

Declarative Flexible Workflows for Trustworthy Pervasive Healthcare Services

Talk at Microsoft Research India, Bangalore
29 July, 2010

Raghava Rao Mukkamala

PhD student,

Advisor: **Thomas T Hildebrandt**

Programming, logic and Semantics Group
IT University of Copenhagen, Denmark

Overview

- PhD thesis Overview
 - Problem statement and research questions
 - Research Methodology and Approach
- Results up to date
 - Formal Process model Dynamic condition response structures (DCRS)
 - Model checking and verification of DCRS processes
- Road map for next 18 months
- Expected Contribution

PhD Thesis Overview

- Part of Trustworthy Pervasive Healthcare Services (TrustCare) project.
- Timeline:



- Thesis proposal presentation at Doctoral Consortium, BPM 2010 Conference, New York on September 11th.
- Thesis proposal defense in September at the university.

Research Questions

Overall Goal: To develop formal foundations for trustworthy and declarative flexible workflows with a key focus on the health care sector.

1. What are the formal semantical models suitable for describing flexible workflow processes for health care and other dynamic services?

Formal workflow language based on declarative modeling primitives.

2. How should one describe interfaces, contracts and interactions for declarative, quantitative and dynamic workflows?

Extensions to Session types/End-point projections for declarative and quantitative contracts.

3. What are the suitable model checking and verification techniques for enhancing trustworthiness of declarative, quantitative and dynamic workflows?

Model checking and verification techniques for declarative processes.

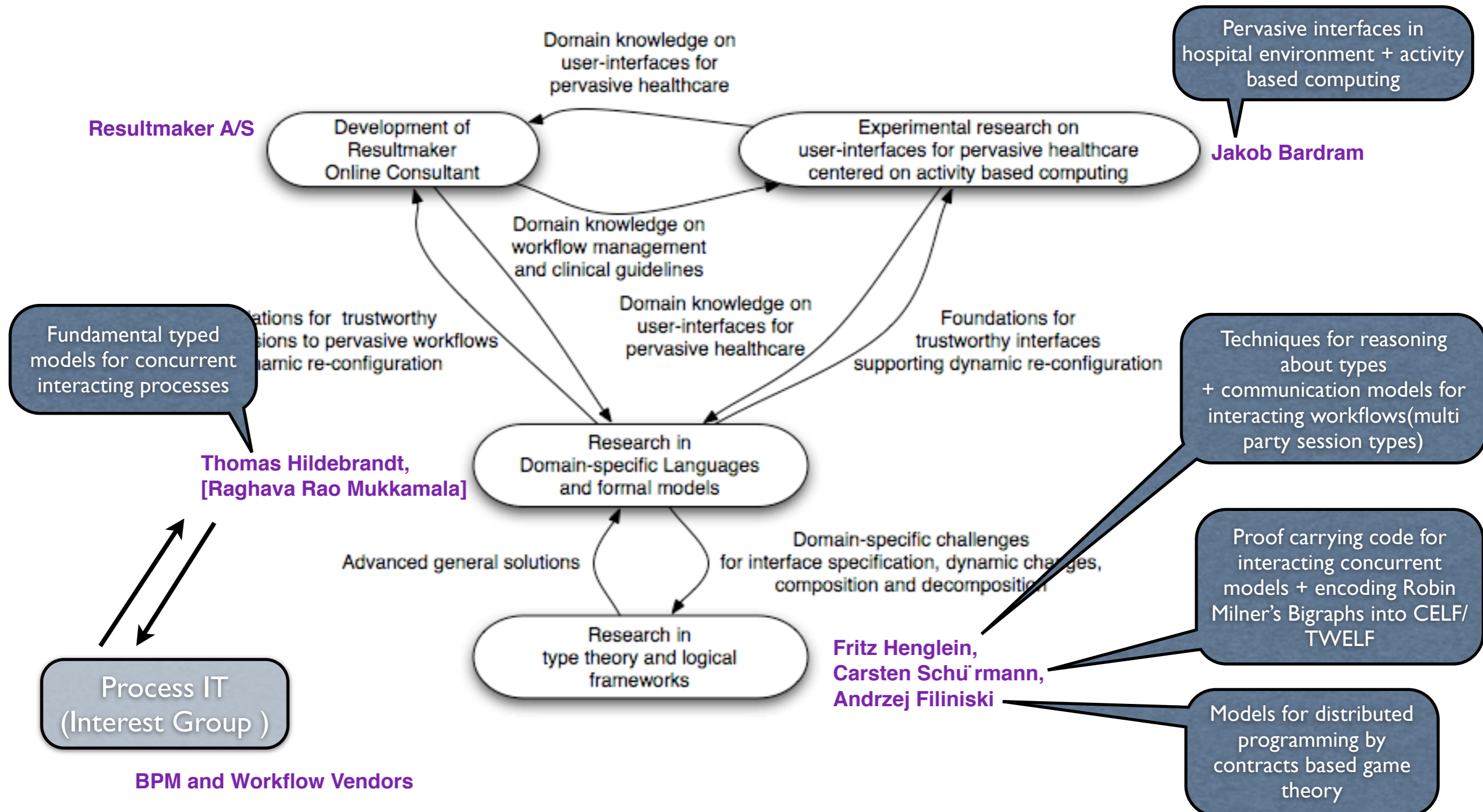
Prior and Related Work

- **Declare tool and DecSerFlow**
 - M. Pesic and W.M.P. van der Aalst: A Declarative Approach for Flexible Business Processes.(2006)
 - W.M.P. van der Aalst and M. Pesic: DecSerFlow: Towards a Truly Declarative Service Flow Language. (2006)
 - W. M. P. van der Aalst, M. Pesic and H. Schonenberg: Declarative workflows: Balancing between flexibility and support. (2009).
- **Event Calculus**
 - Nihan Kesim Cicekli and Ilyas Cicekli: Formalizing the specification and execution of workflows using the event calculus. (2006)
- **Session Types and Choreographies (WS-CDL)**
 - Marco Carbone, Kohei Honda, and Nobuko Yoshida: Structured communication-centered programming for web services
- **Prior work:**
 - Raghava Rao Mukkamala, Thomas T. Hildebrandt, and Janus Boris Tøth: The Resultmaker Online Consultant: From Declarative Workflow Management in Practice to LTL (2008)

Research approach and methodology

TrustCare Project Methodology

Goal: To provide trustworthy it-support for pervasive health care service



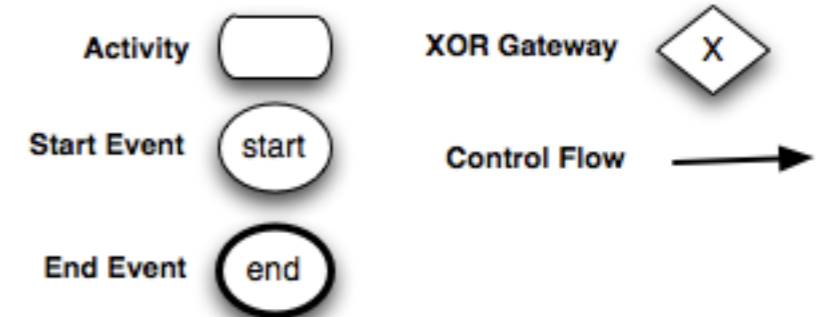
Motivation for Declarative Modeling

Traditional workflow example in BPMN¹

- Specification

- ➔ 3 tasks: *bless*, *curse* and *pray*.
- ➔ Rule: if you *curse* someone, then you **MUST** *pray* afterwards.

