Adaptor synthesis for real-time components

Pascal Fradet (Pascal.Fradet@inrialpes.fr)

Alain Girault (Alain.Girault@inrialpes.fr)

Gregor Goessler (Gregor.Goessler@inrialpes.fr)

Massimo Tivoli (Massimo.Tivoli@inrialpes.fr)

POP-ART project team - INRIA Rhône-Alpes

We believe that…

- Model construction should be part of the development process
  - early identification of problems.
- Model analysis and synthesis
  - increased confidence on the adequacy and validity of the final product;
  - mechanical verification and enforcing (when possible) of properties.
- In particular: design of concurrent systems
  - integration of components can introduce interaction problems that are hard to detect;
  - deadlock, starvation, safety, liveness, etc.…

Component Based Software Engineering (CBSE)

- CBSE focuses on building software systems by integrating previously existing software components [SEI-CMU].
- It embodies the “buy, don’t build” philosophy [Brooks‘87].
- Boosted by:
  - increase in the quality and variety Commercial-Off-The-Shelf (COTS) components;
  - component integration technologies, e.g., COM & CORBA;
  - lower development and maintenance budget.
- It introduces a new approach to system design.

Designing Component Based (CB) systems

- Components are:
  - autonomous, decoupled, concurrent and possibly distributed entities;
  - with well-defined interfaces for communication and synchronization.
- Integration context provides:
  - abstraction framework for integrating components, e.g., network, O.S., data representation, etc…;
  - standard services, e.g., naming, yellow pages, etc…;
  - *limited capabilities* for accessing component services:
    - only simple interactions are supported, e.g., in CORBA, synchronous, deferred synchronous and one-way.

Motivation: why adaptation is needed (and more…)

- Adaptation of software components is an important issue in CBSE.
- SOA & Web Services connectivity
  - heterogeneous services;
  - interaction/architectural mismatches.
- Legacy,embedded and COTS systems integration
  - unavoidable heterogeneity;
  - incompatible and/or non-sufficiently specified interaction behavior
    - e.g., in COM, only interface signature (*it is not enough!!!*);
    - when the time is not critical, signature + protocol might be enough;
    - when the time is critical, signature + protocol + QoS constraints might be enough.
- Interfaces and even frameworks

What is our application context and what's the problem?

- Context:
  - a CB development framework for the *correct-by-construction* and *incremental assembly* of CB real-time systems out of a set of already heterogeneous implemented components.
- Problem:
  - the ability to establish/guaranteeing properties on the assembly code by only assuming a relative knowledge of the properties of the single components.

The role of the Software Architecture (SA)

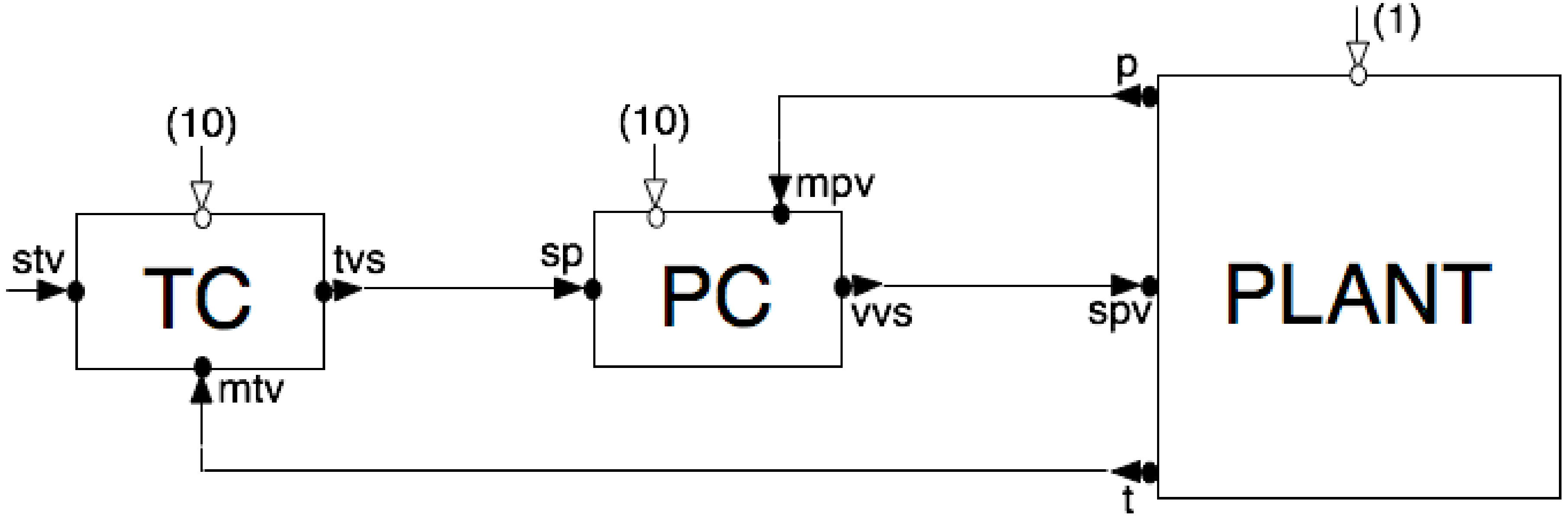
- A SA represents the reference skeleton used to compose components and let them interact
  - interactions among components are represented by the notion of *software connector*.

Basic ideas

- A simple SA structure which exploits the separation between functional behavior (i.e., the components) and integration/ communication behavior (i.e., the connectors).
- Extra information at component level: component assumptions on the expected environment
  - interface signature + interaction protocol + timing information.
- Promoting the use of *automatically derived* component *adaptors* as special components that are used to enhance the behavior of connectors in order to solve possible incompatibilities (black-box component settings).

The reference architectural model

- It belongs to generic pipe-and-filter styles and the components follow a data-flow interaction model.



Component information

- Component behavior *observable* from its *external* environment
  - sequences of reading/writing actions from/to input/output ports plus QoS constraints on these actions.
- *Assumptions on the* component expected *environment* in order to guarantee a property in a specific integration context.
- In the case of deadlock-freeness, we use *“my context never blocks me”*, e.g.:
  - *“if the time is elapsing for me, it has to elapse for the environment as well”;*
  - *“if I can perform a writing action on a port p (within a certain interval of time), the environment must be able to perform the reading from p within that interval”.*
- How do we produce this additional information?

Modeling the system

- The DLiPA specification language:
  - component behavior modeled using finite state machines (i.e., LTSs)
    - a unique model explicitly describing a combination of functional and extra-functional behavior in a operational way (suitable for synthesis purposes);
  - integration through parallel composition of component models;
  - mismatches/incompatibilities
    - clock inconsistency, reading/writing time inconsistency, mismatching interaction protocols, etc…;
    - all modeled as deadlocks: the system model, seen as the parallel composition of the component models, can reach a state where no action is possible.

DLiPA (Duration-Latency-interval Process Algebra)

- It is an extension of Milner's CCS aiming at considering a notion of *controllability*, *latency* and *duration* of actions, and of logical *clock* associated to each process:
  - controllable (i.e., "discardable") vs. uncontrollable (i.e., "mandatory") actions;
  - latency: the number of global time units that can pass before the actions is performed from the time it is enabled
    - earger vs. delayed actions;
  - duration: the number of local time units needed for the action execution
    - time-consuming vs. immediate actions;
  - clock: a periodic stream of Boolean values [Pouzet et al. EMSOFT'05]
    - activation frequency of the component.

DLiPA… continuing

- Two component views:
  - clock-independent
    - only for *sequential processes*;
    - the behavior is parametric with respect to the clock that must be still assigned;
  - clock-dependent
    - for *DLiPA processes* (sequential and composite ones);
    - the clock is fixed since it has been instantiated.

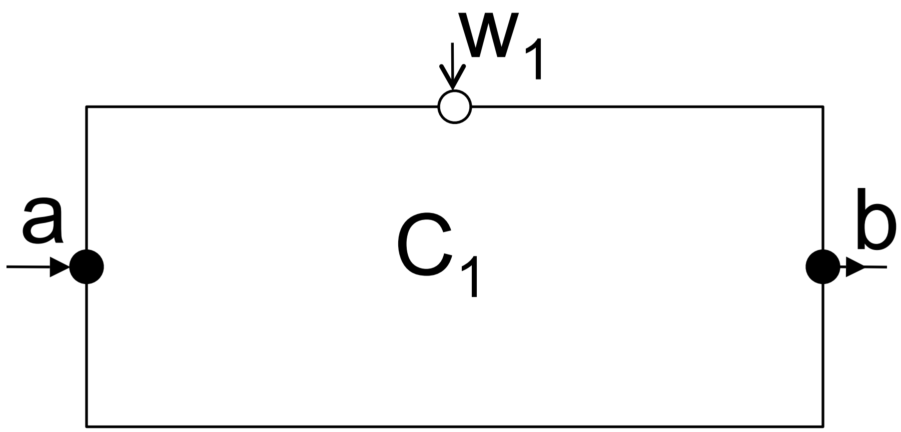
Sequential processes (syntax)

$p := \mu_l^D.p \mid p+p \mid X \mid \text{rec } X.p$

- l is a number standing for $[0,l]$ (latency)
- D is an interval $[d_1, d_2]$ (duration)
- μ can be
  - visible (e.g., a, $\overline{a}$, a^u, $\overline{a^u}$)
  - internal (i.e., τ)

Sequential processes (semantics)

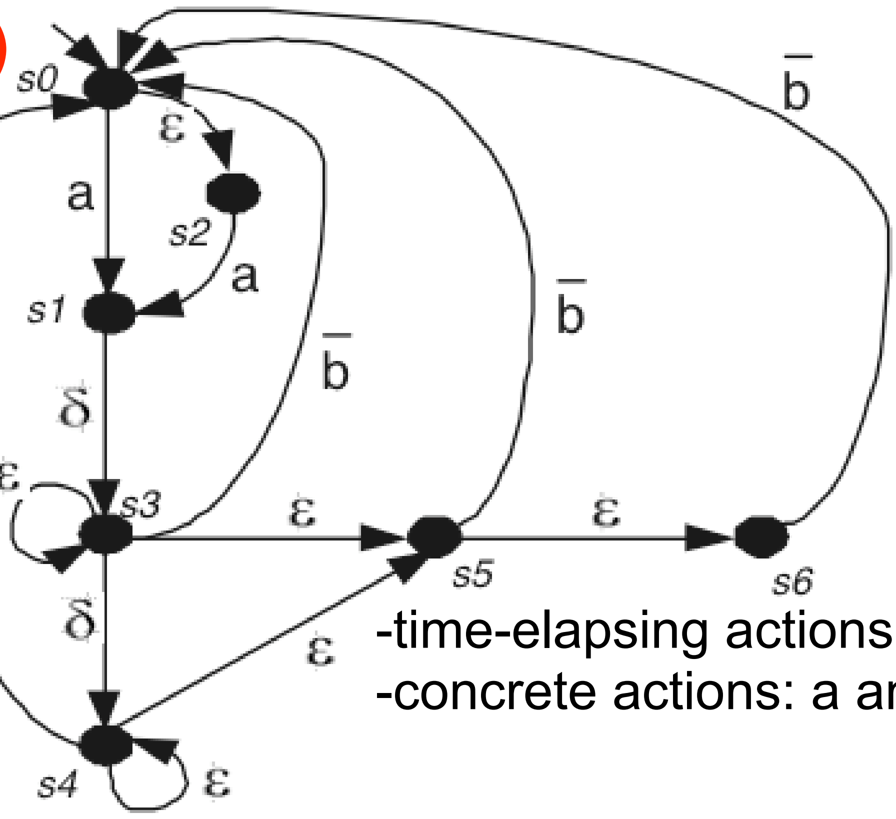
$$C_1 := \text{rec } X.(a_1^{[1,2]}.\overline{b}_2.X)$$



Sequential processes (semantics)

$C_1 := \text{rec } X.(a_1^{[1,2]}.\overline{b}_2.X)$

SeqOS(C_1)



-time-elapsing actions: ε and δ
-concrete actions: a and $\overline{b}$

DLiPA processes (syntax)

P := <p,w> | P|P | P\I | P[f]

- p is a sequential process
- w is a clock constant
  - e.g, (10) = 10101010101010…
- |, \, [f] are the parallel, restriction and relabeling operators, respectively

DLiPA processes (semantics)

- <p,w> behaves like SeqOS(p) where the sequences of actions (concrete and time-consuming) that cannot be performed respect to w are pruned; its LTS is denoted by OS(<p,w>).

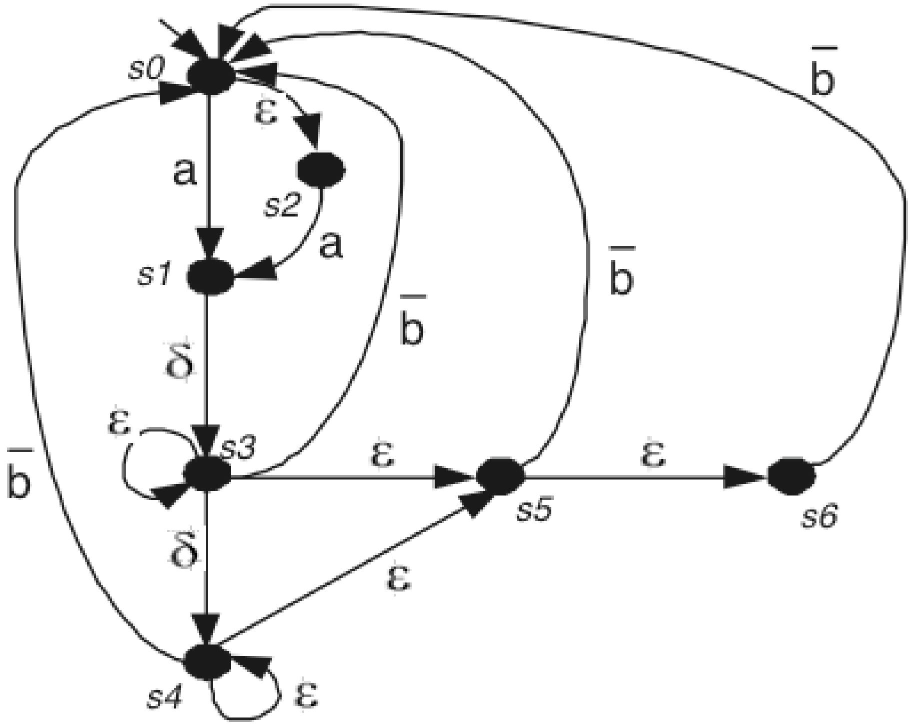
- Model validation:
  - if OS(<p,w>) is empty or it is made only of finite paths, w is an *invalid clock* for p; otherwise, w may be valid;
  - OS(<p,w>) without its finite paths has to preserve the protocol specified for p
    - weak bisimulation between OS(<p,w>) without the finite paths and SeqOS(p) (**tool supported**, e.g., CADP).

