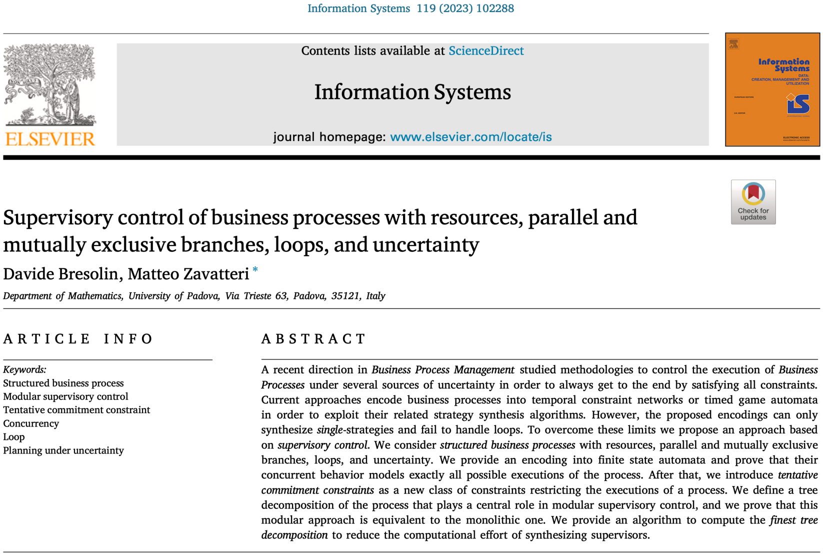
Università
degli Studi
di Padova

Supervisory control of business processes

Matteo Zavatteri

May 15, 2024

First things first



Information Systems 119 (2023) 102288

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

ELSEVIER

Supervisory control of business processes with resources, parallel and mutually exclusive branches, loops, and uncertainty

Davide Bresolin, Matteo Zavatteri *

Department of Mathematics, University of Padova, Via Trieste 63, Padova, 35121, Italy

ARTICLE INFO

Keywords:
Structured business process
Modular supervisory control
Tentative commitment constraint
Concurrency
Loop
Planning under uncertainty

ABSTRACT

A recent direction in *Business Process Management* studied methodologies to control the execution of *Business Processes* under several sources of uncertainty in order to always get to the end by satisfying all constraints. Current approaches encode business processes into temporal constraint networks or timed game automata in order to exploit their related strategy synthesis algorithms. However, the proposed encodings can only synthesize *single*-strategies and fail to handle loops. To overcome these limits we propose an approach based on *supervisory control*. We consider *structured business processes* with resources, parallel and mutually exclusive branches, loops, and uncertainty. We provide an encoding into finite state automata and prove that their concurrent behavior models exactly all possible executions of the process. After that, we introduce *tentative commitment constraints* as a new class of constraints restricting the executions of a process. We define a tree decomposition of the process that plays a central role in modular supervisory control, and we prove that this modular approach is equivalent to the monolithic one. We provide an algorithm to compute the *finest tree decomposition* to reduce the computational effort of synthesizing supervisors.

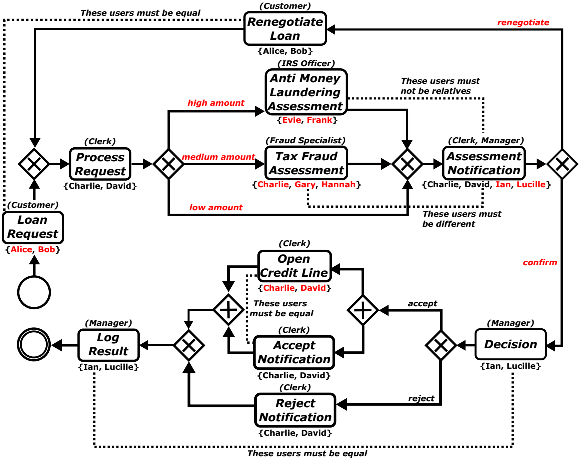
Outline

1. Context and motivation
2. The framework of supervisory control
3. Modeling of business processes as a set of finite state automata
4. Supervisory control of BPs
5. Modular synthesis

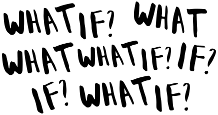
Controllability of BPs

1. Deciding which path to take under conditional and temporal uncertainty
2. Deciding which user to commit for a task under conditional uncertainty
3. Deciding which user to commit for a task under resource uncertainty
4. Deciding which user to commit for a task under conditional and temporal uncertainty

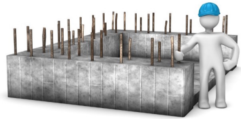
A business process



Weak, Strong and Dynamic Controllability



Weak Controllability: For any uncontrollable behavior, there exists a solution



Strong Controllability: There exists a solution working for all uncontrollable behaviors

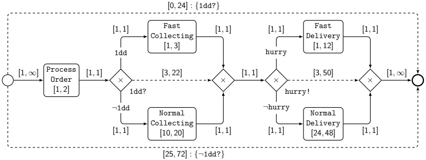
Dynamic Controllability: A solution is generated in real time depending on what is going on

Strong ⇒ Dynamic ⇒ Weak

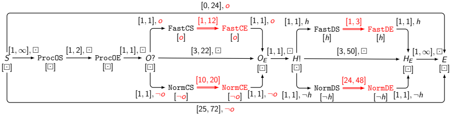
Constraint-based approaches

- Most of them based on (temporal) constraint networks and suitable for acyclic processes
- Can deal with loops by unfolding them up to maximum number of iterations (or a deadline)

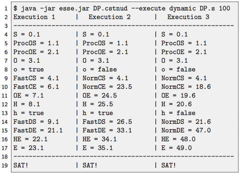
1) BP model (e.g., BPMN)



2) Temporal network encoding (e.g., CSTNUD)



3) Tool (synthesis, execution)

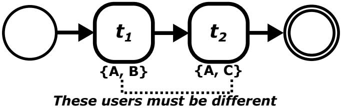


```
$ java -jar esse.jar DP.cstnud --execute dynamic DP.s 100
Execution 1          |  Execution 2          |  Execution 3

---------------------------------------------------------------
S = 0.1              | S = 0.1               | S = 0.1
ProcOS = 1.1         | ProcOS = 1.1          | ProcOS = 1.1
ProcOE = 2.1         | ProcOE = 2.1          | ProcOE = 2.1
O = 3.1              | O = 3.1               | O = 3.1
o = true             | o = false             | o = false
FastCS = 4.1         | NormCS = 4.1          | NormCS = 4.1
FastCE = 6.1         | NormCE = 23.5         | NormCE = 18.6
OE = 7.1             | OE = 24.5             | OE = 19.6
H = 8.1              | H = 25.5              | H = 20.6
h = true             | h = true              | h = false
FastDS = 9.1         | FastDS = 26.5         | NormDS = 21.6
FastDE = 21.1        | FastDE = 33.1         | NormDE = 47.0
HE = 22.1            | HE = 34.1             | HE = 48.0
E = 23.1             | E = 35.1              | E = 49.0
---------------------------------------------------------------
SAT!                 | SAT!                  | SAT!
```

New challenging features

Maximally-permissiveness



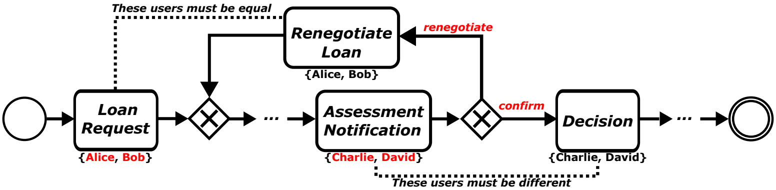
- One strategy: $t_1 = A, t_2 = C$
- Another strategy: $t_1 = B, t_2 = A$
- Yet another strategy: $t_1 = B, t_2 = C$

All strategies instead of only one.

