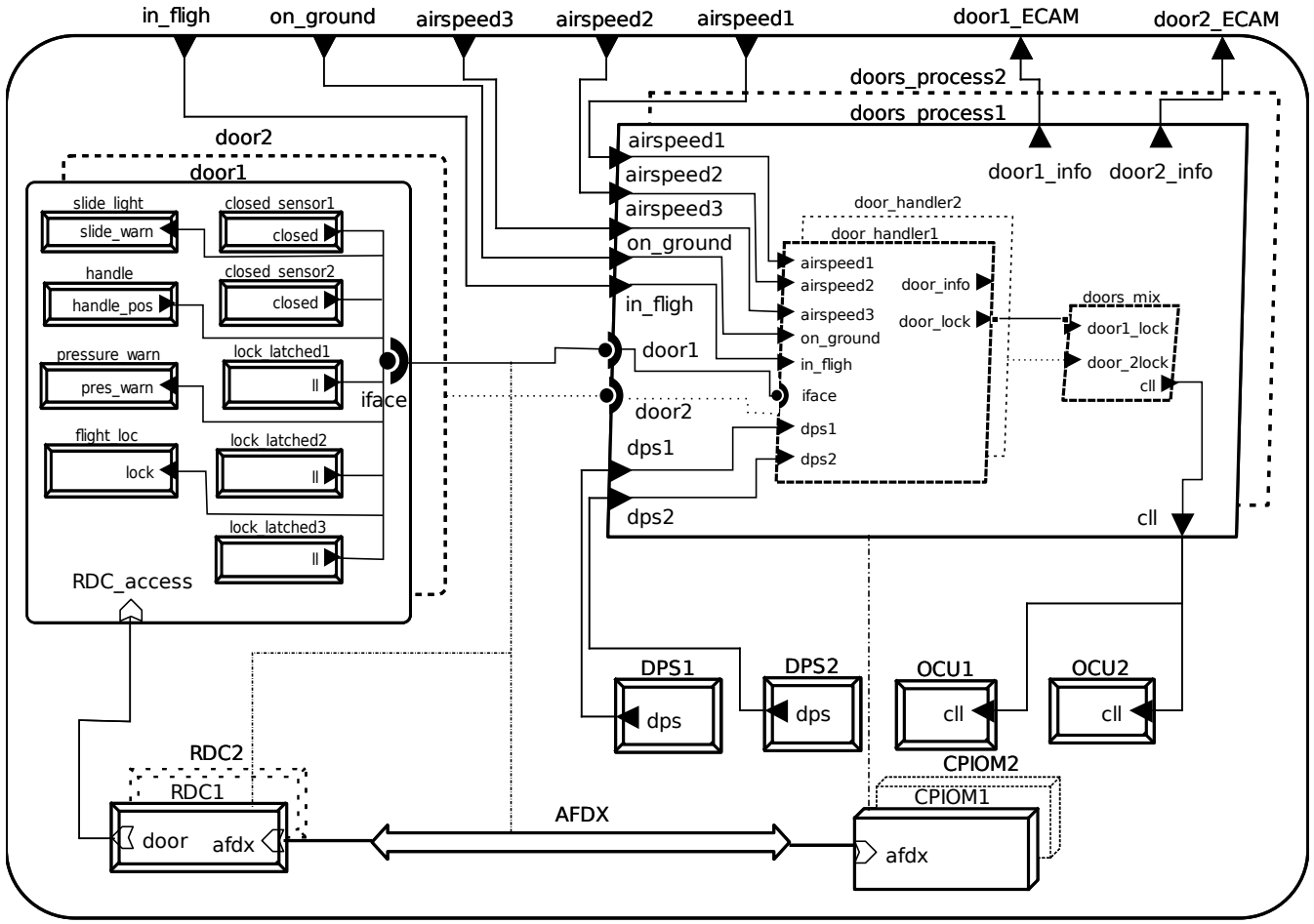
The standard is unambiguous in the first place!

Formal definitions of the standard's constituents: purpose, structure, meaning, relations, extent, and uses ?

➔ A family of related semantics

Theorem 1.

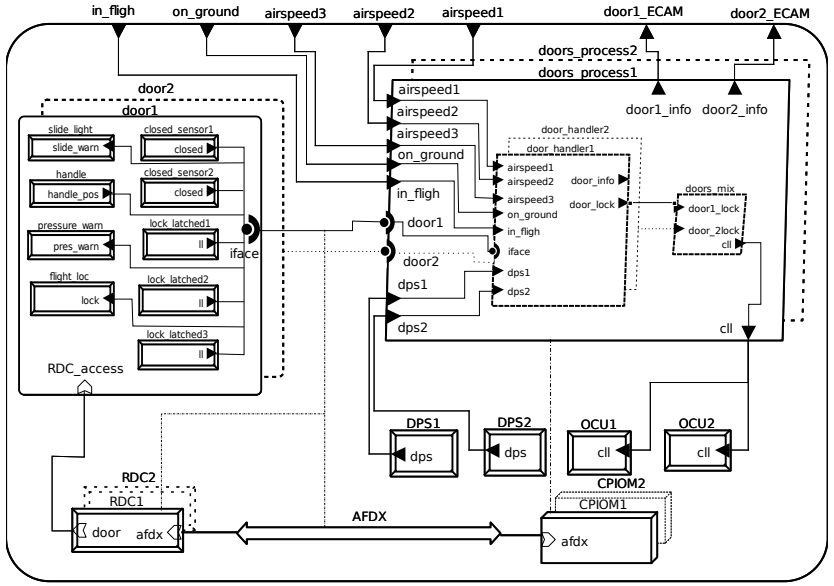
AADL is a formal specification language



Proof →

Systems

```
system door_management
features
  door1_info_FWS: out data port behavior::boolean;
  …/…
  door2_info_ECAM: out data port behavior::boolean;
end door_management ;

system implementation door_management.imp
subcomponents
      door1 : device door.imp {SEI::Priority => 10;};
      door2 : device door.imp {SEI::Priority => 11;};
      lgs : device LGS.imp ;
      …/…
      process_door : process doors_process.imp ;
connections
      conn1: data port door1.handle -> process_door.handle1;
      …/…
      conn14: data port process_door.door2_info -> door2_info_ECAM ;
end door_management.imp ;
```



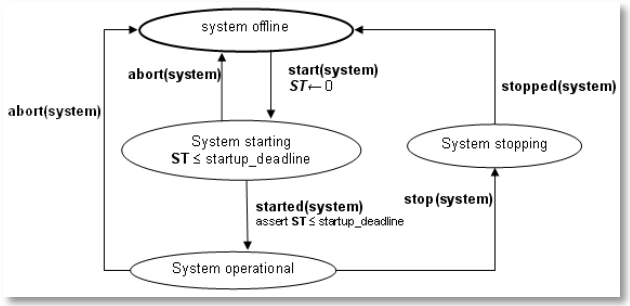


Figure 22 System Instance States, Transitions, and Actions

- Blocks with modes
- Ports and connections
- Logical properties
- States, transitions …

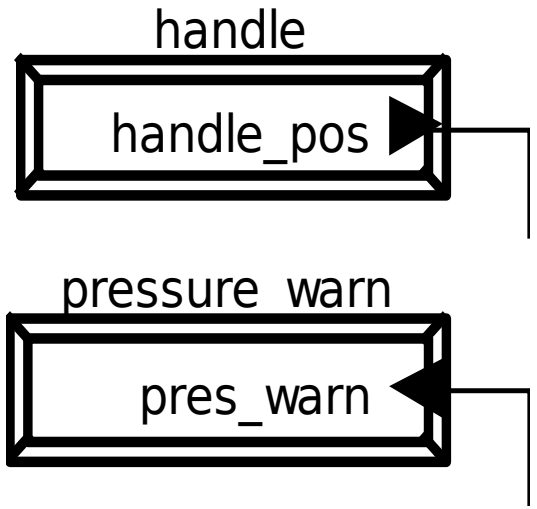
Devices

```
device Handle
  features
    handle: out data port behavior::boolean;
  properties
    Device_Dispatch_Protocol => Periodic;
    Period => 50 Ms;
end Handle;

device PressureWarn
  features
    pressure_warn1: in data port behavior::boolean;
    pressure_warn2: in data port behavior::boolean;
  properties
    Device_Dispatch_Protocol => Periodic;
    Period => 50 Ms;
end PressureWarn;

…/…
```