- In other words, *w is a valid clock if <p,w> exists as sequential process and by abstracting from the time the protocol of p and <p,w> has to be the same*.

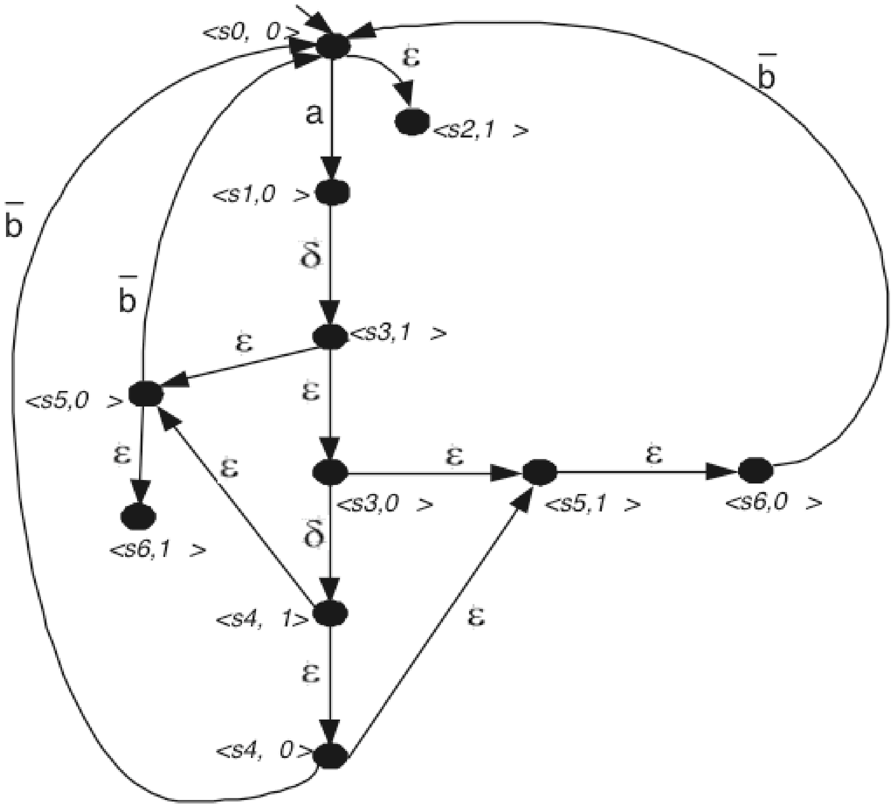
DLiPA semantics… continuing

$$\langle C_1,w_1\rangle := \langle \text{rec } X.(a_1^{[1,2]}.b_2.X),(10)\rangle$$

SeqOS(C_1)



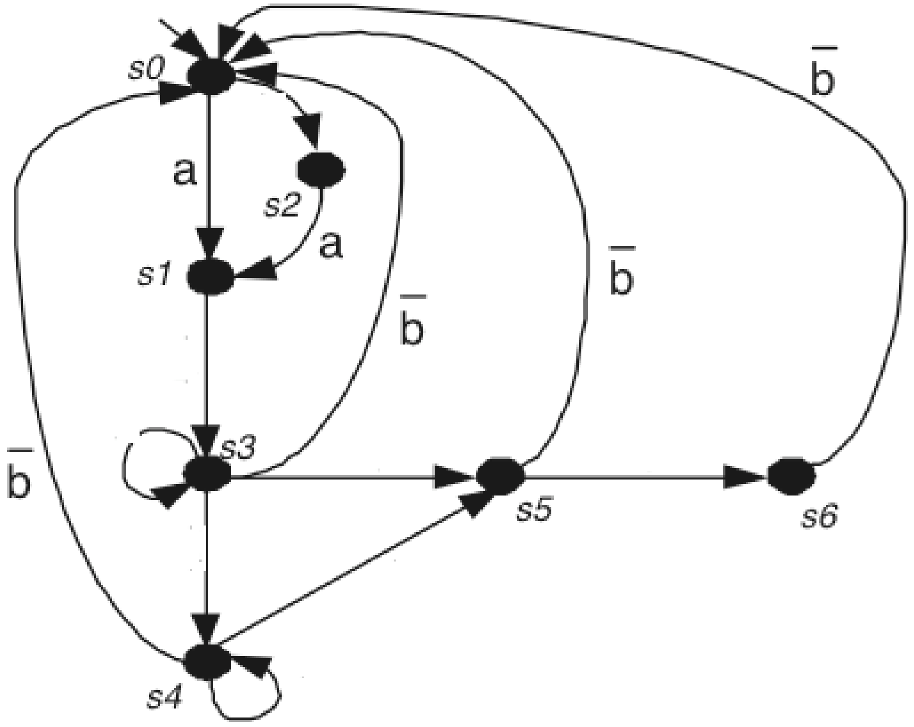
OS($\langle C_1,w_1\rangle$)



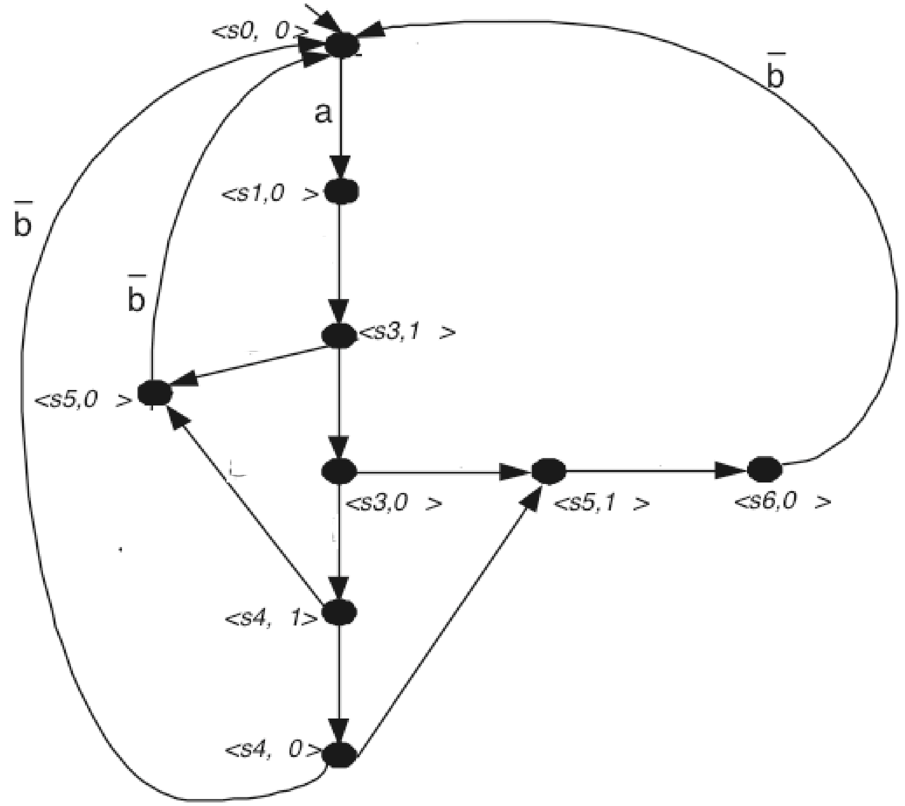
DLiPA semantics… continuing

$\langle C_1, w_1\rangle := \langle \mathrm{rec}\ X.(a_1^{[1,2]}.b_2.X), (10)\rangle$

SeqOS(C_1)



OS($\langle C_1, w_1\rangle$)

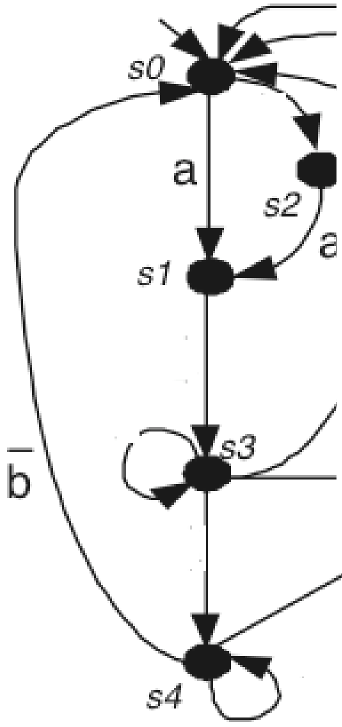


DLiPA semantics… continuing

$<C_1,w_1> := <\text{rec } X.(a_1^{[1,2]}.b_2.X),(10)>$

SeqOS(

$w_1>)$

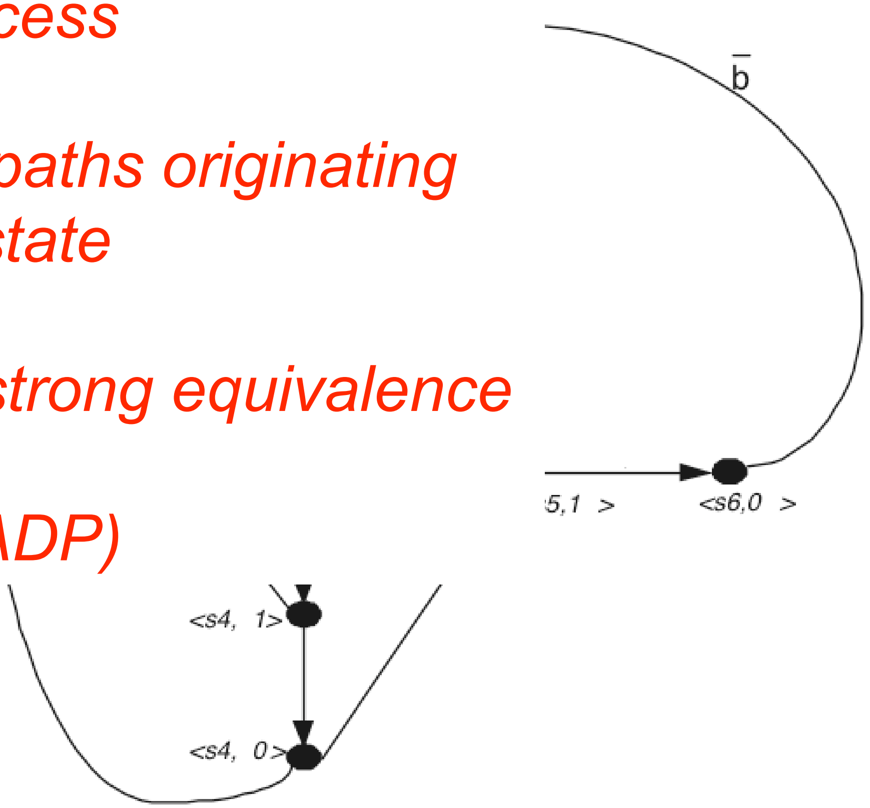


minimization can be required due to the finite paths pruning process

observational identical paths originating from the same source state

LTS reduction modulo strong equivalence

tool supported (e.g., CADP)



DLiPA semantics… continuing

- The restriction operator “\” allows one to define connections among ports (with the same name)
  - in combination with “|”, it forces the synchronization of complementary actions.

- The relabeling operator “[f]” allows one to relabel port names
  - in combination with “\” allows one to redefine connections (i.e., define new architectural configurations, e.g., interpose an adaptor);
  - allows one to match different port names (i.e., to solve interface signature mismatches).

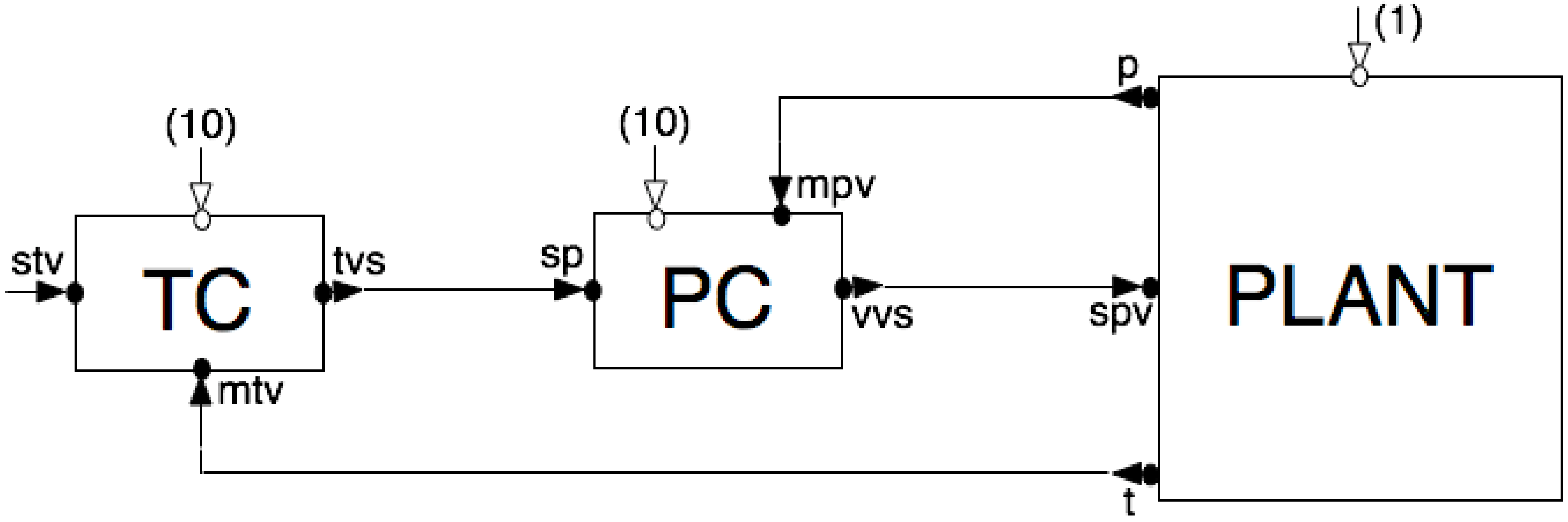
DLiPA semantics… concluding

- The parallel composition operator “|”:
  - a μ action can be executed by performing any its interleaving (except when it is forbidden by means of “\”). Conversely to this, when a timed process lets the time elapse, all other timed processes in the system have to let the time elapse.