BPMN Symbols



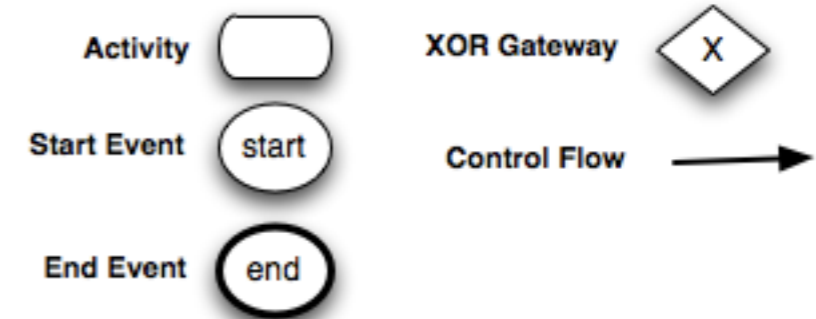
Motivation for Declarative Modeling

Traditional workflow example in BPMN¹

- Specification

- ➔ 3 tasks: *bless*, *curse* and *pray*.
- ➔ Rule: if you *curse* someone, then you **MUST** *pray* afterwards.

BPMN Symbols



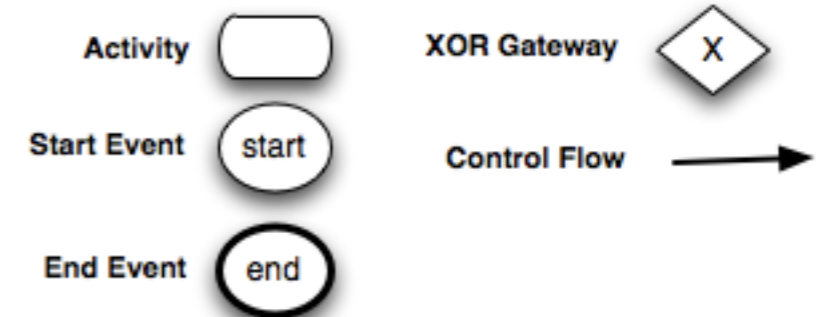
Motivation for Declarative Modeling

Traditional workflow example in BPMN¹

- Specification

- ➔ 3 tasks: *bless*, *curse* and *pray*.
- ➔ Rule: if you *curse* someone, then you **MUST** *pray* afterwards.

BPMN Symbols



[bless]

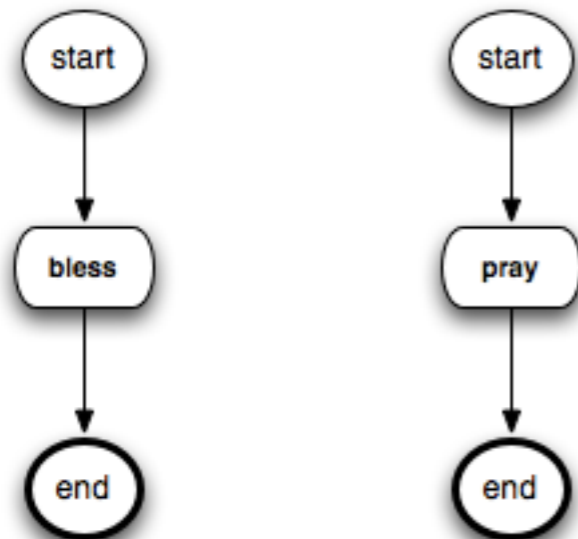
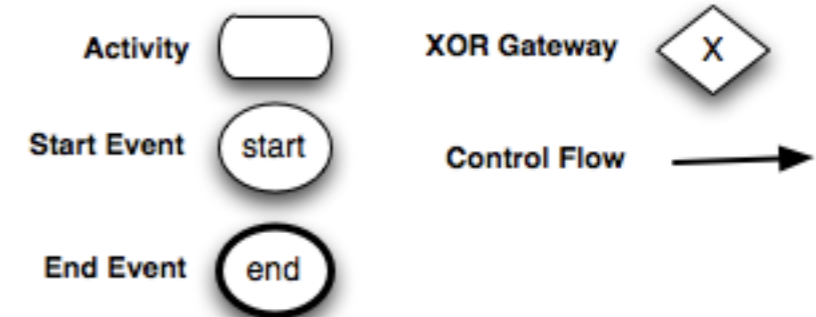
Motivation for Declarative Modeling

Traditional workflow example in BPMN¹

- Specification

- ➔ 3 tasks: *bless*, *curse* and *pray*.
- ➔ Rule: if you *curse* someone, then you **MUST** *pray* afterwards.

BPMN Symbols



[bless]

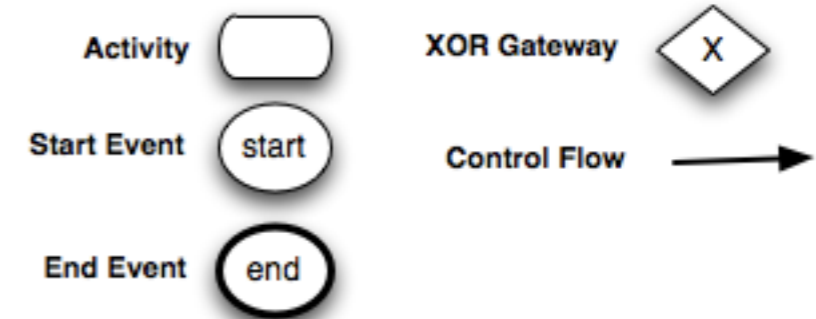
Motivation for Declarative Modeling

Traditional workflow example in BPMN¹

- Specification

- ➔ 3 tasks: *bless*, *curse* and *pray*.
- ➔ Rule: if you *curse* someone, then you **MUST** *pray* afterwards.

BPMN Symbols



[bless]



[pray]

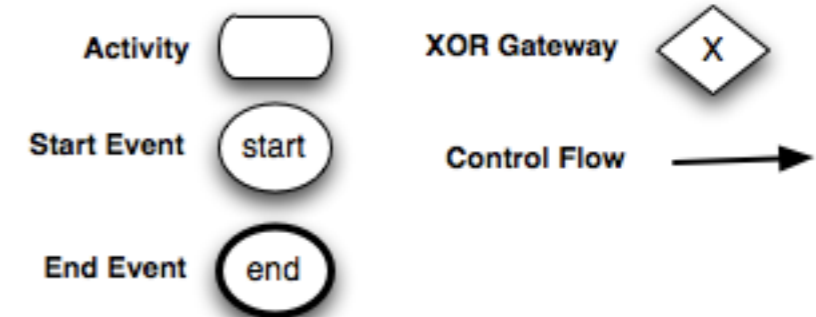
Motivation for Declarative Modeling

Traditional workflow example in BPMN¹

- Specification

- ➔ 3 tasks: *bless*, *curse* and *pray*.
- ➔ Rule: if you *curse* someone, then you **MUST** *pray* afterwards.

BPMN Symbols



[bless]



[pray]



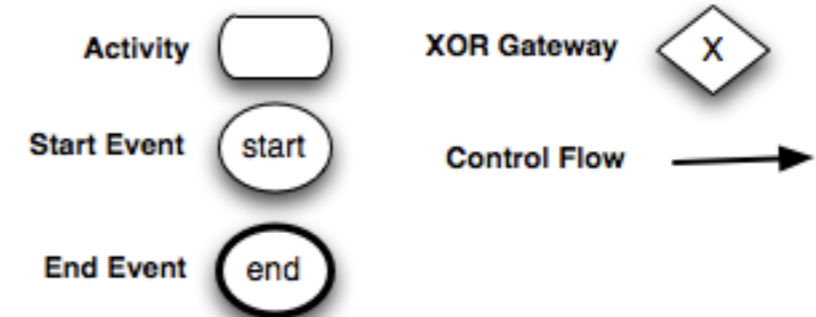
Motivation for Declarative Modeling

Traditional workflow example in BPMN¹

- Specification

- ➔ 3 tasks: *bless*, *curse* and *pray*.
- ➔ Rule: if you *curse* someone, then you **MUST** *pray* afterwards.

