Verification of Declarative Business Processes

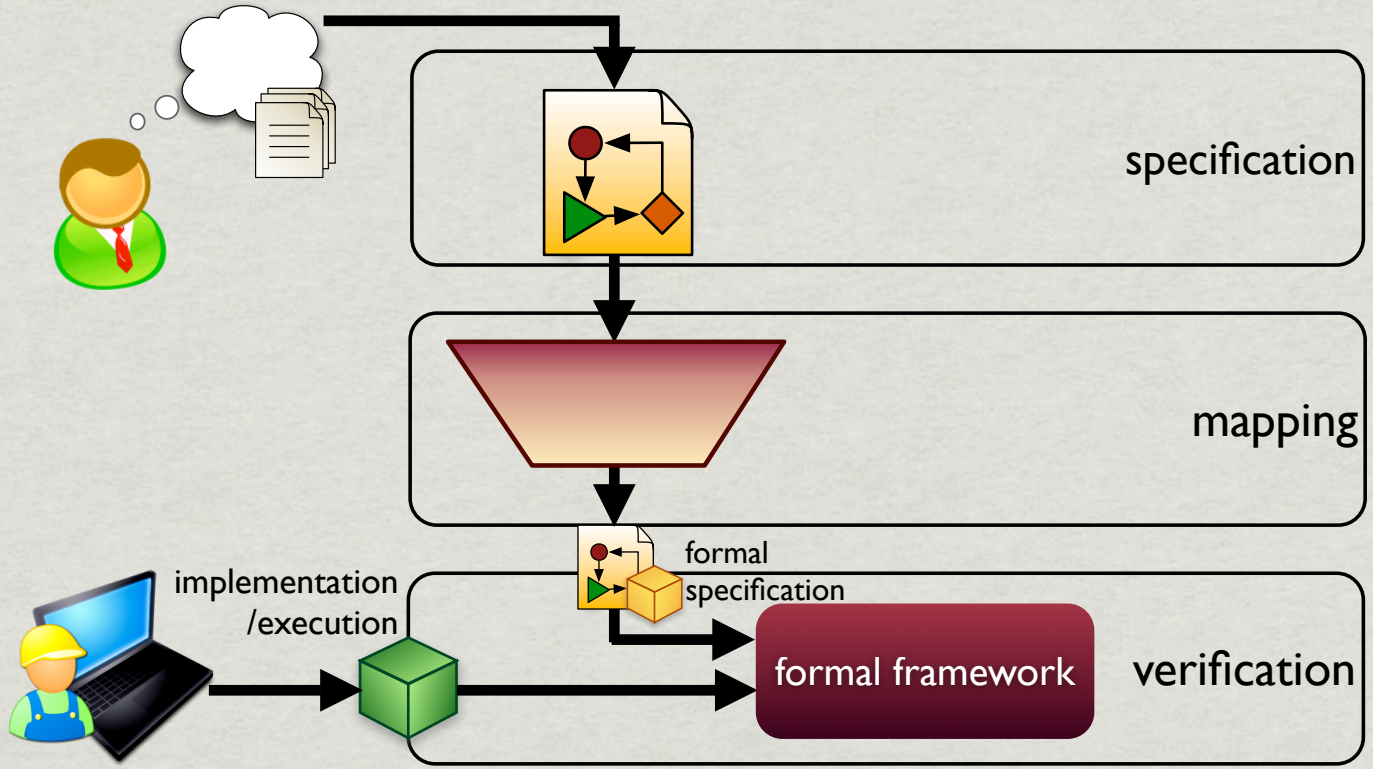
An Approach based on Computational Logic

Marco Montali

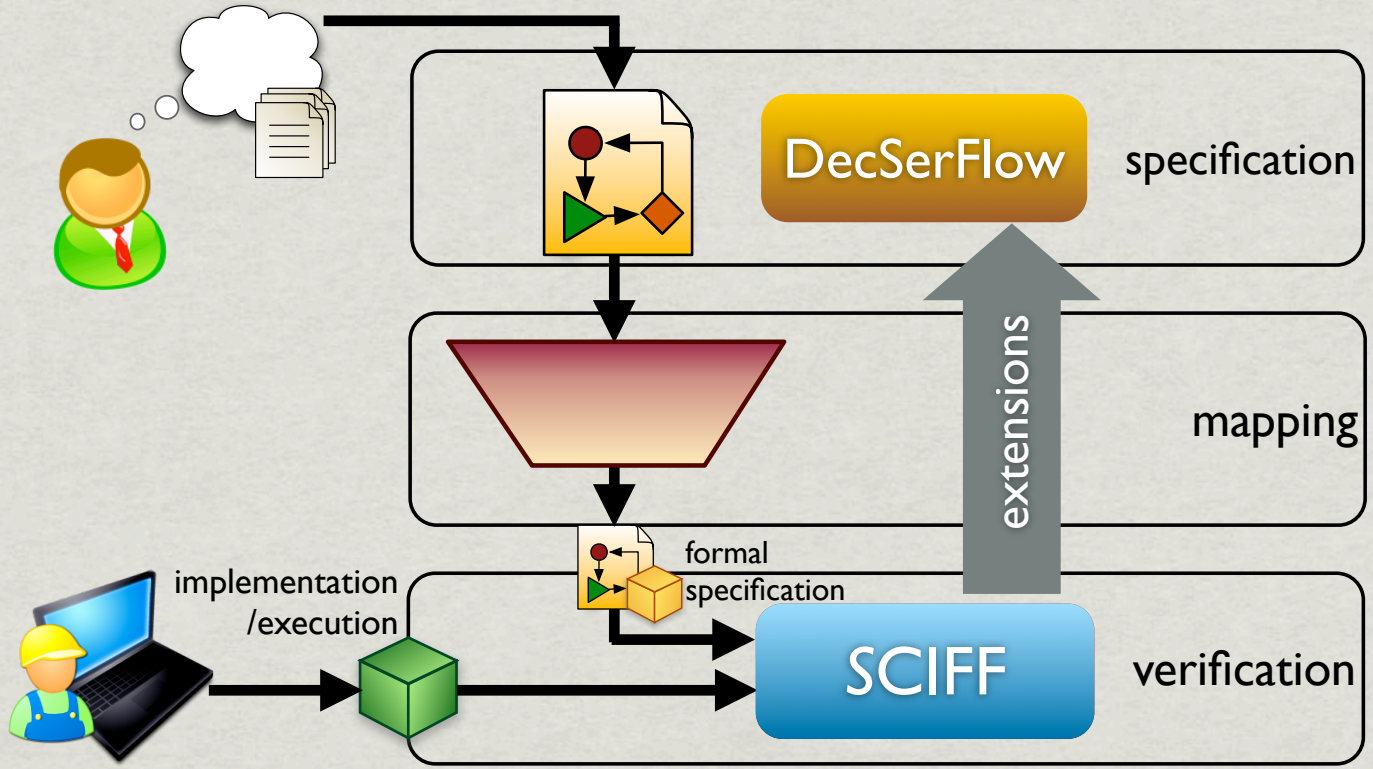
The need of declarativeness

- Different authors claims the importance of adopting **declarative** and **open** paradigms to model interaction/processes
  - van der Aalst et al.(BPM / SOA): DecSerFlow/ConDec/CigDec
  - Singh et al. (MAS): commitment approaches
  - Our work within the SOCS EU Project (MAS) and PRIN projects (together with UNITO): logic-based approaches to verify interaction
- **Declarative** models
  - capture the essential of interaction abstracting away from implementation details
  - focus on the domain problem rather than on a specific (procedural) solution
- **Open** approaches
  - participants should be let freely act where not explicitly forbidden
  - closeness undermines flexibility
- Avoiding to **over-specify** and **over-constrain** interaction is of key importance!
- Need of an underlying formal semantics (verification)

