Business Process Technologies and Management
3. Process Implementation and Execution

Prof. Dr. Stefanie Rinderle-Ma

Dr. Jürgen Mangler

Technical University of Munich

Department of Informatics

Chair of Information Systems

and Business Process Management

stefanie.rinderle-ma@tum.de

juergen.mangler@tum.de

Literature and links

- Weske, M. (2019) Business Process Management - Concepts, Languages, Architectures, Third Edition. Springer 2019, ISBN 978-3-662-59431-5, pp. 1-417 2019
- Marlon Dumas, Marcello La Rosa, Jan Mendling, Hajo A. Reijers: Fundamentals of Business Process Management, Second Edition. Springer 2018, ISBN 978-3-662-56508-7, pp. 1-527
- https://www.signavio.com/bpm-academic-initiative/
- https://cpee.org/

Teaching Objectives

In this chapter we want to

- introduce the notion of execution semantics
- discuss process execution languages
- show the different aspects/perspectives to be considered to develop and implement executable processes and process-oriented applications
- give an overview of both buildtime and runtime components of a process management system (org modeler, work list, etc)
- discuss selected implementation issues related to process management systems

Developing Executable Processes...

... requires the definition and specification of the following aspect (selection):

- Control Flow (Behavioral Aspect)
- Data Flow (Information Aspect)
- Activities (Functional Aspect)
- Deadlines (Temporal Aspect)
- Actor Assignments (Organizational Aspect)
- Application Services (Operational Aspect)

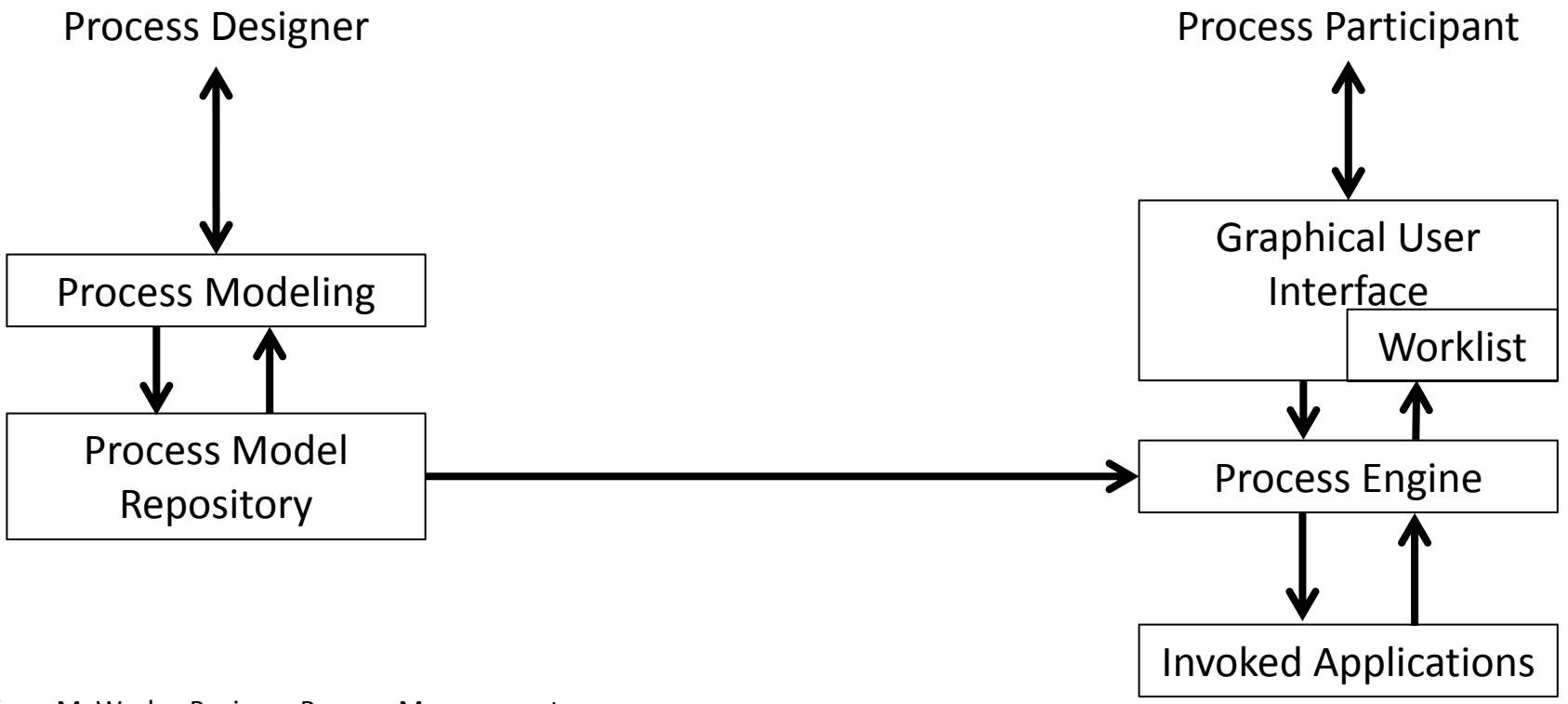
Short recap

- *So far:*
  - Process modeling (on a semantically high level)
  - Process verification
- *Now:*
  - Process modeling in an execution language
  - Concepts for organizational modeling
  - Concepts for application invocation and integration

Process Model / Schema

- Formal representation of (parts of) a business process → Enacted by a process management system or process engine
- Comprises a set of process steps – **tasks** – which are logically related in terms of their contribution to the overall business process
- Describes the **control flow** as well as the **data flow** between activities
- May contain **references** to other model components like sub-processes, organisation model, resource model, and so forth
- Based on a **process modeling language** (= **process meta model**) like Workflow Nets, WSM Nets, or CPEE Trees

Architectural considerations



adapted from M. Weske: Business Process Management. Springer 2019

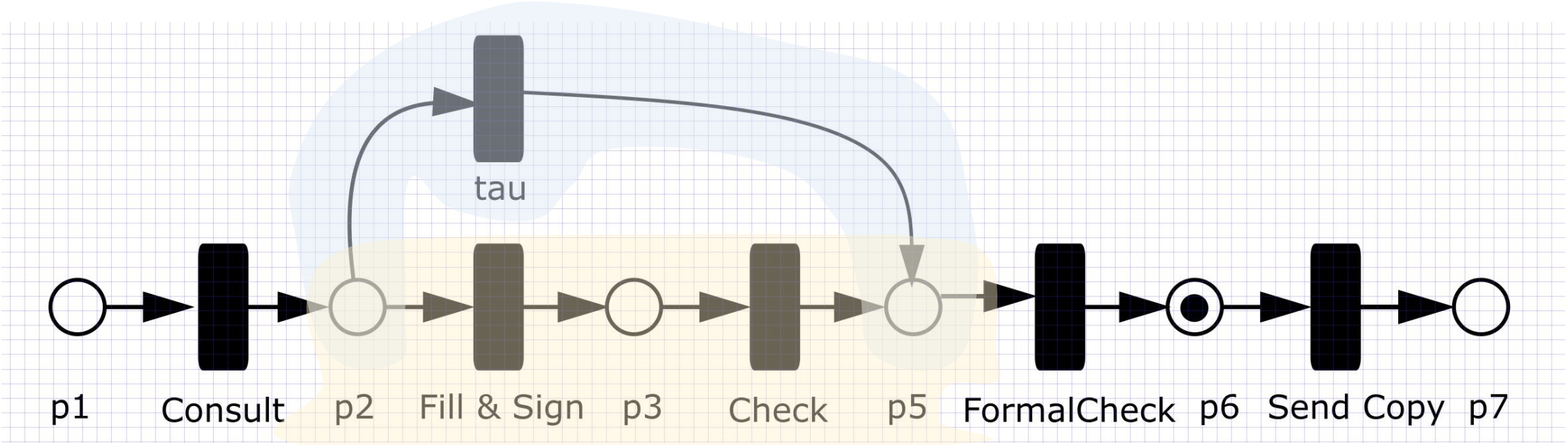
Aspects of Process Execution Modeling

- Formal and operational semantics
- Graphical modeling model-driven coding
- Ease of use
- Expressiveness
- Tools support
- Standard support
- Extensibility