BPMN Symbols



[bless]



[pray]



[curse, pray]

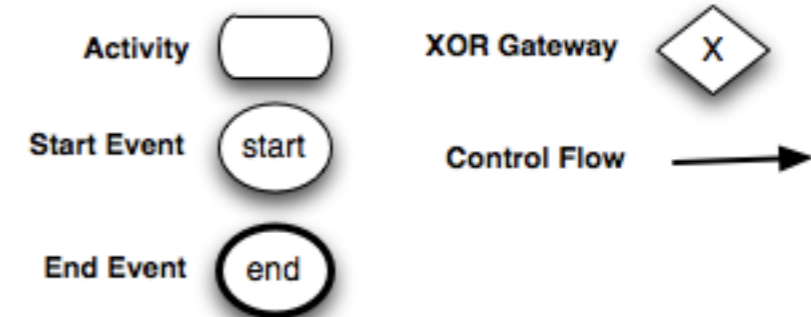
Motivation for Declarative Modeling

Traditional workflow example in BPMN¹

- Specification

- ➔ 3 tasks: *bless*, *curse* and *pray*.
- ➔ Rule: if you *curse* someone, then you **MUST** *pray* afterwards.

BPMN Symbols



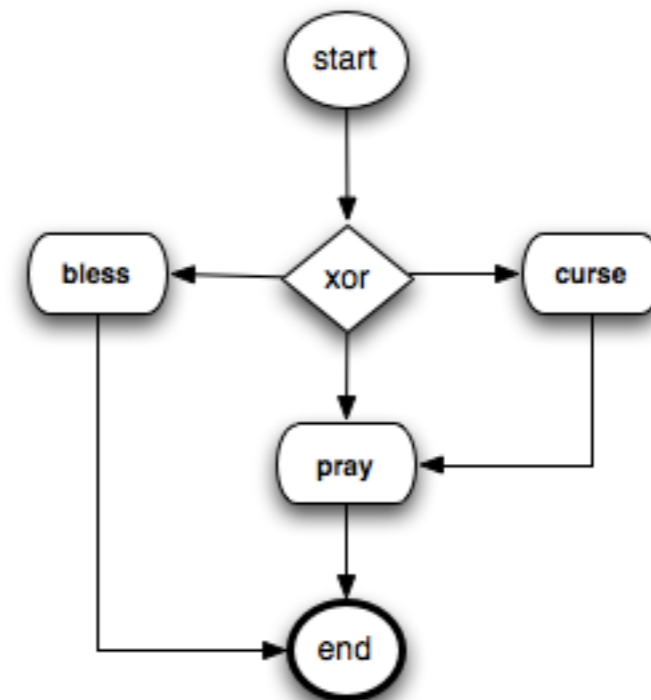
[bless]



[pray]



[curse, pray]



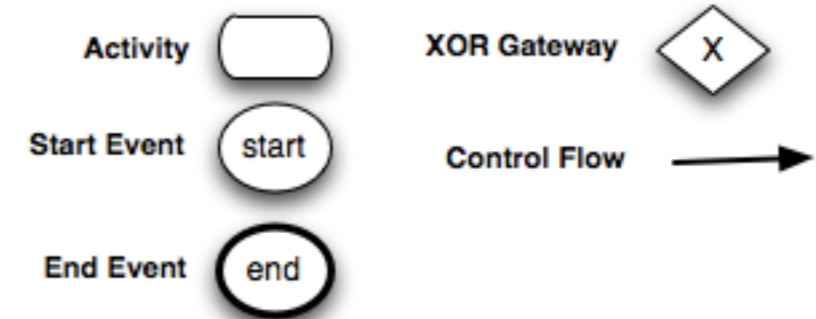
Motivation for Declarative Modeling

Traditional workflow example in BPMN¹

- Specification

- ➔ 3 tasks: *bless*, *curse* and *pray*.
- ➔ Rule: if you *curse* someone, then you **MUST** *pray* afterwards.

BPMN Symbols



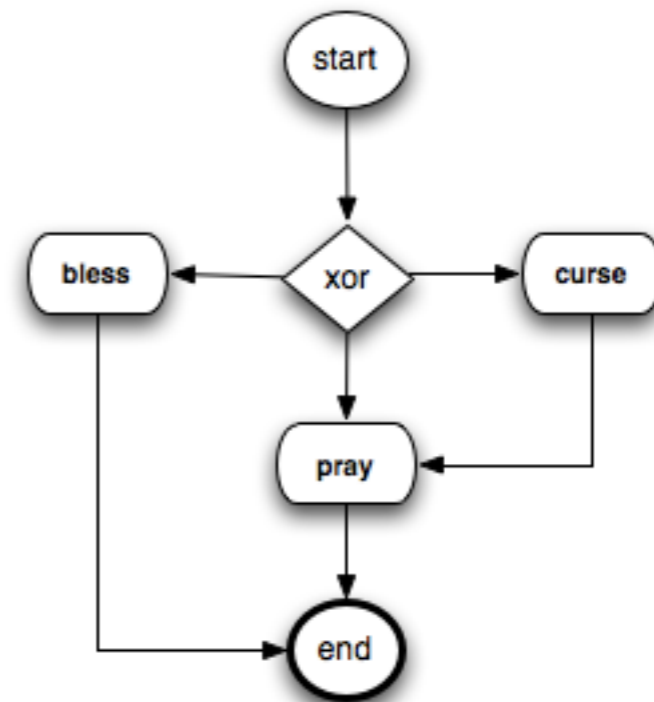
[bless]



[pray]



[curse, pray]



[bless], [pray], [curse, pray]

[1] BPMN: Business Process Modeling Notation

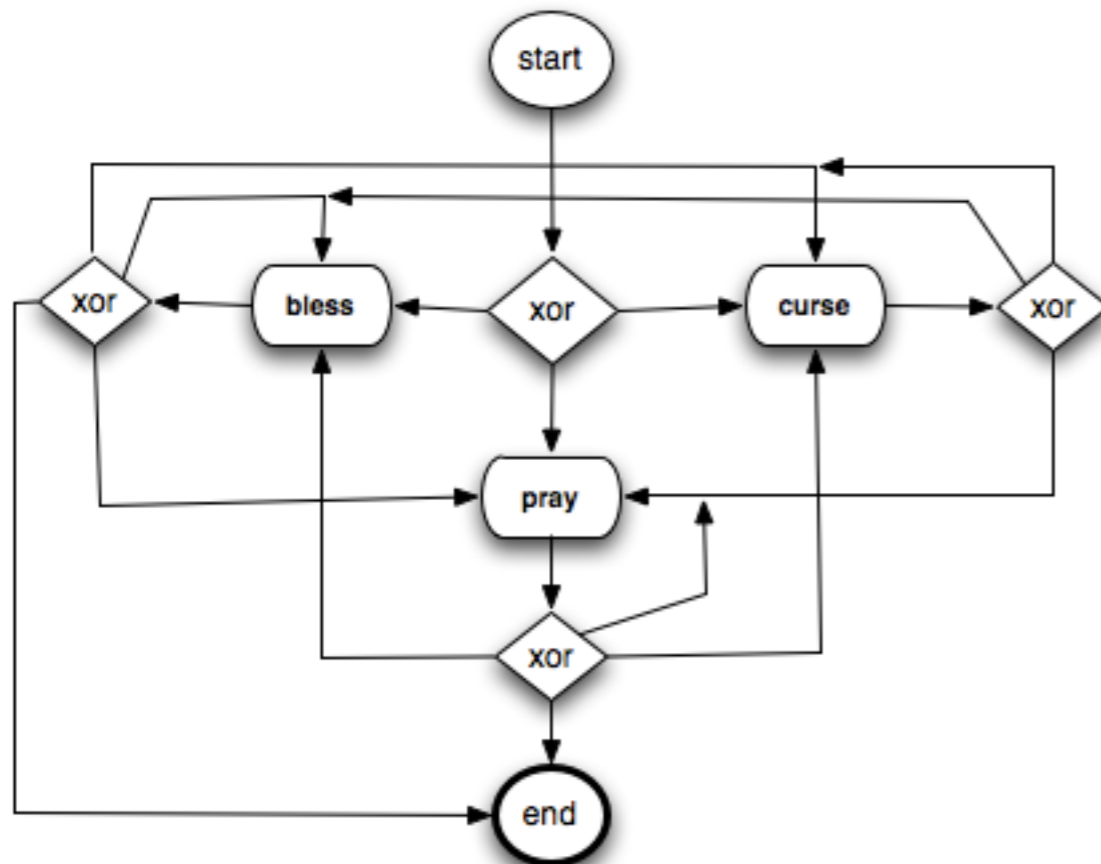
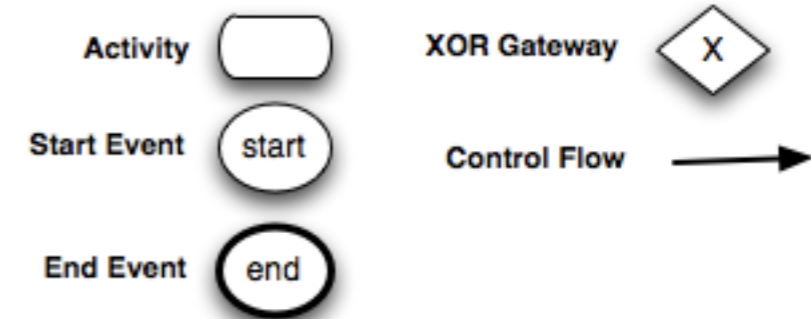
Motivation for Declarative Modeling