- The case of uncontrollable actions
  - concrete action (μ): a process can either perf$\overline{o}$rm μ or let the time elapse while the environment cannot perform μ but it can let the time elapse
    - *mismatch! μ cannot be discarded!*
  - time-elapsing action (ε^u or δ^u, only for duration): a process can either let the time elapse or per$\overline{fo}$rm μ while the environment cannot let the time elapse but it can perform μ
    - *mismatch! it has not been possible to synchronize for all duration values!*

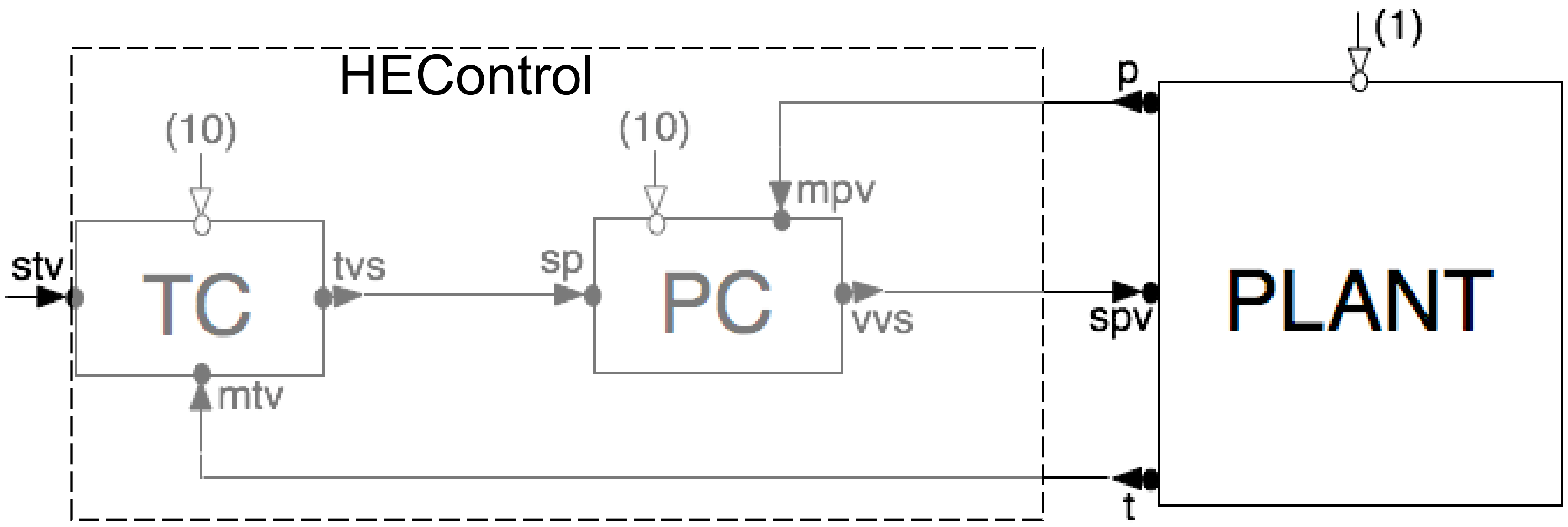
Adaptor synthesis: an example

- TC ::= <rec X.($\overline{\text{tvs}}^{[1,2]}$.stv.mtv^{u}.X), (10)>
- PC ::= <rec X.($\overline{\text{vvs}}^{1}$.mpv.sp$_1$.X), (10)>
- PLANT ::= <rec X.(spv.$\overline{\text{p}}^{1}$.$\overline{\text{t}}^{u1}$.X), (1)>

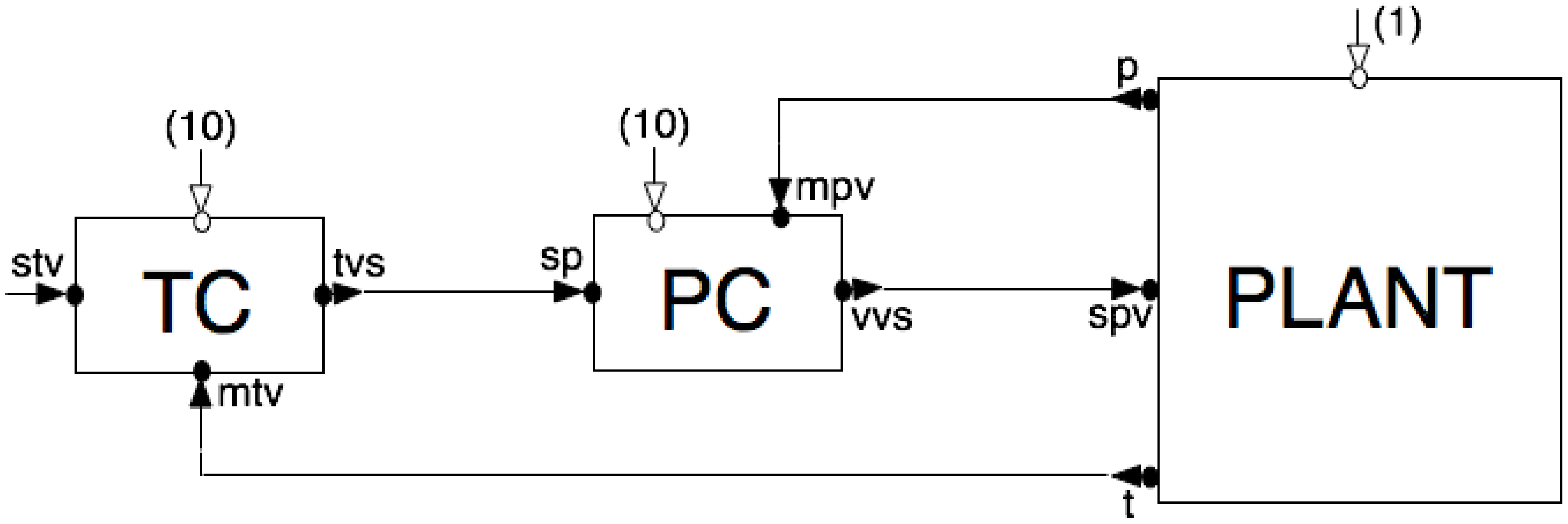


Adaptor synthesis: an example

- TC ::= <rec X.($\overline{tvs}^{[1,2]}$.stv.mtv^{u}.X), (10)>
- PC ::= <rec X.($\overline{vvs}^{1}$.mpv.sp$_1$.X), (10)>
- PLANT ::= <rec X.(spv.$\overline{p}^{1}$.$\overline{t}^{u1}$.X), (1)>