Workflow Nets

- Basic definition of Workflow Nets in Chapter 2
- Execution semantics is precisely defined by marking and firing rules
- Workflow Nets with colored tokens distinguishable process instances
- Modeling control flow
- Execution in YAWL https://yawlfoundation.github.io/

A closer look at execution semantics

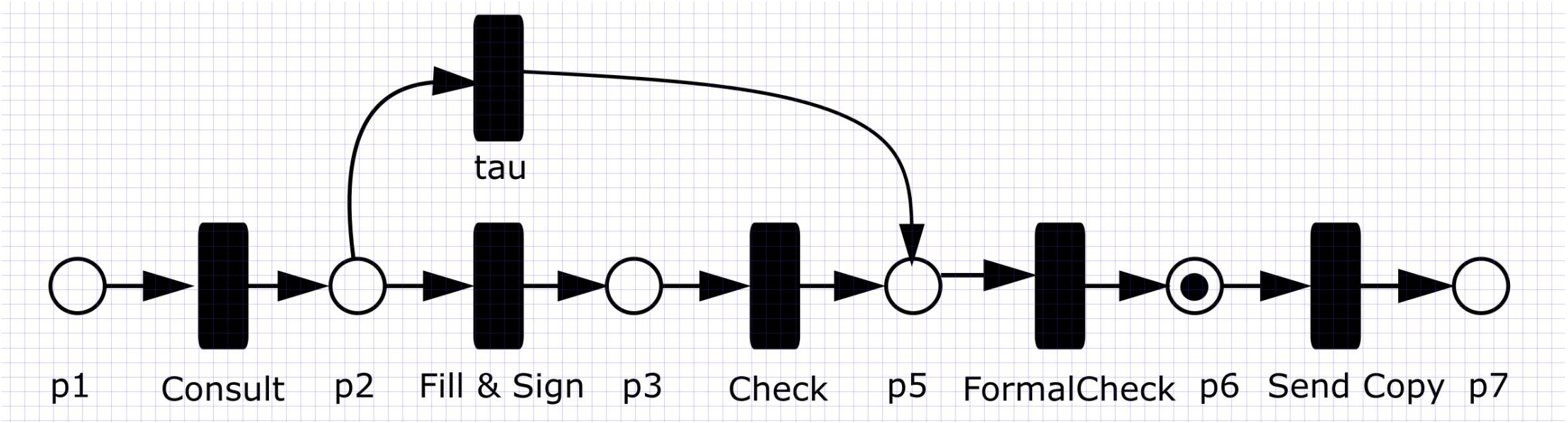


- If you look at the current state, how did we arrive at this state?
- In particular, which of the paths in the alternative branching were chosen?

TRUE Semantics

- Petri Nets and Workflow Nets (without any further information) offer a TRUE semantics
- TRUE semantics mean that we can only see the current state and the paths that have been definitely taken (TRUE), but not the paths that have not been taken (FALSE, SKIPPED)
- However, the information about skipped paths can be interesting for analysis, for example, for decision mining/analysis.
- Thus, a first approach here is to equip modesl with TRUE semantics with additional process history/log information.

TRUE Semantics with Process Log



trace1 = <Consult, Fill&Sign, Check, FormalCheck, SendCopy>
trace2 = <Consult, FormalCheck, SendCopy>

TRUE/FALSE Semantics

- There are also process execution languages with TRUE/FALSE semantics.
- They offer execution markings including markings for SKIPPED tasks and edges that are marked with FALSE.
- In the following:
  - Well-Structured Marking Nets (WSM Nets)
  - Cloud Process Execution Engine Trees (CPEE Trees)
- READ: [RRD04] S. Rinderle, M. Reichert, P. Dadam: Correctness criteria for dynamic changes in workflow systems - a survey. Data Knowl. Eng. 50(1): 9-34 (2004). DOI: 10.1016/j.datak.2004.01.002

Well-Structured Marking Nets

Definition 4.1 (Control Flow Schema (WSM-Net)). A tuple S = (N, D, NT, CtrlE, SyncE, LoopE, EC) is called a control flow schema if the following holds:

– N is a set of activities and D a set of process data elements

– NT: N →{StartFlow, EndFlow, Activity, AndSplit, AndJoin, XOrSplit, XOrJoin, StartLoop, EndLoop}

NT assigns to each node of the WSM-Net a respective node type.

– CtrlE ⊂ N × N is a precedence relation representing “normal” control dependencies between sequential activities

– SyncE ⊂ N × N is a precedence relation between activities of parallel branches

– LoopE ⊂ N × N is a set of loop backward edges

– EC: CtrlE → EdgeCode ∪ {UNDEFINED}, i.e., EC(e) assigns a selection code to the outgoing control edges of an XOR-Split. A global process data element indicates the branch to be selected.

Definition taken from [RRD04a] S. Rinderle, M. Reichert, P. Dadam: Flexible Support of Team Processes by Adaptive Workflow Systems. Distributed Parallel Databases 16(1): 91-116 (2004). DOI: 10.1023/B:DAPD.0000026270.78463.77

Well-Structured Marking Nets

- Control flow modeling is based on a block-structured and graph-based approach
- Concept of regular block structuring:
  - Control structures (sequence, branchings, loops, …) are mapped onto blocks with unique start and end activities (nodes)
  - Blocks must not overlap, but can be arbitrarily nested
  - Leads to „symmetric“ structuring of control flow graphs since splits and joins are done at specific nodes and in a pair-wise manner

Well-Structured Marking Nets

Structural correctness rules (from [RRD04a])

- S_{fwd} = (N, CtrlE, SyncE) is an acyclic graph, i.e., the use of control and sync edges must not cause undesired cycles leading to deadlocks
- For each split (loop start) node there is a unique join (loop end) node, and
- S is structured following a **block concept**; control blocks (sequences, branchings, loops) can be nested but must not overlap.