Traditional workflow example in BPMN¹

- Specification

- ➔ 3 tasks: *bless*, *curse* and *pray*.
- ➔ Rule: if you *curse* someone, then you **MUST** *pray* afterwards.

BPMN Symbols



[bless, bless, curse, pray]
[curse, curse, pray, bless, bless]
[pray, bless, curse, bless, pray]
.....

Control flow is explicit and hence we have to think ahead about all the possible computations we want to support

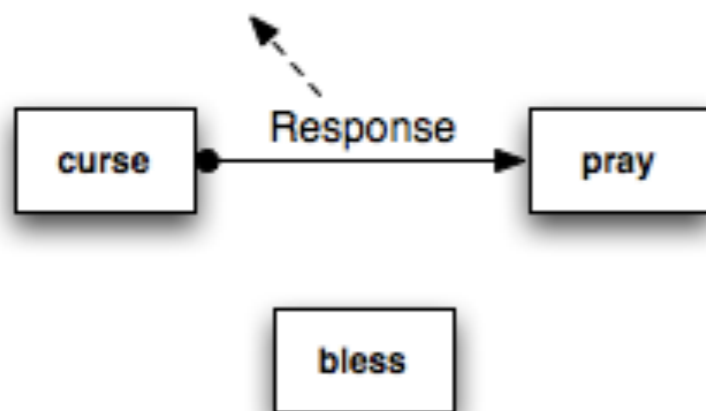
Motivation for Declarative Modeling

Constraint-based Approach

- Specification

- ➔ 3 tasks: *bless*, *curse* and *pray*.
- ➔ Rule: if you *curse* someone, then you **MUST** *pray* afterwards.

$\square (curse \rightarrow \diamond pray)$



- Tasks can be executed
 - ✓ any number of times
 - ✓ in any random orderunless they are prevented by constraints
- Accepting run: any execution ending with accepting state, where all constraints are satisfied

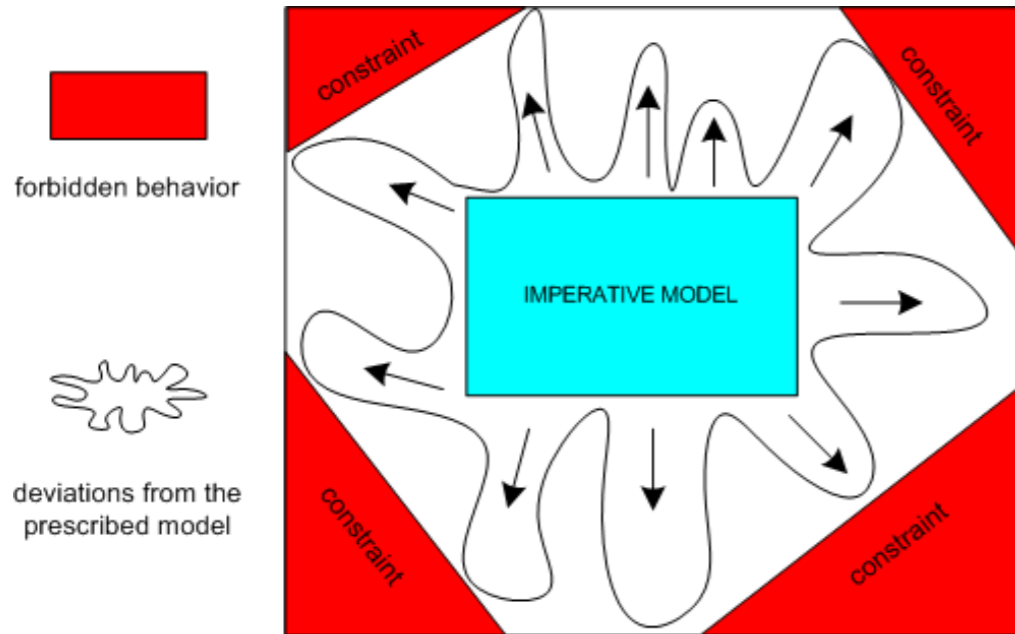
Accepting runs:

[bless, bless]
[bless, bless, curse, pray]
[curse, curse, pray,]
[curse, curse, pray, bless, bless]

Non-accepting runs:

[pray, curse]
[bless, curse, pray, curse, bless]

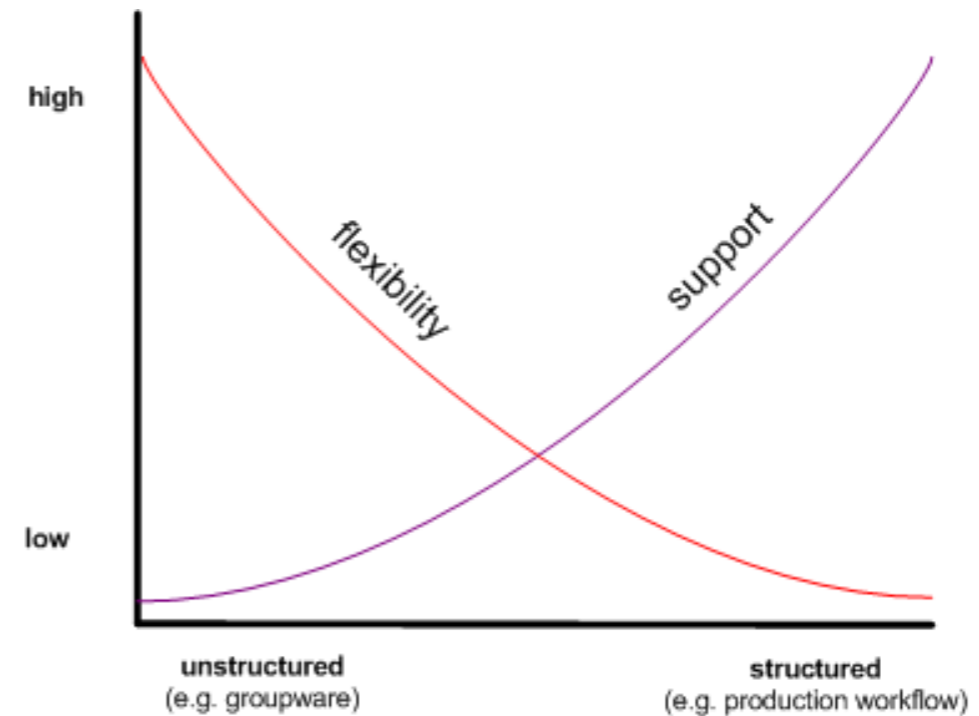
Imperative versus Declarative models



Imperative versus declarative modeling languages¹

Imperative Models

- Control flow is explicit hence over-specification, so less flexible
- Adding a new constraint requires to make a new model
- Focus is on **how** a set of tasks will be performed
- Very successful in sectors where strictly procedural execution is needed!