New challenging features

Università degli Studi di Padova

Loops



- customer makes a loan request
- customer is notified of the assessment (“tentative part”)
- customer can decide to accept or renegotiate: in this last case we must decide what to do with the repeating tasks

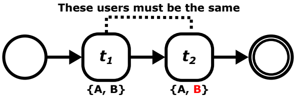
Supervisory control

Supervisory control

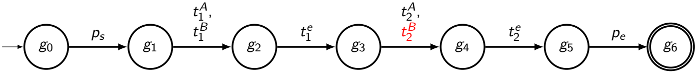
Supervisory control originated from the work of Ramadge and Wonham in the late 80s.

- Theory given for languages $\mathcal{L} \subseteq \Sigma^*$, where Σ is a set of events;
- Separation between plant G and requirements R;
- Goal is to synthesize a maximally-permissive controller S that dynamically controls G so that R is always satisfied;
- When languages $\mathcal{L}$ are regular we can employ finite state automata and related algorithms to synthesize controllers;
- Support for (un)controllable and/or (un)observable events;
- Support for non-blockingness (=executions get to the end).

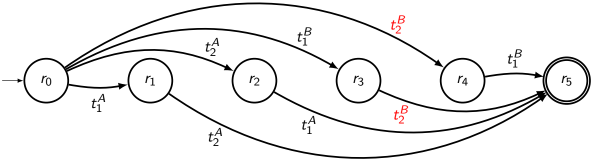
Controller synthesis example



Process P.



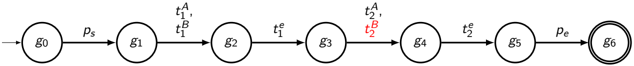
Plant G marking $K := \{p_s t_1^A t_1^e t_2^A t_2^e p_e, p_s t_1^A t_1^e t_2^B t_2^e p_e, p_s t_1^B t_1^e t_2^A t_2^e p_e, p_s t_1^B t_1^e t_2^B t_2^e p_e\}$



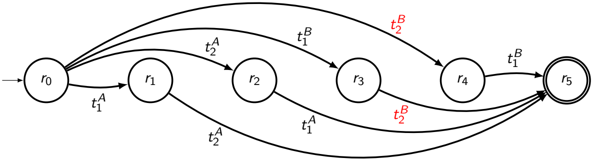
Essential requirement R marking $K_{spec} := \{t_1^A t_2^A, t_2^A t_1^A, t_1^B t_2^B, t_2^B t_1^B\}$

Controller synthesis algorithm

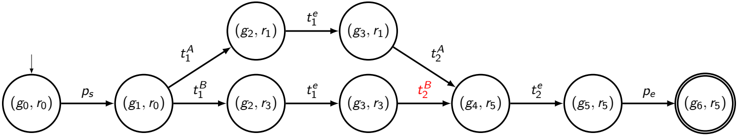
Plant G:



Requirement R:



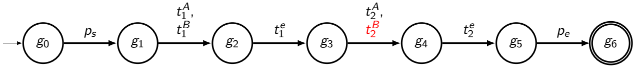
Supervisor $G \| R$:



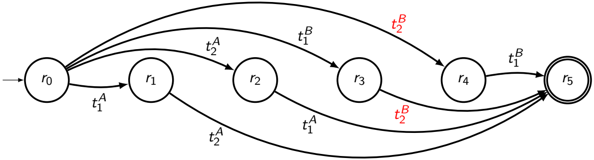
(Admissible behavior)

Controller synthesis algorithm

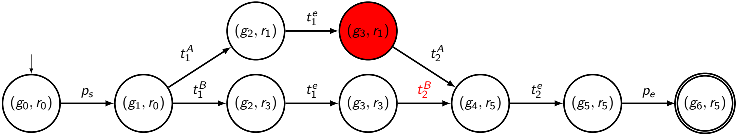
Plant G:



Requirement R:



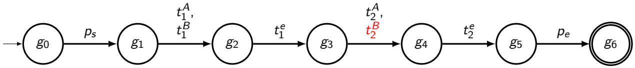
Supervisor $G \| R$:



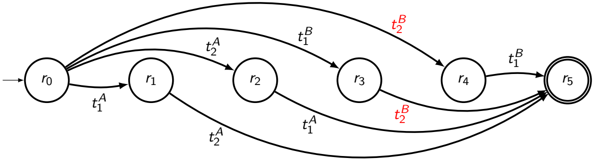
We remove (g_3, r_1) since t_2^B is executable in g_3 of P but not here.

Controller synthesis algorithm

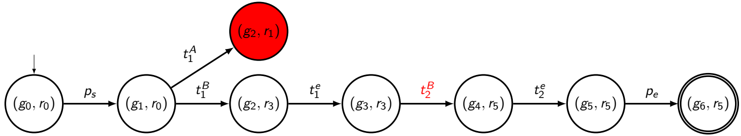
Plant G:



Requirement R:



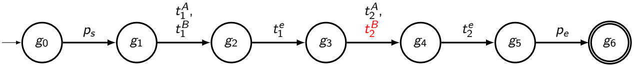
Supervisor $G \| R$:



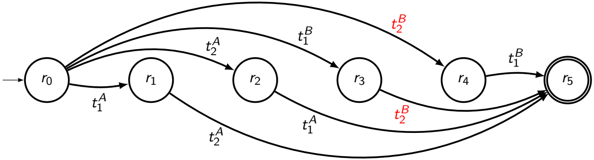
We remove (g_2, r_1) since it's a blocking state.

Controller synthesis algorithm

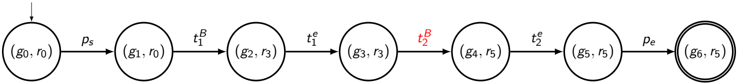
Plant G:



Requirement R:

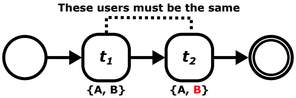


Supervisor $G \| R$:



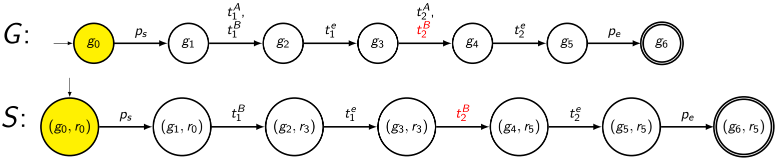
Final S marking $K^{\uparrow C} = \{p_s t_1^B t_1^e t_2^B t_2^e p_e\}$ and generating $\overline{K^{\uparrow C}}$

Supervisor deployment

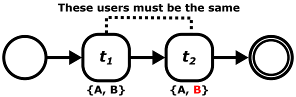


Supervisor deployment

- **runs concurrently with the plant (=$G \| S$)**
- enforces control by disabling events when appropriate (=intersection of events)

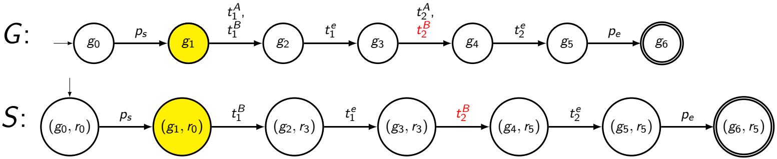


Supervisor deployment

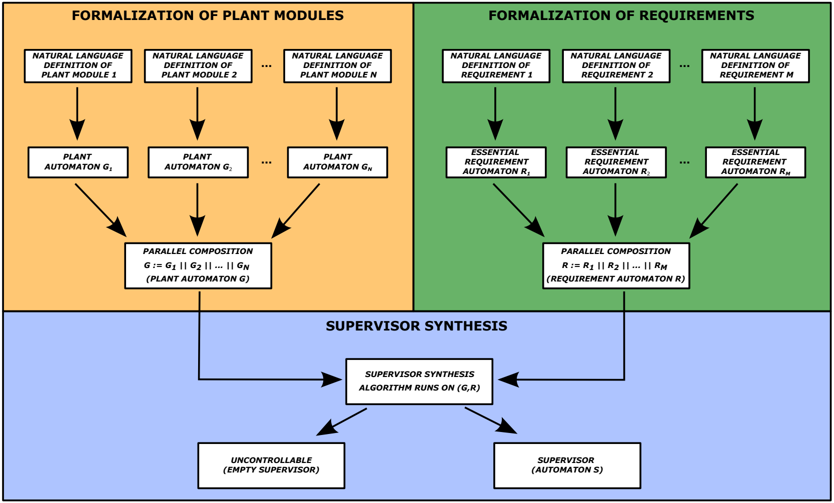


Supervisor deployment

- runs concurrently with the plant $(= G \| S)$
- **enforces control by disabling events when appropriate (=intersection of events)**



Supervisory control workflow



FORMALIZATION OF PLANT MODULES

NATURAL LANGUAGE DEFINITION OF PLANT MODULE 1

NATURAL LANGUAGE DEFINITION OF PLANT MODULE 2

...