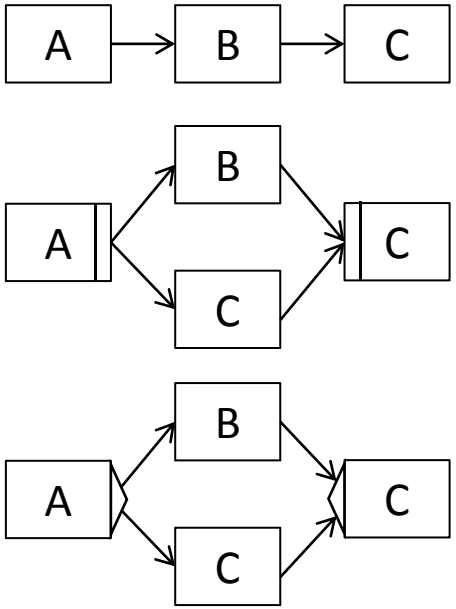
Block structure

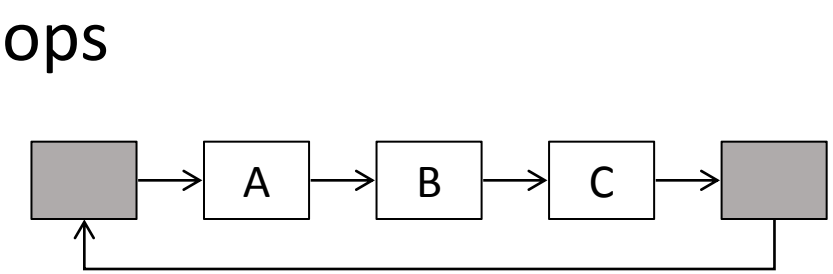
Following a (regular) block structure bears several advantages

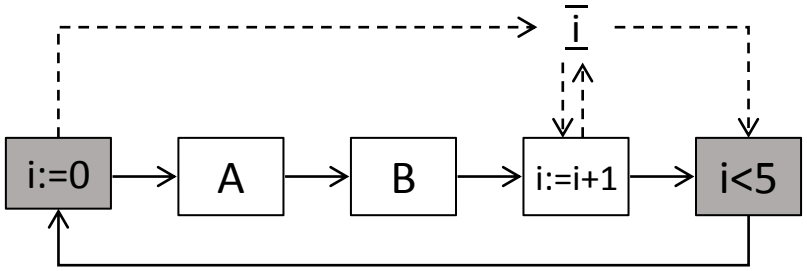
- Better user support in creating, analyzing, and changing workflow models
- Particularly: modeling is less error-prone
- Efficient correctness checks (control flow, data flow, dynamic behavior)
- Certain dynamic properties can be guaranteed „by construction“
- Support by syntax-driven editors
- Change operations can be implemented more easily
- Precise criteria for instance behavior (e.g., when is an instance finished)
- However: expressiveness of process models is restricted for strict block structuring → extensions

Well-Structured Marking Nets

- Sequence
- AND-Split / AND-Join
- XOR-Split / XOR-Join
- Loops







Well-Structured Marking Nets

Definition 4.2 (Process Instance Based on a WSM-Net - control flow perspective). A process instance I is defined by a tuple (S, M^S, Val^S, $\mathscr{D}$) where

– S = (N, D, NT, CtrlE, SyncE,...) denotes the process model the execution of I is based on.

– MS = (NS^S, ES^S) describes node and edge markings of I with:

NS^S: N → {NotActivated, Activated, Running, Completed, Skipped}

ES^S: (CtrlE ∪ SyncE ∪ LoopE) → {NotSignaled, TrueSignaled, FalseSignaled}

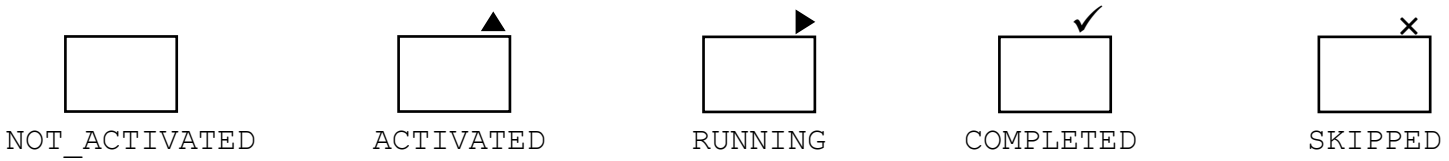
– Val^S is a function on D. It reflects for each data element d ∈ D either its current value or the value UNDEFINED (if d has not been written yet).

– $\mathscr{H}$ = $e_0,..., e_k$ is the execution history of I whereby $e_0,..., e_k$ denote the start and end events of activity executions. For each started activity X the values of data elements read by X and for each completed activity Y the values of data elements written by Y are logged.

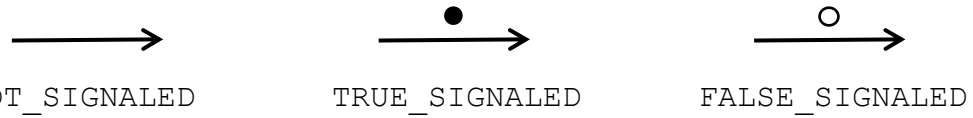
Well-Structured Marking Nets

Operational Semantics:

- Node markings (selection):



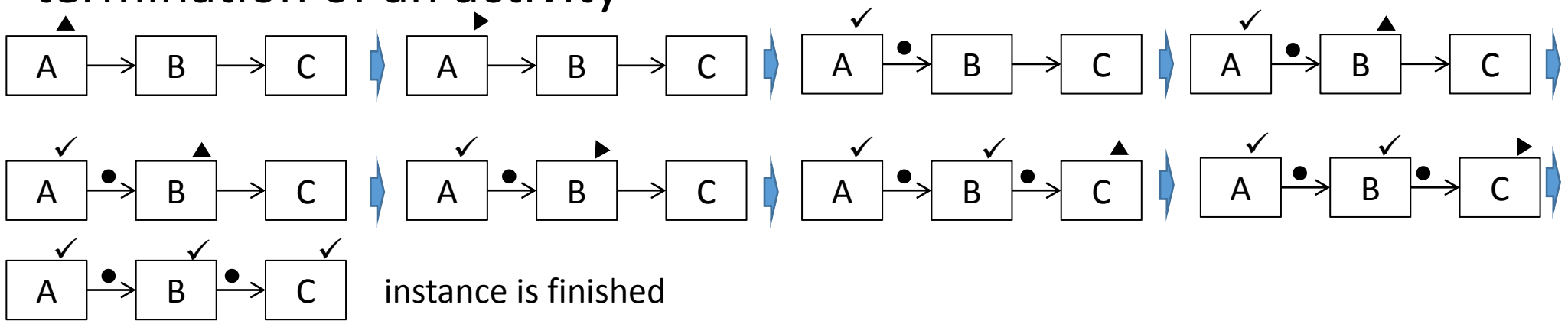
- Edge markings:



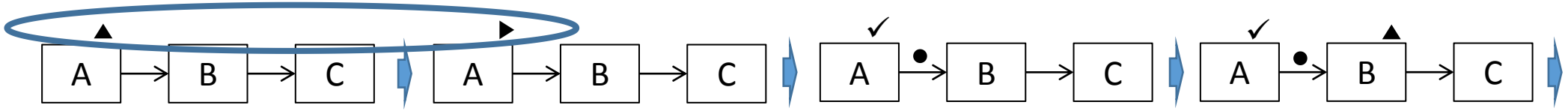
Well-Structured Marking Nets

Control flow is based on a set of well-defined execution and marking rules (cf. firing and marking rules for Petri Nets):