Classical trade-off between flexibility and support¹

Declarative Models

- Control flow is implicit hence under-specification, so more flexible
- Specify constraints to forbid the unwanted behavior
- Focus is on **what** should be done instead of **how**
- Difficult to perceive
 - where to start and where to end
 - what are next possible actions

Dynamic Condition Response Structures (DCRS) - Overview

- DCRS is formal process model for specification and execution of flexible workflows/business processes.
- Using declarative modeling approach
- Based on Event Structures³, a minimal, and declarative model for concurrent processes.
- DCRS also has a graphical notation which closely tied to the formal model.
- Execution semantics
 - For finite runs, mapped to labeled transition system
 - for infinite runs, mapped to generalized Bu"chi Automaton

[3] Glynn Winskel. Event structures. University of Cambridge(1986).

Event Structures³

Definition

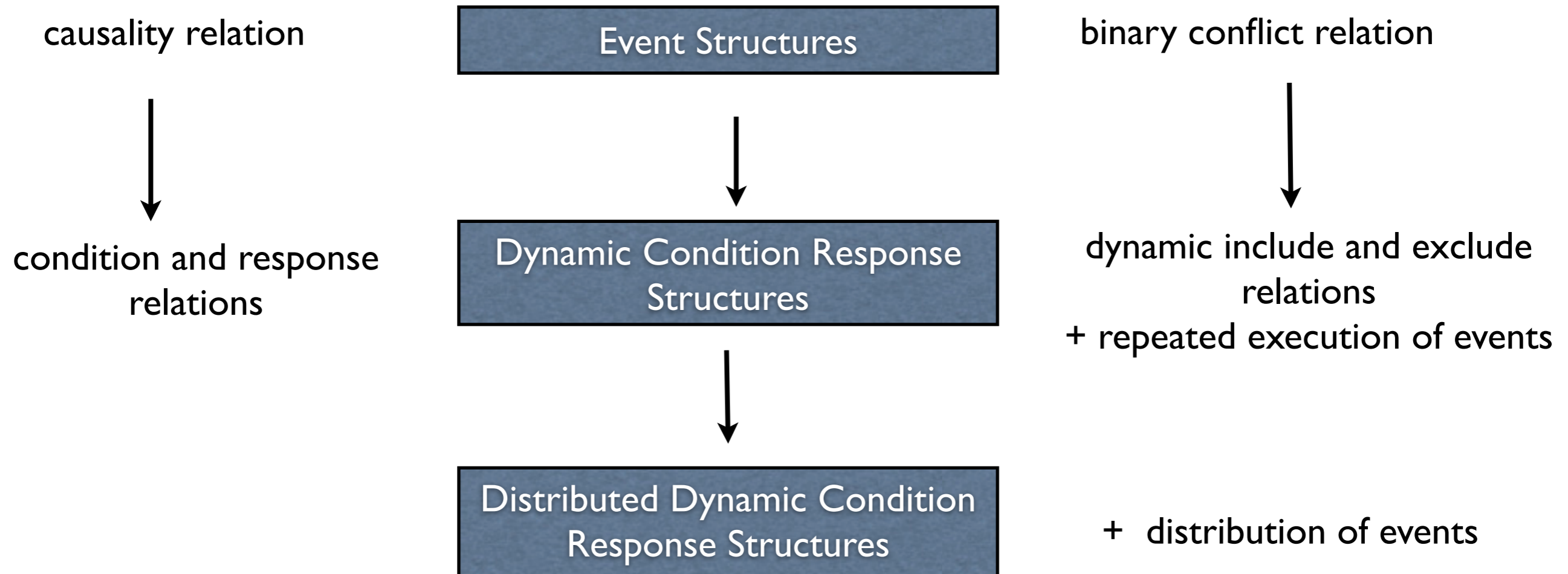
A labeled *prime event structure* (ES) over an alphabet *Act* is a 4-tuple $(E, \leq, \#, l)$ where

- ▶ E is a (possibly infinite) set of events
 - ▶ $\leq \subseteq E \times E$ is the causality relation between events which is partial order
 - ▶ $\# \subseteq E \times E$ is a binary conflict relation between events which is irreflexive and symmetric
 - ▶ $l : E \rightarrow Act$ is the labeling function mapping events to actions
1. Causality relation satisfies principle of finite causes:
 $\forall e, e' \in E : \{e' \mid e' \leq e\}$ is finite.
 2. Conflict relation satisfies principle of conflict heredity:
 $\forall e, e', e'' \in E : e \# e' \wedge e \leq e'' \Rightarrow e' \# e''$
 3. A computation x is a sequence of events such that
 - ▶ $e' \in x \ \& \ e \leq e' \implies e \in x$
(if an event has occurred then all events on which it causally depends have occurred too)
 - ▶ $\forall e, e' \in x. \neg(e \# e')$
(no two conflicting events can occur together in the same computation)

Overview of Dynamic Condition Response Structures

Limitation of events structures

- finite representation of infinite behavior
- accepting condition
- distribution of events



Dynamic Condition Response Structures(DCRS)

A *dynamic condition response structure* (DCR) is a tuple

$D = (E, \text{Act}, \rightarrow\bullet, \bullet\rightarrow, \pm, l)$ where

- ▶ $\rightarrow\bullet \subseteq E \times E$ is the *condition* relation
- ▶ $\bullet\rightarrow \subseteq E \times E$ is the *response* relation
- ▶ $\pm : E \times E \rightarrow \{+, \%, *\}$ is the *dynamic inclusion/exclusion* relation.
- ▶ $l : E \rightarrow \text{Act}$ is a labelling function mapping events to actions.

1. Causality relation (\leq) is replaced with two relations: condition ($\rightarrow\bullet$) and response ($\bullet\rightarrow$) relations.
2. If $e \rightarrow\bullet e'$, then e must happen before e' as a pre-condition.
3. If $e \bullet\rightarrow e'$, then after e happens, e' must eventually happen or be excluded indefinitely for the computation to be accepting.
4. Conflict relation($\#$) is generalized to include ($\rightarrow+$) and exclude ($\rightarrow\%$) relations to include/exclude events dynamically.
5. $\pm(e, e') = +$ if event e gets executed, then it will include event e'
6. $\pm(e, e') = \%$ if event e gets executed, then it will exclude event e'
7. Conflict relation is monotone, but dynamic inclusion/exclusion allows an event to alternate between being in conflict and not.
8. Execution of an event only depends on the condition relation (restricted to the currently included events), where acceptance of a computation depends on response relation.

Distributed Dynamic Condition Response Structures*

Definition

A *distributed* dynamic condition response structure (DDCR) is a tuple

$$(E, \text{Act}, \rightarrow\bullet, \bullet\rightarrow, \pm, I, R, P, \text{as})$$

where $(E, \text{Act}, \rightarrow\bullet, \bullet\rightarrow, \pm, I)$ is a dynamic condition response structure, R is a set of *roles*, P is a set of *principals* (e.g. persons/processors/agents) and $\text{as} \subseteq (P \cup \text{Act}) \times R$ is the role assignment relation to executors and actions.

- assigning roles to actions provide granularity of permissions
- assigning principals to roles gives the permission to execute actions

[*] Thomas Hildebrandt and Raghava Rao Mukkamala. Distributed dynamic condition response structures. Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 10), EATAPS conference Cyprus, March 2010.