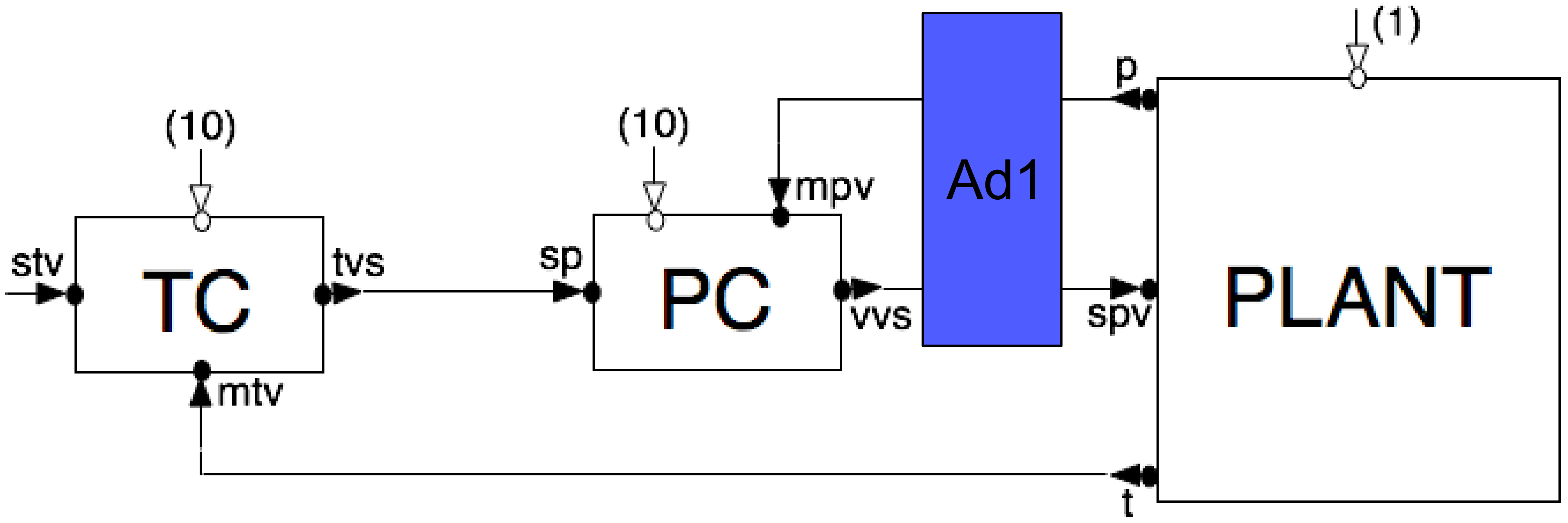


First solution

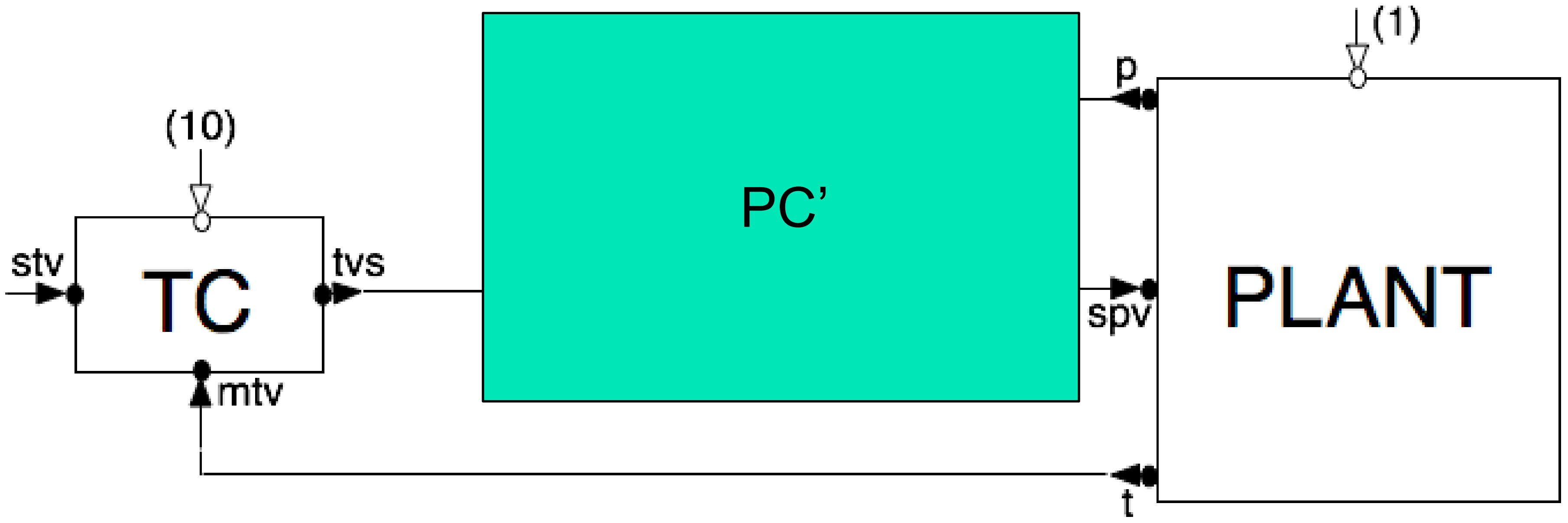


First solution

Step 1

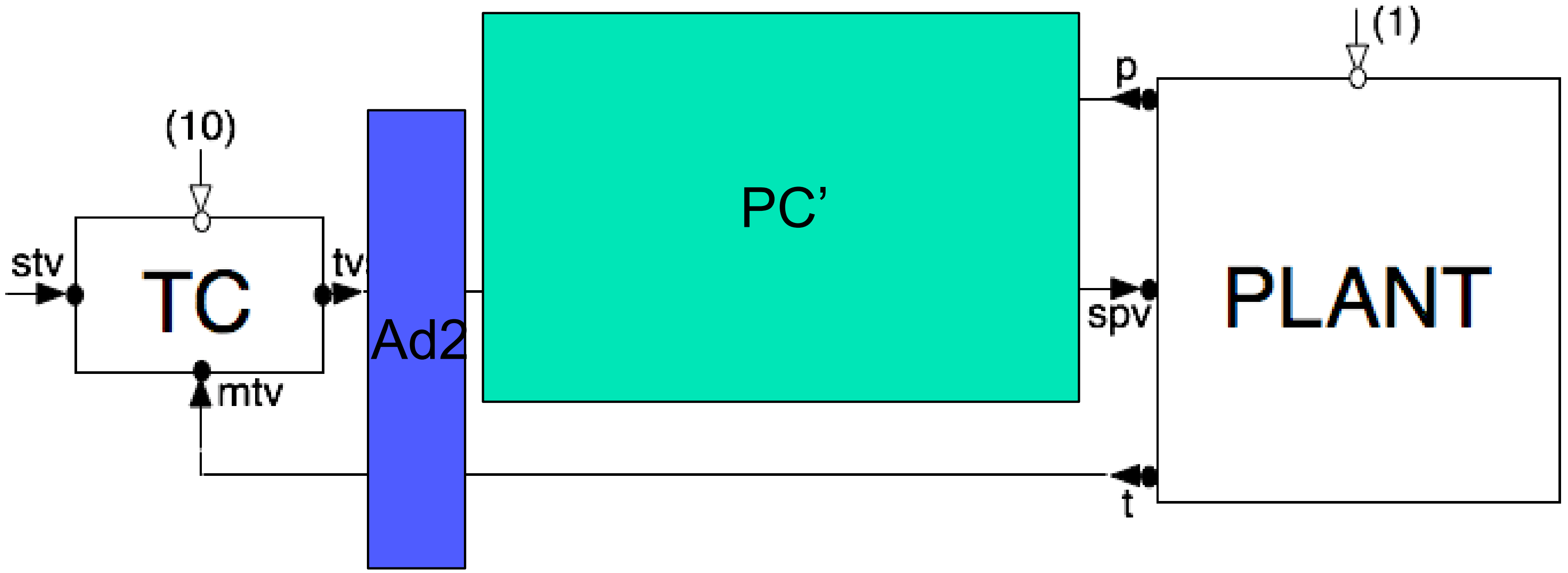


First solution

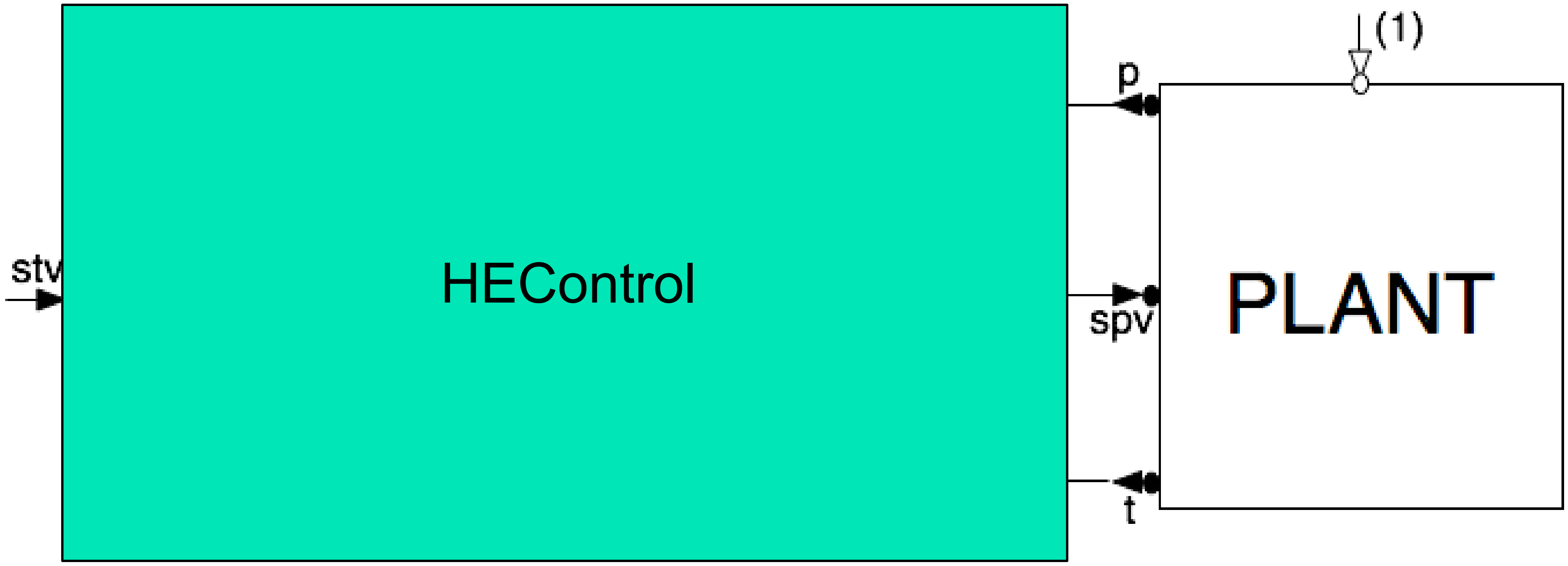


First solution

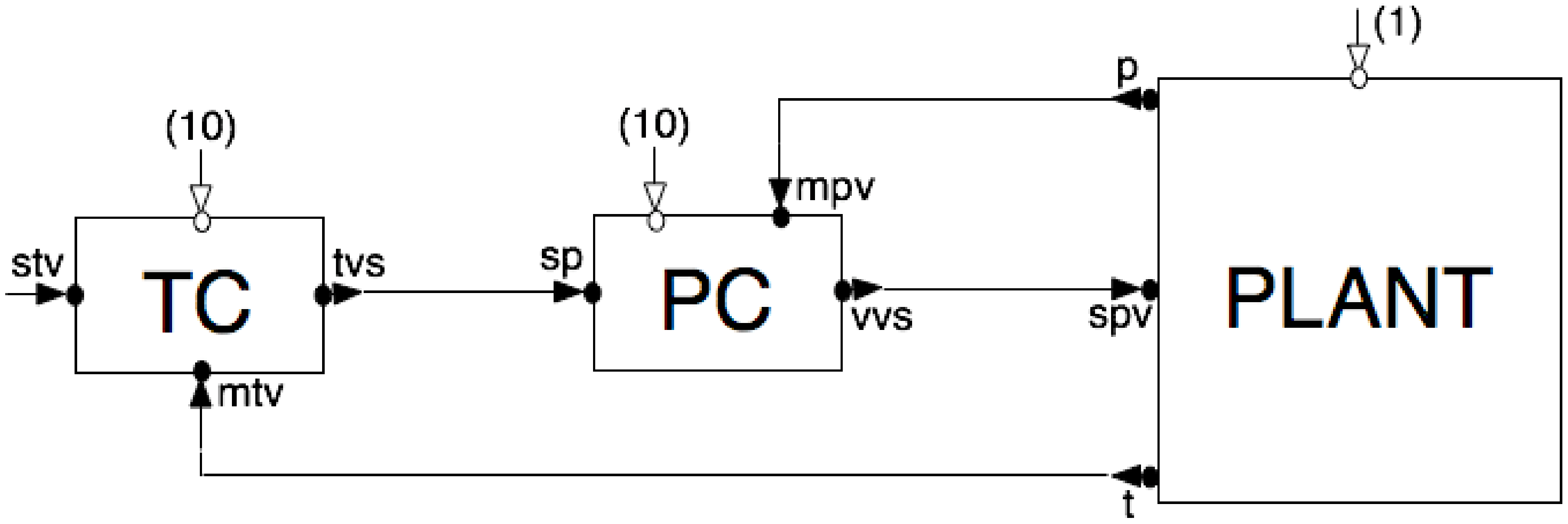
Step 2



First solution

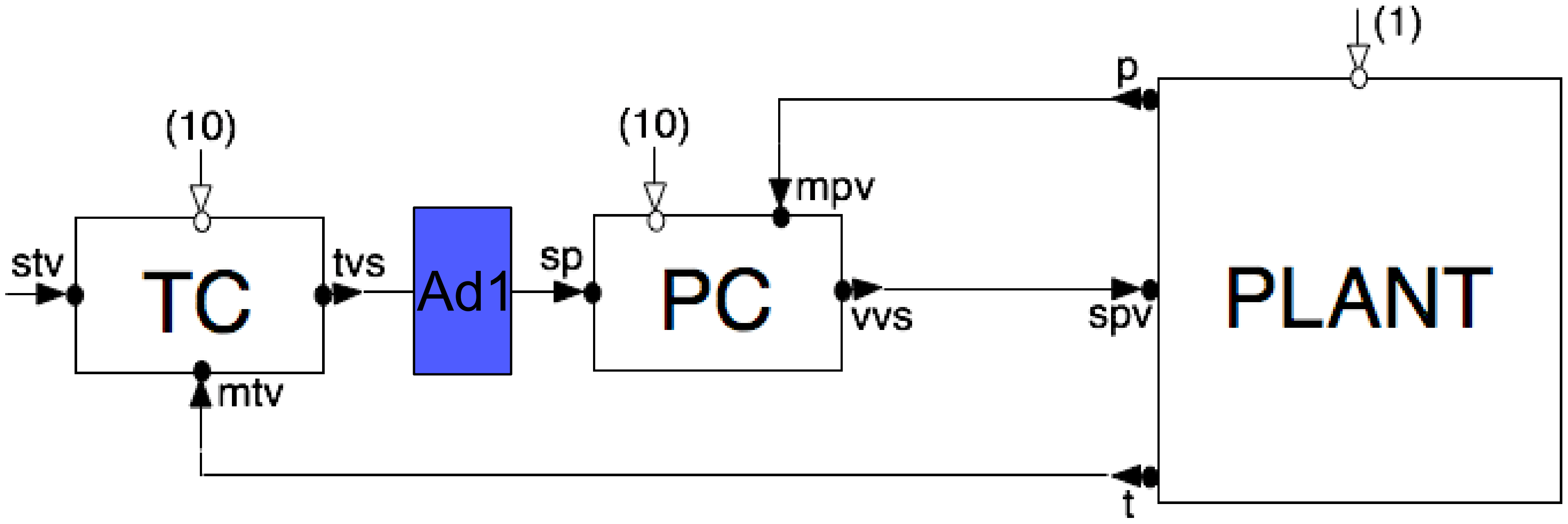


Second solution

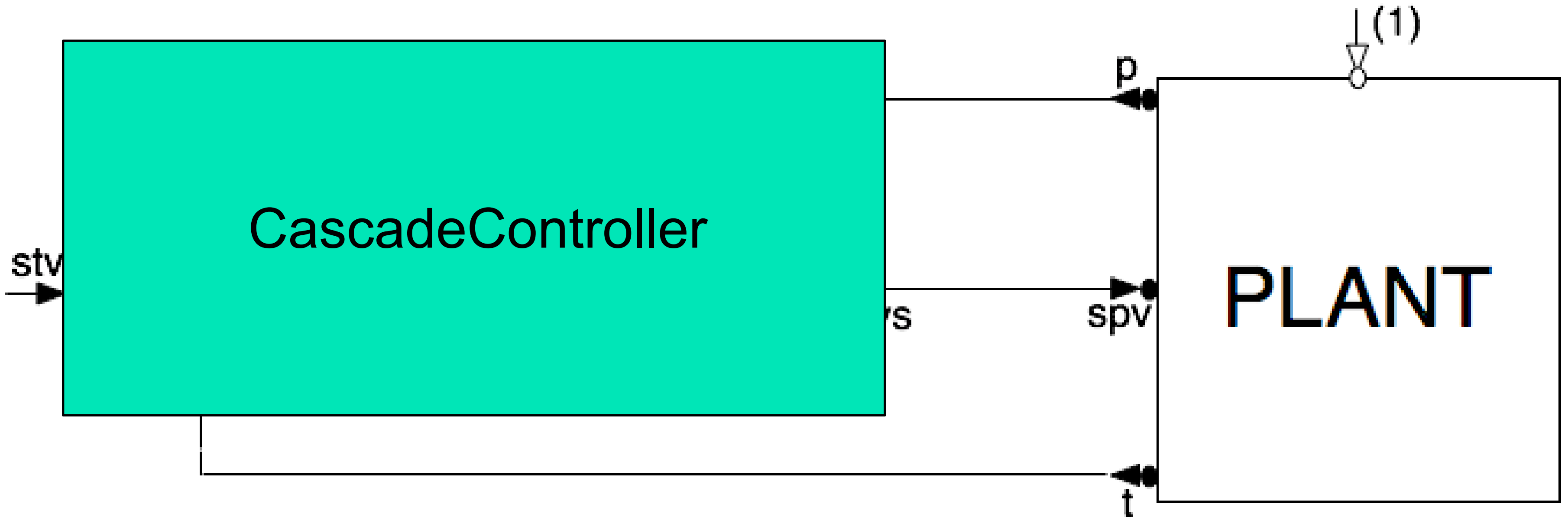


Second solution

Step 1

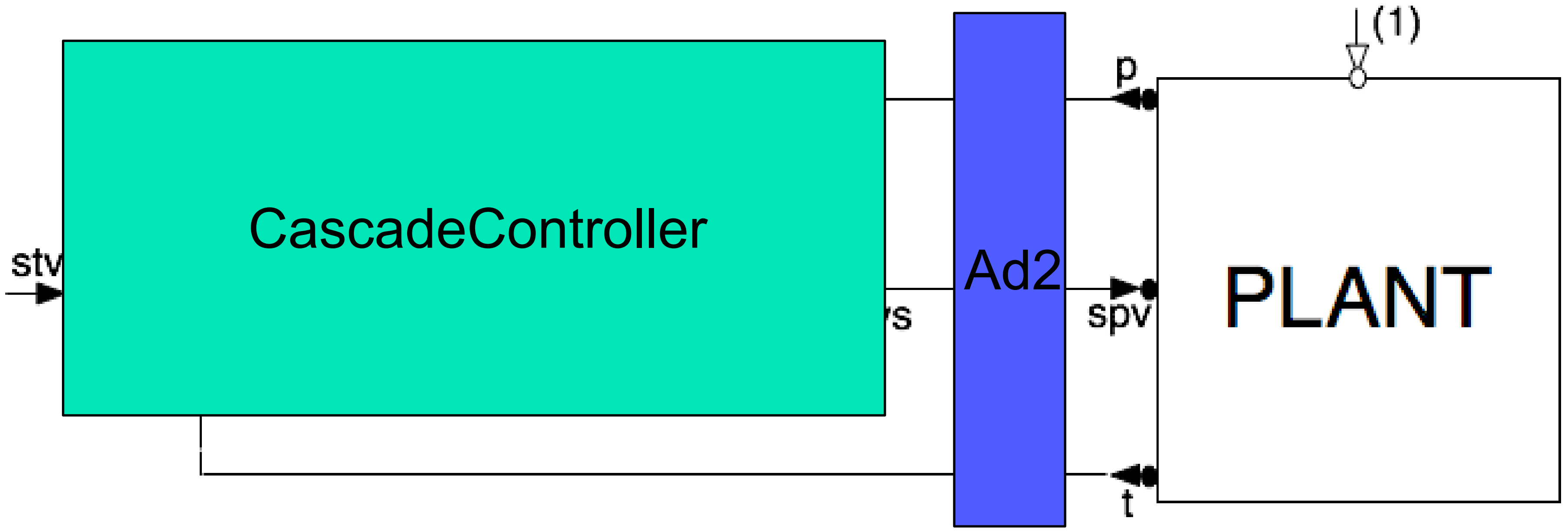


Second solution

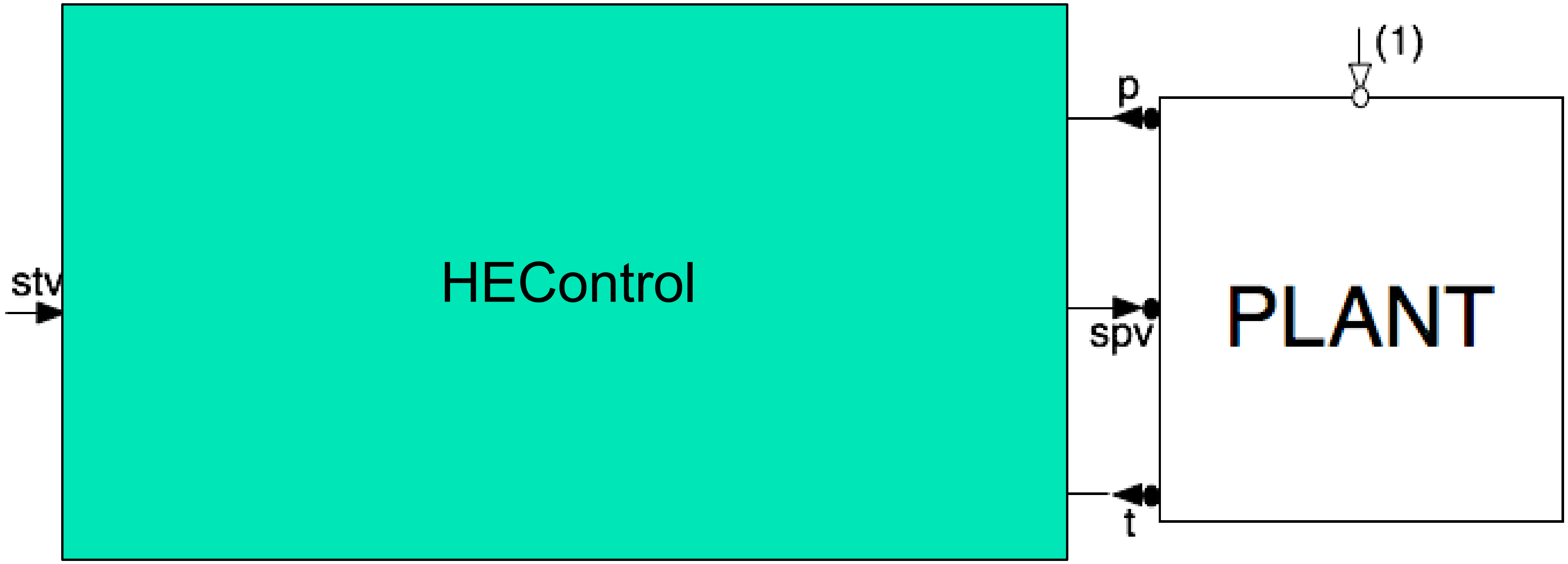


Second solution

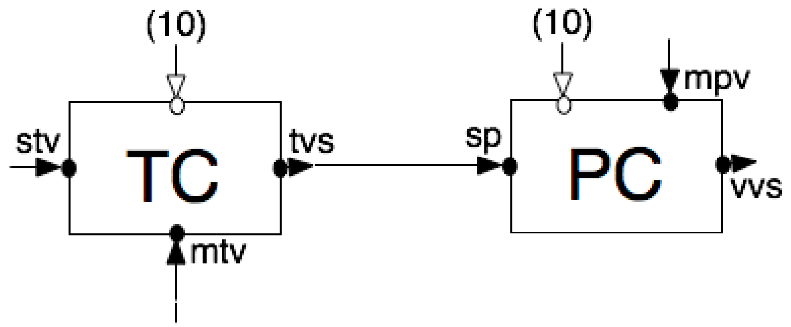
Step 2

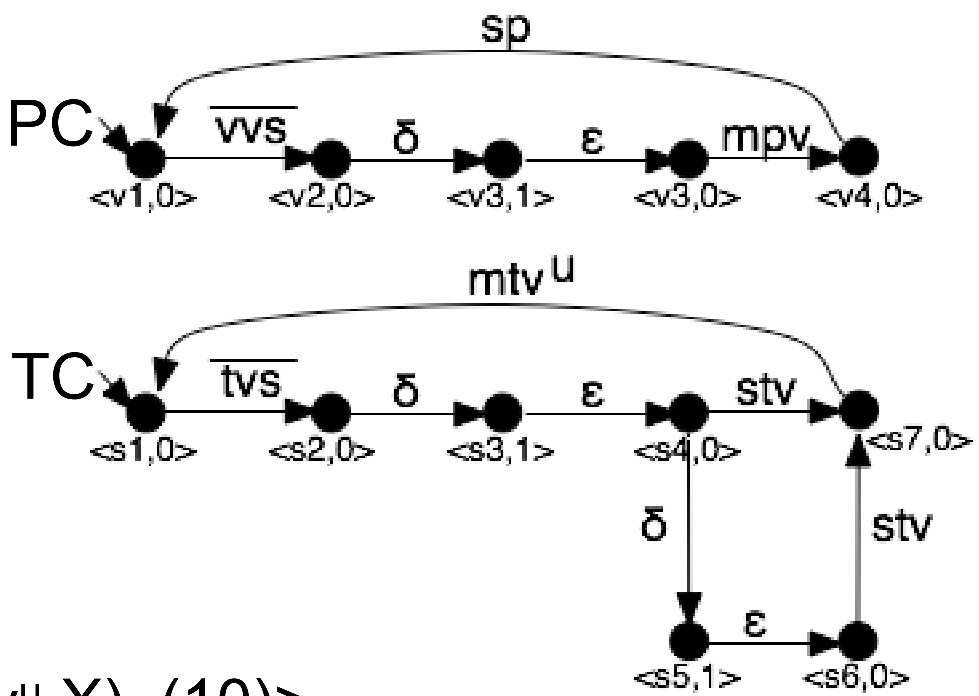


Second solution



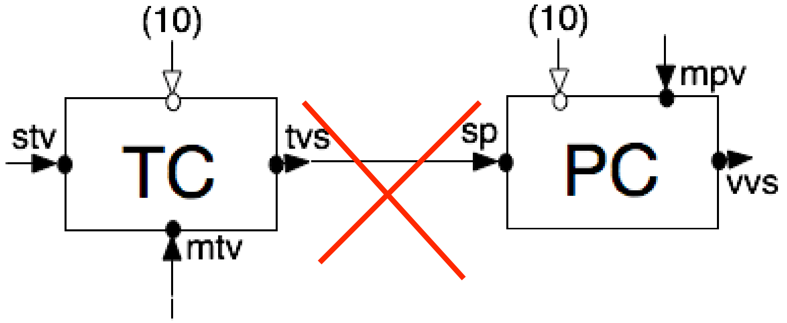
Second solution… continuing

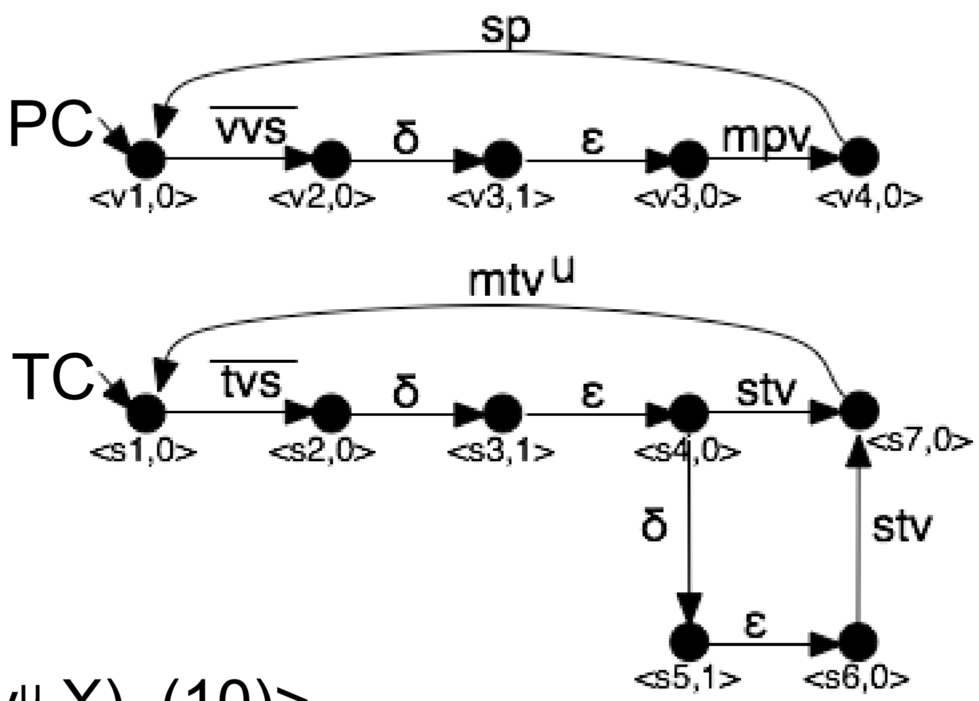