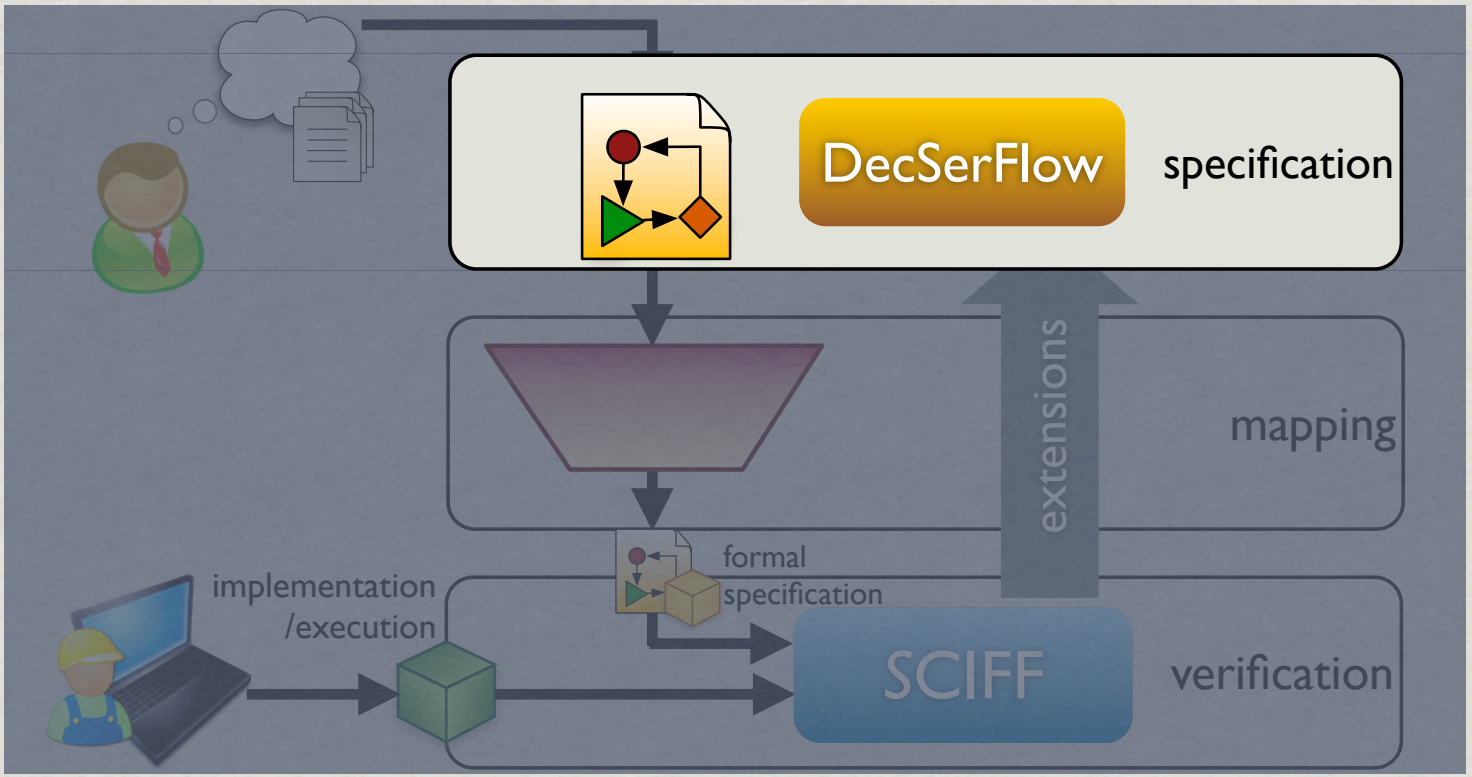
Overall schema of the framework



Overall schema of the framework



Overall schema of the framework



DecSerFlow

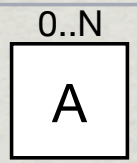
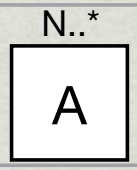
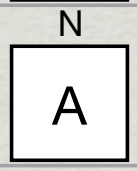
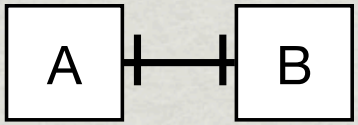
- Graphical declarative language for specifying service flows
- A DecSerFlow models is described by
  - A set of activities (messages)
  - A set of relationships among activities, a là policies / business rules
  - Relationships constrain the way activities should be performed
- Semantics of the model as the conjunction of relationships
- Different applications: CigDec, ConDec, ...

DecSerFlow relationships

- Each relationship has a semantics given in terms of Linear Temporal Logic
  - formulae on “finite traces”, because the choreography should eventually finish
- Extensible language
- Three families of core relationships
  - **Existence** formulas (similar to UML cardinalities)
  - **Relation** formulas (positive relationships between activities)
  - **Negation** formulas (negated version of relation formulas)

Existence Formulas

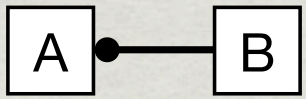
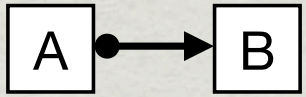
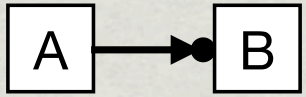
- Constraints on the number of activities execution
- Similar to UML cardinalities...

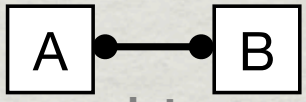
	**absence N**	A should be executed **at most** N times
	**existence N**	A should be executed **at least** N times
	**exactly N**	A should be executed **exactly** N times
	**mutual substitution**	**At least one** between A and B should be executed

Relation Formulas (1)

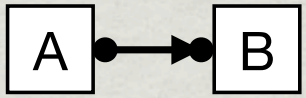
- Constrain the happening of activities w.r.t. other activities execution
- Three basic temporal orderings: no ordering, after, before
- Succession formula constructed by combining the corresponding response and precedence versions
- Number of lines determines how much the relationship is “strict”

Relation Formulas (2)

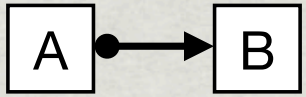
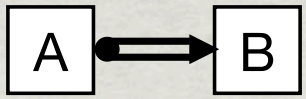
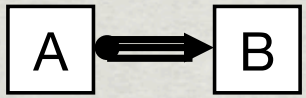
	**responded presence**	if A is executed, B should also be executed
	**response**	if A is executed, B should be executed **after** A
	**precedence**	if B is executed, A should be executed **before** B

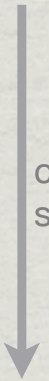


coexistence



succession

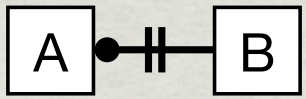
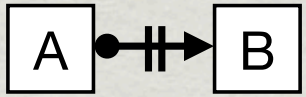
	**response**	if A is executed, B should be executed **after** A
	**alternate response**	B is **response** of A and there should exists at least one B **between** two As
	**chain response**	if A is executed, B should be executed **next** (**immediately after**)

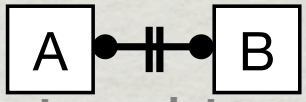


constraint strength

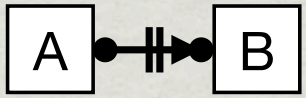
alternate and chain precedence/succession follow straightforward...

Negation Formulas

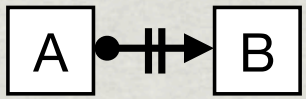
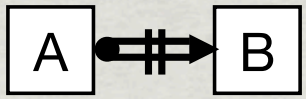
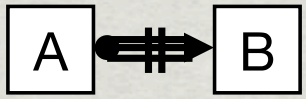
	**responded absence**	if A is executed, B can **never** be executed
	**negation response**	B can **never** be executed **after** A
	**precedence**	if B is executed, A can **never** be executed **before** B



not coexistence



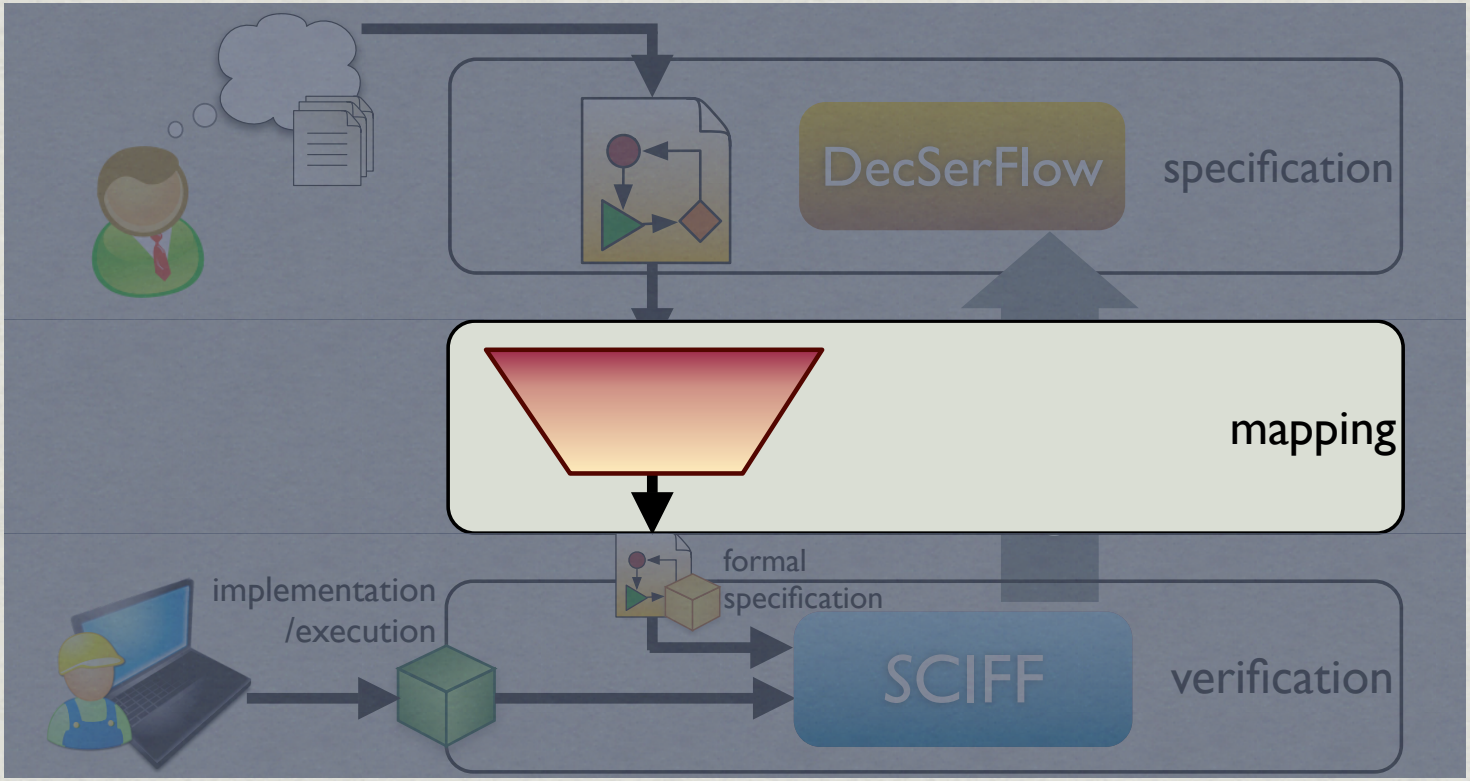
negation succession

	**negation response**	B can **never** be executed **after** A
	**negation alternate response**	There can **never** be a B **between** every two As
	**negation chain response**	B **cannot** be executed **next** to A (the sequence A B is forbidden)

constraint strength

alternate and chain precedence/succession follow straightforward...