DCRS Execution semantics for Finite Runs

Definition

For a distributed DCR $D = (E, \text{Act}, \rightarrow\bullet, \bullet\rightarrow, \pm, l, R, P, \text{as})$ the corresponding labelled transition systems $T(D)$ to be the tuple $(S, (\emptyset, E, \emptyset), \rightarrow\subseteq S \times \text{Act} \times S)$ where $S = \mathcal{P}(E) \times \mathcal{P}(E) \times \mathcal{P}(E)$ is the set of states, $(\emptyset, E, \emptyset) \in S$ is the initial state, $\rightarrow\subseteq S \times (\text{P} \times \text{Act} \times R) \times S$ is the transition relation given by

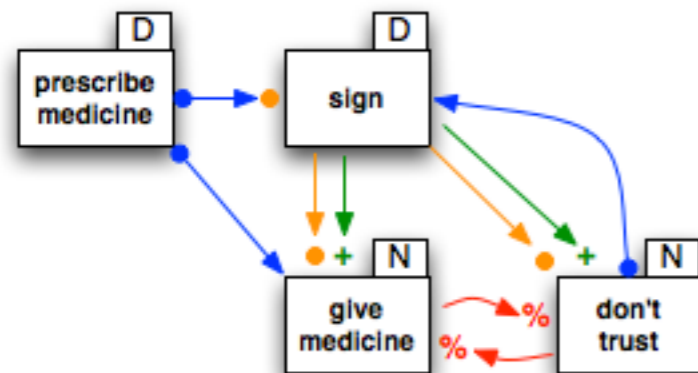
$$(E, l, R) \xrightarrow{(e, (p, a, r))} (E \cup \{e\}, l', R') \text{ where}$$

- ▶ $e \in l, l(e) = a, p \text{ as } r, \text{ and } a \text{ as } r$
 - ▶ $\{e' \in l \mid e' \rightarrow\bullet e\} \subseteq E$
 - ▶ $l' = (l \cup \{e' \mid \pm(e, e') = +\}) \setminus \{e' \mid \pm(e, e') = \%\}$
 - ▶ $R' = (R \setminus \{e\}) \cup \{e' \mid e \bullet\rightarrow e'\}$
-
- map to a labelled transition system to defined accepting runs
 - states of transition system will be (E, l, R) where $E \subseteq E$ is set of happened events, $l \subseteq E$ represents set of currently included events, $R \subseteq E$ represents set of pending response events
 - first condition says that only currently include events can be executed, the label (p, a, r) says that the label of event e must be a , which must be assigned to a role r and principal p
 - second condition says that all condition events to e must have been executed
 - third and fourth conditions are updates to sets of included and pending responses.
 - **Accepting State:** Any state with no pending responses ($R \cap l = \emptyset$)
 - **Accepting run:** Any run which is ending with accepting state ($R \cap l = \emptyset$)

DCRS Graphical Notation- Health Care Workflow

Rules:

- doctor has to sign whenever he prescribes a medicine
- medicine can be given only after doctor's sign
- nurse can check medicine and say that "I don't trust medicine", in that case doctor has to either to sign again or change medicine and sign.



- ▶ $E = \{pm, s, gm, dt\}$
- ▶ $\rightarrow\bullet = \{(pm, s), (s, gm), (s, dt)\}$
- ▶ $\bullet\rightarrow = \{(pm, s), (pm, gm), (dt, ss)\}$
- ▶ $\rightarrow+ = \{(s, gm), (s, dt)\}$
- ▶ $\rightarrow\% = \{(gm, dt), (dt, gm)\}$

• OK and Accepting

- ✓ $[pm, s, gm, gm]$
- ✓ $[pm, s, dt, s, gm]$
- ✓ $[pm, s, dt, pm, s, gm]$
- ✓ $[pm, pm, s, gm]$
- ✓ $[pm, s, pm, s, gm]$

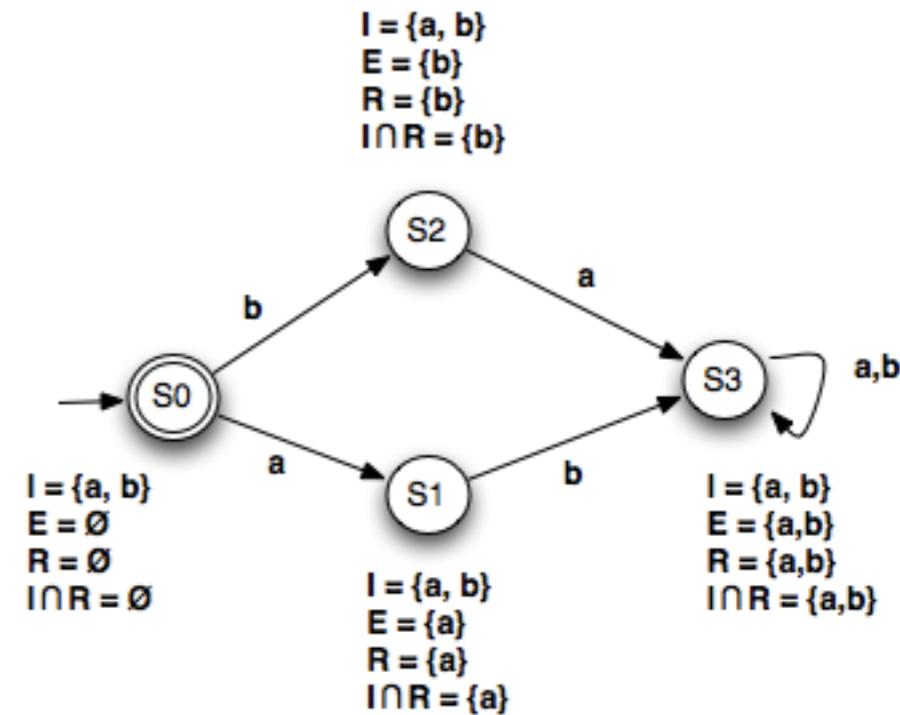
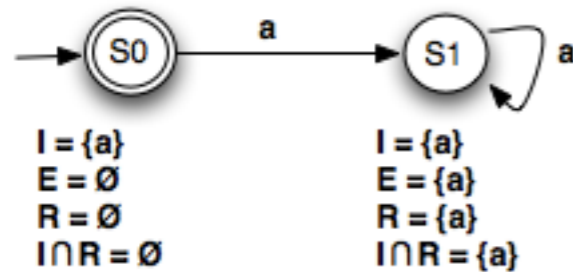
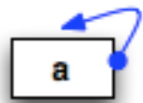
• Not Possible

- ⊙ $[pm, gm]$
- ⊙ $[pm, s, dt, gm]$

• Possible, but not Accepting

- ⊙ $[pm, s, gm, pm]$ (pending : gm, s)
- ⊙ $[pm, s, dt]$ (pending : s)

DCRS for Infinite Runs - Motivation



- In the first example a run a^ω should be accepting.
- In the second example, a run $a^* b^* (ab)^\omega$ should be accepting.
- But strong accepting condition ($R \cap I = \emptyset$) does not allow those runs accepting.