- **Execution rules** define under which graph markings an activity can be activated
- **Marking rules** define which node and edge markings result after termination of an activity



Well-Structured Marking Nets



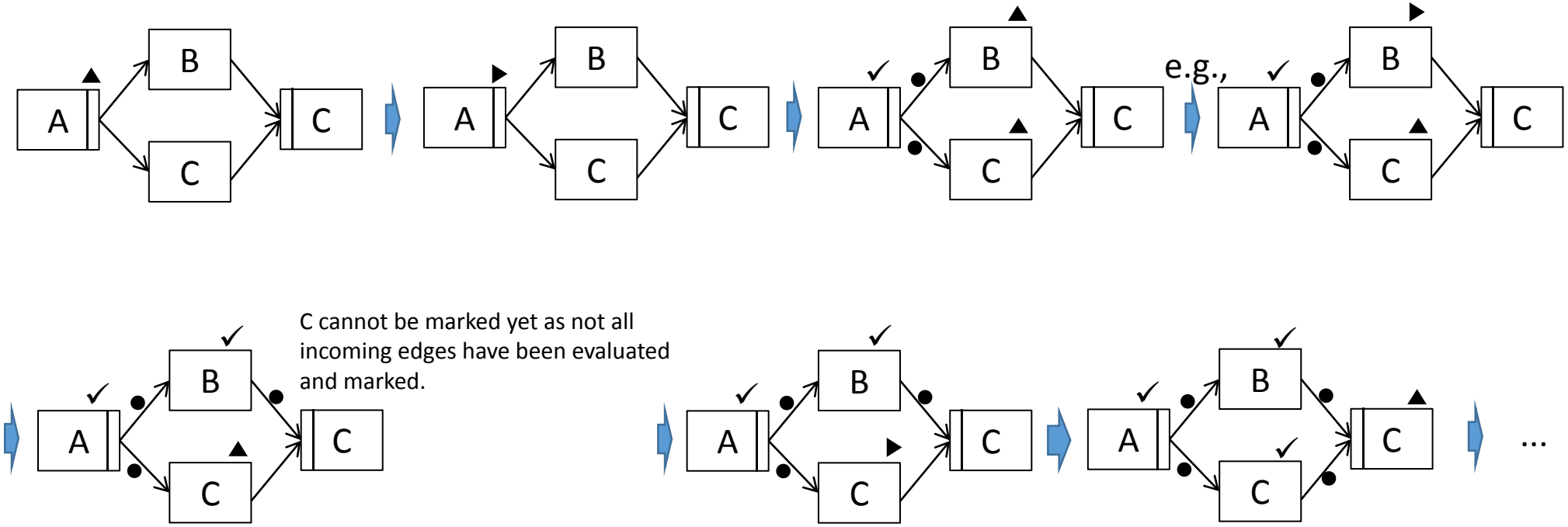
- What can we see here in comparison to executing Petri Nets?
- Distinction between states `ACTIVATED` and `RUNNING`
- This means that we can express that an activity is doing something that takes some time.
- Without time extensions, transitions in Petri Nets just fire.

Well-Structured Marking Nets

- Marking and execution rules on parallel branches
- After completing and AND-Split, all outgoing edges are marked as `TRUE_SIGNALED` and their target activities are marked as `ACTIVATED`.
- The activities in parallel branches can be then executed in arbitrary order (but have to follow the order within the branches).
- The AND-Join can only be `ACTIVATED` if all incoming edges have been evaluated and are marked as `TRUE_SIGNALED`.

Well-Structured Marking Nets

- Marking and execution rules on parallel branches based on example

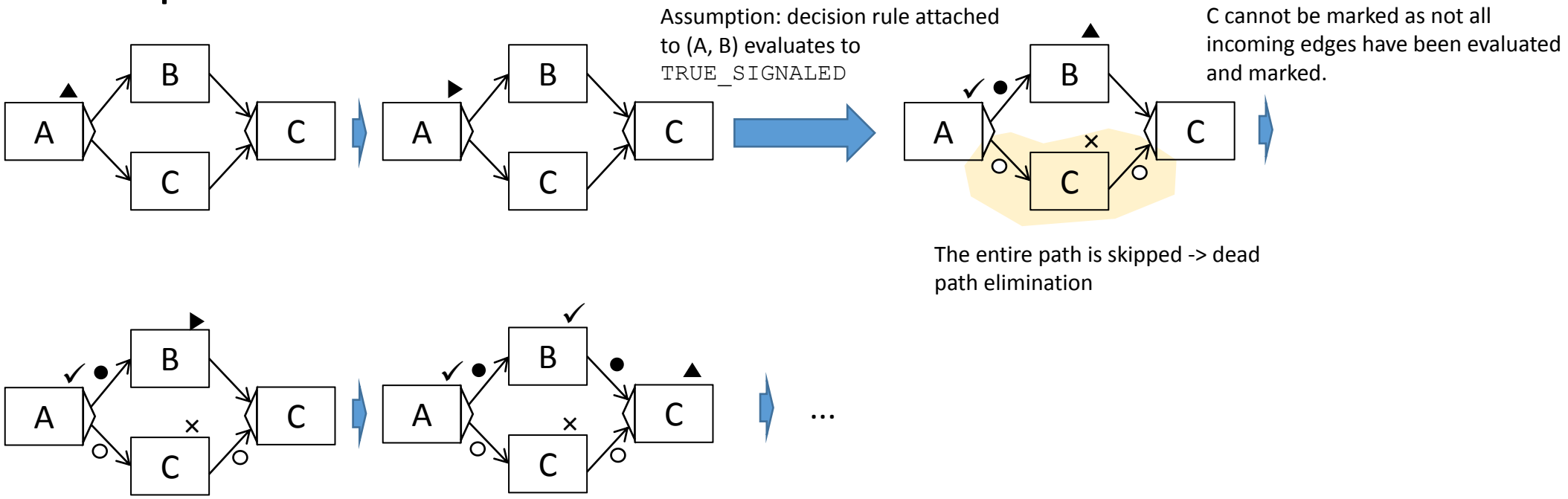


Well-Structured Marking Nets

- Marking and execution rules on alternative branches
- After completing the XOR-Split - based on a decision rule - it is determined which outgoing edge is marked as `TRUE_SIGNALED` (one) and all the other outgoing edges are marked as `FALSE_SIGNALED`.
- All activities with incoming edges that are marked as FALSE are marked as `SKIPPED`.
- If an activity is marked as `SKIPPED` all outgoing edges can be marked as `FALSE_SIGNALED` again.
- Iteratively done until XOR-Join is reached dead path elimination.
- The XOR-Join is marked as `ACTIVATED` if all incoming edges are marked and one of them is marked as `TRUE_SIGNALED` and all other are marked as `FALSE_SIGNALED`.