- TC ::= <rec X.($\overline{\text{tvs}}^{1,}$ -.stv.mtv^{u}.X), (10)>
- PC ::= <rec X.($\overline{\text{vvs}}^{1}$.mpv.sp$_{1}$.X), (10)>

Second solution… continuing



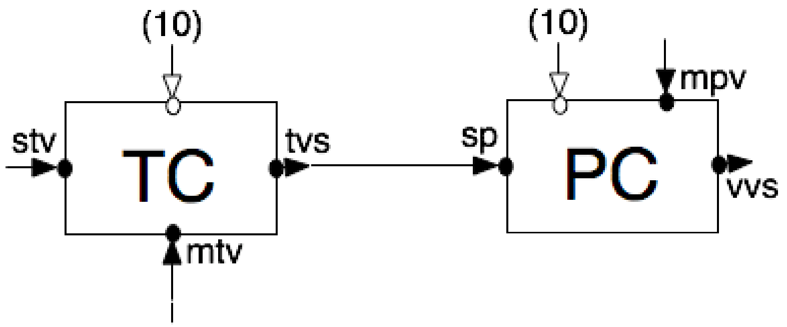


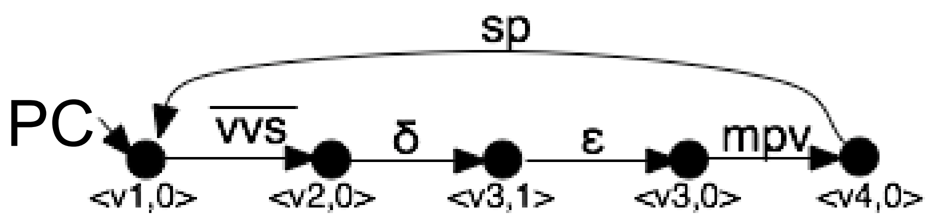
(TC|PC)\{tvs,sp} -> $\overline{vvs}$.nil

interface signature mismatch!, i.e., different connected port names

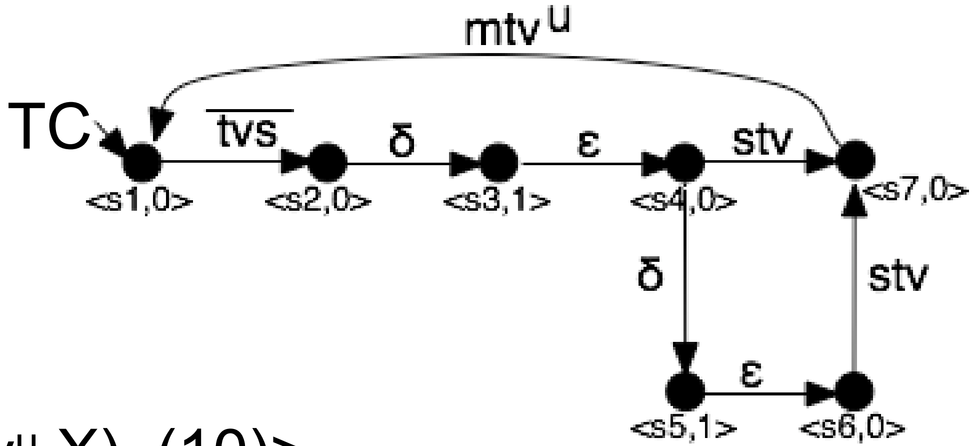
- TC ::= <rec X.($\overline{tvs}^{1}$.stv.mtv^u.X), (10)>
- PC ::= <rec X.($\overline{vvs}^1$.mpv.sp_1.X), (10)>

Second solution… continuing



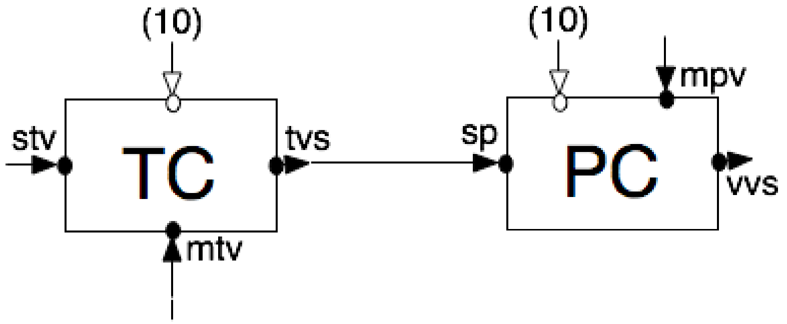


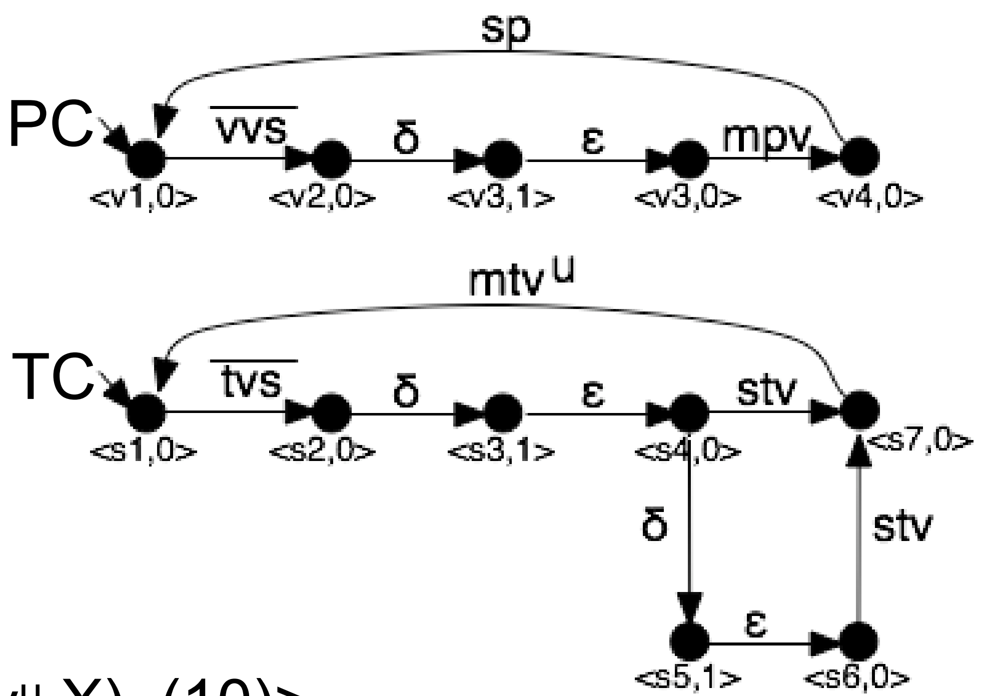
$(TC[\{^{sp}/_{tvs}\}]|PC)\backslash\{sp\} \rightarrow \overline{vvs}.nil$



- TC ::= <rec X.($\overline{tvs}^{1,}$ .stv.mtv^u.X), (10)>
- PC ::= <rec X.($\overline{vvs}^1$.mpv.sp$_1$.X), (10)>