DCRS Execution semantics for Infinite Runs*

For a finite distributed DCR $D = (E, \text{Act}, \rightarrow, \bullet, \bullet \rightarrow, \pm, I, R, P, \text{as})$ where $E = \{e_1, \dots, e_n\}$ we define the corresponding Büchi-automaton $\text{Aut}(D)$ to be the tuple $(S, s, \rightarrow_{\subseteq} S \times \text{Act} \times S, F)$ where $S = \mathcal{P}(E) \times \mathcal{P}(E) \times \mathcal{P}(E) \times \{1, \dots, n\} \times \{0, 1\}$ is the set of states and $s = (\emptyset, E, \emptyset, 1, 1) \in S$ is the initial state and $F = \mathcal{P}(E) \times \mathcal{P}(E) \times \mathcal{P}(E) \times \{1, \dots, n\} \times \{1\}$ is the set of final or accepting states. $\rightarrow_{\subseteq} S \times (\mathcal{P} \times \text{Act} \times \mathcal{R}) \times S$ is the transition relation given by

$$(E, I, R, i, j) \xrightarrow{(p, a, r)} (E \cup \{e\}, I', R', i', j') \text{ where}$$

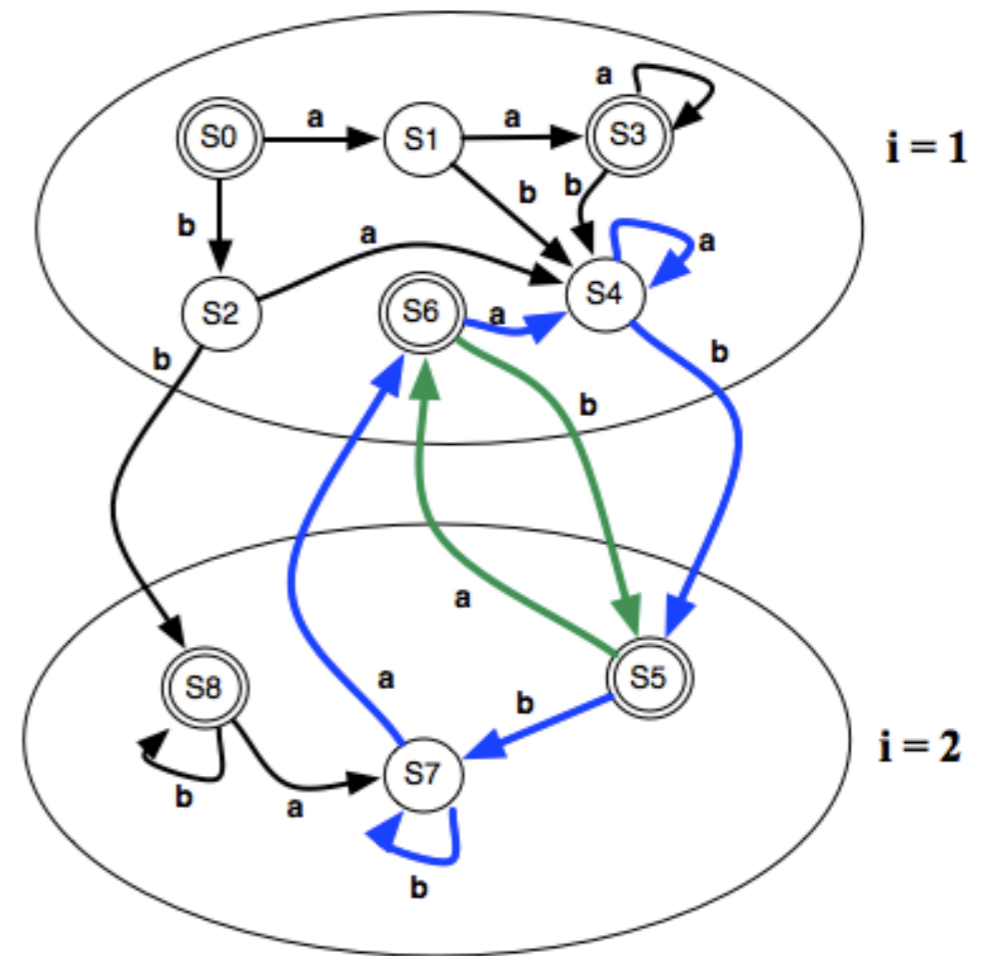
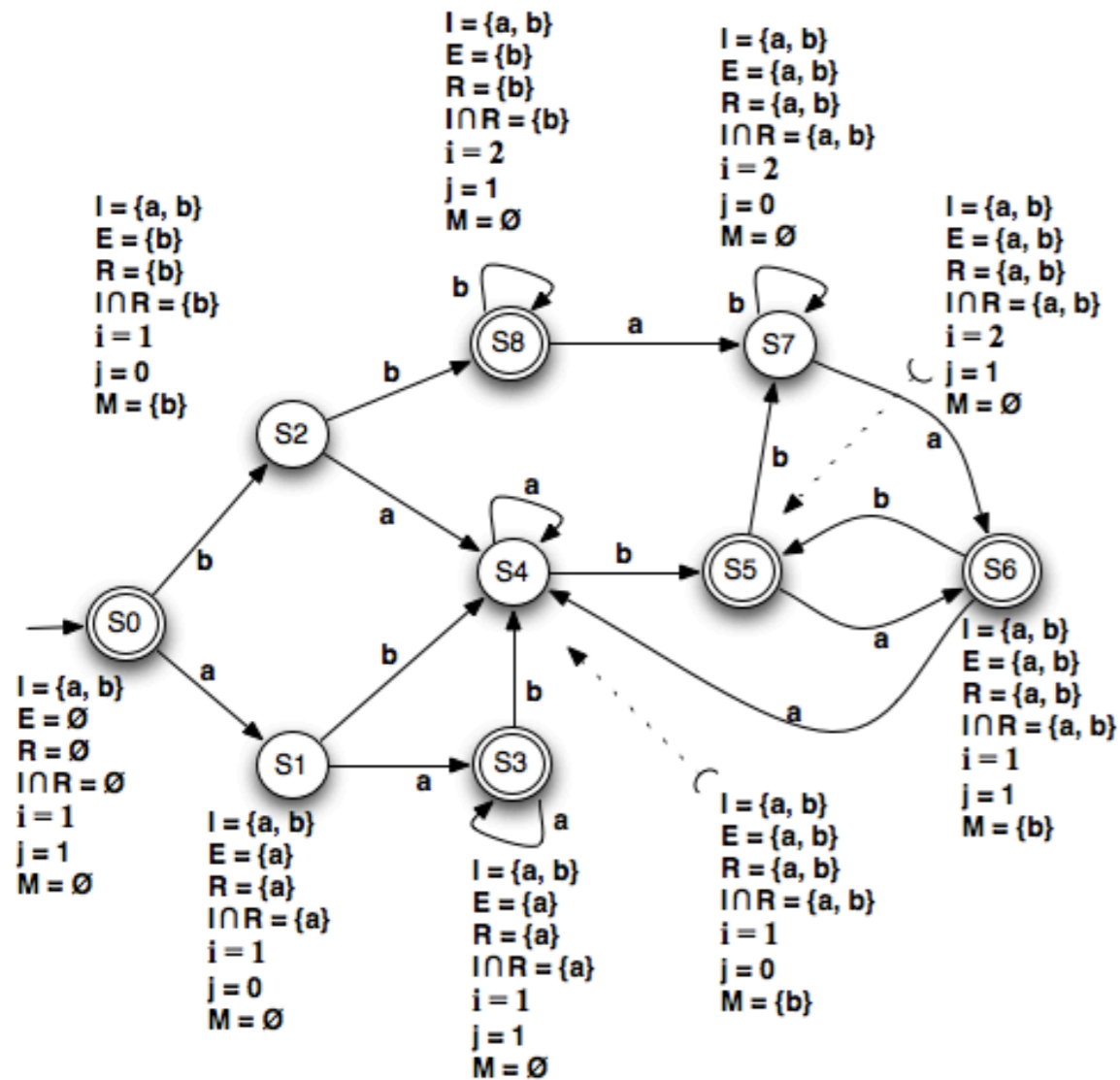
- ▶ Semantics of E, I, R are same as previous accepting condition
- ▶ $j' = 1$ if
 1. $I' \cap R' = \emptyset$
 2. $\min(M) \in (I \cap R \setminus (I' \cap R')) \cup \{e\}$
 3. $M = \emptyset$ and $\min(I \cap R) \in (I \cap R \setminus (I' \cap R')) \cup \{e\}$
 otherwise $j' = 0$.
- ▶ $i' = \text{rank}(\min(M))$ if $\min(M) \in (I \cap R \setminus (I' \cap R')) \cup \{e\}$
- ▶ $i' = \text{rank}(I \cap R)$ if $M = \emptyset$ and $\min(I \cap R) \in (I \cap R \setminus (I' \cap R')) \cup \{e\}$
- ▶ $i' = i$ otherwise.

for $M = \{e \in I \cap R \mid \text{rank}(e) > i\}$.