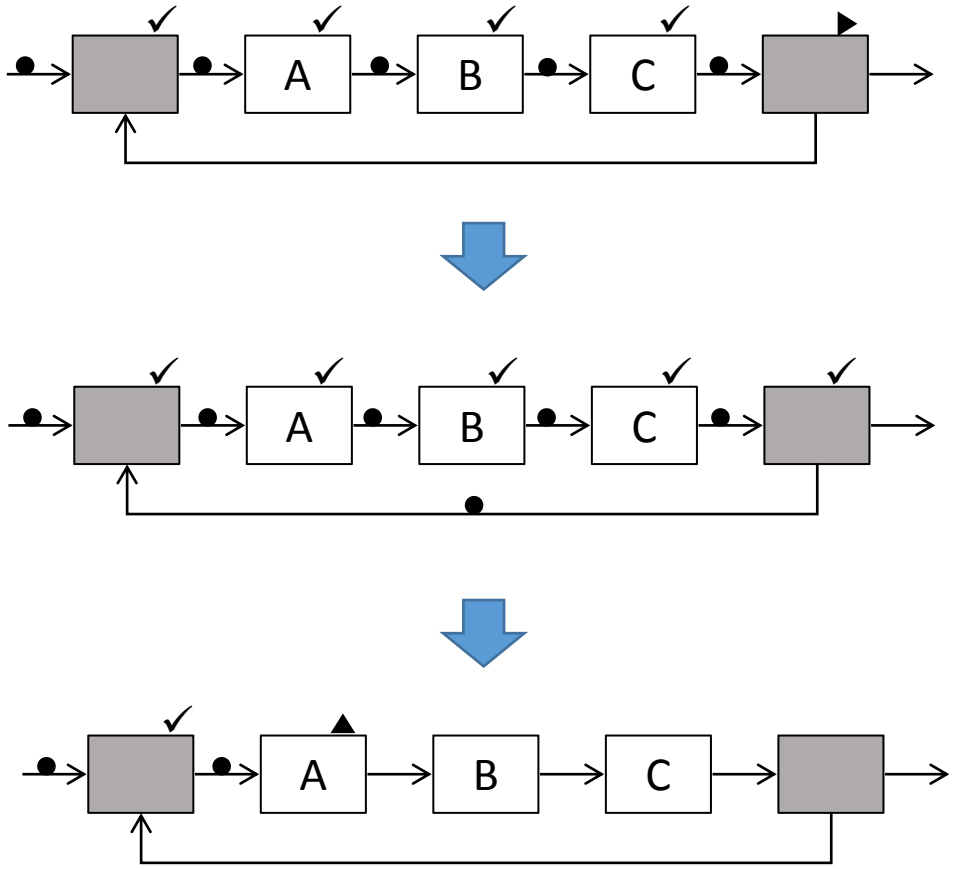
Well-Structured Marking Nets

- Marking and execution rules on alternative branches based on example



Well-Structured Marking Nets

- Marking and execution rules on alternative branches
- When the loop end node is reached, it is directly `COMPLETED` and the loop condition is evaluated.
- If the loop back edge is evaluated as `TRUE SIGNALED`, the loop start node is directly marked as `COMPLETED`.
- All other activities in the loop are marked as `NOT ACTIVATED`, except for the direct predecessor of the loop start activity. This node is marked as `ACTIVATED`.



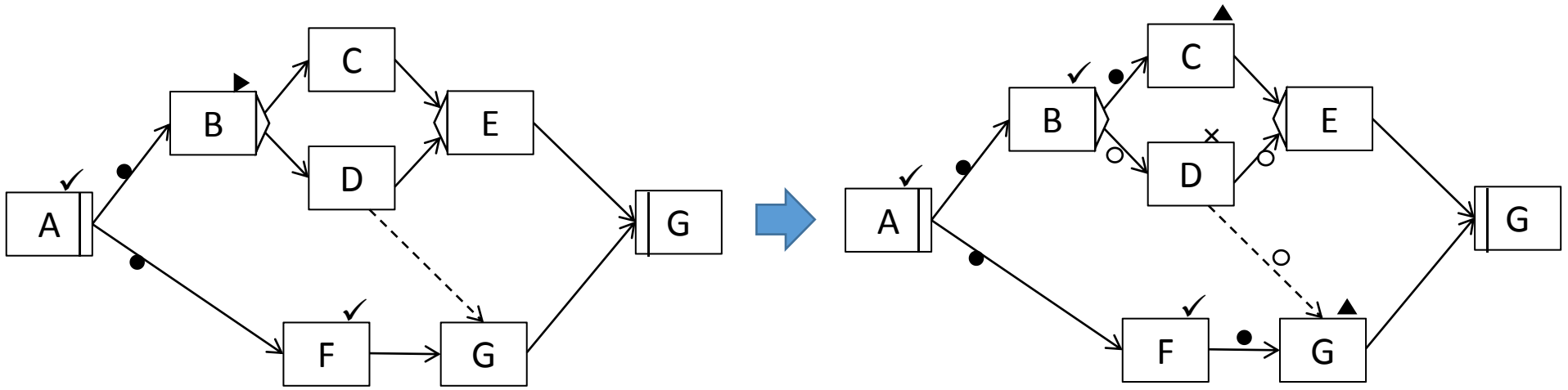
Well-Structured Marking Nets

Strict block structuring is perceived as too restrictive.

- Therefore, synchronization edges are used:
- Construct for synchronizing the execution of activities in different parallel branches
- A (soft) sync edge X → Y describes a delay between parallel activities X and Y
- Semantics: Y can be activated only if S has been successfully finished or if X cannot be executed anymore (e.g., if X has been skipped)

Well-Structured Marking Nets

Marking and execution with synchronization edges, example:



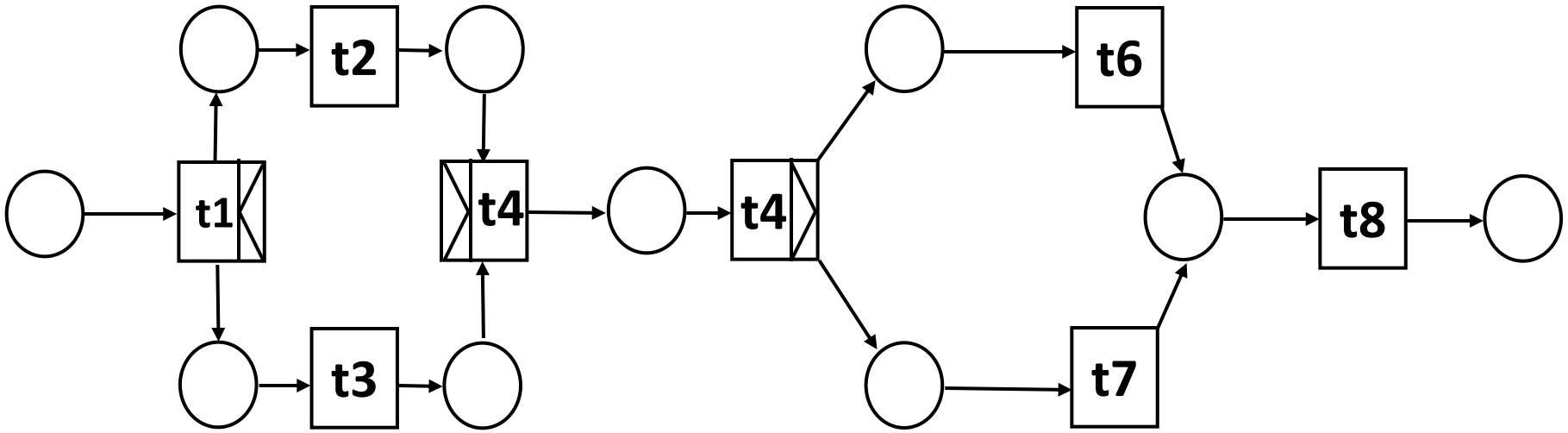
Well-Structured Marking Nets

Based on the marking and execution rules we can prove that

- For each activity of a process instance marked as `COMPLETED` or `SKIPPED` holds that each of its predecessor nodes is marked as either `COMPLETED` or `SKIPPED`.
- Under each of the reachable end markings of a workflow instance, all nodes are marked as either `COMPLETED` or `SKIPPED`
- There will be no deadlocks at runtime if it is guaranteed that the using sync edges does not produce a cycle.
- Remark: Under certain fairness assumptions (e.g., excluding infinite loops) we can conclude that all workflows terminate correctly.