Second solution… continuing



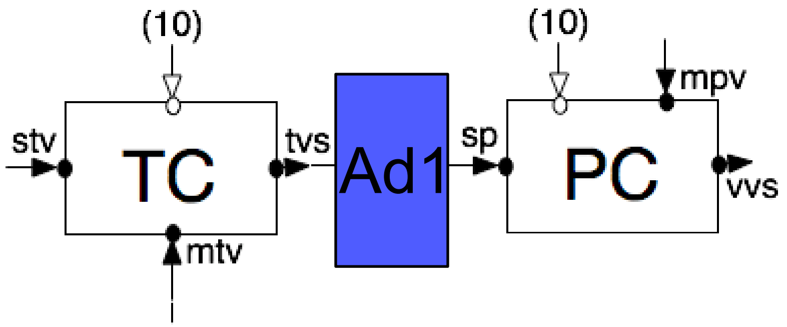


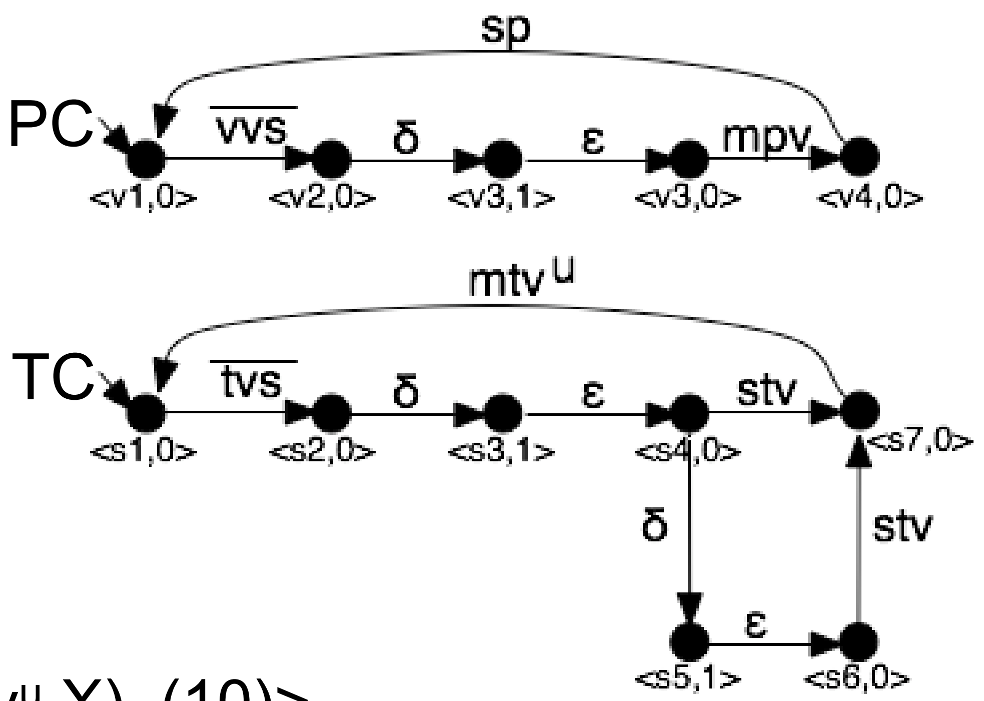
(TC[{$^{sp}/_{tvs}$}]|PC)\{sp} -> $\overline{\text{vvs}}$.nil

timing assumption inconsistency!

- TC ::= <rec X.($\overline{\text{tvs}}$| ^{1,} -.stv.mtv^{u}.X), (10)>
- PC ::= <rec X.($\overline{\text{vvs}}^{1}$.mpv.sp$_{1}$.X), (10)>

Second solution… continuing



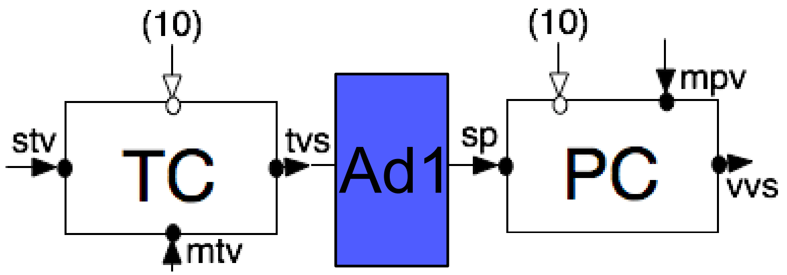


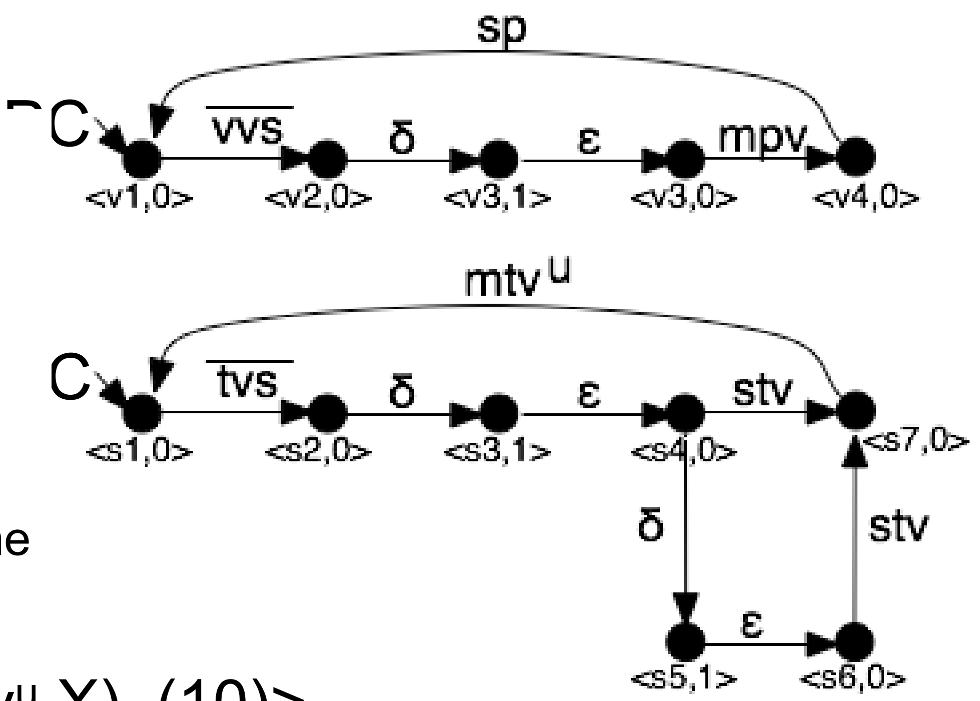
(TC[{$^{sp}/_{tvs}$}]|PC)\{sp} -> $\overline{\text{vvs}}$.nil

timing assumption inconsistency!

- TC ::= <rec X.($\overline{\text{tvs}}^{1,}$ -.stv.mtv^{u}.X), (10)>
- PC ::= <rec X.($\overline{\text{vvs}}^{1}$.mpv.sp$_{1}$.X), (10)>

Second solution… continuing



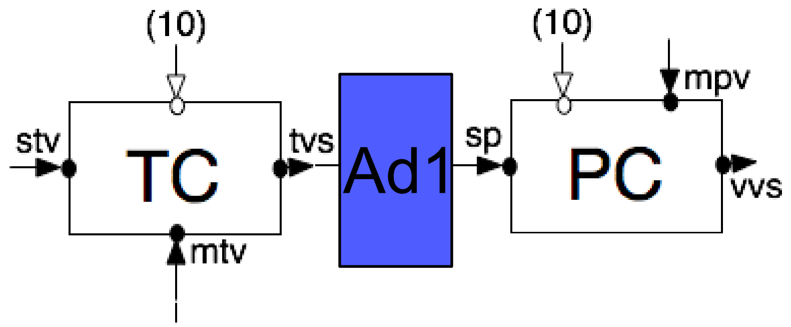


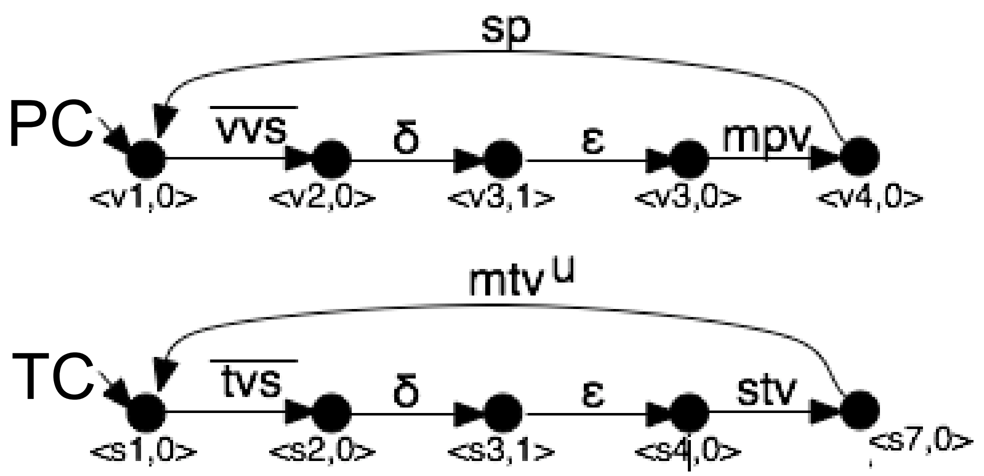
What's about checking the existence of the adaptor?
bounded: proportional time scale;

BUT (controllable vs. uncontrollable):
is it sufficient for deadlock- and livelock-freeness?
deadlock- and livelock-free: identical time scale.

- TC ::= <rec X.($\overline{tvs}^{1,}$ -.stv.mtv^u.X), (10)>
- PC ::= <rec X.($\overline{vvs}^1$.mpv.sp_1.X), (10)>

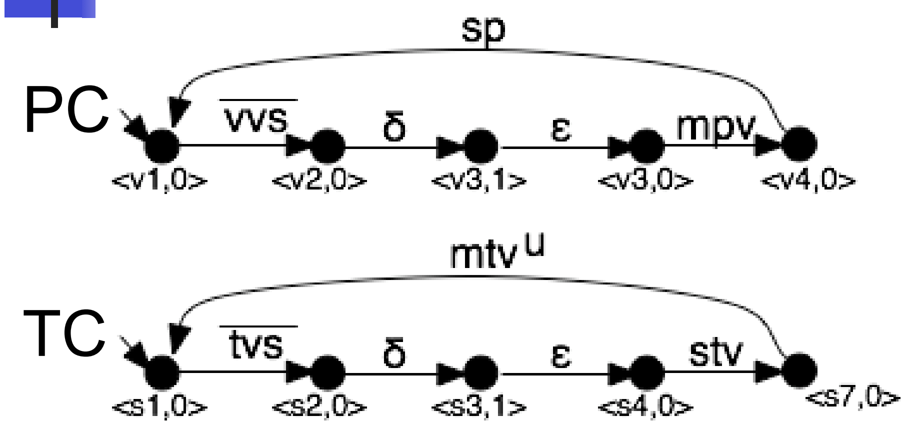
Second solution… continuing





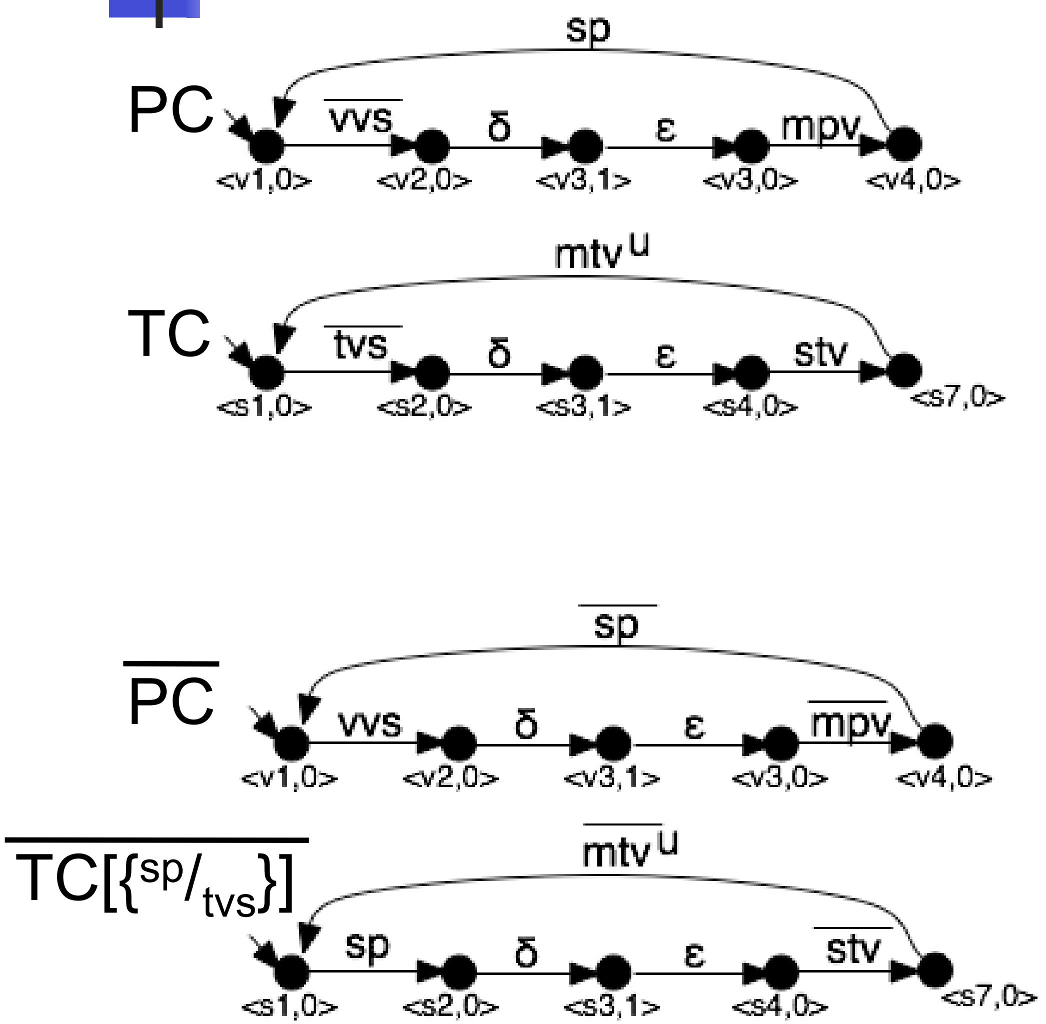
- TC ::= <rec X.($\overline{tvs}^{1,}$ -.stv.mtv^{u}.X), (10)>
- PC ::= <rec X.($\overline{vvs}^{1}$.mpv.sp$_{1}$.X), (10)>