Overall schema of the framework

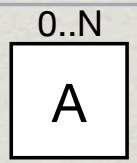
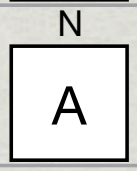
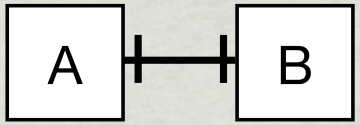


Mapping DecSerFlow onto *SCIFF*

- Very intuitive translation (close to the natural language description of DecSerFlow)

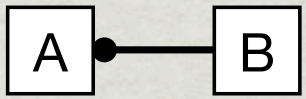
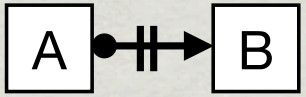
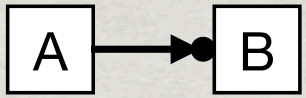
DecSerFlow	***SCIFF***
Conjunction of constraints (formulas)	Conjunction of rules (IC)
Activity	Event
Formula	(set of) IC
existence and relation formulas	rules involving positive expectations
absence and negation formulas	rules involving negative expectations

Mapping of existence formula

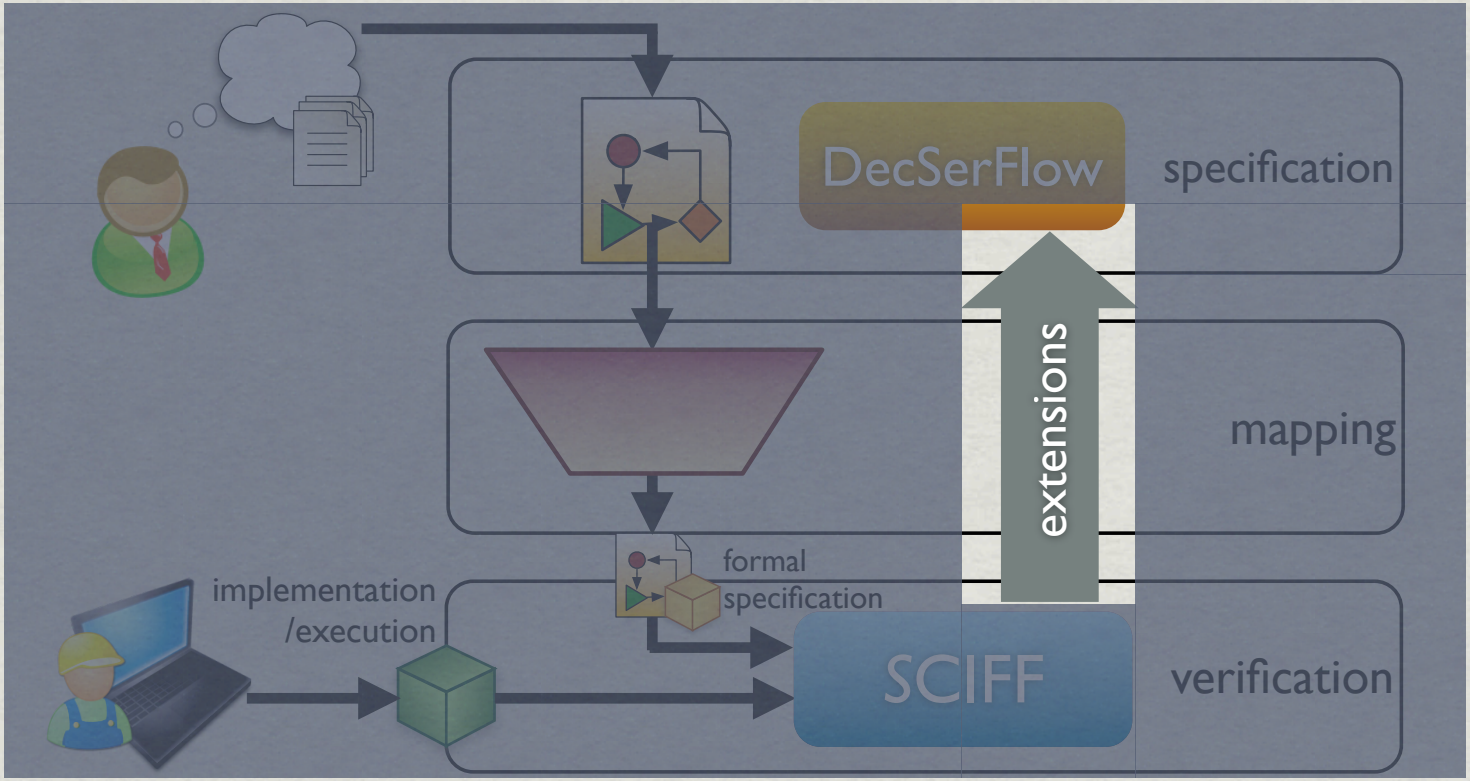
	**H**(A, T_1) /\ **H**(A, T_2) /\ $T_2 > T_1$ /\ ... /\ **H**(A, T_N) /\ $T_N > T_{N-1}$ →**EN**(A,T) /\ $T > T_N$.
	→**E**(A, T) /\ **E**(A, T_2) /\ $T_2 > T$ /\ ... /\ **E**(A, T_N) /\ $T_N > T_{N-1}$.
	combination of existence_N and absence_N formulas
	→ **E**(A, _) \/ **E**(B, _).

Mapping of relation and negation formulas

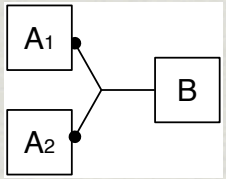
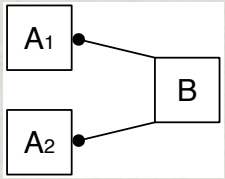
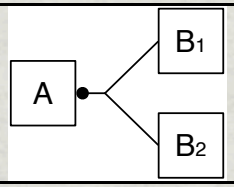
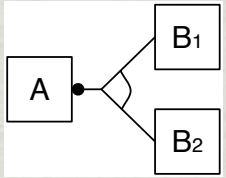
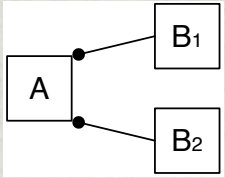
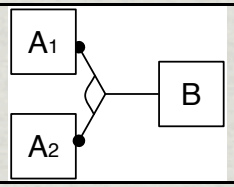
- Straightforward mapping, by interpreting the natural language description
- The mapping of negation formulas is similar, but all **E** are substituted by **EN**

	**H**(A, _) → **E**(B, _).		
	**H**(A, T_A) → **E**(B, T_B) /\ $T_B > T_A$.		**H**(A, T_A) → **EN**(B, T_B) /\ $T_B > T_A$.
	**H**(B, T_B) → **E**(A, T_A) /\ $T_A < T_B$.		
	response and **H**(A, T_{A1}) /\ **H**(A, T_{A2}) /\ $T_{A2} > T_A$ → **E**(B, T_B) /\ $T_B > T_A$ /\ $T_B < T_{A2}$		