NATURAL LANGUAGE DEFINITION OF PLANT MODULE N

PLANT AUTOMATON G_1

PLANT AUTOMATON G_2

...

PLANT AUTOMATON G_N

PARALLEL COMPOSITION
$G := G_1 \,||\, G_2 \,||\, \ldots \,||\, G_N$
(PLANT AUTOMATON G)

FORMALIZATION OF REQUIREMENTS

NATURAL LANGUAGE DEFINITION OF REQUIREMENT 1

NATURAL LANGUAGE DEFINITION OF REQUIREMENT 2

...

NATURAL LANGUAGE DEFINITION OF REQUIREMENT M

ESSENTIAL REQUIREMENT AUTOMATON R_1

ESSENTIAL REQUIREMENT AUTOMATON R_2

...

ESSENTIAL REQUIREMENT AUTOMATON R_M

PARALLEL COMPOSITION
$R := R_1 \,||\, R_2 \,||\, \ldots \,||\, R_M$
(REQUIREMENT AUTOMATON R)

SUPERVISOR SYNTHESIS

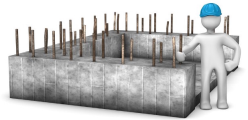
SUPERVISOR SYNTHESIS ALGORITHM RUNS ON (G,R)

UNCONTROLLABLE (EMPTY SUPERVISOR)

SUPERVISOR (AUTOMATON S)

Encoding BPs into FSA

Structure *matters*.

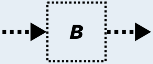


Encoding BPs into FSA

The encoding

- works at process block level;
- exploits synchronization of enter/exit events of blocks;
- encodes loops naturally;
- **controllability of events decided arbitrarily**.

Main idea is simple

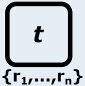


- every block must be entered and exited (if relevant)
- subblocks abstracted by only keeping their enter/exit events
- synchronization with enter/exit events of the blocks models the partial order of the BP

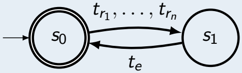
Encoding BPs into FSA

Task

Block



Automaton

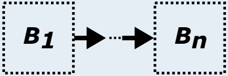


- The enter events are $E_t^{\searrow} = \{t_{r_1}, \ldots, t_{r_n}\}$;
- The exit event is $E_t^{\nearrow} = \{t_e\}$;
- No repeat events;
- Marking in initial state (=block might not be executed)

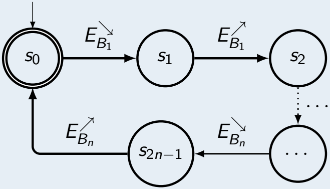
Encoding BPs into FSA

Sequence

Block



Automaton

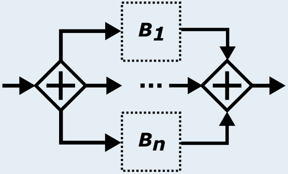


- The enter events are $E_S^{\searrow} = E_{B_1}^{\searrow}$;
- The exit events are $E_S^{\nearrow} = E_{B_n}^{\nearrow}$;
- No repeat events;
- Marking in initial state (=block might not be executed).

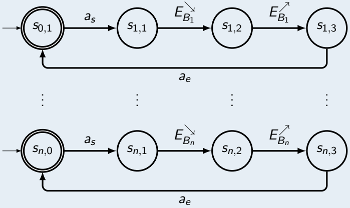
Encoding BPs into FSA

AND

Block



Set of automata



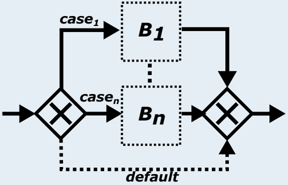
- The enter event is $E_A^{\searrow} = \{a_s\}$;
- The exit event is $E_A^{\nearrow} = \{a_e\}$;
- No repeat events.
- Marking in initial states (=block might not be executed).

* the only block encoded into more than one automaton

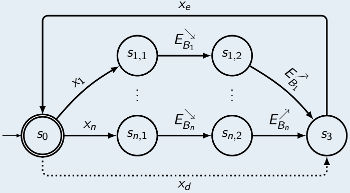
Encoding BPs into FSA

XOR

Block



Automaton

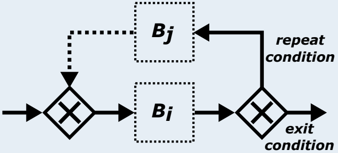


- The enter events are $E_X^{\searrow} = \{x_1, \dots, x_n, x_d\}$;
- The exit event is $E_X^{\nearrow} = \{x_e\}$;
- No repeat events;
- Marking in initial state (=block might not be executed).

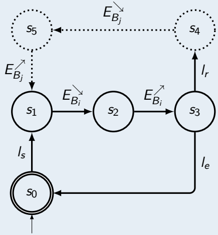
Encoding BPs into FSA

Loop

Block



Automaton

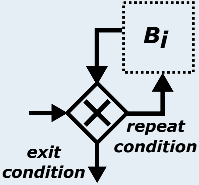


- The enter event is $E^{\searrow}_{X} = \{l_s\}$;
- The exit event is $E^{\nearrow}_{X} = \{l_e\}$;
- The repeat event is $Rep(X) = \{l_r\}$;
- Marking in initial state (=block might not be executed).

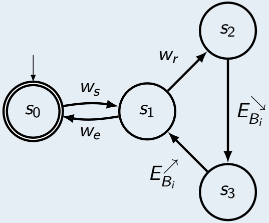
Encoding BPs into FSA

While

Block



Automaton

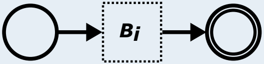


- The enter event is $E^{\searrow}_{W} = \{w_s\}$;
- The exit event is $E^{\nearrow}_{W} = \{w_e\}$;
- The repeat event is $Rep(W) = \{w_r\}$;
- Marking in initial state (=block might not be executed).

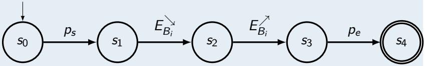
Encoding BPs into FSA

Process

Block

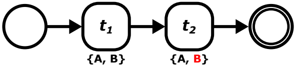


Automaton

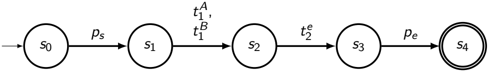


- The enter event is $E_P^{\searrow} = \{p_s\}$;
- The exit event is $E_P^{\nearrow} = \{p_e\}$;
- No repeat events;
- Marking in final state (=block must be executed).