pressureWarn every 50ms

every pressure_warn1

every pressure_warn2

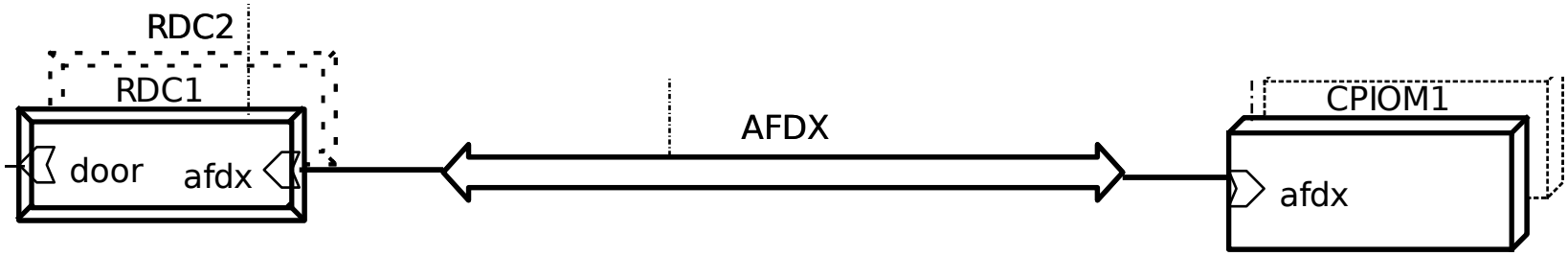


- Blocks and connections
- Logical properties

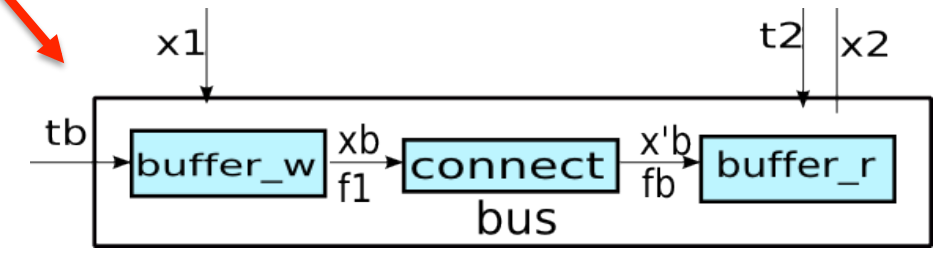
…/…

- Protocols (periodic, aperiodic, …)
- Timed properties and constraints (inherited from connections)

Buses and processors



- Blocks and connections
- Logical and timed properties
- Logical and timed constraints
- …/…
- User manuals, APIs, specifications, requirements
- Behavioral semantic: protocols, modes, transitions



Connections and ports

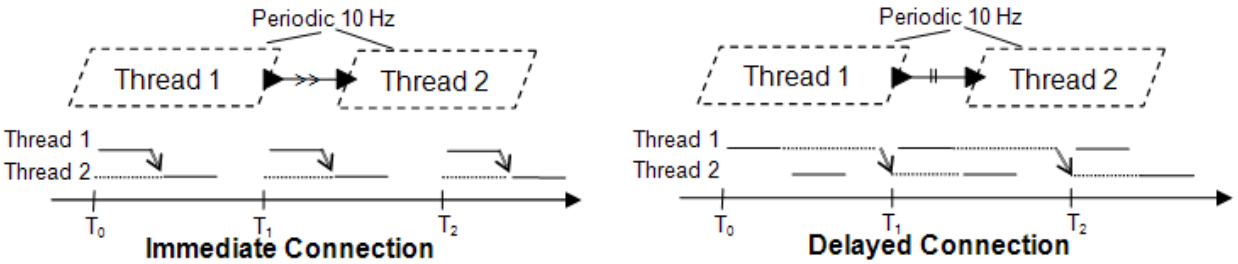
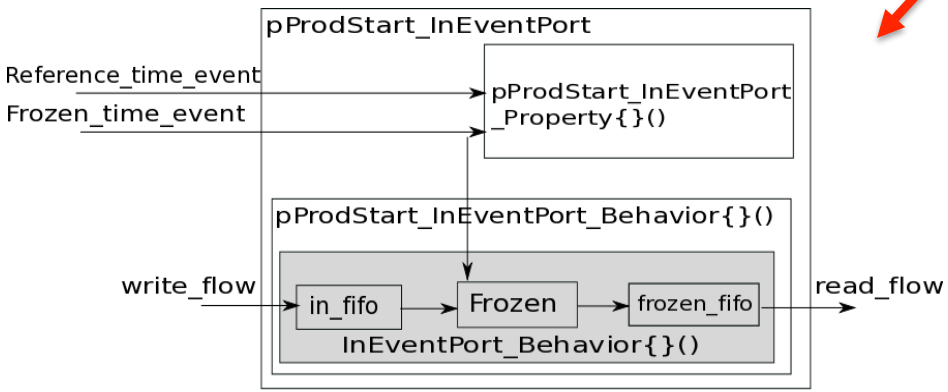


Figure 17 Timing of Immediate & Delayed Data Connections

```
system implementation door_management.imp
subcomponents
      …/…
connections
   conn1: data port door1.handle ->
      process_door.handle1;
   conn2: data port door1.locked ->
      process_door.locked1;
 ’
```

(27) The following property values for `Output_Time` are supported to specify the output time to be the dispatch time (`Dispatch`), any time during execution relative to the amount of execution time from the start (`Start`) or from the completion (`Completion`) including at completion time, the deadline (`Deadline`), and the fact that no input occurs (`NoIO`):

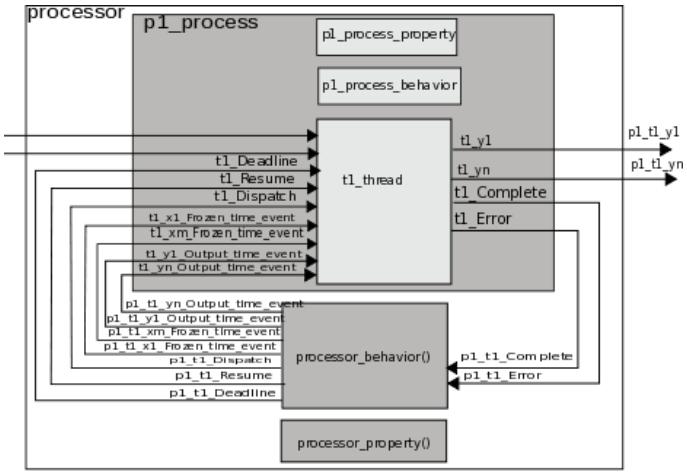
- `Start`, time range: output is transmitted at a specified amount of execution time relative to the beginning of execution. The time is within the specified time range. The time range must have positive values. $Start_{low} \leq c \leq Start_{high}$.
- `Completion` time range: output is transmitted at a specified amount of execution time relative to execution completion. The time is within the specified time range. A negative time range indicates execution time before completion. $c_{complete} + Completion_{low} \leq c \leq c_{complete} + Completion_{high}$, where $c_{complete}$ represents the value of c at completion time. The default is completion time with a time range of zero, i.e., it occurs at $c = c_{complete}$.
- `Deadline`: output is transmitted at deadline time; the time reference is clock time rather than execution time. $t = Deadline$. This allows for static alignment of output time of one thread with the `Dispatch_Time` input time of another thread with a `Dispatch_Offset`.
- `NoIO`: output is not transmitted. In other words, the port is excluded from transmitting new output from the source text. This allows users to specify that a subset of ports to provide output. The property value can be mode specific, i.e., a port can be excluded in one mode and included in another mode.

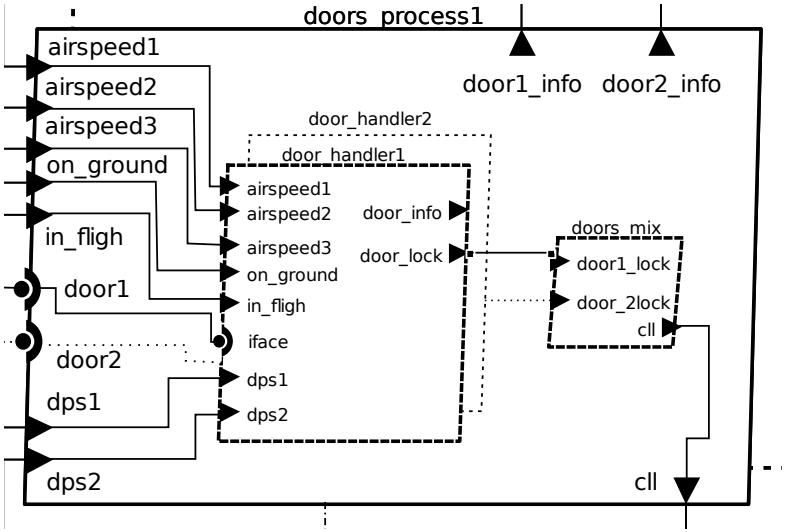


- Connections
- Logical and timed properties and constraints
- Behavioral semantic: protocols, modes, transitions