Overall schema of the framework

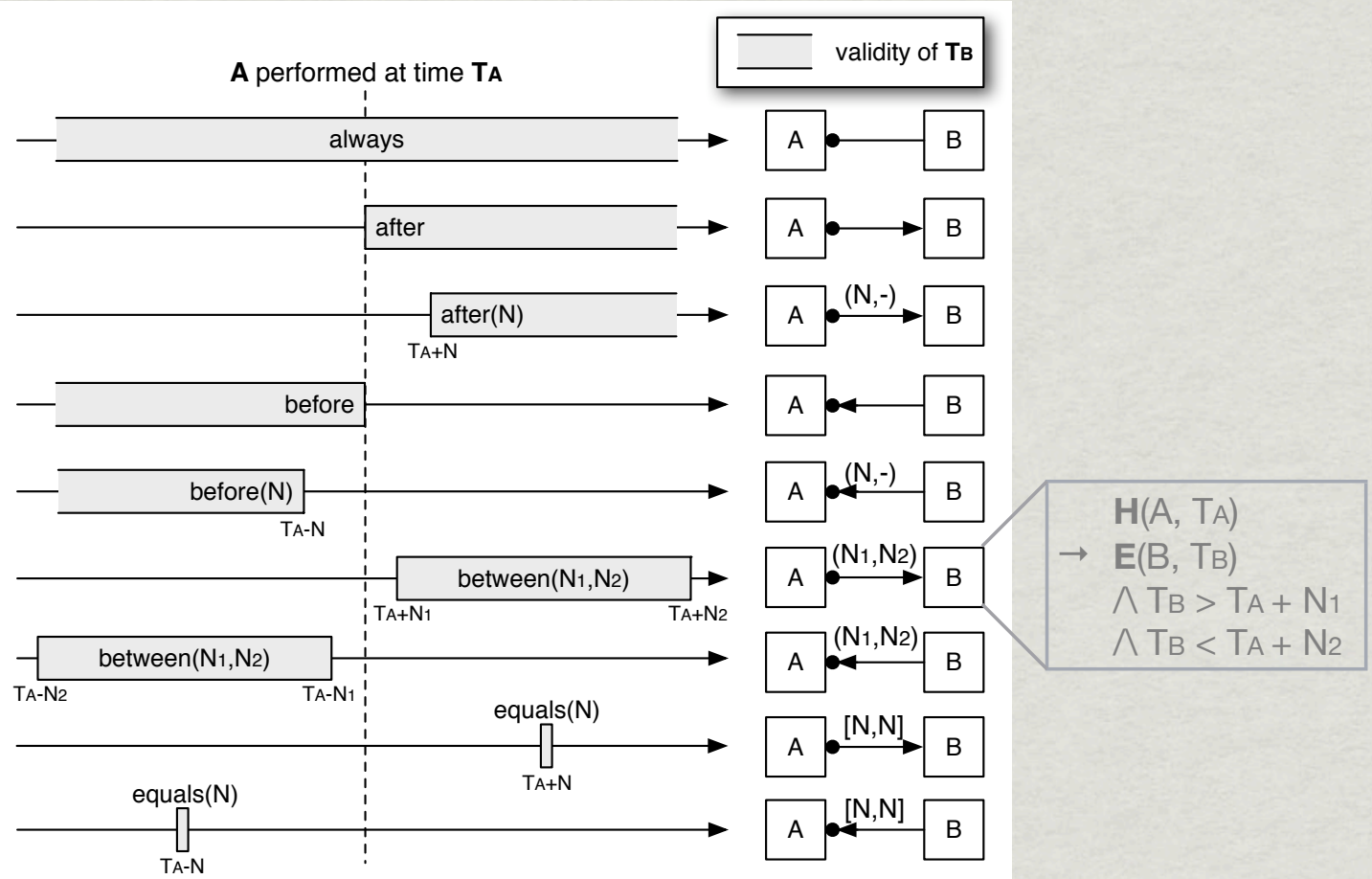


Extending branching formulas

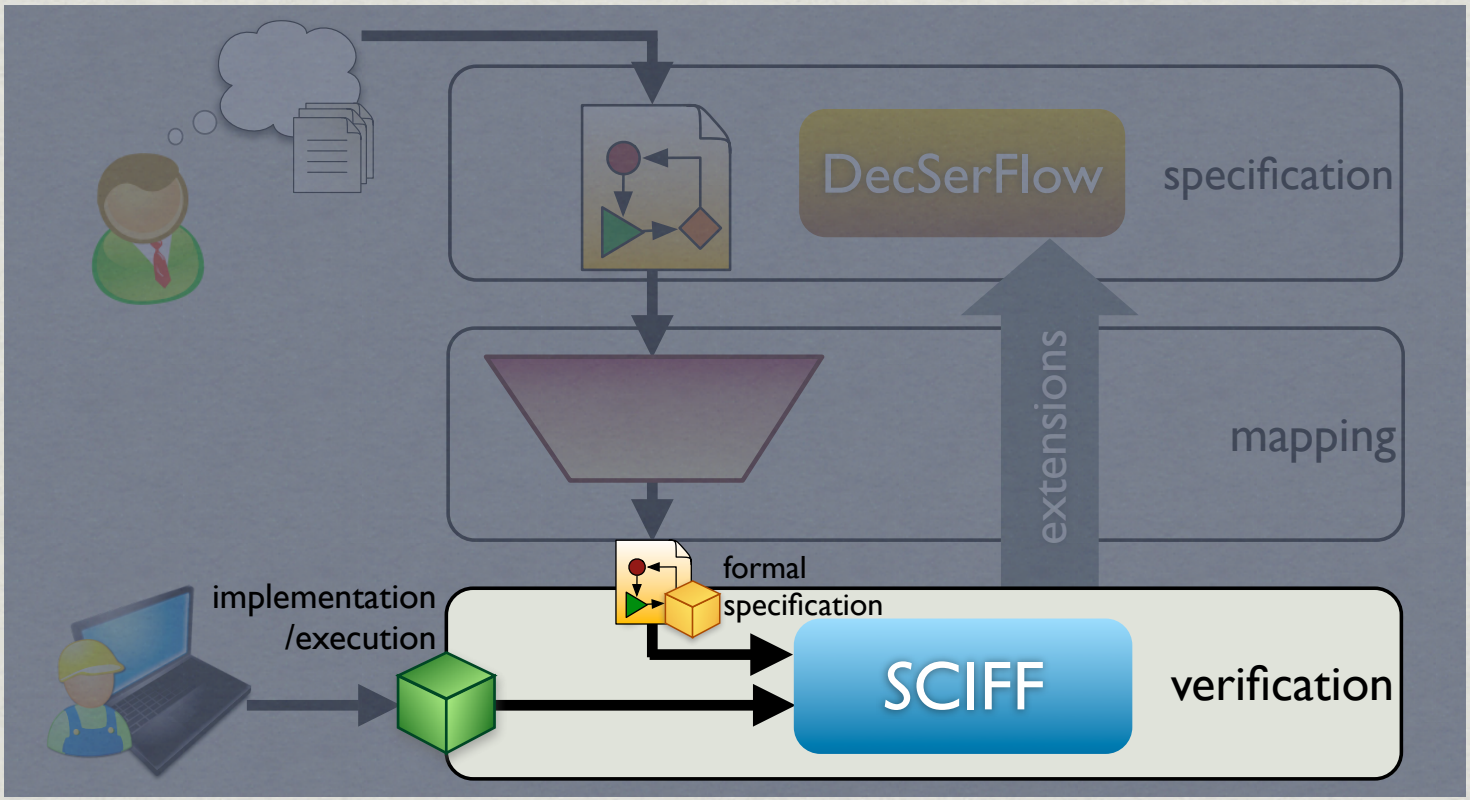
	type	representation	equivalent representation	formalization
OR JOIN	$responded\_existence$ $(A_1 \vee A_2, B)$			$\mathbf{H}(performed(A_1), T_A)$ $\rightarrow \mathbf{E}(performed(B), T_B).$ $\mathbf{H}(performed(A_2), T_A)$ $\rightarrow \mathbf{E}(performed(B), T_B).$
OR SPLIT	$responded\_existence$ $(A, B_1 \vee B_2)$			$\mathbf{H}(performed(A_1), T_A)$ $\rightarrow \mathbf{E}(performed(B_1), T_{B1})$ $\vee \mathbf{E}(performed(B_2), T_{B2}).$
AND SPLIT	$responded\_existence$ $(A, B_1 \wedge B_2)$			$\mathbf{H}(performed(A), T_A)$ $\rightarrow \mathbf{E}(performed(B_1), T_B).$ $\mathbf{H}(performed(A), T_A)$ $\rightarrow \mathbf{E}(performed(B_2), T_B).$
AND JOIN	$responded\_existence$ $(A_1 \wedge A_2, B)$			$\mathbf{H}(performed(A_1), T_{A1})$ $\wedge \mathbf{H}(performed(A_2), T_{A2})$ $\rightarrow \mathbf{E}(performed(B), T_B).$