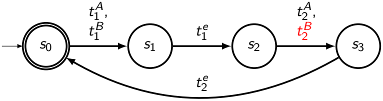
BP encoding example



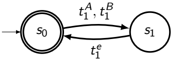
Process block G_P:



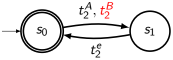
Sequence block G_S:



First Task G_{t_1} :

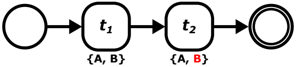


Second Task G_{t_2} :

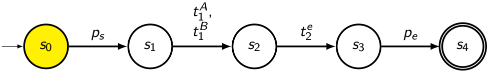


$G_P \| G_S \| G_{t_1} \| G_{t_2}$ encodes all possible unconstrained executions.

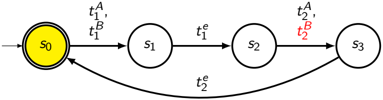
BP exeucution



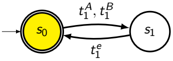
Process block G_P:



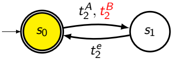
Sequence block G_S:



First Task G_{t_1} :

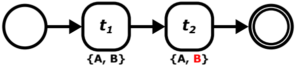


Second Task G_{t_2} :

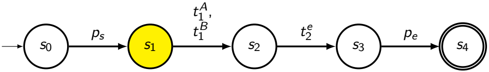


Initial state

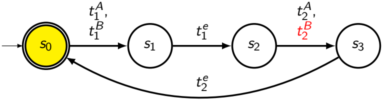
BP exeucution



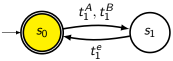
Process block G_P:



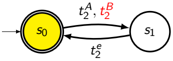
Sequence block G_S:



First Task G_{t_1} :

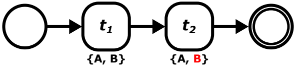


Second Task G_{t_2} :

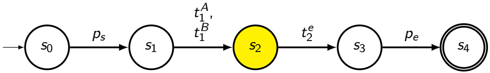


Process starts

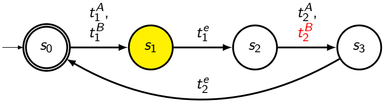
BP exeucution



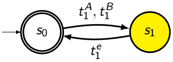
Process block G_P:



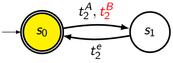
Sequence block G_S:



First Task G_{t_1} :

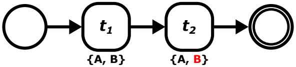


Second Task G_{t_2} :

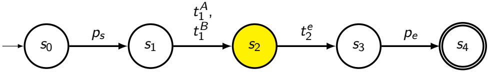


Sequence/Task t_1 starts by committing either A or B for its execution

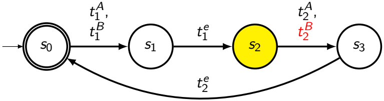
BP exeucution



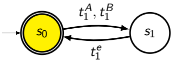
Process block G_P:



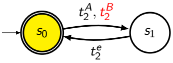
Sequence block G_S:



First Task G_{t_1} :

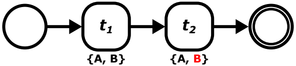


Second Task G_{t_2} :

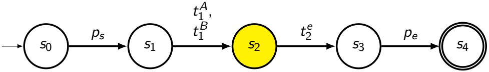


Task t_1 ends

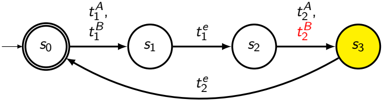
BP exeucution



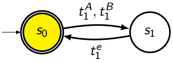
Process block G_P:



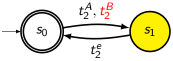
Sequence block G_S:



First Task G_{t_1} :

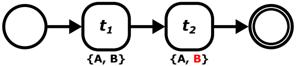


Second Task G_{t_2} :

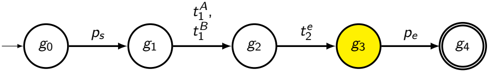


Task t_2 starts by committing either A or B for its execution

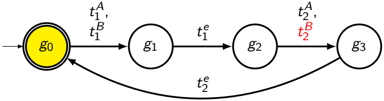
BP exeucution



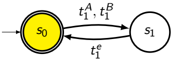
Process block G_P:



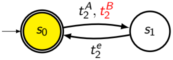
Sequence block G_S:



First Task G_{t_1} :

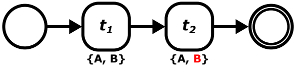


Second Task G_{t_2} :

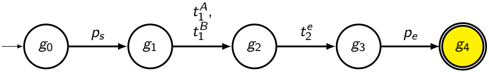


Task t_2/Sequence ends

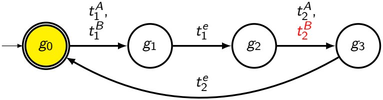
BP exeucution



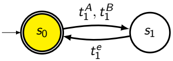
Process block G_P:



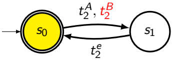
Sequence block G_S:



First Task G_{t_1} :



Second Task G_{t_2} :



Process ends

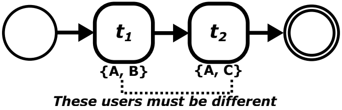
Summary of FSA encoding

Encoding BPs into FSA

- runs in linear time in the number of blocks;
- computes a set of automata whose **concurrent run** allows for all possible unconstrained executions;
- at this stage there is no need to compute parallel composition explicitly;
- the encoding is correct-by-construction. . .

. . . because structure *matters*.

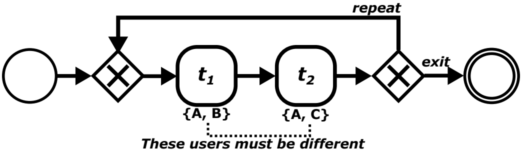
Tentative commitment constraints



Situation 1: t_1 and t_2 can't be repeated

We must assign the right users.

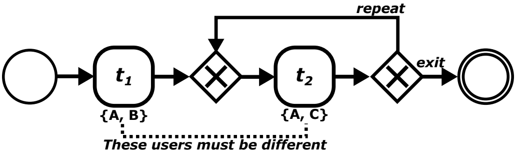
Tentative commitment constraints



Situation 2: t_1 and t_2 can both be repeated

No different from the previous, but at every iteration all user assignments are overwritten.

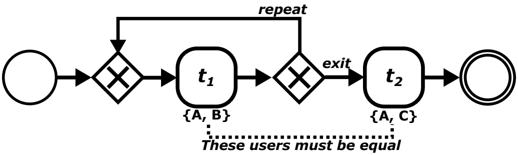
Tentative commitment constraints



Situation 3: t_1 cannot repeat, whereas t_2 can.

We can't make mistakes when assigning a user to t_2.

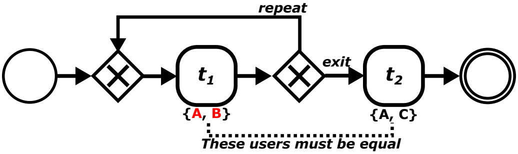
Tentative commitment constraints



Situation 4: t_1 can be repeated, whereas t_2 can't

We can exit the loop only if there is a way to extend the partial assignment to t_1 to a complete one to t_1 *and* t_2. Otherwise, we repeat and overwrite the partial assignment to t_1.

Tentative commitment constraints

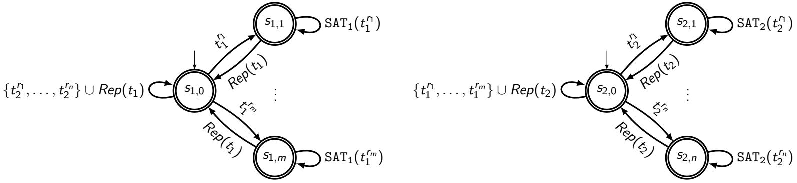


Tentative commitment constraints

- are relational constraints evaluated under activity repetition;
- are centered around the idea that by repeating they **overwrite** (partial) assignments;
- relational constraint is evaluated only when the resource assignment is complete (=users assigned to both tasks).