- Mapped to a Generalized Buchi automata
- Instead of looking for $R \cap I = \emptyset$, we make sure that all pending response events ($R \cap I$) will be eventually executed infinitely often.
- In other words, infinite run is not accepting if one or more pending response event(s) will never executed.

[*] Raghava Rao Mukkamala and Thomas Hildebrandt. From Dynamic Condition Response Structures to Büchi Automata. 4th IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2010), Taiwan, August 2010.

Accepting condition for infinite runs - example



Accepting run for case where both a and b are executed indefinitely often with i copies of state

Infinite runs: Buchi Automaton for example

Model checking and Verification of DCRS processes

- ◆ Declarative models offer little support and hard to perceive.
- ◆ Model checking/verification enhances reliability and trustworthiness among processes.

Model checking with Spin model checker:

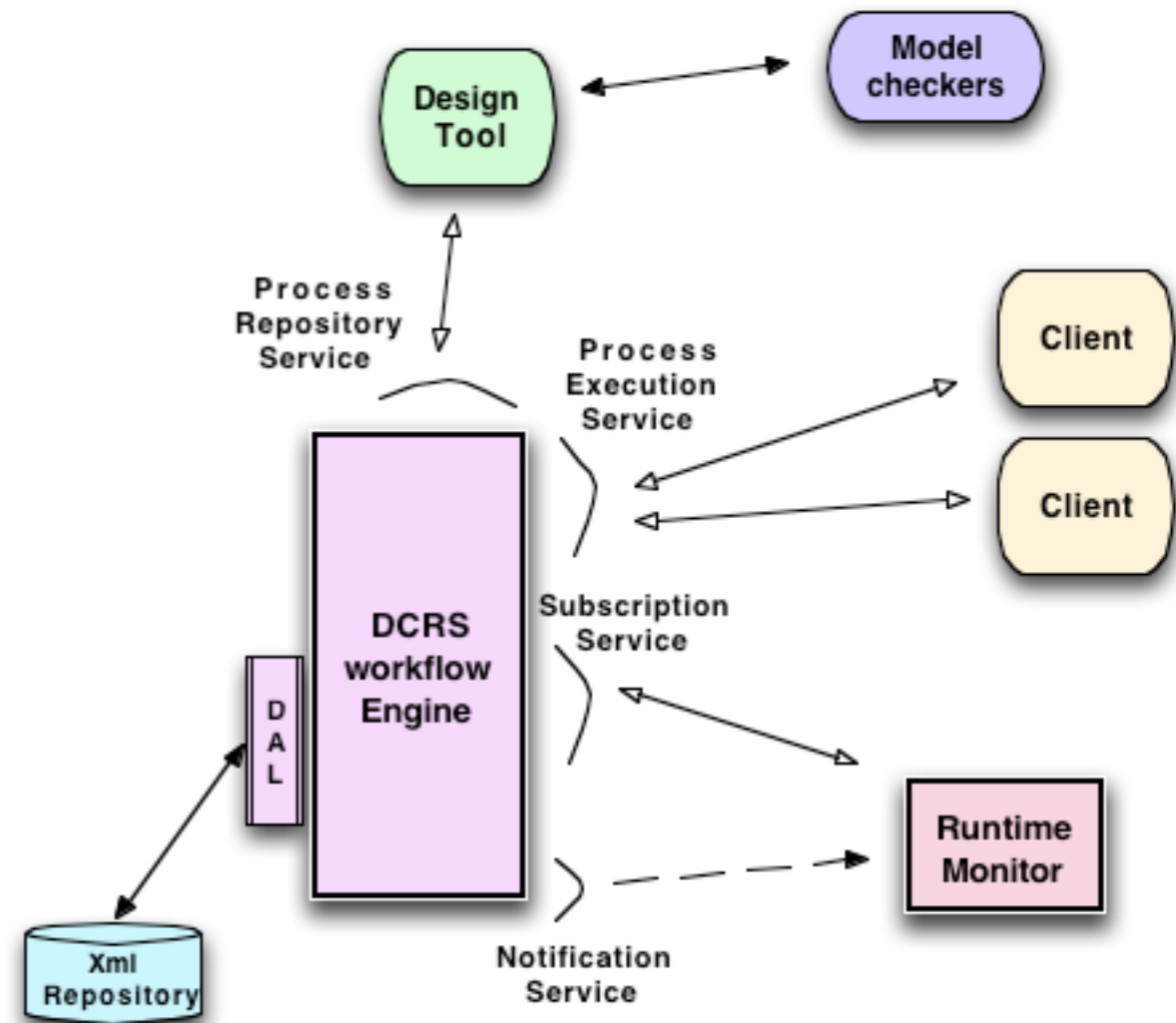
- ✓ A prototype application with interface to Spin model checker
- ✓ DCRS model can be automatically translated to Promela code.
- ✓ Both safety and liveness properties (expressed in LTL) can be verified for both finite and infinite runs.
- ✓ State space in Spin for DCRS models is un-necessarily large due to limited language constructs in Promela

Model checking with Zing model checker:

- ✓ Prototype application upgraded to generate code for Zing.
- ✓ State space in Zing is much smaller than Spin due to rich language constructs supported by Zing
- ✓ Only safety properties can be verified.

Runtime Monitoring of DCRS processes (work at MSRI)

- ◆ Verification of properties can be expensive for large process specifications.
- ◆ Runtime monitoring of DCRS processes makes sure that certain desired properties are preserved during execution.
- ◆ A predefined set of property specification patterns¹ can be used to specify properties.
- ◆ Properties transformed to a finite state machine.



DCRS Prototype Implementation

[1] Matthew B. Dwyer, George S. Avrunin and James C. Corbett: Property Specification Patterns for Finite-state Verification.(1998)
<http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml>

Roadmap for next 18 months

1) What are the formal semantical models suitable for describing flexible workflow processes for health care and other dynamic services?

- ✓ Extensions to formal model
 - ✓ Data, Time,
 - ✓ sub-processes,
 - ✓ Exceptions and Compensation
- ✓ Support for quantitative workflows(??)

3) What are the suitable model checking and verification techniques for enhancing trustworthiness of declarative, quantitative and dynamic workflows?

- ✓ DCRS prototype implementation to updated with extensions to formal model
- ✓ Support for more model checking and verification tools

Roadmap for next 18 months

2) How should one describe interfaces, contracts and interactions for declarative, quantitative and dynamic workflows?

- ✓ A choreography/contract describes a global view on how different participants interact.
- ✓ Session Types studied how to project such global interactions to individual participants called End-point projections¹.
- ✓ An interface to workflow is the observable behavior from outside (similar to choreography/contract)
- ✓ Extensions to theory of end-point projections will be studied to include quantitative and declarative contracts.

[1] Carbone, M., Honda, K. and Yoshida, N., Structured communication-centred programming for web services. 16th European Symposium on Programming, ESOP 2007. In: LNCS, Volume 4421/2007,

Expected Contribution

1. Contribution to Theory

- ✓ A formal process model for flexible declarative workflows based on inspiration from
 - ❖ Resultmaker workflow method
 - ❖ Declare¹
- ✓ Suitable extensions to theory of end-point projections² for declarative and quantitative contracts.

2. Contribution to Practice

- ✓ New ideas/methods from thesis will provide the foundations for development of real time workflow systems for complex/rapidly changing sectors, where process flexibility is important.
- ✓ Prototype implementation based on formal model will serve as good model/proof of concept, for application of model checking and verification techniques to real-time workflows.

[1] W. M. P. van der Aalst, M. Pesic: Declarative workflows: Balancing between flexibility and support. (2009).

[2] Carbone, M., Honda, K. and Yoshida, N., Structured communication-centred programming for web services. 16th European Symposium on Programming, ESOP 2007. In: LNCS, Volume 4421/2007,

Thank You
Questions & Comments ?