Well-Structured Marking Nets

- Transformation between different formalisms is often essential.
- For example, to transform for verification and execution.
- Exercise: transform the following Workflow Net into as WSM Net



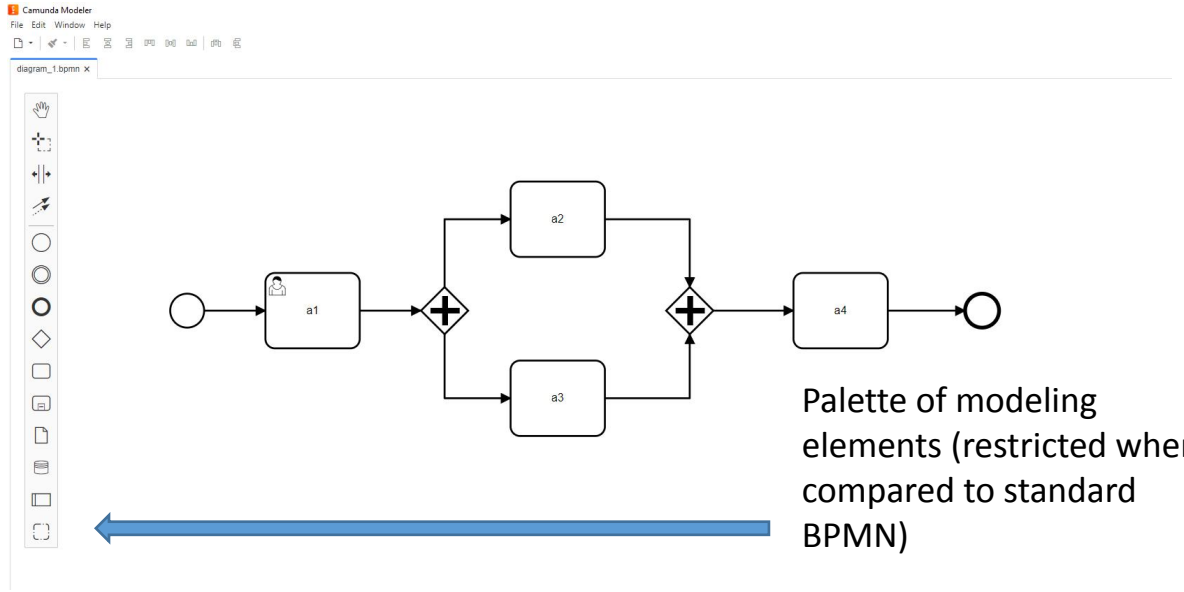
Execution on BPMN

BPMN can be used for

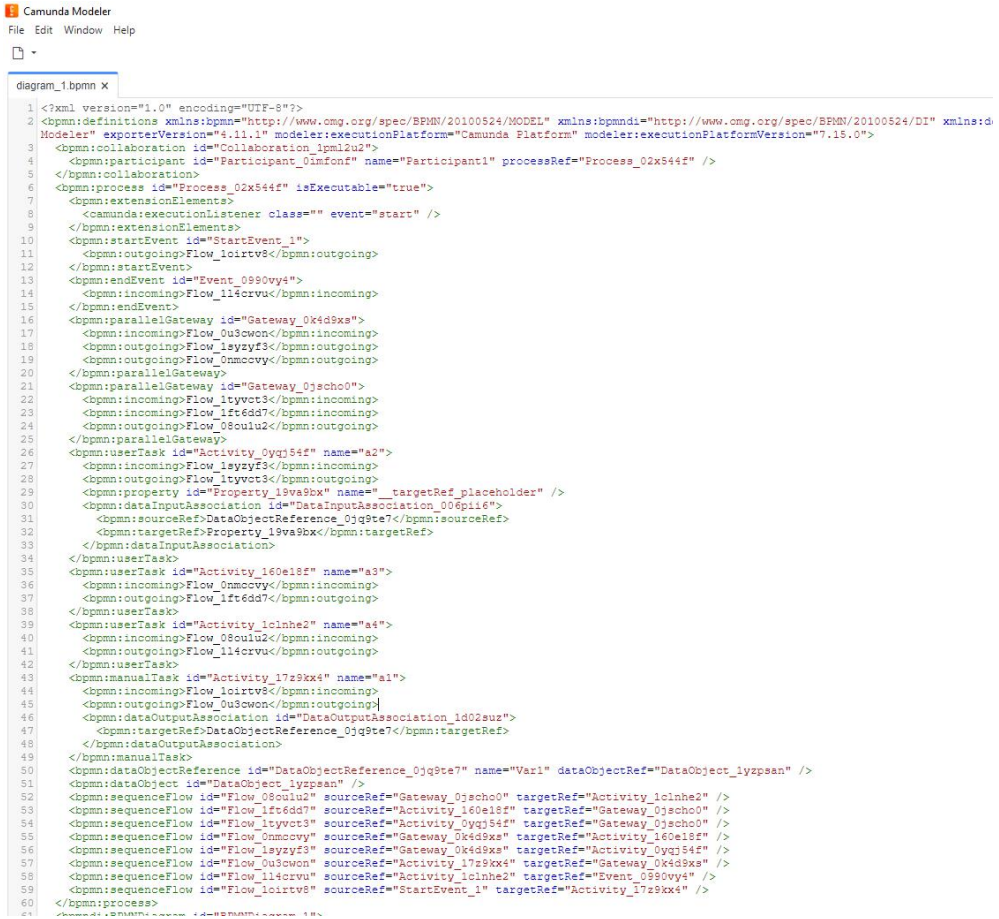
- defining executable models under strict modeling guidelines/restrictions and refinement of data flow, authorization, and invocation of web services/applications.
- defining conceptual models that can be mapped to executable models in a straigthforward way, again by employing modeling guidelines and of data flow, authorization, and invocation of web services/applications.

Execution in Camunda

- https://camunda.com/
- commercial and open source
- Will be used in the exercises.