TCCs become more intriguing under uncertainty: what if the resource commitment to t_1 were uncontrollable?

Encoding of TCC


$s_{1,1}$
$\mathtt{SAT}_1(t_1^{r_1})$
$t_1^{r_1}$
$Rep(t_1)$
$\{t_2^{r_1},\dots,t_2^{r_n}\} \cup Rep(t_1)$
$s_{1,0}$
⋮
$t_1^{r_m}$
$Rep(t_1)$
$s_{1,m}$
$\mathtt{SAT}_1(t_1^{r_m})$
$s_{2,1}$
$\mathtt{SAT}_2(t_2^{r_1})$
$t_2^{r_1}$
$Rep(t_2)$
$\{t_1^{r_1},\dots,t_1^{r_m}\} \cup Rep(t_2)$
$s_{2,0}$
⋮
$t_2^{r_n}$
$Rep(t_2)$
$s_{2,n}$
$\mathtt{SAT}_2(t_2^{r_n})$

(a) Automaton R_1.

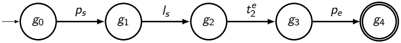
(b) Automaton R_2.

Encoding a TCC in two automata

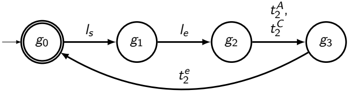
- Two automata (we don't know which task is executed first).
- If $t_1^{r_i}$ is executed (r_i is committed for t_1), then $\mathtt{SAT}_1(t_1^{r_i})$ is the set of resource commitments allowed for t_2. Similarly for t_2.
- $Rep(t_i)$ is the set of repeating events for t_i (=reset of the current tentative assignment).

BP+TCC encoding example

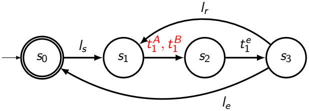
P:



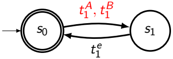
S:



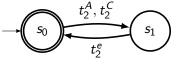
G_L :

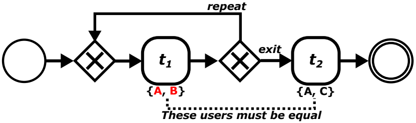


G_{t_1} :



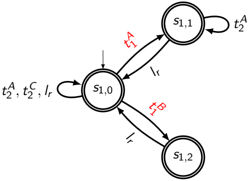
G_{t_2} :





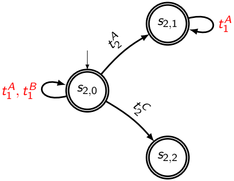
If t_1 executes first:

R_1 :



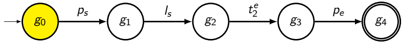
If t_2 executes first (never happens):

R_2 :

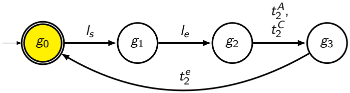


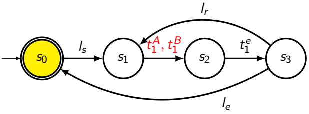
BP+TCC execution example

P:

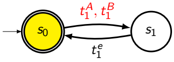


S:

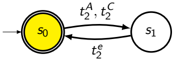




G_L :

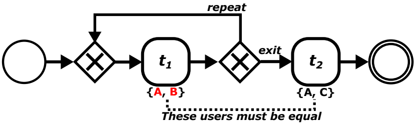


G_{t_1} :



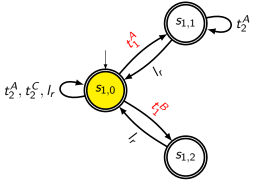
G_{t_2} :

Initial state



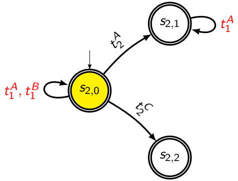
If t_1 executes first:

R_1 :



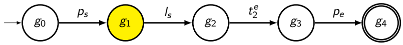
If t_2 executes first (never happens):

R_2 :

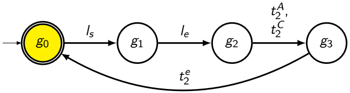


BP+TCC execution example

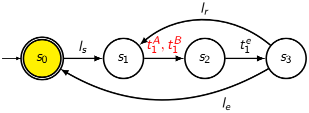
P:



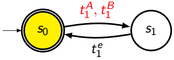
S:



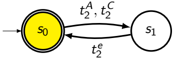
G_L :



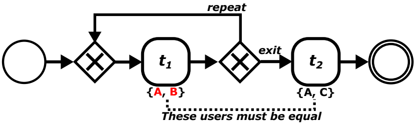
G_{t_1} :



G_{t_2} :

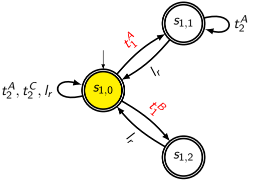


Process block starts



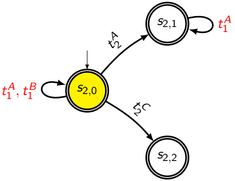
If t_1 executes first:

R_1 :

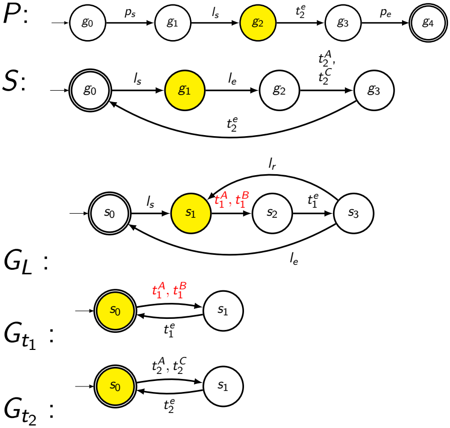


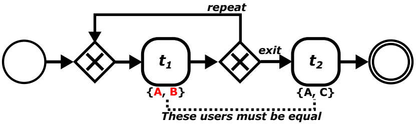
If t_2 executes first (never happens):

R_2 :

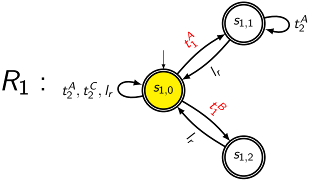


BP+TCC execution example

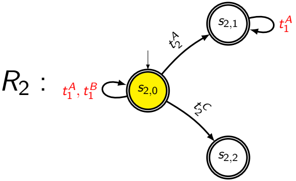

P:
g_0 $\xrightarrow{p_s}$ g_1 $\xrightarrow{l_s}$ g_2 $\xrightarrow{t_2^e}$ g_3 $\xrightarrow{p_e}$ g_4
S:
g_0 $\xrightarrow{l_s}$ g_1 $\xrightarrow{l_e}$ g_2 $\xrightarrow{t_2^A, t_2^C}$ g_3 $\xrightarrow{t_2^e}$ g_0
G_L :
s_0 $\xrightarrow{l_s}$ s_1 $\xrightarrow{t_1^A, t_1^B}$ s_2 $\xrightarrow{t_1^e}$ s_3
l_r
l_e
G_{t_1} :
s_0 $\xrightarrow{t_1^A, t_1^B}$ s_1 $\xrightarrow{t_1^e}$ s_0
G_{t_2} :
s_0 $\xrightarrow{t_2^A, t_2^C}$ s_1 $\xrightarrow{t_2^e}$ s_0



If t_1 executes first:

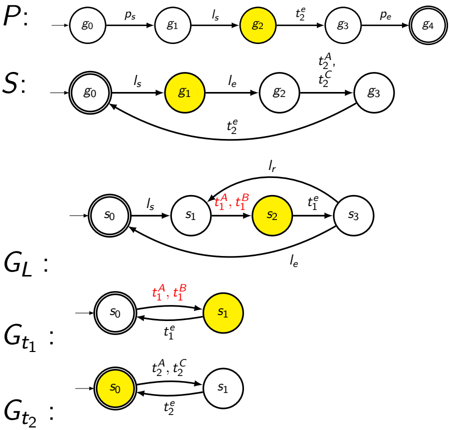


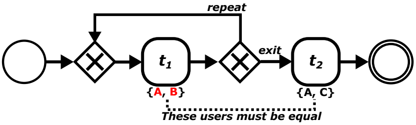
If t_2 executes first (never happens):



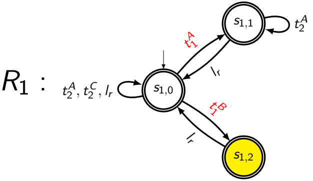
Sequence/Loop starts

BP+TCC execution example

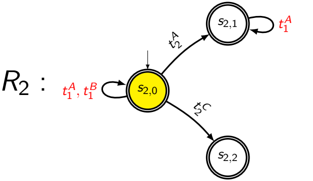




If t_1 executes first:



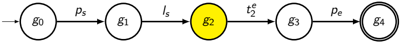
If t_2 executes first (never happens):



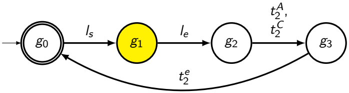
Task t_1 starts ($t_1 = B$)

BP+TCC execution example

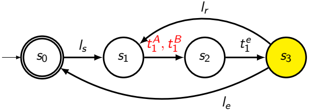
P:



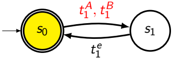
S:



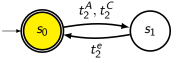
G_L :



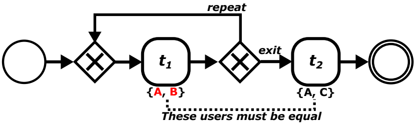
G_{t_1} :



G_{t_2} :

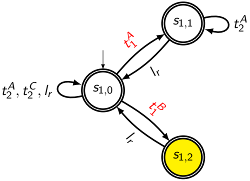


Task t_1 ends



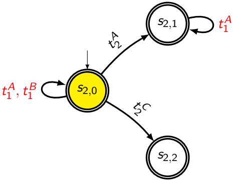
If t_1 executes first:

R_1 :



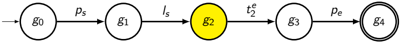
If t_2 executes first (never happens):

R_2 :

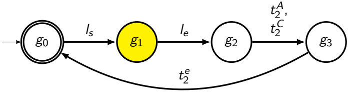


BP+TCC execution example

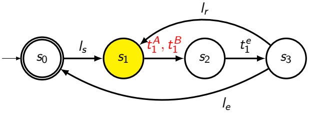
P:



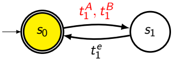
S:



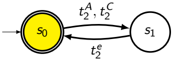
G_L :



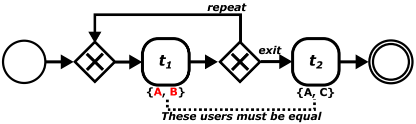
G_{t_1} :



G_{t_2} :

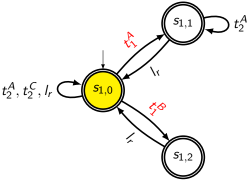


Loop repeats ($t_1 = B$ is forgotten)



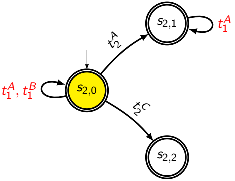
If t_1 executes first:

R_1 :



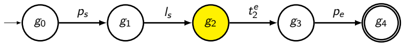
If t_2 executes first (never happens):

R_2 :

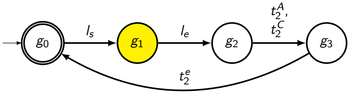


BP+TCC execution example

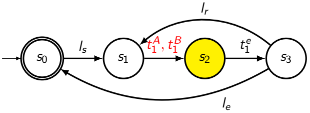
P:



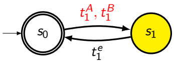
S:



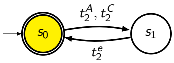
G_L :



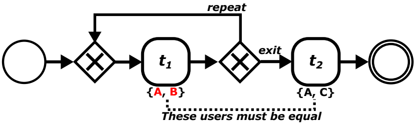
G_{t_1} :



G_{t_2} :

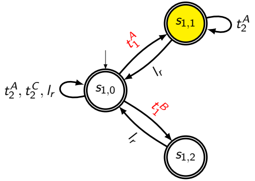


Task t_1 starts again ($t_1 = A$)



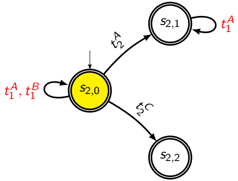
If t_1 executes first:

R_1 :



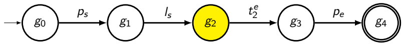
If t_2 executes first (never happens):

R_2 :

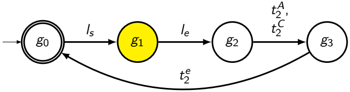


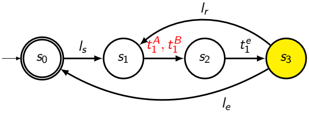
BP+TCC execution example

P:

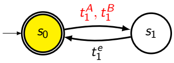


S:

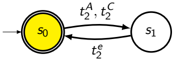




G_L :

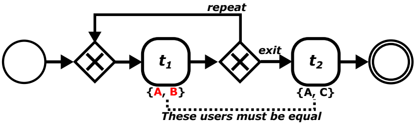


G_{t_1} :



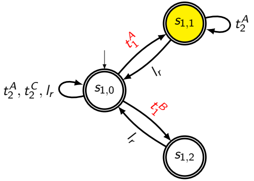
G_{t_2} :

Task t_1 ends



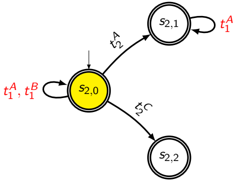
If t_1 executes first:

R_1 :



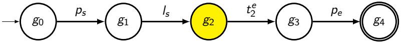
If t_2 executes first (never happens):

R_2 :

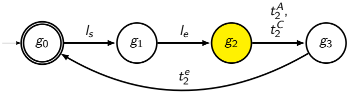


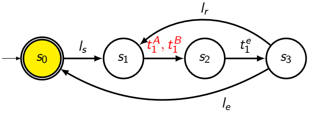
BP+TCC execution example

P:

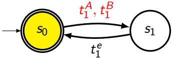


S:

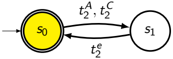




G_L :

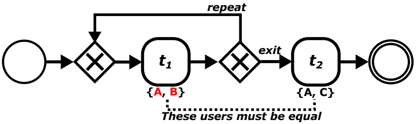


G_{t_1} :



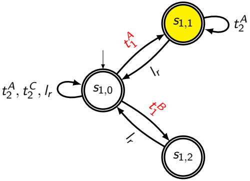
G_{t_2} :

Loop exists



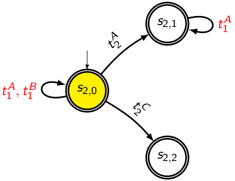
If t_1 executes first:

R_1 :



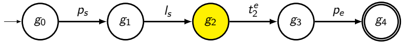
If t_2 executes first (never happens):

R_2 :

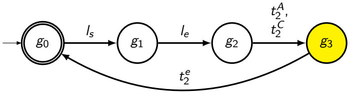


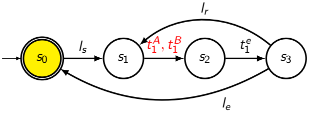
BP+TCC execution example

P:

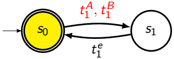


S:

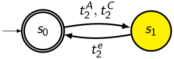




G_L :

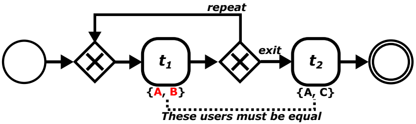


G_{t_1} :



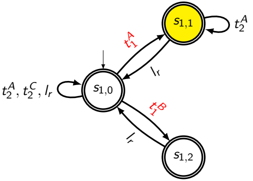
G_{t_2} :

Task t_2 starts ($t_2 = A$)



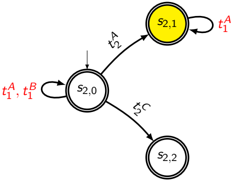
If t_1 executes first:

R_1 :



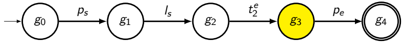
If t_2 executes first (never happens):

R_2 :

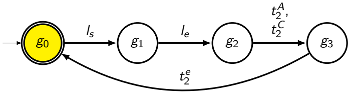


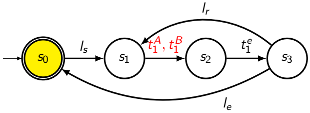
BP+TCC execution example

P:

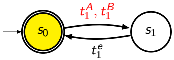


S:

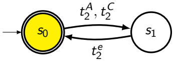




G_L :

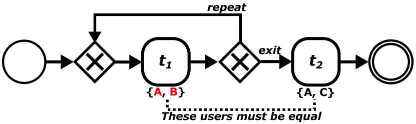


G_{t_1} :



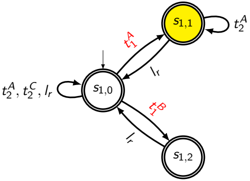
G_{t_2} :

Task t_2/Sequence ends



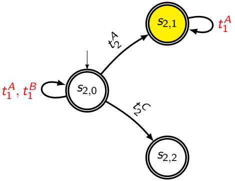
If t_1 executes first:

R_1 :



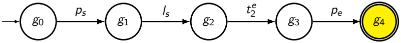
If t_2 executes first (never happens):

R_2 :

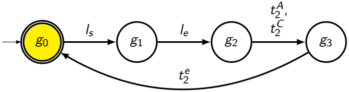


BP+TCC execution example

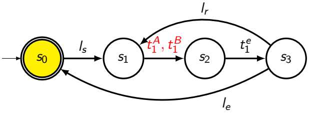
P:



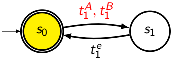
S:



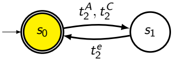
G_L :



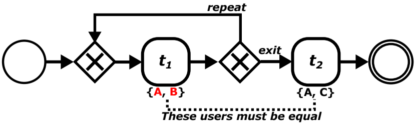
G_{t_1} :



G_{t_2} :

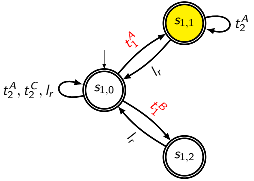


Process ends



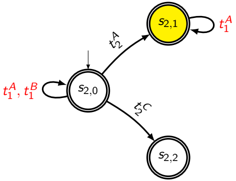
If t_1 executes first:

R_1 :

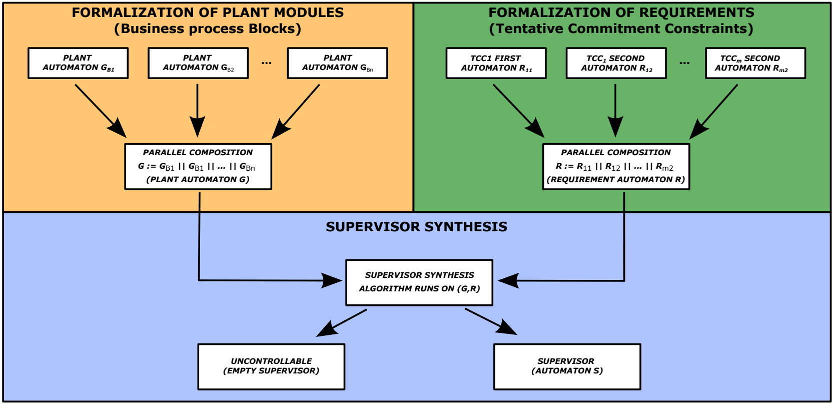


If t_2 executes first (never happens):

R_2 :



Supervisor Synthesis: Workflow



FORMALIZATION OF PLANT MODULES
(Business process Blocks)

PLANT AUTOMATON G_{B1}

PLANT AUTOMATON G_{B2}

...

PLANT AUTOMATON G_{Bn}

PARALLEL COMPOSITION
$G := G_{B1} \mid\mid G_{B1} \mid\mid \ldots \mid\mid G_{Bn}$
(PLANT AUTOMATON G)

FORMALIZATION OF REQUIREMENTS
(Tentative Commitment Constraints)

TCC1 FIRST AUTOMATON R_{11}

TCC_1 SECOND AUTOMATON R_{12}

...

TCC_m SECOND AUTOMATON R_{m2}

PARALLEL COMPOSITION
$R := R_{11} \mid\mid R_{12} \mid\mid \ldots \mid\mid R_{m2}$
(REQUIREMENT AUTOMATON R)

SUPERVISOR SYNTHESIS

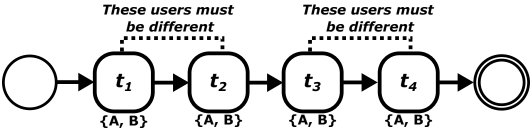
SUPERVISOR SYNTHESIS ALGORITHM RUNS ON (G,R)

UNCONTROLLABLE
(EMPTY SUPERVISOR)

SUPERVISOR
(AUTOMATON S)

Deployment of controller: $G \| S$.

Modular Supervisory Control

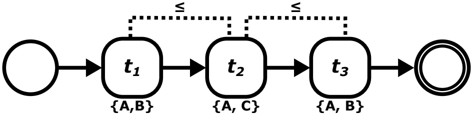


- The user assignment to t_1 and t_2 *has nothing to do* with the user assignment to t_3 and t_4;
- synthesizing a single controller is not wrong but we can do much better

Decomposition (=detecting independent parts of the BP)

- 1 controller to handle $t_1 \neq t_2$
- 1 controller to handle $t_3 \neq t_4$

What about this one?



- Assume $t_i \leq t_j$ means that the user committed to t_i must not be more expert of that committed for t_j;
- Suppose we synthesize a controller for $t_1 \leq t_2$ and another controller for $t_2 \leq t_3$.

Blocking problem!

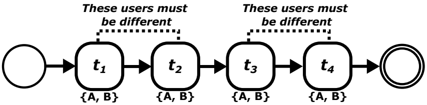
- Suppose that $t_1 = B$. Both controllers allow that assignment.
- Now, the first controller allows for the assignment $t_2 = C$, whereas the second doesn't.
- Overall, since all automata run in parallel, by synchronous composition we have that there is no assignment for t_2.

Tree decomposition

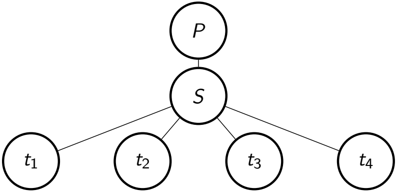
We need to understand how to decompose BPs properly.

- Every *structured* BP is a tree
- Every node of the tree matches a specific block of the BP
- Every block matches an automaton in the encoding

Process



Process tree



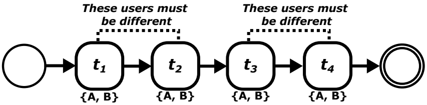
Structure *matters*.