Adding quantitative time constraints

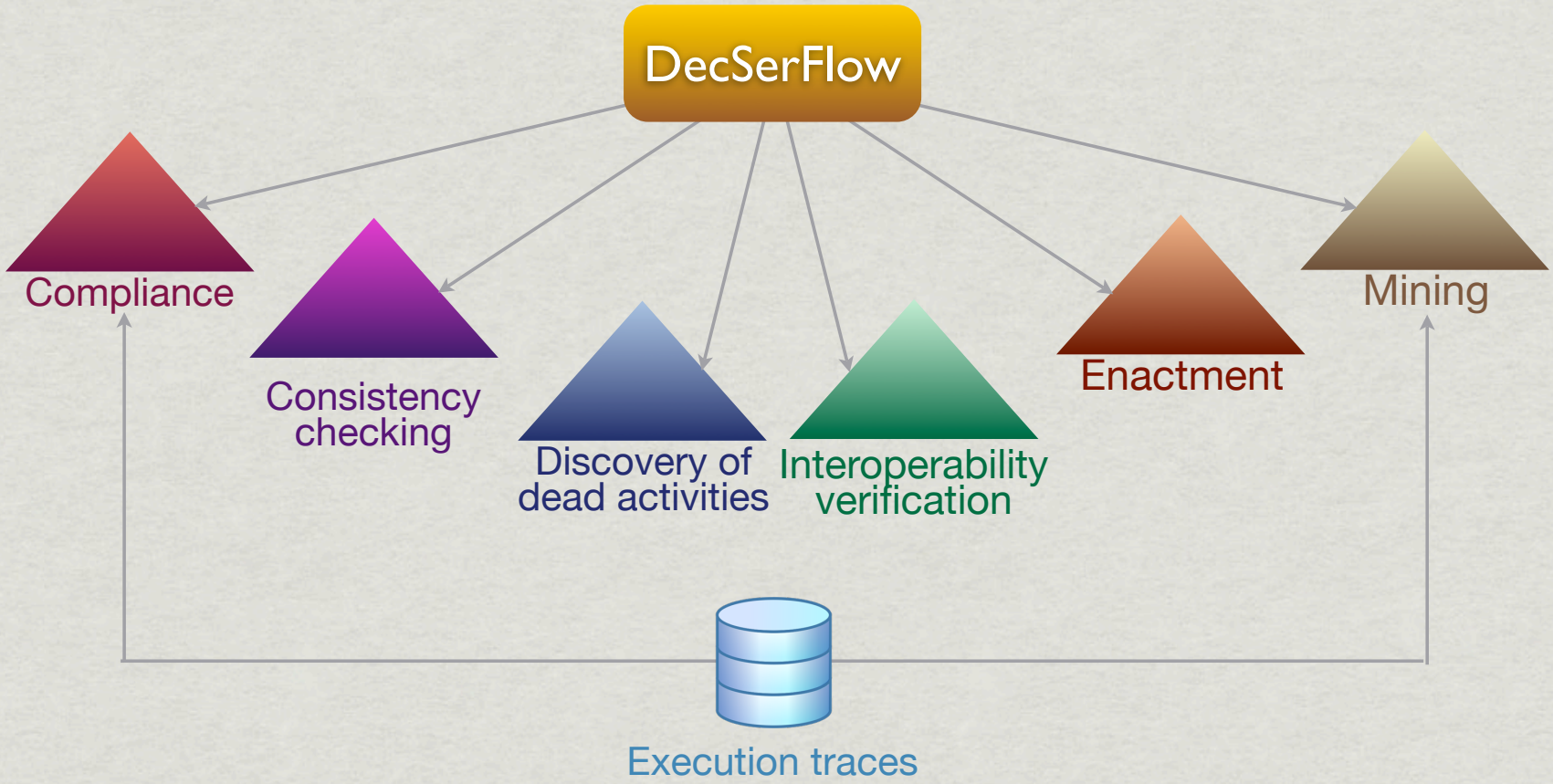
when A is performed, B should be performed at a time...



Overall schema of the framework

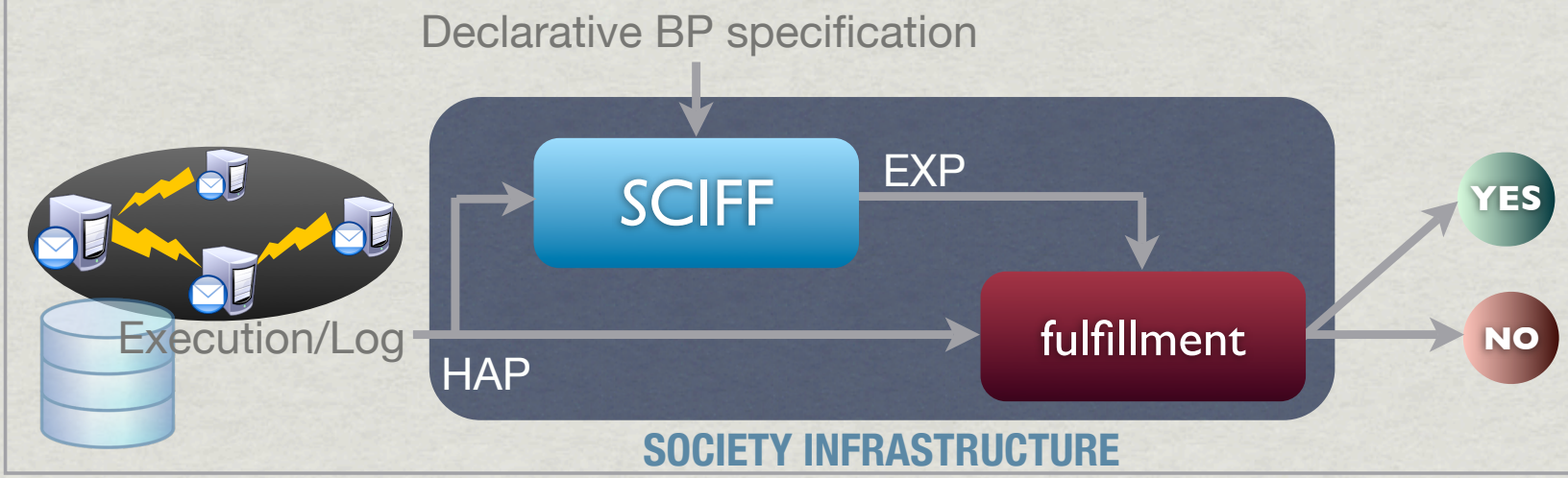


DecSerFlow verification

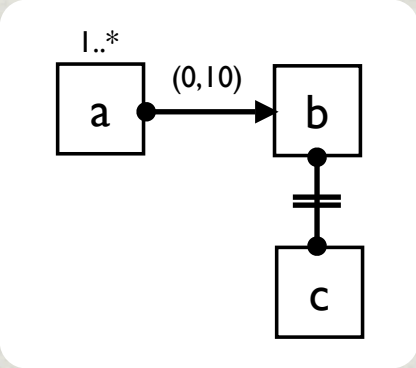


Conformance Verification - Monitoring

- SCIFF proof procedure: verify whether a set of happened events complies with the specification
  - At run-time, it dynamically performs reasoning and halts the computation waiting for further events
  - A-posteriori, by analyzing the entire execution trace of an instance



DecSerFlow Monitoring



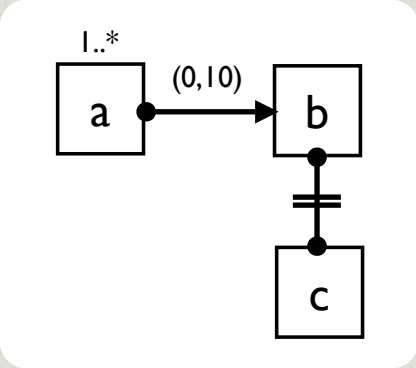
→ **E**(performed(a), Ta).
H(performed(a), Ta) → **E**(performed(b), Tb) /\ Tb > Ta /\ Tb < Ta + 10.
H(performed(b), Tb) → **EN**(performed(c), Tc). (and viceversa)

H(performed(a),1).
H(performed(b),8).

H(performed(a),1).
H(performed(b),12).

H(performed(a),1).
H(performed(b),9).
H(performed(c),15).

DecSerFlow Monitoring



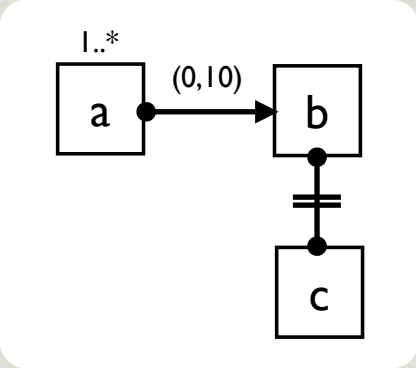
H(performed(a),1).
H(performed(a),1).
H(performed(b),8).

H(performed(a),1).
H(performed(b),12).

H(performed(a),1).
H(performed(b),9).
H(performed(c),15).

→ **E**(performed(a),Ta).	**E**(performed(a),T1).	fulfilled with T1=1	fulfilled with T1=1	fulfilled with T1=1
H(performed(a),Ta) → **E**(performed(b),Tb) /\ Tb > Ta /\ Tb < Ta + 10.				
H(performed(b),Tb) → **EN**(performed(c),Tc). (and viceversa)				

DecSerFlow Monitoring



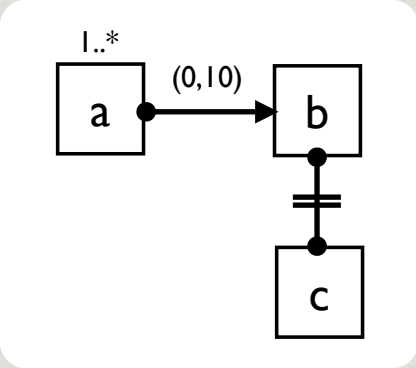
H(performed(a),1).
H(performed(a),1).
H(performed(b),8).

H(performed(a),1).
H(performed(b),12).

H(performed(a),1).
H(performed(b),9).
H(performed(c),15).

→ **E**(performed(a),Ta).	**E**(performed(a),T1).	fulfilled with T1=1	fulfilled with T1=1	fulfilled with T1=1
H(performed(a),Ta) → **E**(performed(b),Tb) /\ Tb > Ta /\ Tb < Ta + 10.	**E**(performed(b),T2) /\ T2 > 1 /\ T2 < 11.	fulfilled with T2=8	VIOLATED	fulfilled with T2=9
H(performed(b),Tb) → **EN**(performed(c),Tc). (and viceversa)				

DecSerFlow Monitoring



H(performed(a),1).
H(performed(a),1).
H(performed(b),8).