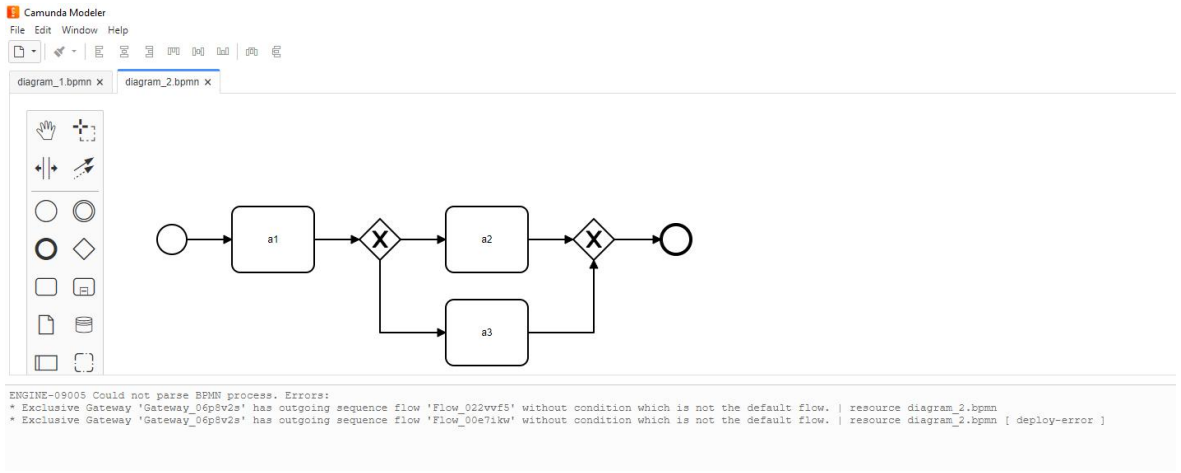


Palette of modeling elements (restricted when compared to standard BPMN)



Execution in Camunda

- Modeling rules to guide users and correctness checks before deployment to the engine
- When starting exactly one start event exists.
- Before deployment several checks are conducted.



CPEE Trees

CPEE (Cloud Process Execution Engine - cpee.org) Trees are a RPST (Refined Process Structure Tree) inspired implementation for storing process models.

Properties:

- XML: a small set of elements and their potential children (XML NS http://cpee.org/ns/description/1.0).
- No edges. The edges are implicit, i.e. derived from the structure of the tree.
- Deleting nodes does not require adjustment of edges (see: no edges).
- Inserting nodes does not require insertion of edges (see: no edges).
- As long as the tree XML is correct and adheres to an xml schema, no structural deadlocks are possible (of course there might be hidden dependencies in the individual implementation of task, which might deadlock parallel branches).

Advantages:

- Easy to parse in comparison to Workflow Graphs.
- Deletion and Insertion are trivial.
- Transformation to different formats (including workflow graphs, programming languages such as python, ...) is trivial (cmp. how difficult the parsing of workflow graphs and transforming them into an RPST is).
- Good basis for analysis and execution.
- Enforces syntactical correctness of the process model (no structural deadlocks).

CPEE Trees - **Elements 1**

ROOT <description xmlns="http://cpee.org/ns/description/1.0"/>

task *<call id="a1" endpoint="..."/>*
call to functionality, has input arguements, potentially modifies the instance context (data). A task might delegate work to humans, hardware, or arbitrary software - as long as they accept the input, and generate the expected output. This element can have no children.

parallel *<parallel wait="-1" cancel="last"/>*
The children of this element will be evaluated. The result of this evaluation determines the number of parallel branches. The mode determines the join semantic: **-1** means that all parallel branches have to be finished. Any number **bigger than 0** means that after this number of branches (or all), all remaining branches will be canceled. The cancel attribute further refines the join semantic: if wait is 1 and cancel is **last**, then after 1 branch ended, all other branches are canceled, no matter at which task they currently are. If wait is 1 and cancel is **first**, after the first task of any branch finishes, all other branches are canceled, and the original branch continues normally. This is the equivalent of an event-based gateway.

CPEE Trees - **Elements 2**

loop *<loop mode="pre_test" condition="..."/>*
The children of this element will be repeated over and over. The mode determines if the condition is evaluated before or after the first iteration. The loop continues until the condition is false.

- **escape** *<escape/>*
  jump out of the loop if execution is inside a loop, even if the loop is not the parent. If no ancestor is a loop, the escape is ignored.

choose *<choose mode="exclusive"/>*
No children except the following two are allowed. The mode allows for differentiation between exclusive and inclusive or. Important note: the order of the children determines the order of the evaluation.

- **alternative** *<alternative condition=""/>*
  if the condition evaluates to true, an alternatives' children are traversed.
- **otherwise** *<alternative condition=""/>*
  if no alternative has been found the children of the otherwise element are traversed. If no otherwise can be found, an empty otherwise is assumed. The otherwise does not have to be the last child of ***choose*** *to work properly.*

CPEE Trees - **Elements 3**

script task *<manipulate id="a2" label="...">...</manipulate>*
can hold arbitrary code, it is up to execution to make use of this code. The code may modify instance context (data).

terminate *<terminate/>*
immediately terminates execution if encountered. Is not intended to have children.

stop *<stop id="a2"/>*
is intended to pause the execution of an instance. The difference between terminate and stop is, that termination is final - an instance is finished after execution. A stopped instance might continue executing at exactly the position it stopped. Is not intended to have children.

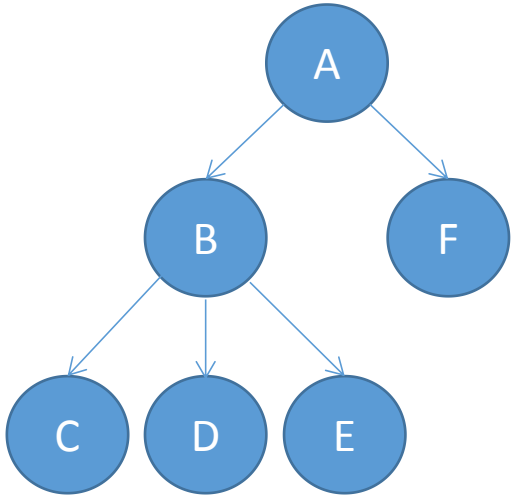
Generic note - only elements where a potential execution can stop have an id:

- task
- script task
- stop

All other elements can not be interrupted during exection.

CPEE Trees - **CPEE Trees are ...**

- Ordered Trees: the order of elements in each node is important.
- Depth first traversed (see exception: parallel branch).
- Pre-order traversed (ABCDEF).
- Leaves are either (if any other element is a leaf, it can be ignored):
  - Task
  - Script
  - Terminate
  - Stop
  - Escape
- Leaves have no other elements as children.
- Branches are either
  - Choose - Alternative and Otherwise are allowed children
  - Alternative - all elements are allowed children
  - Otherwise - all elements are allowed children
  - Parallel - all elements are allowed children (use only Parallel Branches for standard BPMN syntax)
  - Parallel Branch - all elements are allowed children



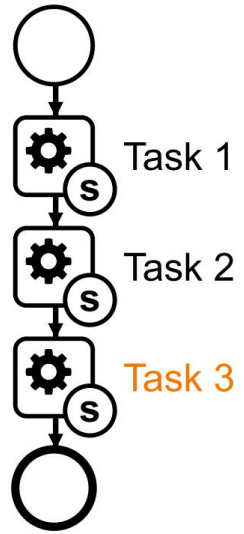
CPEE Trees - **Traversal - Sequence**

depth-first, pre-order traversal; only one possible traversal:

- ABCD

order of elements B, C, D is important and influences the execution

```
A <description xmlns="http://cpee.org/ns/description/1.0">
B   <call id="a1" endpoint=""/>
C   <call id="a2" endpoint=""/>
D   <call id="a3" endpoint=""/>
  </description>
```