Processes and threads

```
process implementation doors_process.imp
  subcomponents
    door_handler1: thread door_handler.imp;
    door_handler2: thread door_handler.imp;
    door_mix: thread doors_mix.imp;
  connections
    conn1: data port engine_running ->
           door_handler1.engine_running;
   …/…
    conn45: data port door_handler2.cll -> cll2;
end doors_process.imp;
```





- Blocks, connections, ports
- Logical and timed properties and constraints (as behaviors abstractions)
- …/…

Threads and scheduling

```
thread door_handler
  features
    handle_pos: in data port behavior::boolean;
   …/…
    pres_warn: out data port behavior::boolean;
end door_handler;

thread implementation door_handler.imp
  properties
    Dispatch_Protocol => Periodic;
    Period => 50 Ms;
end door_handler.imp;
```

- Blocks and connections
- Logical and timed properties and constraints (as behaviors abstractions)
- …/…
- Modes, states, transitions

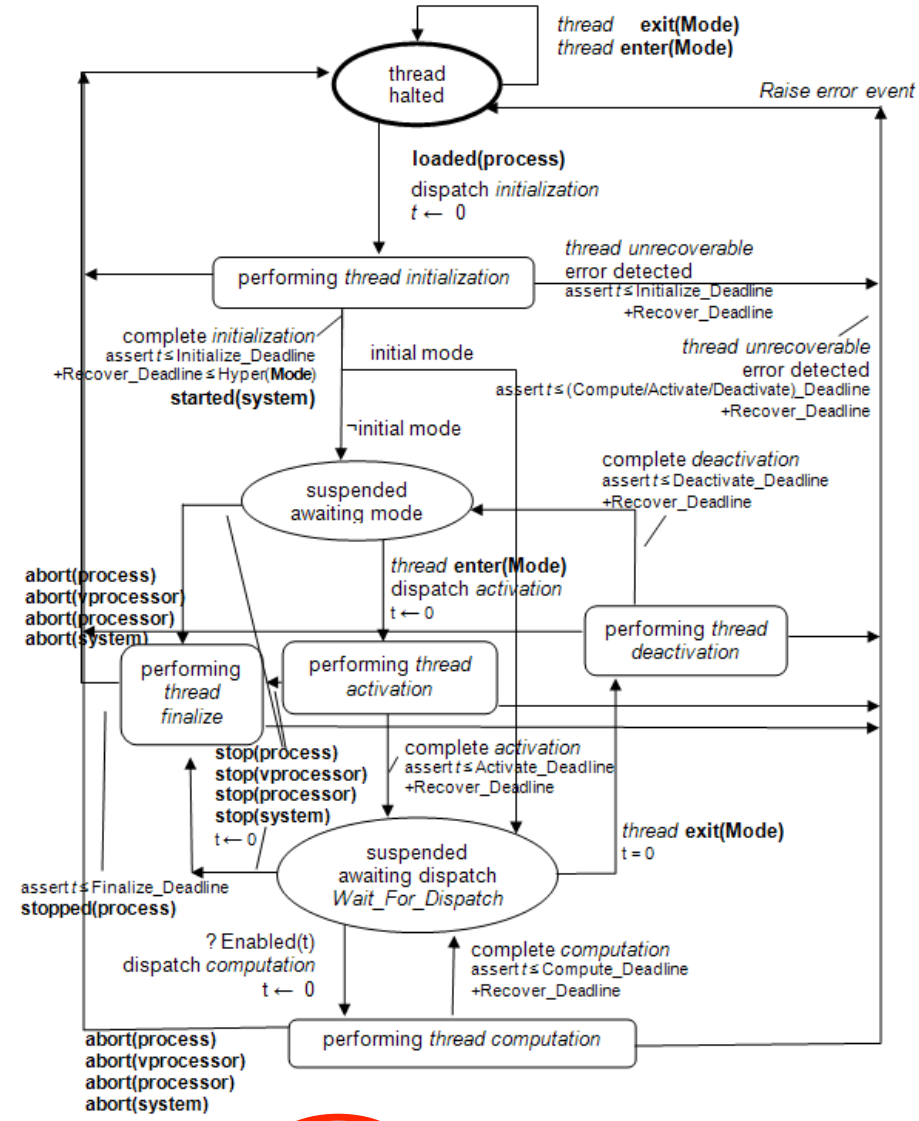
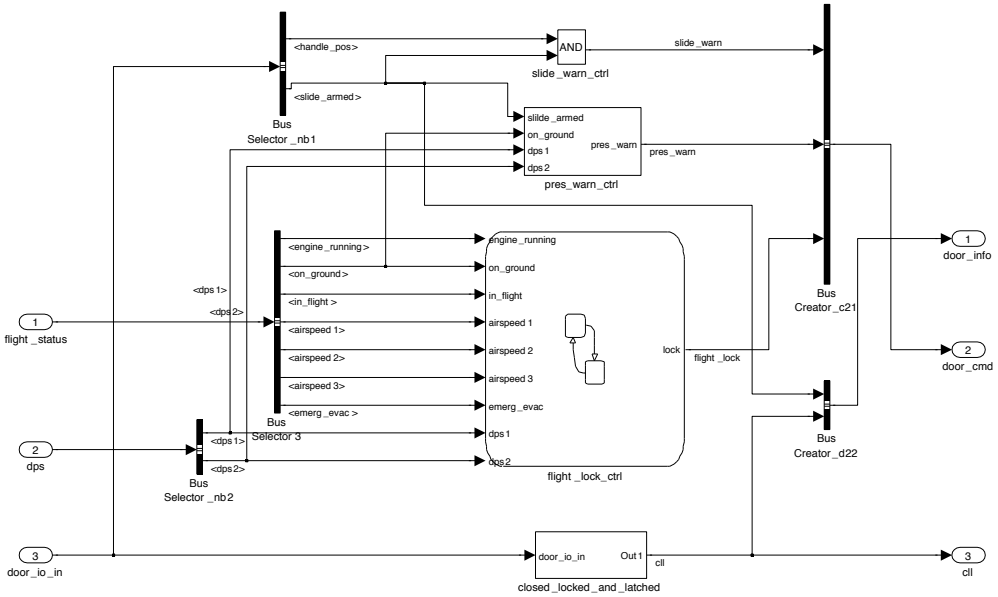


Figure 5 Thread States and Actions

Threads and behaviors

```
thread implementation door_handler.imp
annex behavior_specification {**
  states
   s0: initial complete state;
  transitions
   s0 -[on (dps > 3) and handle]-> s0 {
      computation(1ms);
      warn_diff_pres:=true;
      door_info:=locked;
      door_locked:=locked;
      };
   s0 -[on not (dps > 3 and handle)]-> s0 {
      computation(1ms);
      door_info:=locked;
      door_locked:=locked;
      };
  **};
end door_handler.imp;
```



- Untimed automata and action blocks
- Abstractions of external Simulink, Stateflow, Scade diagrams …

Rationale
Formal semantics for the AADL

- First-order formulas
  For logical, timed, quantitative properties and constraints
- Constrained, small-step, automata with variables
  For modes, automata and action blocks
- Data-flow networks
  For everything else
- Model-theoretic semantic interpretations (viewpoints)
  - Untimed, asynchronous, small-step semantics
  - Timed semantics
  - Synchronous big-step semantics
- Trace relations between semantic view-points

First-order formulas

F_A set of well-typed first-order formulas f defined on the vocabulary in scope of object A

- Ports P, variables V, states S
- Formulas over Booleans

  $true_A$, *false, present(p),* ^*p, v, v', not f,* f_1 *and* f_2, f_1 *or* f_2, …

- Formulas over integers m, n

  0, 1, …, time(p), @p, $m + n$, $m \leq n$, …

First-order formulas

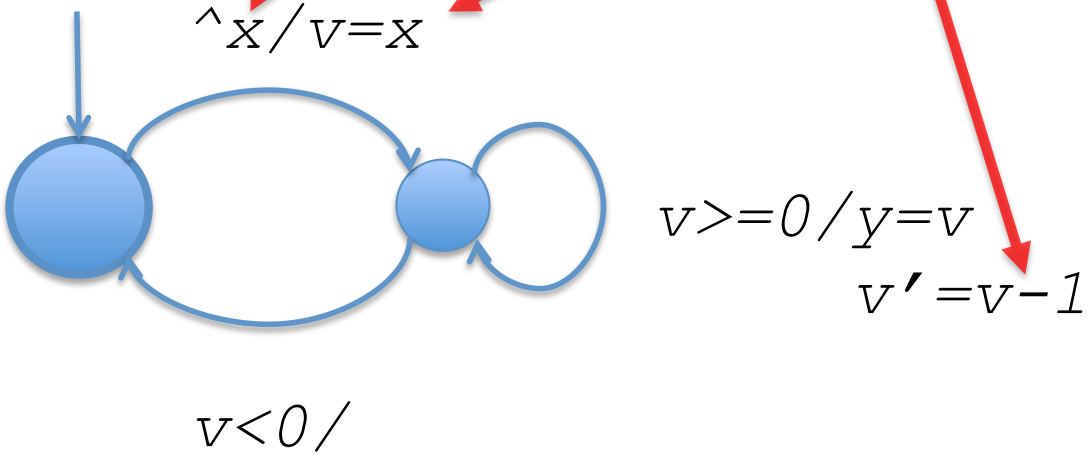
- Variables are valuated on a value domain D
- Ports are valuated on $D^{\perp} = D \cup \{\perp\}$ (absence)
  - $true_A$ true with respect to A
  - $not_A(f)$ false with respect to A
  - $\hat{}p$ event p is present
  - $@p$ time of event p