H(performed(a),1).
H(performed(b),12).

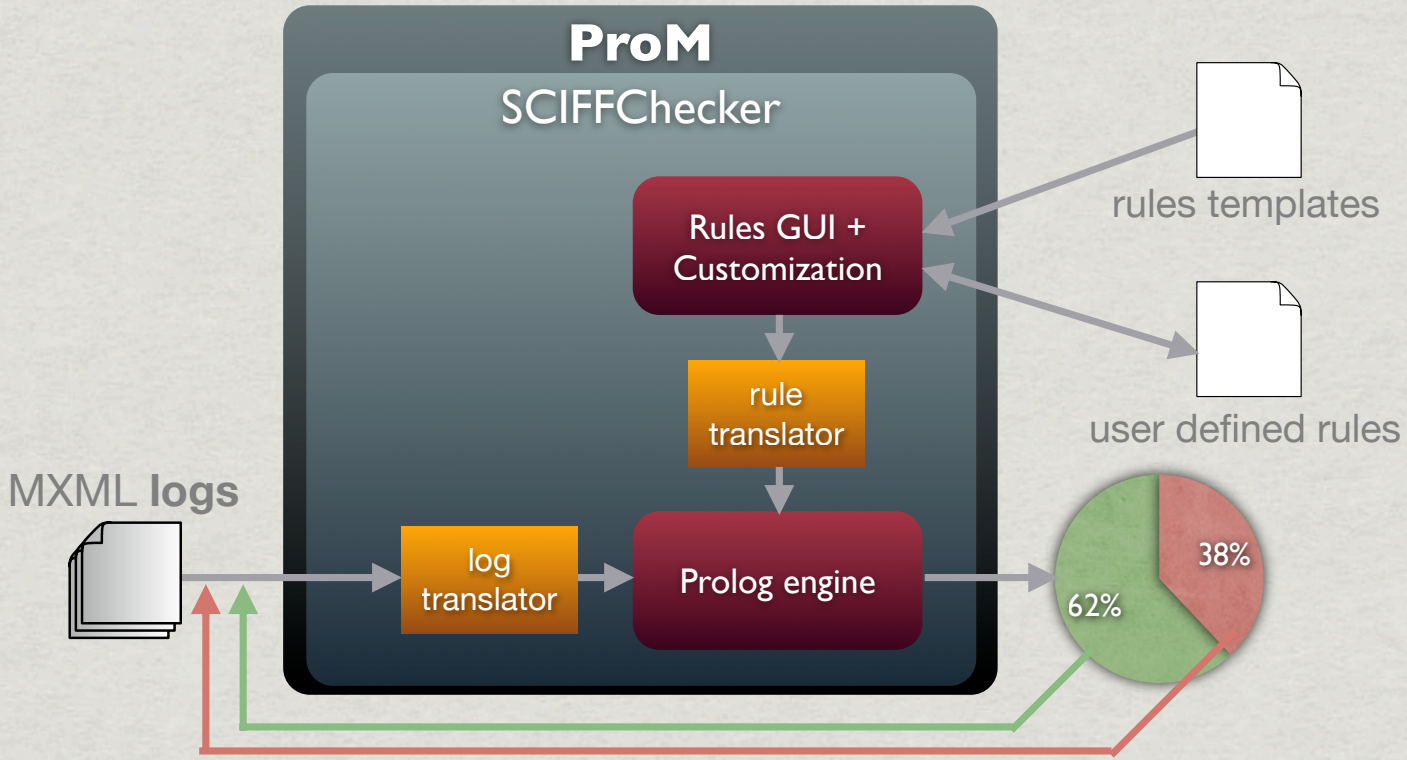
H(performed(a),1).
H(performed(b),9).
H(performed(c),15).

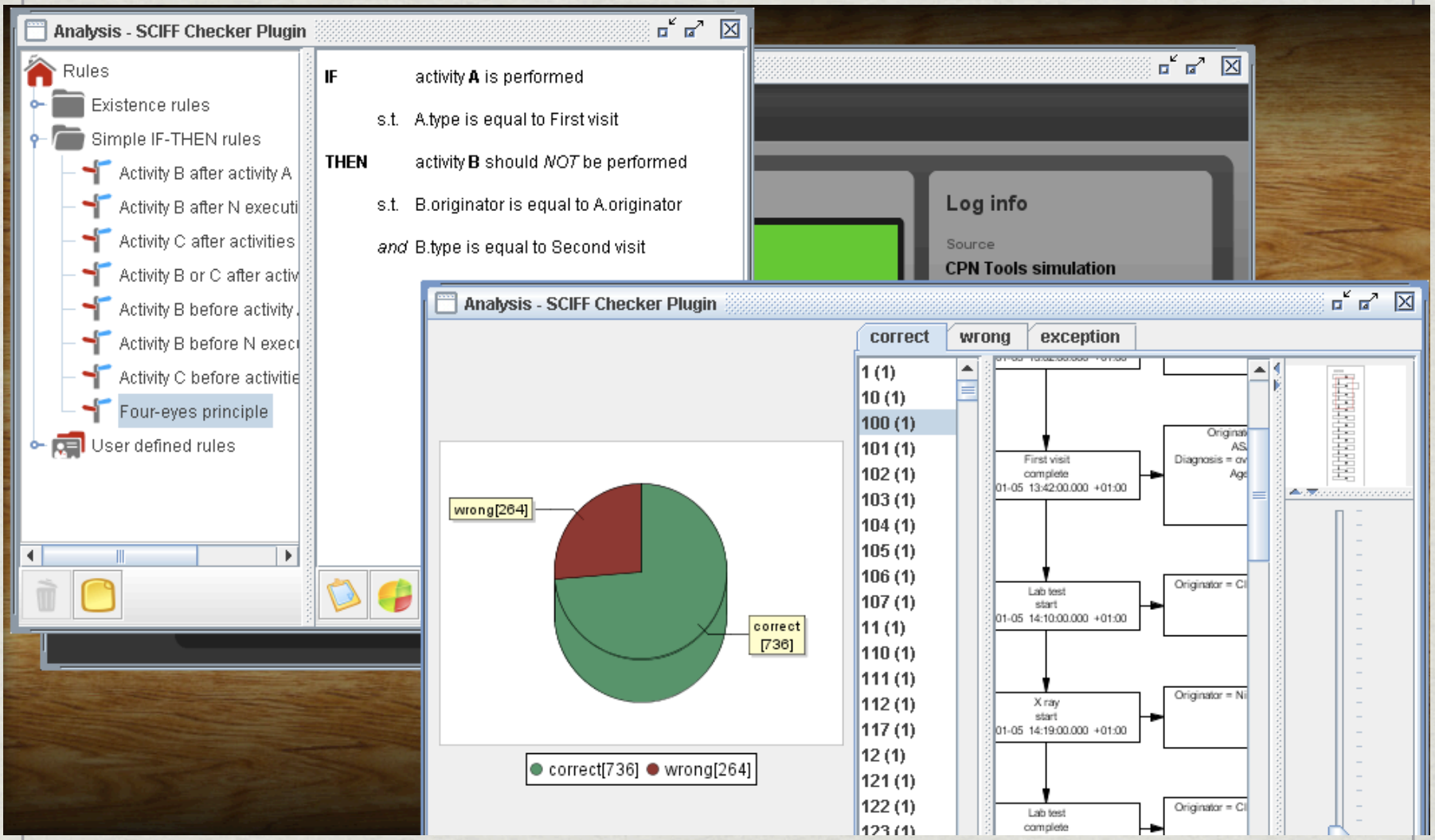
→ **E**(performed(a),Ta).	**E**(performed(a),T1).	fulfilled with T1=1	fulfilled with T1=1	fulfilled with T1=1
H(performed(a),Ta) → **E**(performed(b),Tb) /\ Tb > Ta /\ Tb < Ta + 10.	**E**(performed(b),T2) /\ T2 > 1 /\ T2 < 11.	fulfilled with T2=8	VIOLATED	fulfilled with T2=9
H(performed(b),Tb) → **EN**(performed(c),Tc). (and viceversa)	**EN**(performed(c), _).	fulfilled		VIOLATED

SCIFF-Checker

- SCIFF Checker is a ProM plug-in for performing log analysis, in the style of LTL-Checker
- It uses SCIFF-like rules to classify execution traces
- User interface which gives a user-friendly textual rules representation, and allows the user to CUSTOMIZE the rule by adding constraints between ACTIVITY TYPES, ORIGINATORS, TIMES (bot absolute or relative w.r.t. another activity type, originator, time)