Ad1 adaptor synthesis

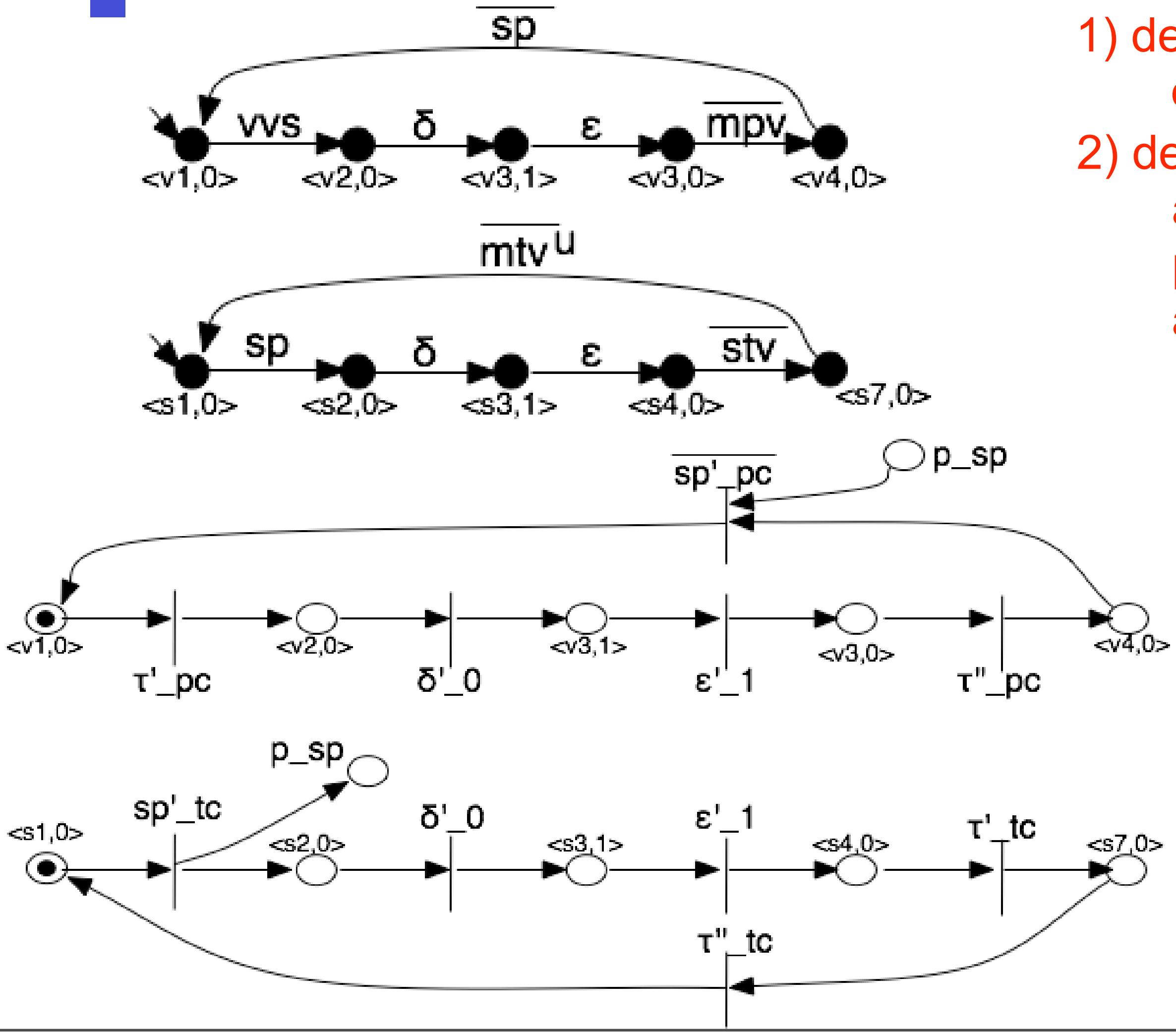


Ad1 adaptor synthesis

1) derive the component expected environment in order to not block;

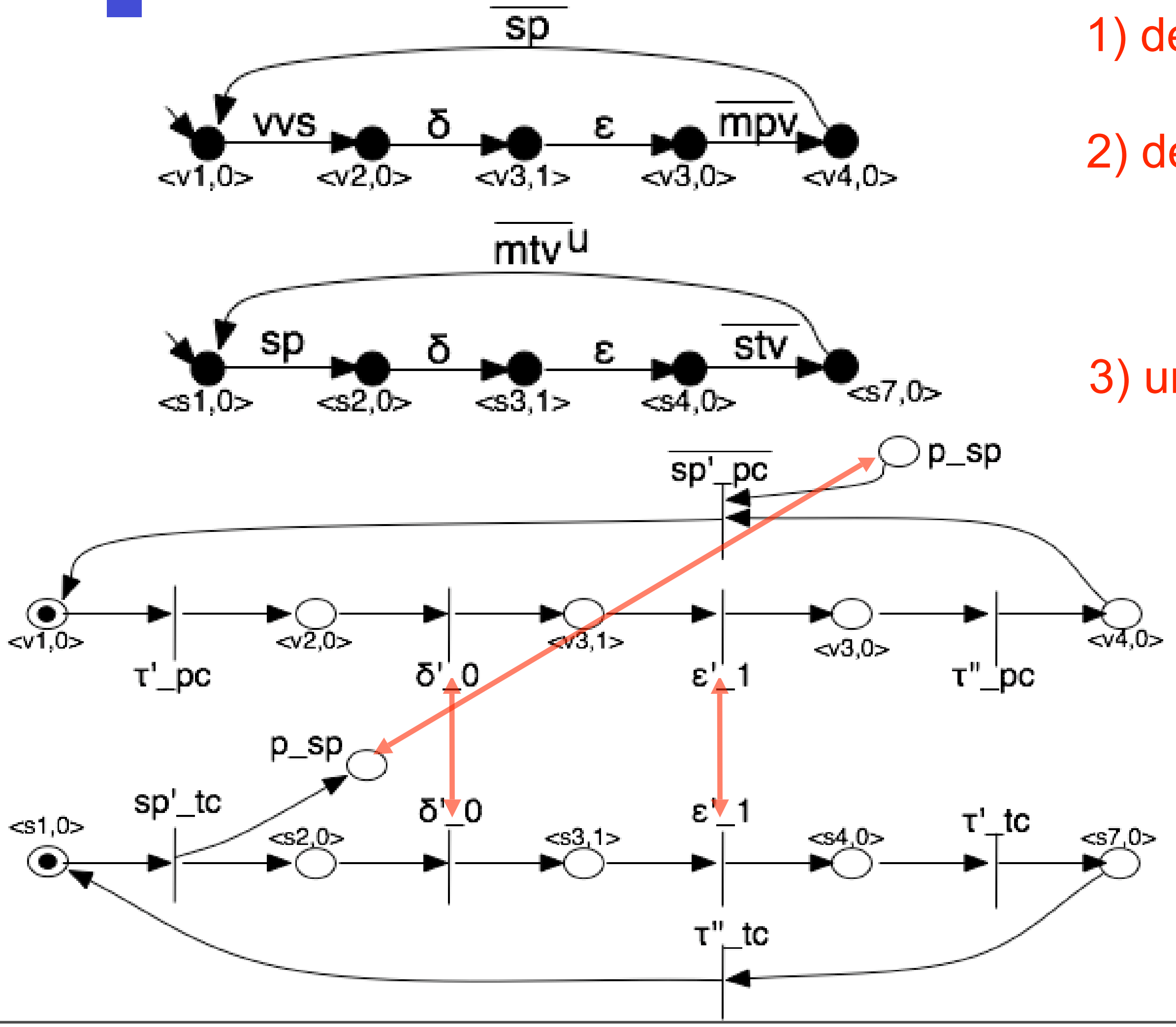


Ad1 adaptor synthesis



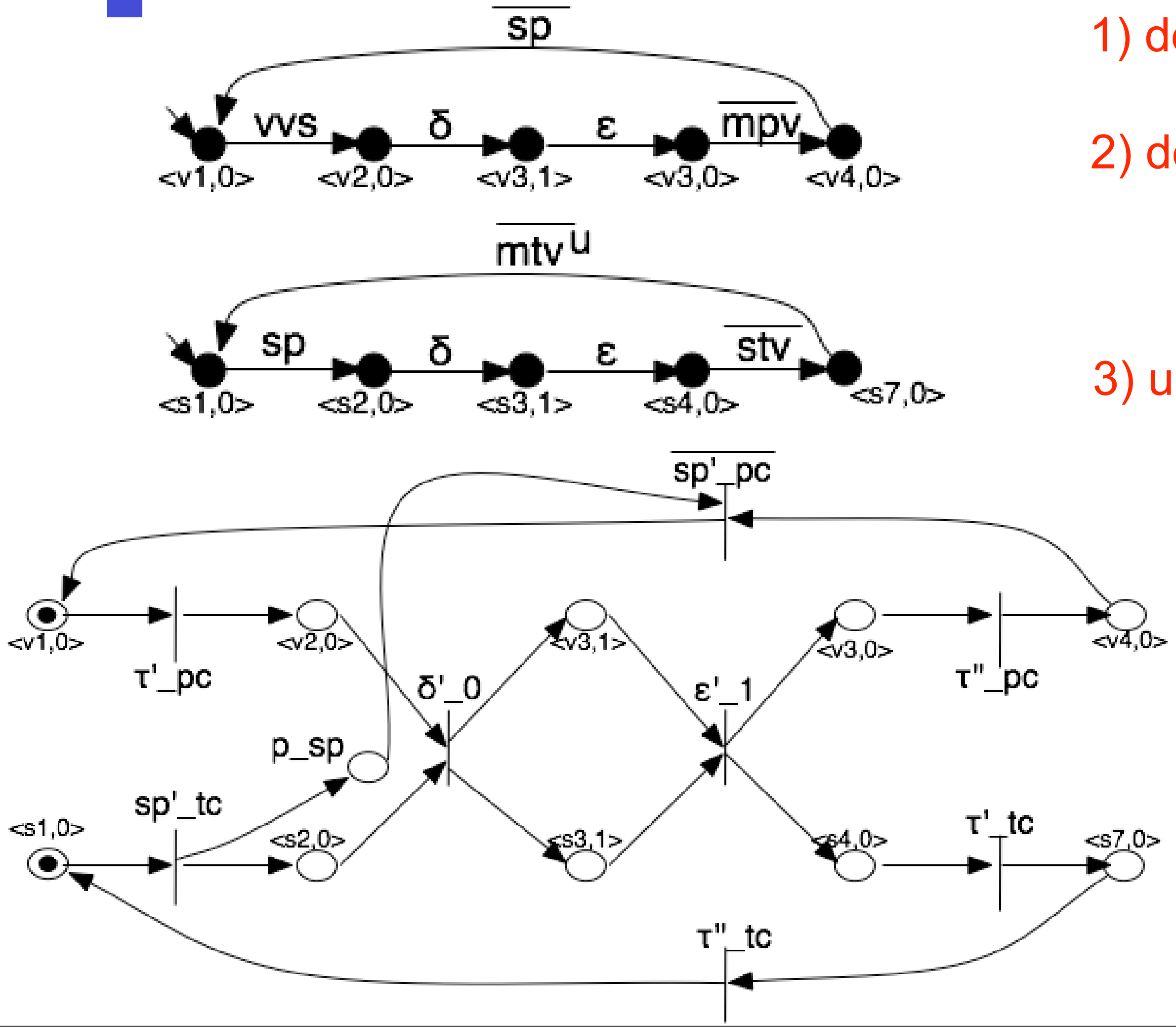
1) derive the component expected environment in order to not block;
2) derive the component PN model according to the restriction and parallel operator (i.e., the ***partial*** adaptor view of the component);

Ad1 adaptor synthesis



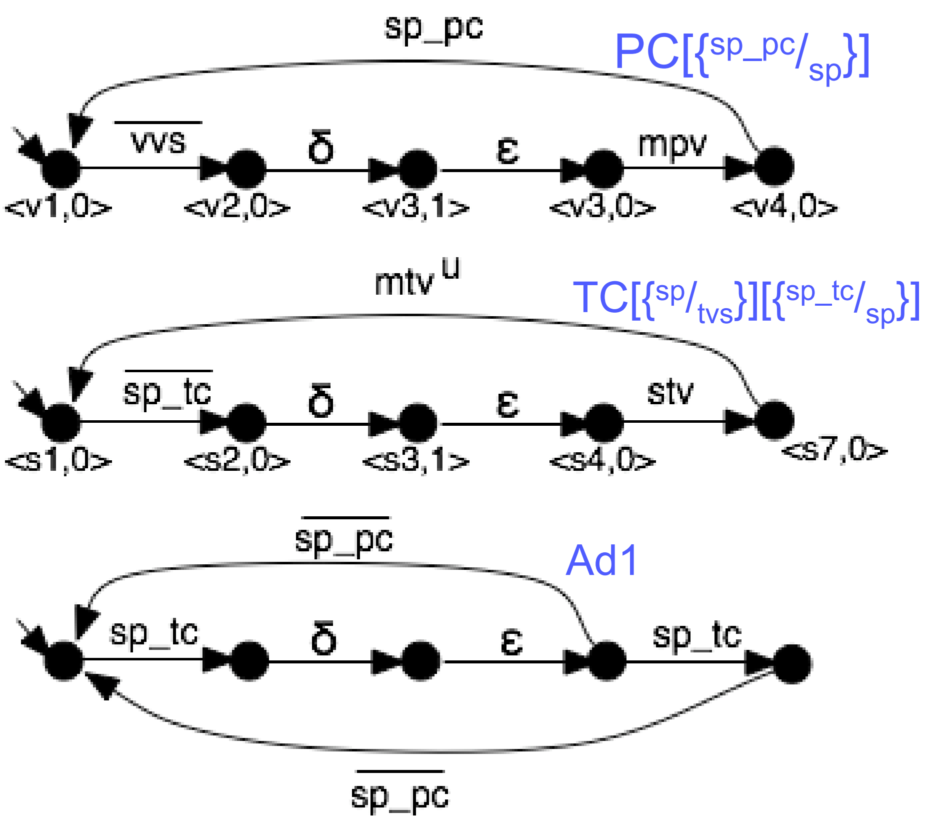
1) derive the component expected environment in order to not block;
2) derive the component PN model according to the restriction and parallel operator (i.e., the ***partial*** adaptor view of the component);
3) unify all the component partial views of the adaptor according to the parallel operator.

Ad1 adaptor synthesis



1) derive the component expected environment in order to not block;
2) derive the component PN model according to the restriction and parallel operator (i.e., the ***partial*** adaptor view of the component);
3) unify all the component partial views of the adaptor according to the parallel operator.