Examples of formula

```
variables v: integer;
states s1: complete state, s2: state;
transitions
     s1 -[on dispatch x]-> s2 {x?v}
     s2 -[v < 0]-> s1
     s2 -[v >=0]-> s2 {y!v; v=v-1}
```

Examples of formula

```
variables v: integer;
states s1: complete state, s2: state;
transitions
      s1 -[on dispatch x]-> s2 {x?v}
      s2 -[v < 0]-> s1
      s2 -[v >=0]-> s2 {y!v; v=v-1}
```

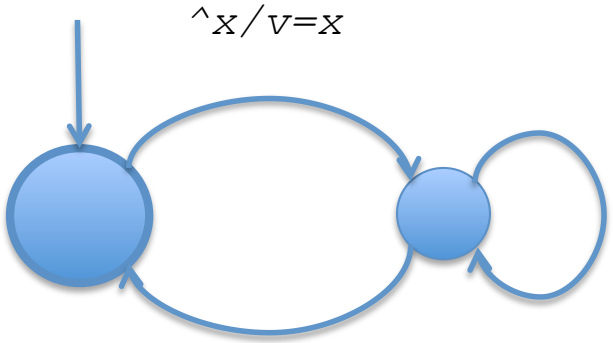


Constrained automata

- Automata $A = (S, s_0, P, V, T, C)$
- States S and initial state s_0
- Formulas defined on $F_A = F_{P+V+S}$
- Transitions T are quadruples (s_1, g, a, s_2)
  - Source s_1 and target s_2
  - Guard formula g and action formula a
- C is the constraint formula
  - must remain false

Example of automaton

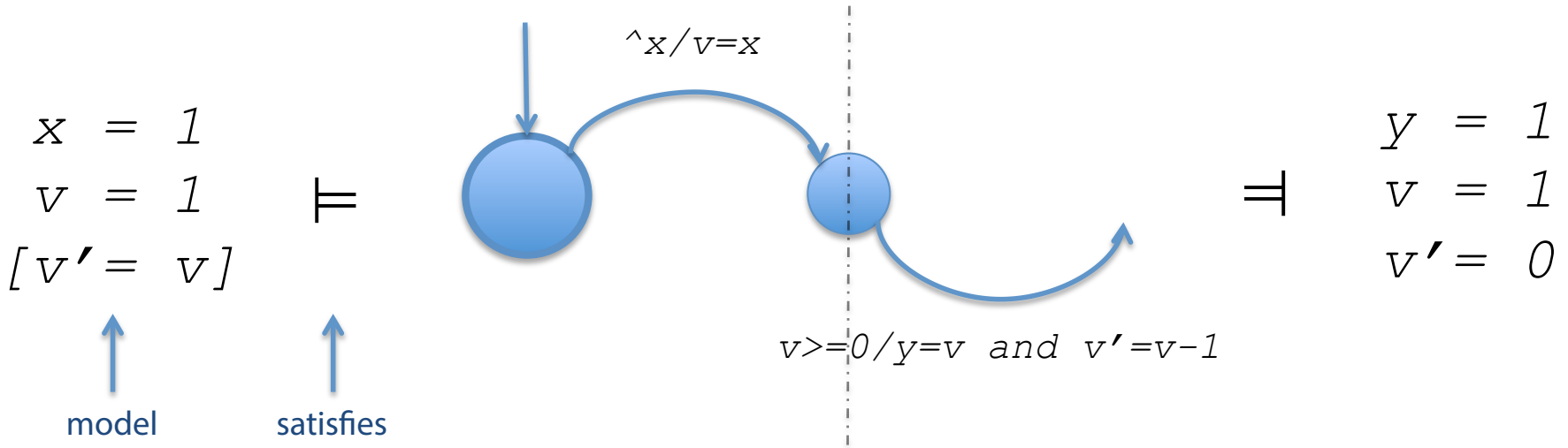
```
variables v: integer;
states s1: complete state, s2: state;
transitions
      s1 -[on dispatch x]-> s2 {x?v}
      s2 -[v < 0]-> s1
      s2 -[v >=0]-> s2 {y!v; v=v-1}
```



(s1, ^x, v=x, s2)
(s2, v<0, true, s1)
(s2, v>=0, v'=v-1 and y=v, s2)

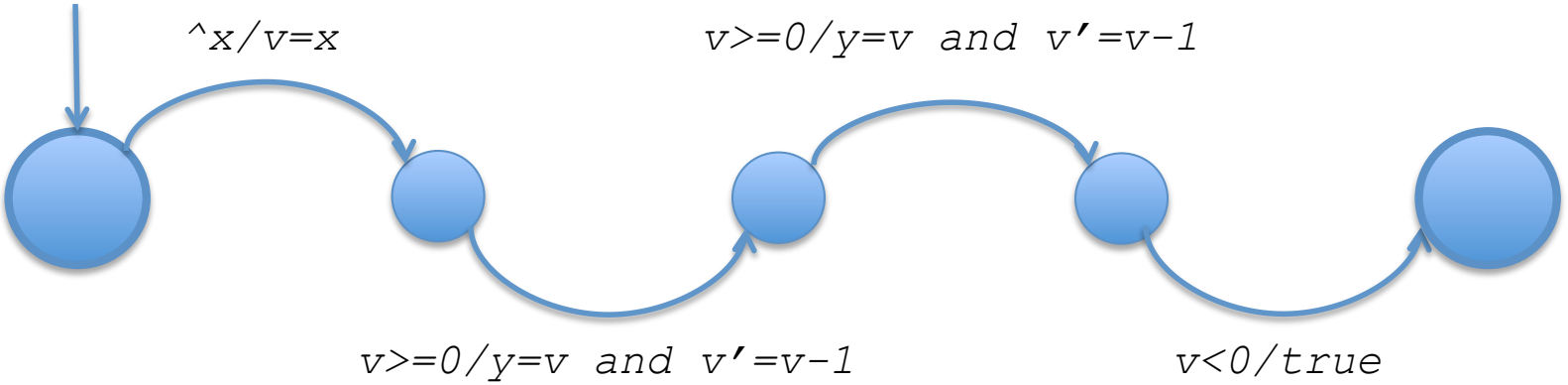
Models of behaviors

A model M is a valuation of variables and ports on the value domains D and $D^{\perp}$