Architecture



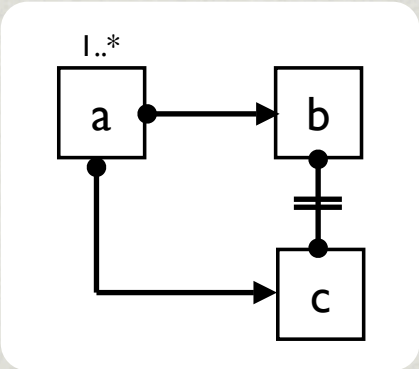

Analysis - SCIFF Checker Plugin
Rules
Existence rules
Simple IF-THEN rules
Activity B after activity A
Activity B after N executi
Activity C after activities
Activity B or C after activ
Activity B before activity
Activity B before N exec
Activity C before activitie
Four-eyes principle
User defined rules
IF activity A is performed
s.t. A.type is equal to First visit
THEN activity B should NOT be performed
s.t. B.originator is equal to A.originator
and B.type is equal to Second visit
Log info
Source
CPN Tools simulation
Analysis - SCIFF Checker Plugin
correct
wrong
exception
1 (1)
10 (1)
100 (1)
101 (1)
102 (1)
103 (1)
104 (1)
105 (1)
106 (1)
107 (1)
11 (1)
110 (1)
111 (1)
112 (1)
117 (1)
12 (1)
121 (1)
122 (1)
123 (1)
wrong[264]
correct [736]
correct[736] wrong[264]
First visit complete
01-05 13:42:00.000 +01:00
Lab test start
01-05 14:10:00.000 +01:00
X ray start
01-05 14:19:00.000 +01:00
Lab test complete

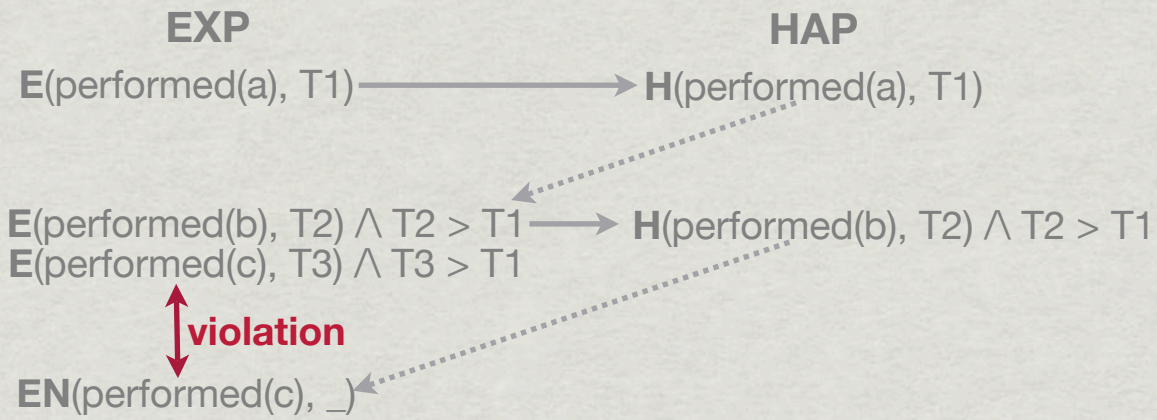
Verifying models

- To verify (extended) DecSerFlow models, we can exploit the generative variant of the *S*CIFF proof procedure
- Intuitively, it checks if a positive expectation has been fulfilled and, if this is not the case, automatically generates a (partially instantiated) happened event

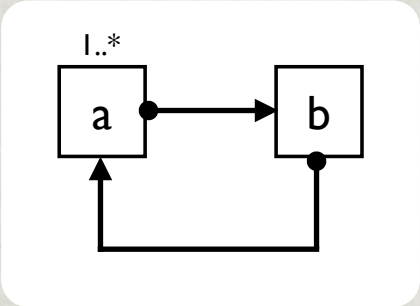
Verifying consistency

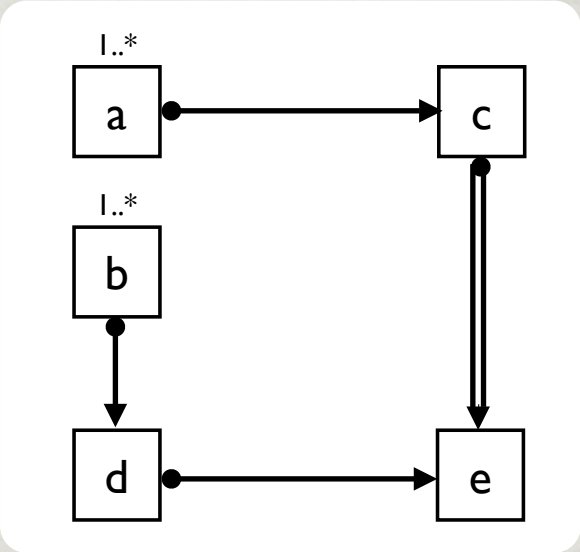
- Aim: check whether a DecSerFlow model admits at least one execution trace
- If g-SCIFF is able to find an execution trace, the model is consistent





Some experiments...





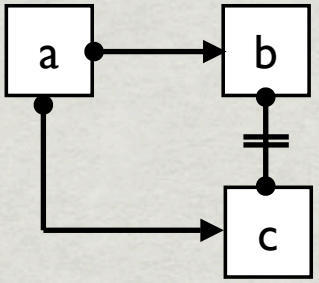
- inconsistent model
- SCIFF loops
- LTL answers immediately

- consistent model
- SCIFF answers immediately
- LTL answers in 1 minute

Discovering dead activities

- Aim: finding activities which can never be executed (i.e. discovering inconsistencies in sub-models)
- This task can be reduced to the consistency verification
- Basic algorithm:

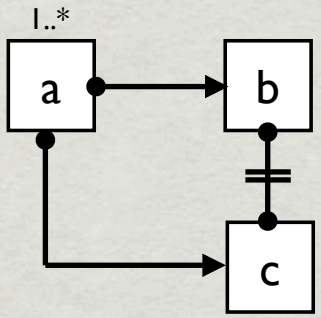
Input: $\mathcal{S}_M$, $\mathcal{S}$CIFF formalization of the DecSerFlow model $\mathcal{M}$
Output: $\mathcal{D}$, the set of dead activities
$\mathcal{D} \leftarrow \emptyset$;
foreach *Activity* $A \in \mathcal{M}$ **do**
  $\mathcal{S}'_M \leftarrow \mathcal{S}_M \cup existence\_1(A)$;
  if *call(g-*$\mathcal{S}$*CIFF(*$\mathcal{S}'_M$*)) fails* **then**
    $\mathcal{D} \leftarrow \mathcal{D} \cup A$;
  end
end



Discovering dead activities

- Aim: finding activities which can never be executed (i.e. discovering inconsistencies in sub-models)
- This task can be reduced to the consistency verification
- Basic algorithm:

Input: $\mathcal{S}_M$, $\mathcal{S}$CIFF formalization of the DecSerFlow model $\mathcal{M}$
Output: $\mathcal{D}$, the set of dead activities
$\mathcal{D} \leftarrow \emptyset$;
foreach *Activity* $A \in \mathcal{M}$ **do**
  $\mathcal{S}'_M \leftarrow \mathcal{S}_M \cup existence\_1(A)$;
  if *call(g-*$\mathcal{S}$*CIFF(*$\mathcal{S}'_M$*)) fails* **then**
    $\mathcal{D} \leftarrow \mathcal{D} \cup A$;
  end
end

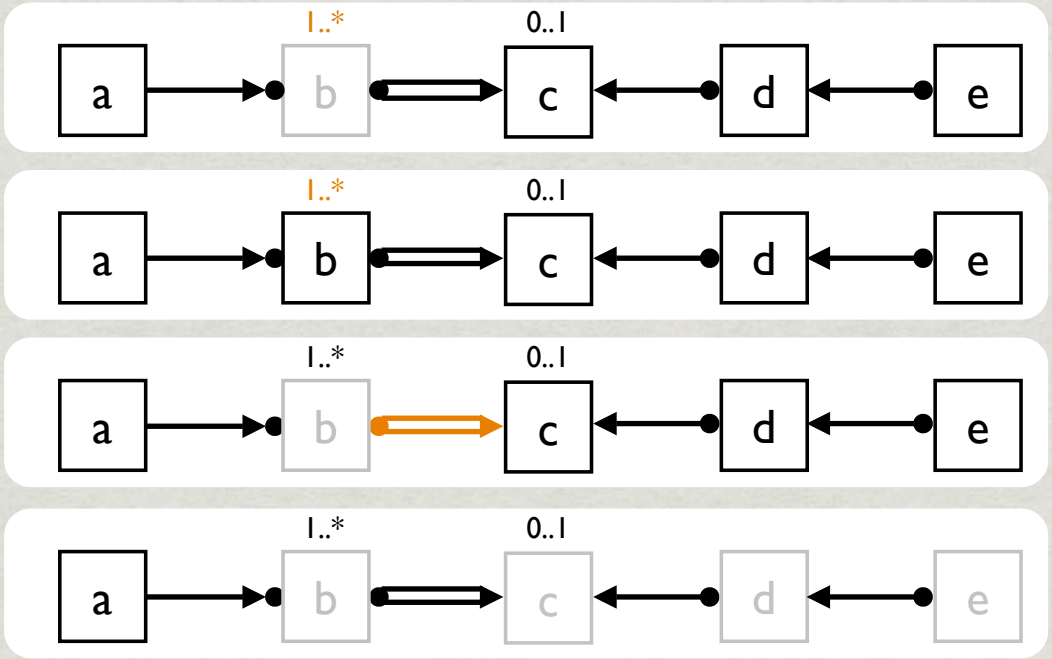


inconsistent

➪ 'a' dead activity

DecSerFlow Enactment

- Aim: supporting users when executing DecSerFlow models, by blocking a-priori the possibility to violate constraints



Enactment through *SCIFF*

Ongoing work

- Enactment of extended DecSerflow models
- Our idea: exploiting the consistency check with an already acquired (partial) history

1)	T ← 0, LOG ← ∅
2)	for each activity A, suppose to do A at time T: LOG' ← LOG U {**H**(A,T)} verify if the model is consistent by considering LOG': if **NOT**, block the possibility to do A at time T
3)	perform an activity and update LOG, or do nothing
4)	T ← T + 1, back to 2) (MANDATORY if there are still pending expectations)

Interoperability

Ongoing work

- **Interoperability checking** between a DecSerFlow choreography and a DecSerFlow service
- As far as now, only an existential interoperability (consistency of the joint model)
- Our aim is to extend the notation by considering sender/receiver and to study more complex notions of interoperability (see [Baldoni et al. 2006, Alberti et al. 2006])

SCIFF and LTL

- DecSerFlow has an underlying semantics in terms of LTL formulas
  - only “finite” formulas are envisaged (a process should EVENTUALLY TERMINATE)
- What about the relationship between LTL and SCIFF
- Is SCIFF able to represent all the different LTL formulas?