Tree decomposition

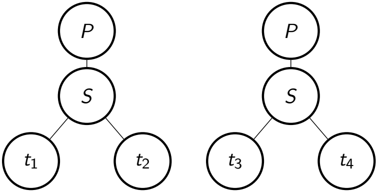
First (incomplete) approach

- Build a subtree for each TCC;
- Synthesize a controller for each tree: plant consists of all automata appearing as nodes of the tree, whereas requirements of all TCCs involving the leaves.

Process

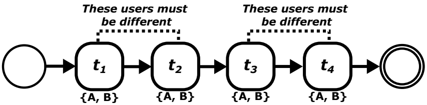


Process trees

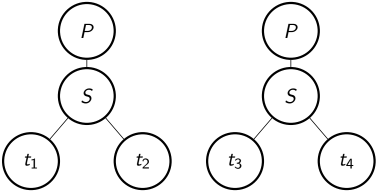


Tree decomposition

Process



Process trees



Two supervisors

- S_1 synthesized from $G_1 := G_P \| G_S \| G_{t_1} \| G_{t_2}$ and $R(t_1; t_2)$;
- S_2 synthesized from $G_2 := G_P \| G_S \| G_{t_3} \| G_{t_4}$ and $R(t_3; t_4)$;

Deployment: $\overbrace{G_P \| G_S \| G_{t_1} \| G_{t_2} \| G_{t_3} \| G_{t_4}}^{\text{plant encoding}} \quad \| \quad \overbrace{S_1 \| S_2}^{\text{all supervisors}}$

Tree decomposition

Workflow

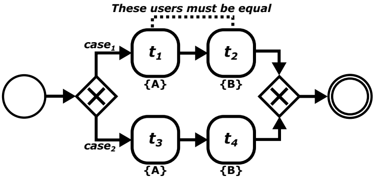
1. Find a particular set of trees that meets some properties
2. Synthesize a controller for each such tree
   - If one of these controller does not exist the whole process is uncontrollable;
   - Otherwise, we can deploy all controllers in parallel and we are equivalent to the monolithic approach.

Tree decomposition rules

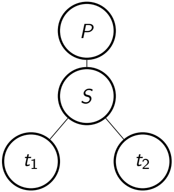
Rule 1

Every tree in the decomposition is a subtree of the process tree rooted at P

Process



Process tree



- TCC is unsatisfiable.
- If we can take the branch $case_2$ we can avoid the problem.
- If the XOR is uncontrollable, then we will detect the problem (and in case try to control before).

Tree decomposition

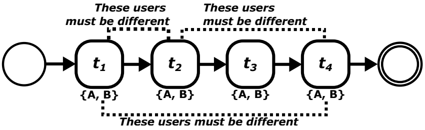
Dependent blocks

Two blocks B_1 and B_2 are dependent is there exists $t_1 \in B_1$ and $t_2 \in B_2$ such that t_1 and t_2 are involved in a TCC.

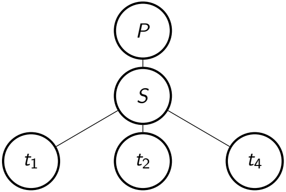
Rule 2

If two task blocks are dependent, then they belong to the same tree

Process


These users must be different
These users must be different
t1
t2
t3
t4
{A, B}
{A, B}
{A, B}
{A, B}
These users must be different

Process tree

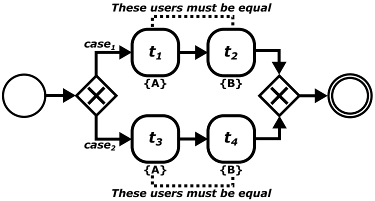


Tree decomposition

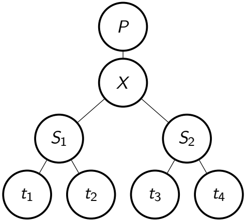
Rule 3

If all children blocks of a XOR block without default branch are dependent with another block of the process, then the XOR block belongs to exactly one tree (similarly for Loop blocks).

Process



Process tree



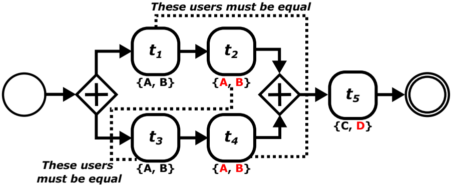
If we computed one supervisor for each branch we would get non-empty supervisors, each one saying "take the other branch".

Tree decomposition

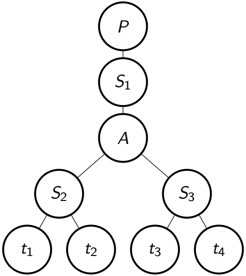
Rule 4

If two children blocks and of an AND node are dependent, then they belong to exactly one tree.

Process



Process tree



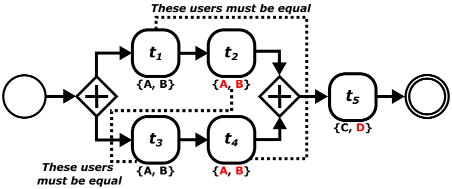
If we computed one supervisor for each branch we would get non-empty supervisors, each one trying to execute the task with uncontrollable resource commitment first.

Tree decomposition

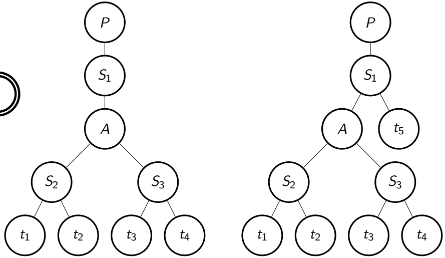
Rule 5

There are no redundant trees (=subtrees of already existing trees) in the decomposition.

Process



Process trees



Trees are "minimal". Here, the leftmost is a subtree of the rightmost (which adds nothing).

Algorithm

Finest Tree Decomposition

- creates initial trees from tasks “up to” process node following the parent relation;
- exploits strongly connected component of graphs to detect dependent blocks;
- merges trees in case some rules applies;
- at the end returns a set of (minimal) trees.

Complexity

Computing the finest tree decomposition runs in time $\mathcal{O}(M \times N)$, where M is the number of of TCCs and N is the number of blocks in the process.

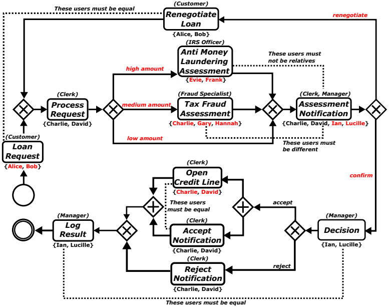
Modular supervisory synthesis

Approach

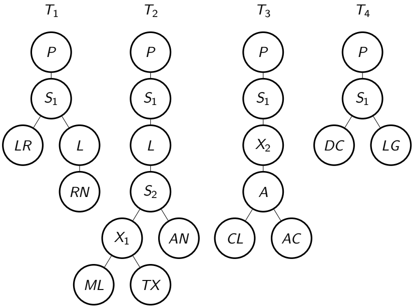
- Synthesize a controller for each tree
  - take as plant all automata corresponding to the nodes of the trees
  - take as requirement all automata of TCCs involving tasks in the trees (leaves)
- If one controller is empty, the whole process is uncontrollable.
- Otherwise, we deploy all controllers in parallel with the process.

Loan origination process

Process



Finest Tree Decomposition



We synthesize 4 controllers (one for each tree).

Conclusions

Achieved results

- Dynamic controllability under task repetitions
- Encoding from BPs into FSAs
- Tentative commitment constraints
- Supervisory control synthesis
- Maximally-permissiveness
- Tree decomposition for modular synthesis

Future work

What's next?

- Definition of more expressive classes of constraints (e.g., multi-tasking limitation)
- Definition of more complex properties on process blocks and trees to get better tree decompositions
- Development of symbolic synthesis algorithms
- Modeling of quantitative time
- Support of other kinds of uncertainty