Untimed trace

Small steps of the automaton



$T^{\#} =$

$x = 1$
$v = 1$

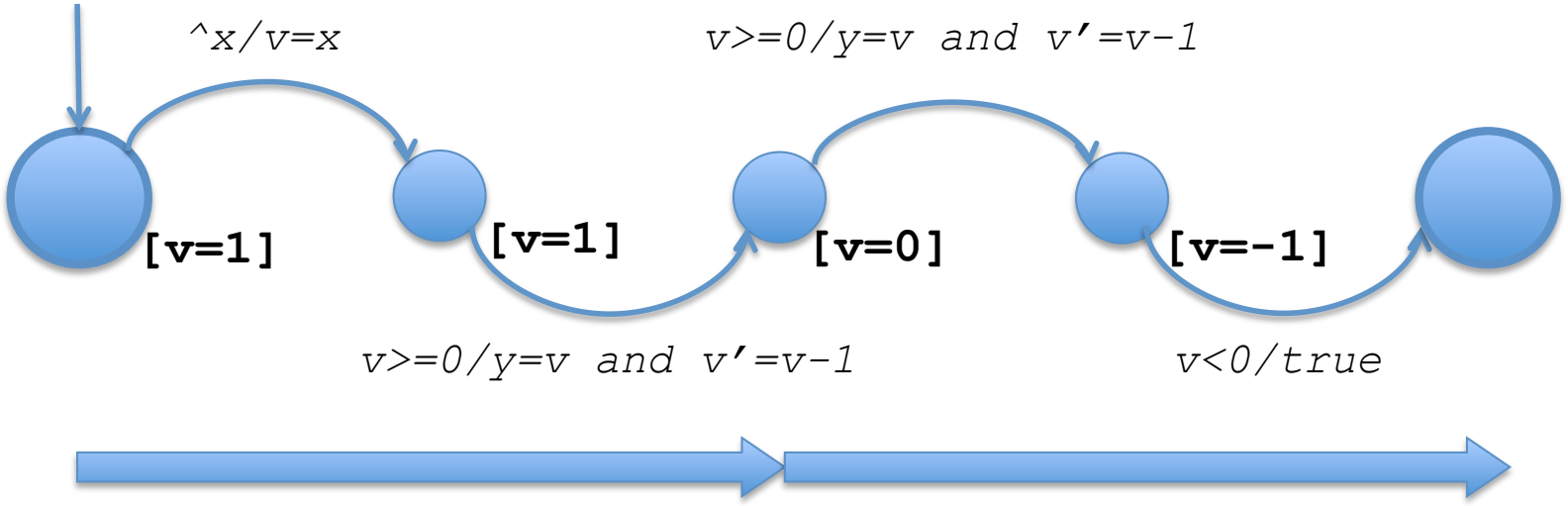
$v = 1$
$v' = 0$
$y = 1$

$v = 0$
$v' = -1$
$y = 0$

$v = -1$

Logically timed traces

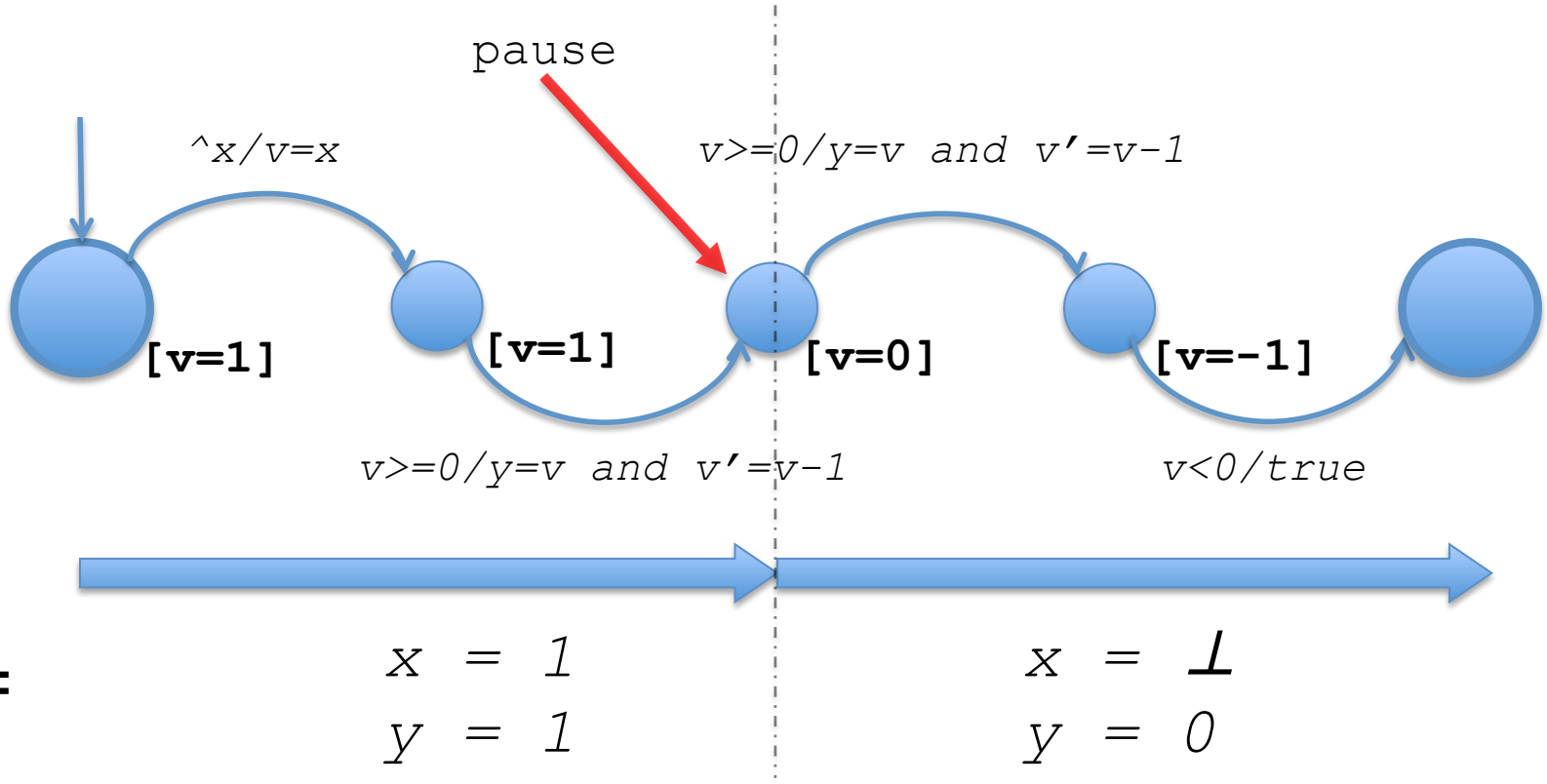
Big steps of the extended automaton until a complete state is reached of the model invalidated*…



T = $x = 1$, $y = 1$; $x = \perp$, $y = 0$

Logically timed traces

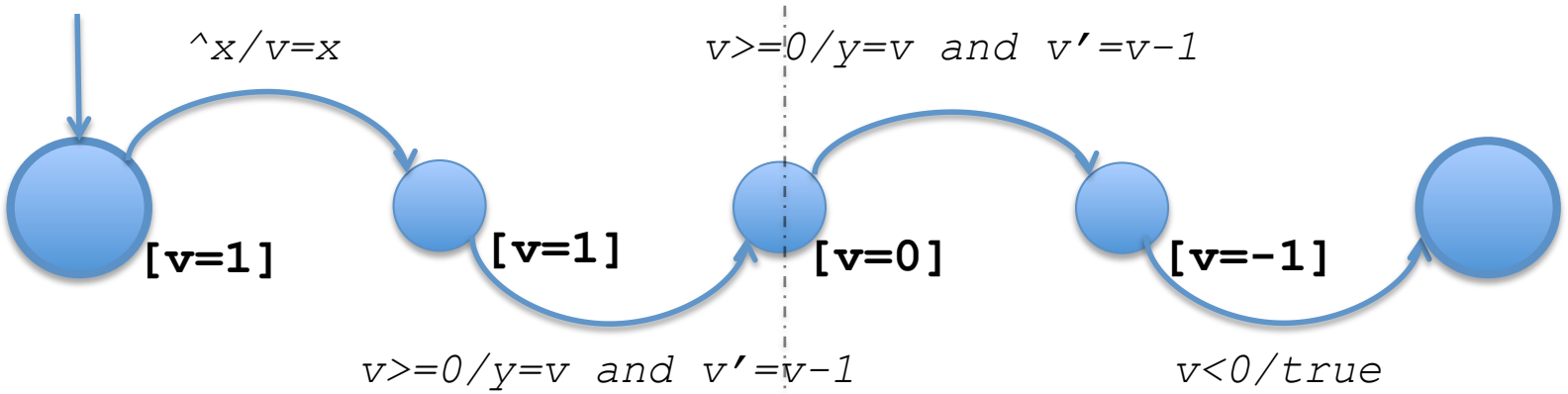
*… or a `pause`$^{(new)}$ state tells where to stop



T = $x = 1$, $y = 1$; $x = \perp$, $y = 0$

Big-step, real-timed traces

A tagged trace T of the extended automaton of period p



T = $x_p = 1$ $Y_{2p} = 1$ $Y_{3p} = 0$

Trace relations

Synchrony and asynchrony

$(T)^{\#} =$ $\quad x = 1 \quad y = 1$ $\quad$ ~~$x = -1 \quad v = 0 \quad v' = -1$~~ $\quad y = 0$ $\quad \leq T^{\#}$

Synchrony and real-time

$(T)^{@} =$ $\quad x_{t1} = 1 \quad 0 \leq t1 < p$
$y_{t2} = 1 \quad p \leq t2 < 2p$
$y_{t3} = 0 \quad 2p \leq t3 < 3p$ $\quad \leq T^{@}$