What we want to prove

- There exists a model mapping μ and a formula mapping τ s.t.

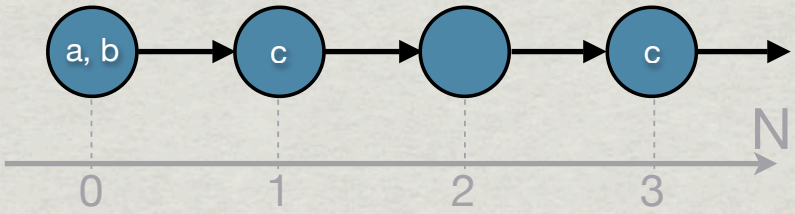
$$\sigma \models_{LTL} f \Rightarrow \mu(\sigma) \models_{SCIFF} \tau(f)$$

Model Mapping

- LTL model: **M = (T, <, v)**
  - **(T, <)** strict total order (flow of time) → in our case, **N**
  - **v** valuation function (to denote validity of propositions)
- SCIFF models: execution traces
- Model Mapping:

$$\mathcal{M}: \quad (\mathbb{N}, v) \longrightarrow \mathcal{H} \tag{1}$$

$$\sigma = (\mathbb{N}, v_\sigma) \longmapsto HAP = \{\mathbf{H}(a, T) | T \in v_\sigma(a)\} \tag{2}$$



v(a)={0}, v(b)={0}, v(c)={1,3}

H(a, 0)
H(b, 0)
H(c, 1)
H(c, 3)

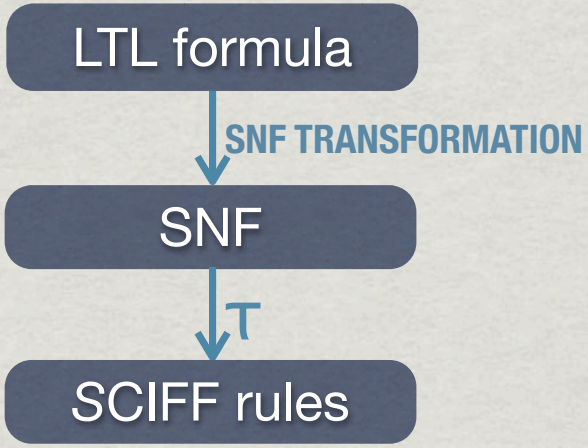
Formula Mapping

- We exploit Fisher's Separated Normal Form

$$\mathbf{start} \implies \bigwedge_c l_c \qquad \text{(an initial LTL-clause)}$$

$$\Box\left(\bigwedge_a k_a \implies \bigcirc \bigvee_d l_d\right) \qquad \text{(a step LTL-clause)}$$

$$\Box\left(\bigwedge_b k_b \implies \Diamond l\right) \qquad \text{(a sometime LTL-clause)}$$



FISHER, M., DIXON, C., AND PEIM, M. 2001.
CLAUSAL TEMPORAL RESOLUTION.
ACM TRANS. COMPUT. LOGIC 2, 1 (JAN. 2001), 12-56.

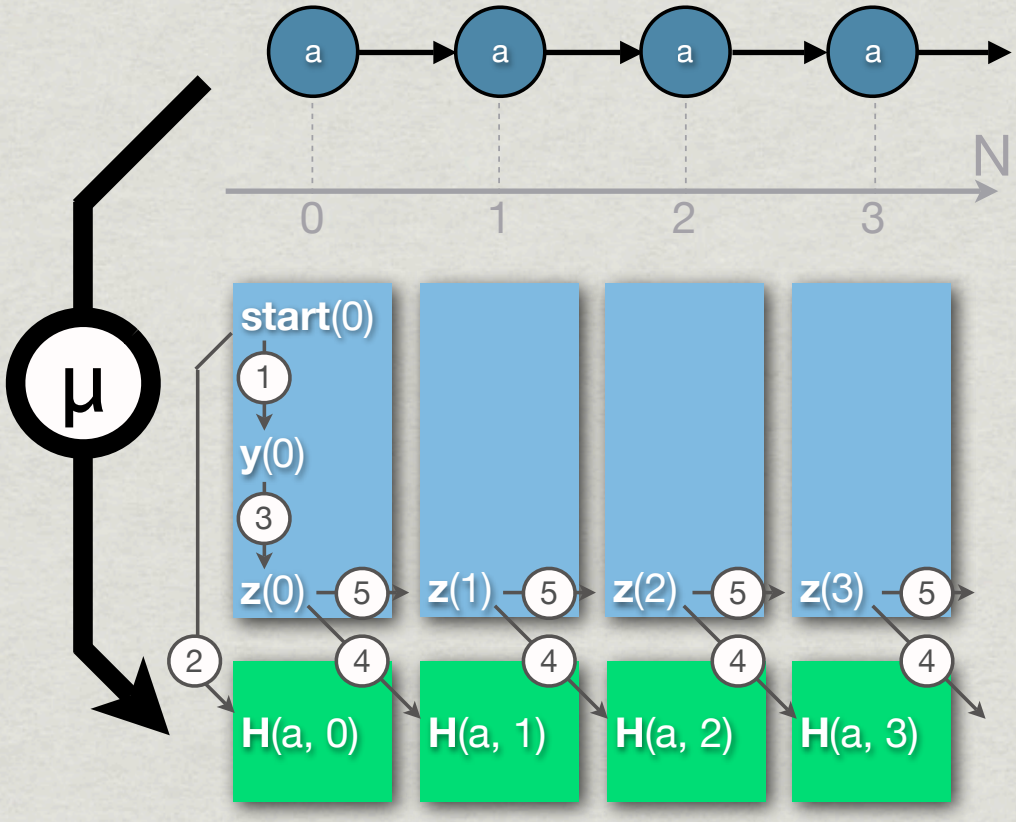
Some transformations

	SNF	SCIFF
$\Box a$	$\Box(\mathbf{start} \Rightarrow \mathbf{y})$ $\Box(\mathbf{y} \Rightarrow a)$ $\Box(\mathbf{y} \Rightarrow \mathbf{z})$ $\Box(\mathbf{z} \Rightarrow Oa)$ $\Box(\mathbf{z} \Rightarrow O\mathbf{z})$	$\mathbf{start}(0) \rightarrow \mathbf{y}(0).$ $\mathbf{y}(T) \rightarrow \mathbf{H}(a, T).$ $\mathbf{y}(T) \rightarrow \mathbf{z}(T).$ $\mathbf{z}(T) \rightarrow \mathbf{H}(a, T2) \wedge T2 == T + 1.$ $\mathbf{z}(T) \rightarrow \mathbf{z}(T2) \wedge T2 == T + 1.$
$\Diamond a$	$\Box(\mathbf{start} \Rightarrow \mathbf{y})$ $\Box(\mathbf{y} \Rightarrow \Diamond a)$	$\mathbf{start}(0) \rightarrow \mathbf{y}(0).$ $\mathbf{y}(T) \rightarrow \mathbf{H}(a, T2) \wedge T2 > T.$
$\Box(a \Rightarrow \Diamond b)$	$\Box(a \Rightarrow \Diamond b)$	$\mathbf{H}(a, T) \rightarrow \mathbf{H}(b, T2) \wedge T2 > T.$

Mapping Example

□a

1. **start**(0)→**y**(0).
2. **y**(T)→**H**(a, T).
3. **y**(T)→**z**(T).
4. **z**(T)→**H**(a, T2) ∧ T2 == T + 1.
5. **z**(T)→**z**(T2) ∧ T2 == T + 1.



Conclusions

- There is a need of declarative languages and tools when developing flexible business processes
- We propose to adopt
  - (an extended version of) **DecSerFlow** for the graphical specification
  - **SCIFF** as the underlying formal framework
- The **mapping** of DecSerFlow onto SCIFF is intuitive and **automatic**
- SCIFF can be fruitfully used to **monitor** services w.r.t. a choreographic model, to **verify consistency** of a DecSerFlow model and **discovery dead activities**, and even to **mine** DecSerFlow models starting from execution traces
- Deeply test the usability of the language
- Future works:
  - Modeling data-related aspects (e.g. data-driven conditions)
  - Deeply study the relationships between LTL and SCIFF