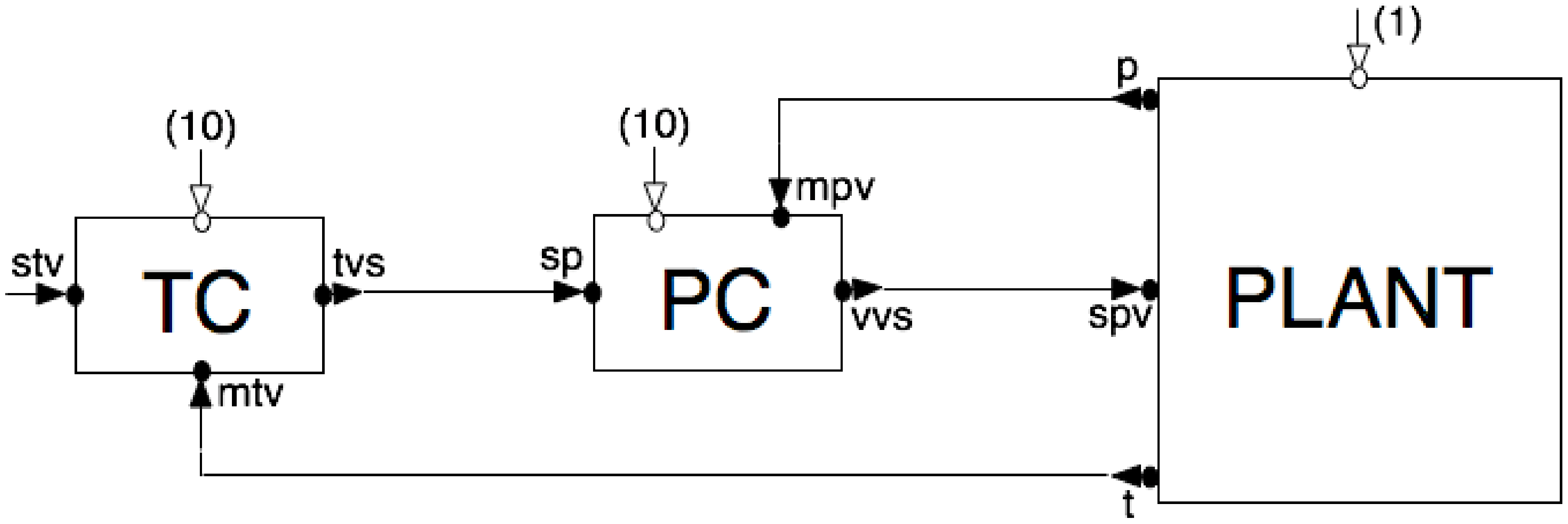
Ad1 adaptor synthesis



1) derive the component expected environment in order to not block;
2) derive the component PN model according to the restriction and parallel operator (i.e., the ***partial*** adaptor view of the component);
3) unify all the component partial views of the adaptor according to the parallel operator.
4) the coverability graph is computed, minimized (TINA + CADP toolboxes) and *"cleaned"*

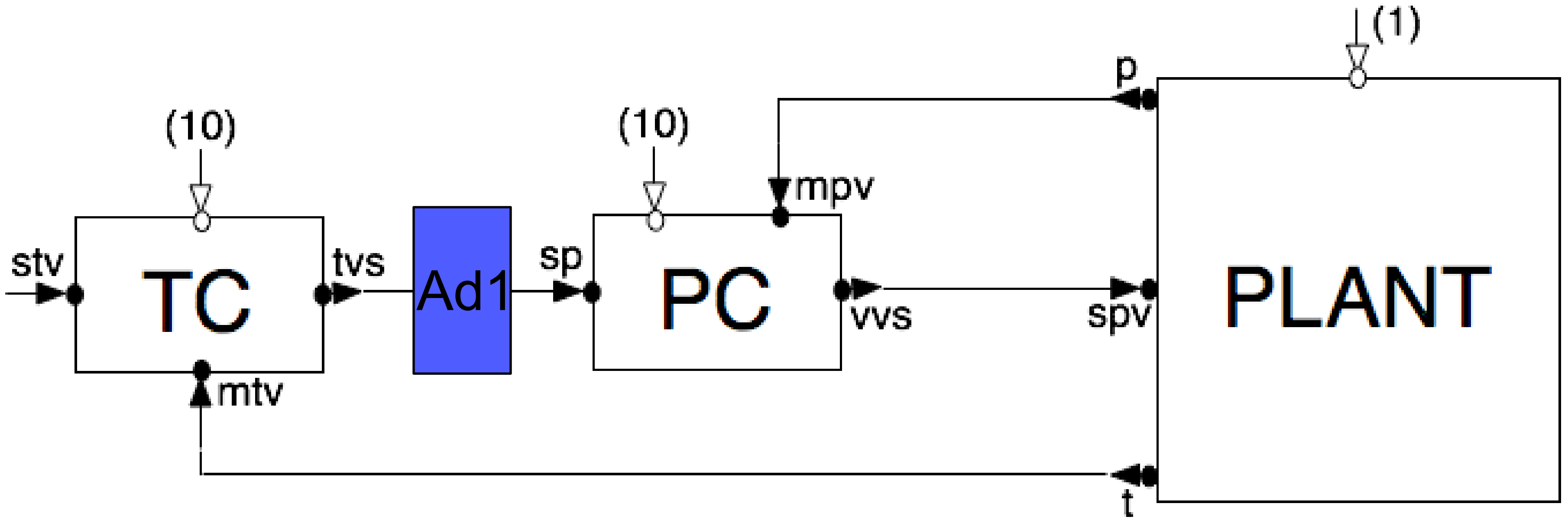
CascadeController = $(TC[\{^{sp}/_{tvs}\}][\{^{sp\_tc}/_{sp}\}] \mid Ad1 \mid PC[\{^{sp\_pc}/_{sp}\}])\backslash\{sp\_tc, sp\_pc\}$

Second solution… again

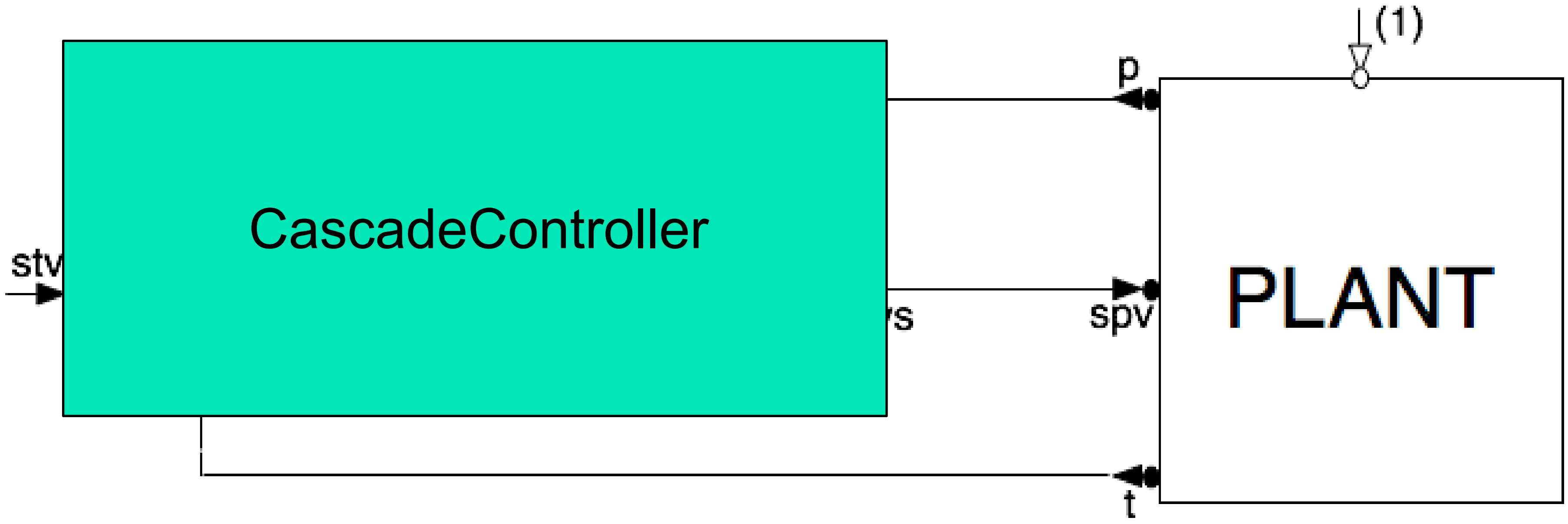


Second solution… again

Step 1

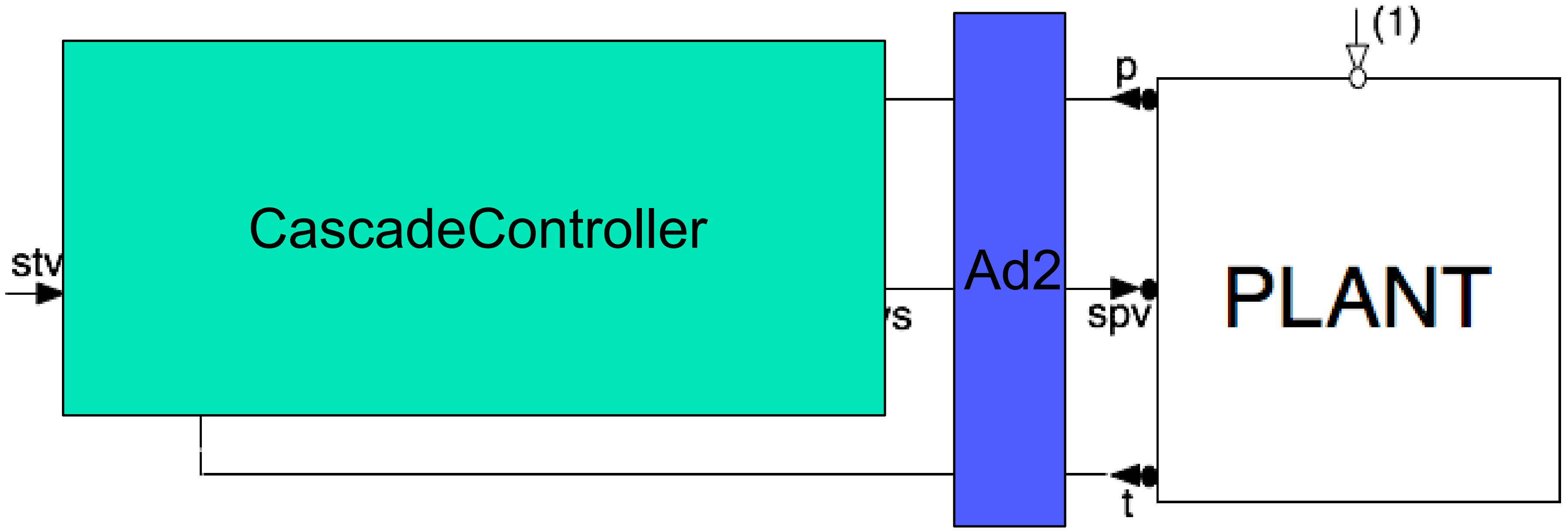


Second solution… again

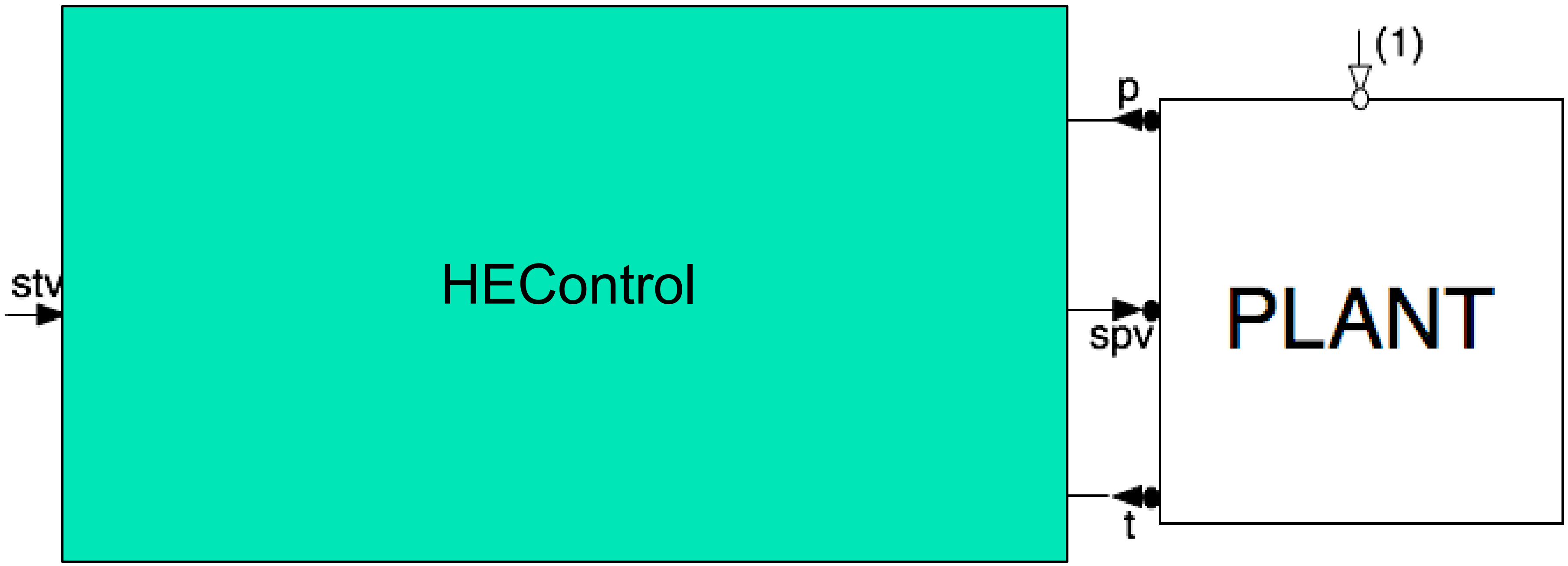


Second solution… again

Step 2



Second solution… again



Conclusion and future works

- What’s good:
  - automatic derivation of the correct assembly code
  - by implementing a DLiPA compiler and the adaptor synthesis algorithm the entire approach can be automated being integrated with TINA and CADP toolboxes
    - TINA is for PNs analysis (coverability graph and its properties);
    - CADP for automata analysis (minimization, *tau*-reduction, bi-simulation);
  - the approach may be carried on incrementally hence allowing the architect to manage the system complexity, although the approach is exponential.
- What’s bad:
  - more validation is needed
    - so far, only for HeatExchanger and ACC;
  - is the approach always incremental? Nope! :(
    - it is tightly coupled with the system SA;
    - it depends on the system architectural configuration (e.g., Dining

Open issues

- Beyond periodic clocks
  - in many cases, the activation frequency of a component might depend on values computed at run-time (e.g., operational modes).
- Supporting the architect while instantiating clocks
  - it would be useful to infer the n-tuple of *"allowable"* clocks (with respect to the adaptor to be built).