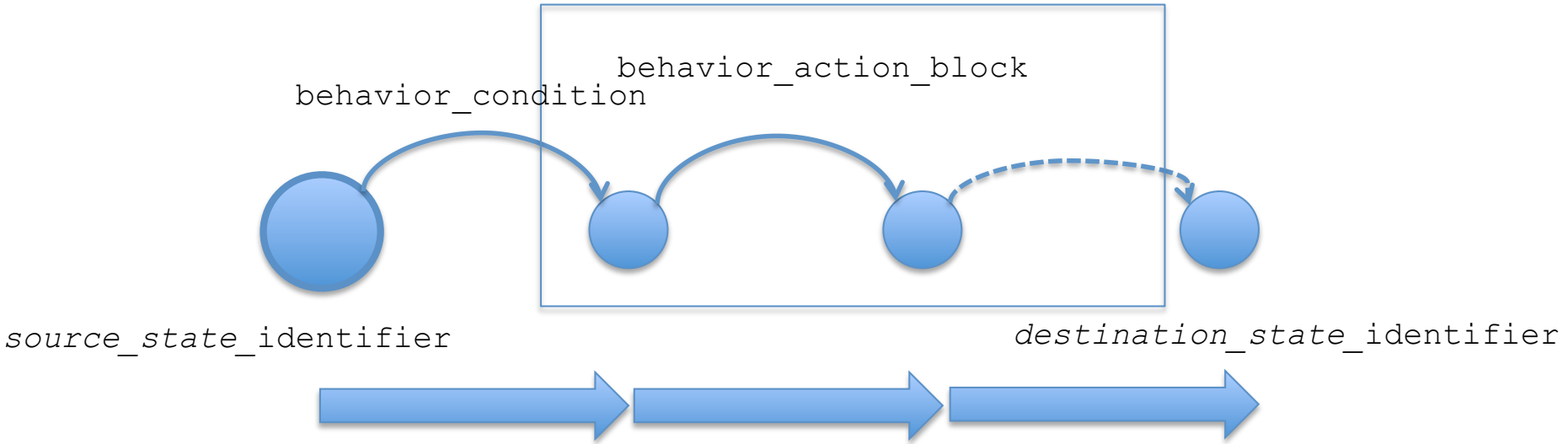
Composition, abstraction, refinement, …

Extension to action blocks

The imperative language to describe behaviors in AADL

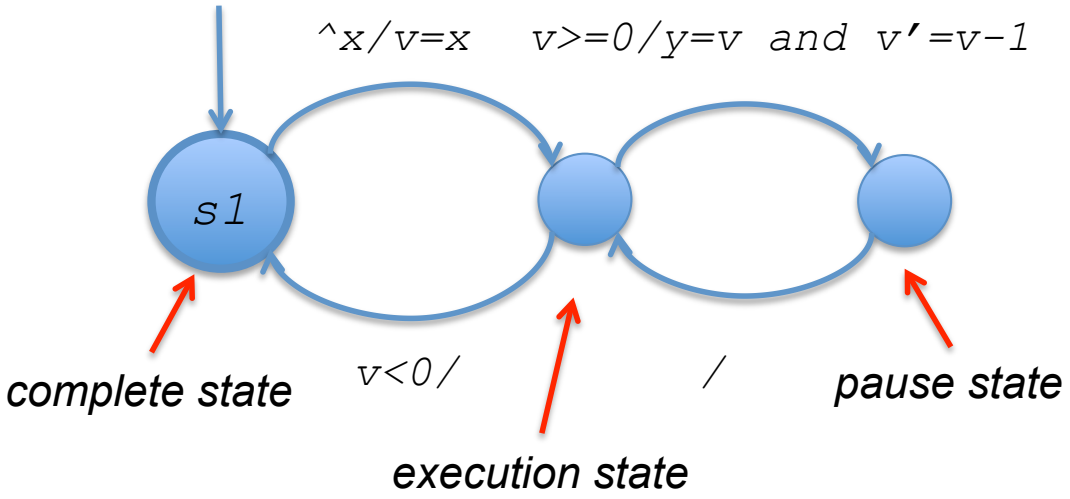
`send, receive, assign, loop while, until, for, call, …`

Action blocks are attached to source states of transitions



Example of action block

```
transitions
      s1 -[on dispatch x]-> s1 {
         x?v;
         while v >=0 {
            y!v;
            v=v-1;
            pause;
      }}
```

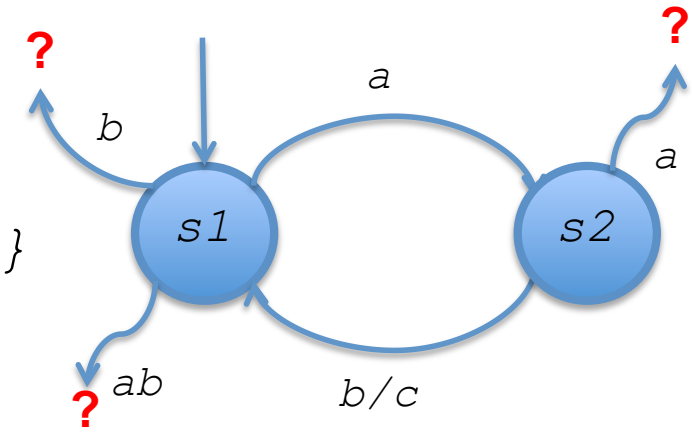


Constraints for specification refinement

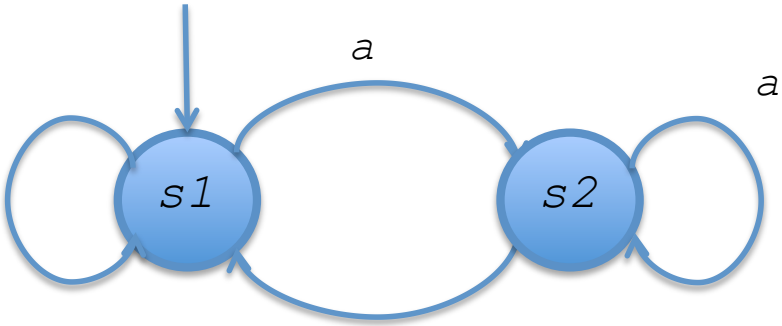
Specifications are, most of the time, incomplete

```
s1 -[on dispatch a]-> s2
s2 -[on dispatch b]-> s1 {c!}
```

`aaaaabcabbabaaaabc` ?



Constraints (logical and/or timed) help refine/complete them



`b and not a`

`a and not b`

`never a and b`

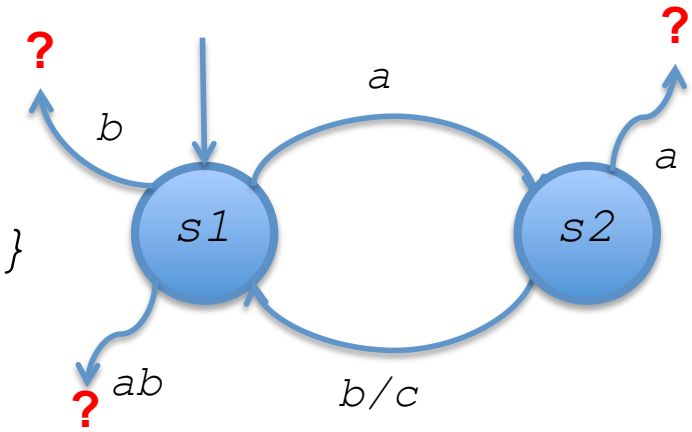
`aaaaaabcbbbbbbbaaaaaabcaaa`

Regexps for specification refinement

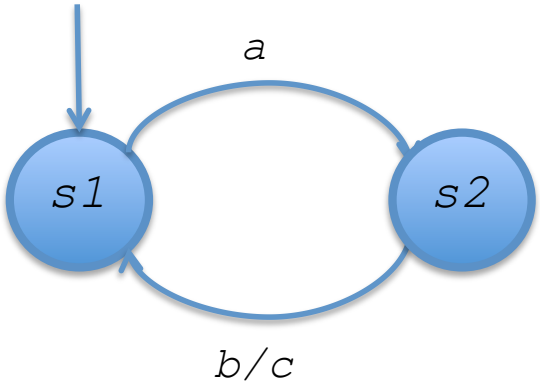
Specifications are, most of the time, incomplete

```
s1 -[on dispatch a]-> s2
s2 -[on dispatch b]-> s1 {c!}
```

`aaaaabcabbabaaaabc` ?



Regular expressions help refine more (controller synthesis)




```
always a;(b and c)
```

`abcabcabc`

All action blocks translate to automata

if *g* **then** actions$_1$ **else** actions$_2$ **end if** $T_1 \cup T_2$ with $T_1 = Tg,s,s'$ [actions$_1$] and $T_2 = T\neg g,s,s'$ [actions$_2$]

while *g* **{** actions **}** $T_1 \cup T_2$ with $T_1 = Tg,s,s$ [actions] and $T_2 = (s,\neg g,true,s')$

.../...

do actions **until (***g***)**

for (identifier **in** values**) { **actions **}**

forall (identifier **in** values**) { **actions **}**

Tg,s,s'[target **:=** value] = (s, g, target'=value, s')

Tg,s,s'[*port***!**value] = (s, g, *port*=value, s')

Tg,s,s'[*port***?**target] = (s, g, *target'*=value, s')

.../...

This is being specified in the standard

Formal Semantics

(31) In a behavior specification, a transition denotes the consumption of time sensed by a component from the time the enclosing thread is dispatched to the time it releases control. This perception of time can take the form of a time trigger, of a port dispatch, or a port output. Complete states temporally delimit the interaction of a behavior with its parent, static or dynamic, environment.

(32) **Definition (Values and Identifiers).** Dispatch conditions and behavior actions are defined by formulas defined on disjoint sets of states S, behavior variables V, and ports P. These identifiers are valuated on the domain $\mathbb{D} = \mathbb{B} + \mathbb{Z} + \mathbb{R}$ of Boolean, integer and real numbers. We write $\mathbb{D}_x$ for the value domain of a typed identifier x. Port identifiers are valuated and timed on $\mathbb{D}^{\perp} = \mathbb{D} \cup \{\perp\}$. The bottom sign $\perp$ represents the absence of a value at a given time sample.

(33) The addition of $\perp$, to mean absence, implies a timing relation between the port that carries it and the status and value of other ports at the given time sample of their evaluation. A port value can be absent if its value is unused, unneeded, irrelevant or unavailable at a given time. We write ^p for the clock or period of a port p, i.e., the condition that p carries a value: true for present, false for absent. We write v' for the next value of a variable v, i.e., the value that v will carry at the next time sample.

(34) **Definition (Logical Formula).** The set of Boolean formulas F_{SVP} on states S, behavior variables V, and ports P over the domain $\mathbb{D}^{\perp}$ is defined by induction from:

1. Constants **0**, to mean never, **1**$_{SVP}$, to mean always in the scope of S, V and P (e.g. a thread)
2. Atoms s, a state, p, a port value, ^p, a port clock, v, a Boolean variable value, v', its next value, and d a Boolean constant.
3. Unary expressions **not** f, for all f in F_{SVP}
4. Binary expressions f = g, f **and** g, f **or** g, for all f, g in F_{SVP}

For instance, the formula ^a and ^b = 0 stipulates that port a and port b should never occur (be dispatched and/or written) at the same time. Ill-formed formula (e.g. of non-Boolean result or wrong arity) are always false.

(35) **Definition (Automaton).** A behavior annex is equivalent to incomplete and synchronous automaton[1] with variables[2] $A = (S_A, s_0, V_A, P_A, T_A, C_A)$ defined by

1. S_A, the set of complete and execution states of the behavior annex, s_0 is the initial state.
2. V_A, the set of local variables of the behavior annex
3. P_A, the set of ports of the behavior annex, partitioned into inputs I_A and outputs O_A.
4. F_A is the set of Boolean formulas of A, defined on S_A, V_A, and P_A
5. The transition function $T_A \in S_A \times F_A \rightarrow F_A \times S_A$ defines a transition system where
   a. The source formula is the guard, defined on V_A and I_A
   b. The target formula is the update, defined from V_A and P_A
6. The control formula $C_A \in F_A$ (or constraint) must equal **0**. It defines the invariants (requirements) of the behavior pertaining to its timing, synchronization, and causal properties, in a form of a logic formula.

By construction, a transition (s, g, f, d) $\in T_A$ corresponds to the set of all transitions of its equivalent form $X_A = Q_A \times F_P \rightarrow F_P \times Q_A$ on extended states $Q_A = S_A \times \mathbb{D}^{V_A}$ with valuations $\mathbb{D}^{V_A} = \prod_{v \in V_A} \mathbb{D}_v$ for all persistent state variables V_A:

[1] L. BESNARD , A. BOUAKAZ , T. GAUTIER , P. LE GUERNIC , Y. MA , J.-P. TALPIN , H. YU . Timed behavioural modelling and affine scheduling of embedded software architectures in the AADL using Polychrony , in "Science of Computer Programming", July 2014. https://hal.inria.fr/hal-01095010

[2] M. Sköldstam, K. Akesson and M. Fabian. "Modeling of Discrete Event Systems using Finite Automata With Variables". 46th Conference on Decision and Control. IEEE, 2007.

Formal Semantics

(19) The action block attached to a transition in a behavior annex is represented by a transition system whose source state is that of the transition and whose final state the transition's target state. The transition system of an action block can either decompose the action sequence into micro steps.

(20) A `dispatch_condition` is represented by a guarding formula g that is formed by referring to the clock ^p of the logical combination of ports specified as its `dispatch_trigger_condition` or its `frozen_ports`. Timeout and stop conditions are equally regarded as occurrences of port events.

~~(21)~~ An `execute_condition` is represented by a guarding formula that encodes its `logical_value_expression` using the current state of its persistent variables V. ~~The `otherwise` clause is handled as the guard of least priority (D.3, (45)).~~

(22) The recursive function $\mathcal{T}(g,s,d)$[behavior_actions] = T associates the action block `behavior_actions` guarded by a `behavior_condition` of formula g, of source and target states s and d, to a transition system T. It is performed by case analysis on `behavior_actions`.

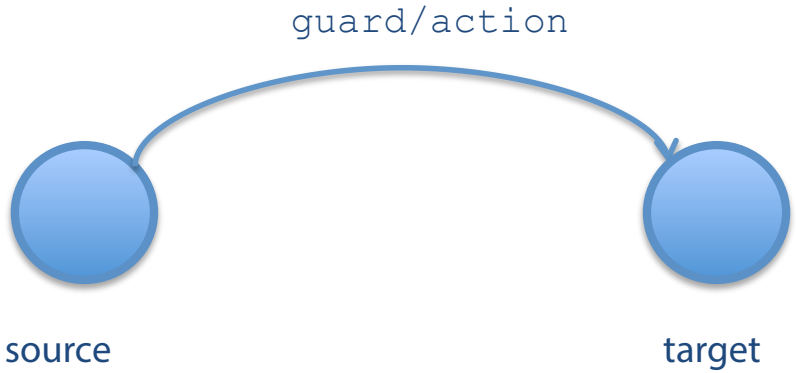
- A `behavior_action_sequence` is represented by concatenating the transition systems of its elements. For instance, $\mathcal{T}(g,s,d)$[action$_1$; action$_2$] is translated by the union $T_1 \cup T_2$ of its transition systems $T_1 = \mathcal{T}(g,s,e)$[action$_1$] and $T_2 = \mathcal{T}(true,e,d)$[action$_2$], by introducing a new execution state e.
- A `behavior_action_set` is represented by composing the transition systems of its elements. For instance, $\mathcal{T}(g,s,s')$[action$_1$ & action$_2$] is translated by the synchronous composition $(T_1 \mid T_2)$[(s, s)/s, (d, d)/d] of its transition systems $T_1 = \mathcal{T}(g,s,d)$[action$_1$] and $T_2 = \mathcal{T}(g,s,d)$[action$_2$], substituting the composed states (s, s) and (d, d) by s and d.
- A `behavior_action` is translated by case analysis of its form
  - **if** (*logical*_expression) actions$_1$ **else** actions$_2$ **end if** is translated by representing the *logical*_expression using a guard formula g and returning the union $T_1 \cup T_2$ of its transition systems $T_1 = \mathcal{T}(g,s,d)$[actions$_1$] and $T_2 = \mathcal{T}(\neg g,s,d)$[actions$_2$].
  - **while** (*logical*_expression) { actions } is translated by the union $T_1 \cup T_2 \cup T_3$ of its transition systems $T_1 = \mathcal{T}(g \wedge h,s,d)$[actions], $T_2 = \{(d,h,true,s)$ and and $T_3 = \{(s,\neg h,true,d)$ where the guard formula h is the translation of the *logical*_expression.
  - **do** actions **until** (*logical*_expression) is translated by the union $T_1 \cup T_2$ of its transition systems $T_1 = \mathcal{T}(g,s,d)$[actions] and $T_2 = \{(d,h,true,s)$ where the guard formula h is the translation of the *logical*_expression.
  - **for** (identifier **in** values) { actions } can be translated by a while loop or by the sequence actions$_1$;… action$_n$, where action$_i$ results from the substitution of `identifier` by the i[th] element of `values` in actions. *An **invariant** logical_expression can be specified in the scope of identifiers that must hold for all iterations of the loop.*
  - **forall** (identifier **in** values) { actions } can be translated by a while loop or by the composition actions$_1$&… action$_n$, where action$_i$ results from the substitution of `identifier` by the i[th] element of the `values` in actions.
- A `basic_action` is translated by case analysis of its grammar's sub-clauses
  - An `assignment_action` to a variable `target := value` is represented by represented by updating `target` with `value` as $\mathcal{T}(g,s,d)$[target := value] = {(s, g, target'=value, d)}.
  - *A simultaneous assignment statement (target{,target}+:=(value){,(value)}+) is equivalent to the composition of assignments target:=value (& target:=value)+*

(AS5506/2)

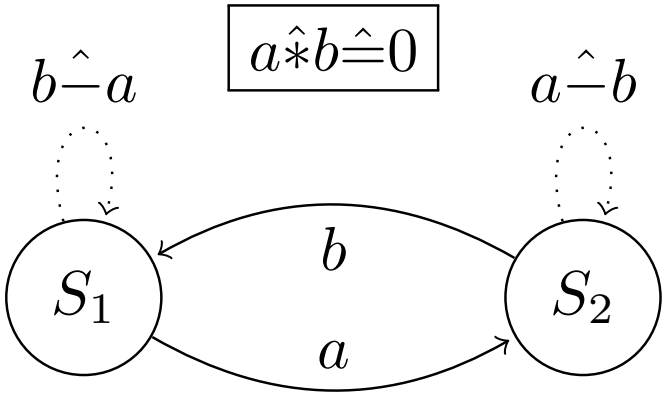
Definition of subsets

- Untimed, asynchronous (Fiacre)
- Timed (Uppaal, Cheddar, …)
- Aperiodic, sporadic: uses pause, needs ⊥
- Multi-rate, periodic (MRS, RADL)
- Synchronous (Lustre, Signal, Esterel, Bless, …)
- Strictly synchronous (Scade, Simulink)

Implementation of constrained automata using synchronous data-flow



- action when guard and source
- target' = true when source and guard … default target
- source' = false when source and guard … default source

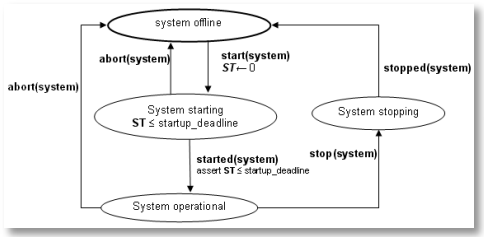


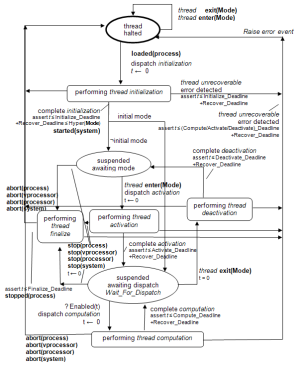

```
  S1 :: P1()
| S2 :: P2()
| Initial_State (S1)
| Transition (S1, S2, a)
| Transition (S2, S1, b)
| Never (a ^* b)
```

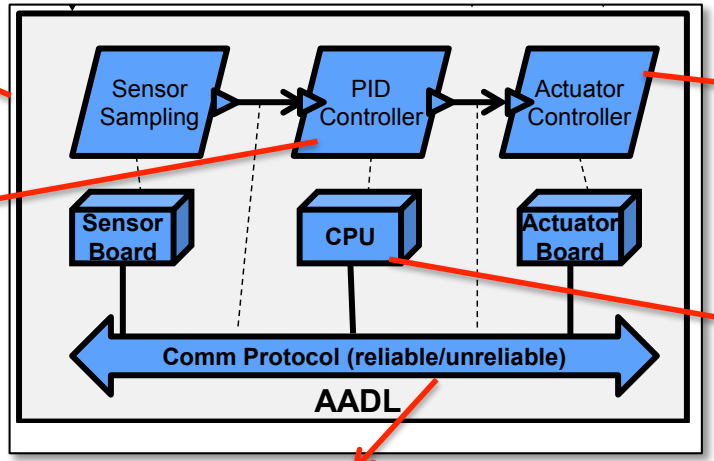
Future works

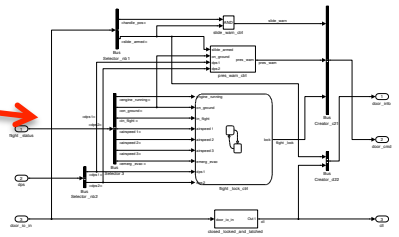
Formal behavioral models of AADL objects

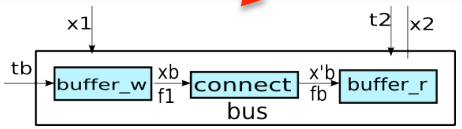
- Ports, threads, CPUs, buses, …
- Use of refinement for simulation: bus ≤ AFDX
- Use of abstractions for verification: FIFO ≤ bus

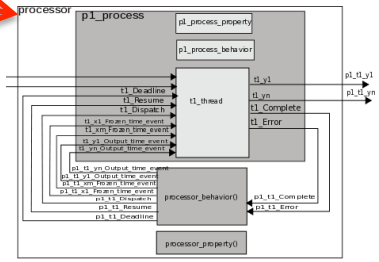






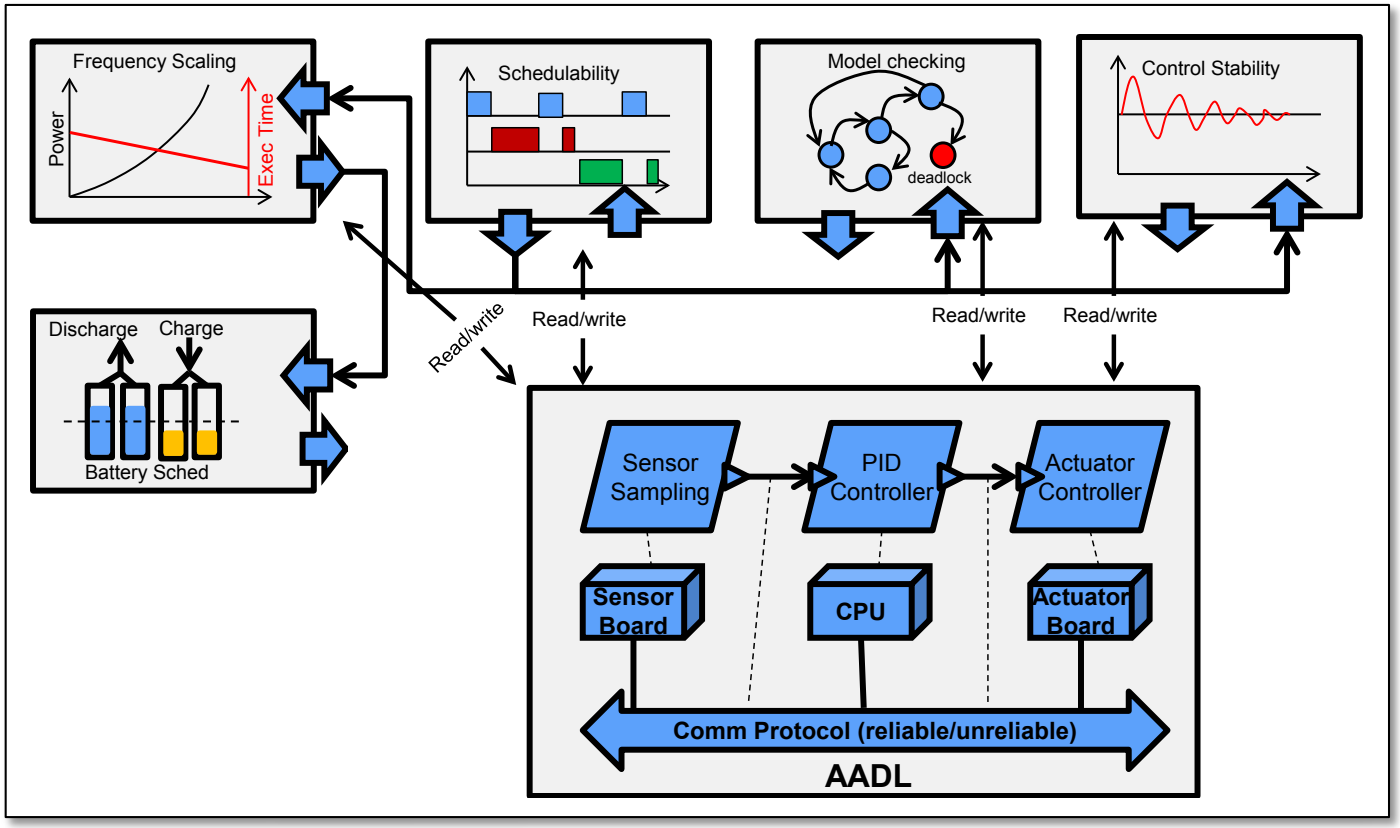






Future works

- Contract-based timed, quantitative, analysis
- SAT/SMT verification, refinement types



(CMU/SEI)

Conclusion

Started from a practical use case with a background on synchronous programming

Evolved as a formal, scalable, adaptive family of semantics for the AADL standard

Thank you

Questions ?