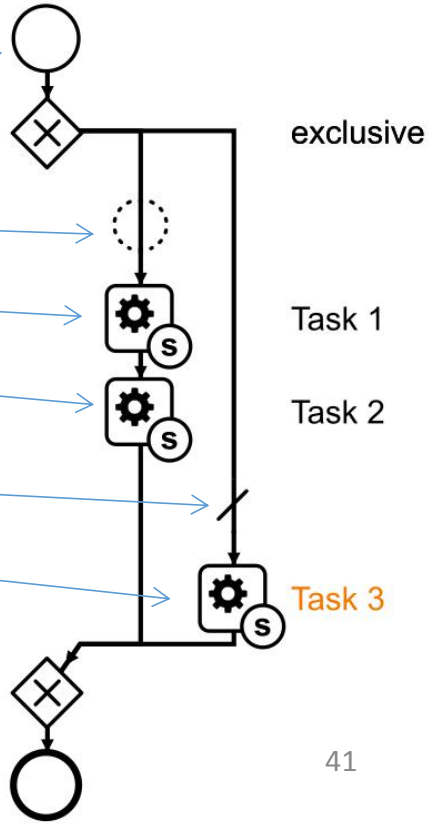


CPEE Trees - XOR (BPMN Exclusive Gateway)

depth-first, pre-order traversal; potential traversals:

- ABCDE
- ABCFG

```
A <description xmlns="http://cpee.org/ns/description/1.0">
B  <choose mode="exclusive">
C   <alternative condition="">
D    <call id="a1" endpoint=""/>
E    <call id="a2" endpoint=""/>
    </alternative>
F   <otherwise>
G    <call id="a3" endpoint=""/>
    </otherwise>
   </choose>
  </description>
```

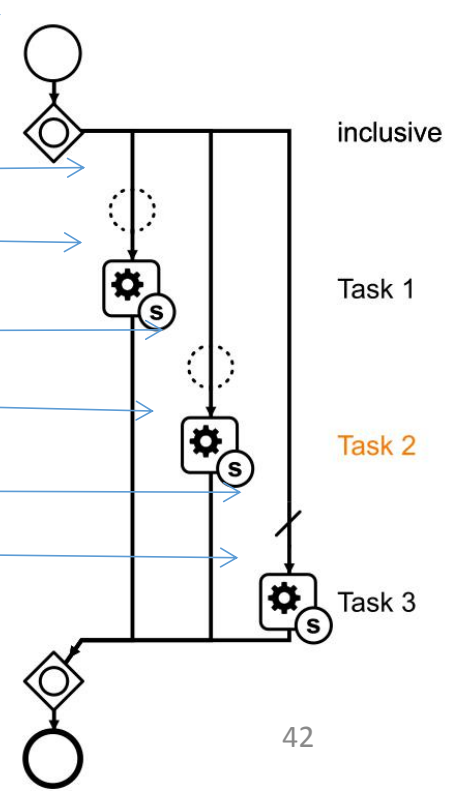


CPEE Trees - **OR (BPMN Inclusive Gateway)**

depth-first, pre-order traversal; potential traversals:

- ABCDE, ABCEF, ABCEGH only one branch or otherwise
- ABCDEF multiple branches

```
A <description xmlns="http://cpee.org/ns/description/1.0">
B  <choose mode="inclusive">
C   <alternative condition="">
D    <call id="a1" endpoint=""/>
    </alternative>
E   <alternative condition="">
F    <call id="a2" endpoint=""/>
    </alternative>
G   <otherwise>
H    <call id="a3" endpoint=""/>
    </otherwise>
   </choose>
  </description>
```

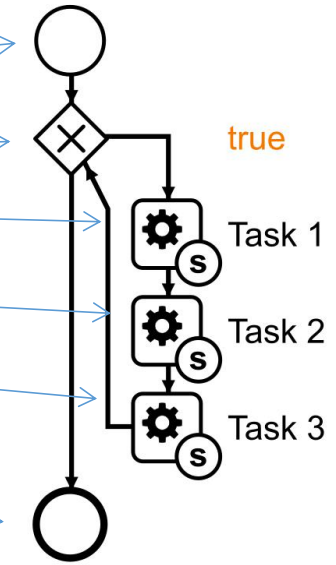


CPEE Trees - **Pre-Test Loop**

depth-first, pre-order traversal; potential traversals:

- A(BCDE)*B (condition from B is tested before each CDE)
- e.g. for 3 iterations: ABCDEBCDEBCDEB
- * means 0..n

```
A <description xmlns="http://cpee.org/ns/description/1.0">
B   <loop mode="pre_test" condition="true">
C     <call id="a1" endpoint=""/>
D     <call id="a2" endpoint=""/>
E     <call id="a3" endpoint=""/>
    </loop>
  </description>
```

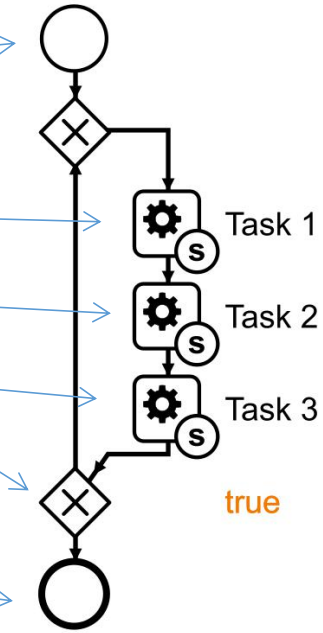


CPEE Trees - **Post-Test Loop**

depth-first, pre-order traversal; potential traversals:

- AB(CDEB)+ (condition from B is tested after each CDE)
- + means 1..n
- The grey B occurs, but is not tested

```
A <description xmlns="http://cpee.org/ns/description/1.0">
B   <loop mode="post_test" condition="true">
C     <call id="a1" endpoint=""/>
D     <call id="a2" endpoint=""/>
E     <call id="a3" endpoint=""/>
    </loop>
  </description>
```

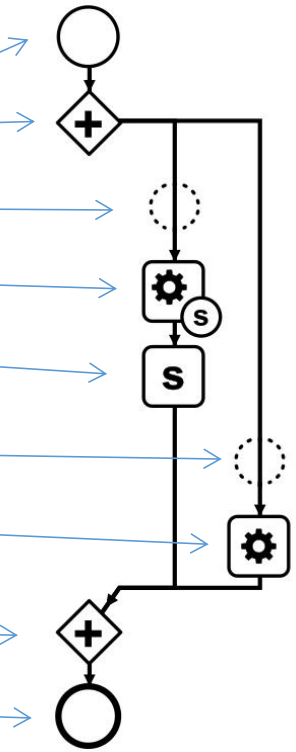


CPEE Trees - **Parallel**

depth-first + breadth, pre-order traversal; potential traversals:

- ABCFDEG
- ABCFDGE
- ABCFGDE

```
A <description xmlns="http://cpee.org/ns/description/1.0">
B   <parallel wait="-1" cancel="last">
C     <parallel_branch>
D       <call id="a1" endpoint=""/>
E       <manipulate id="a3" label=""/>
      </parallel_branch>
F     <parallel_branch>
G       <call id="a2" endpoint=""/>
      </parallel_branch>
    </parallel>
  </description>
```

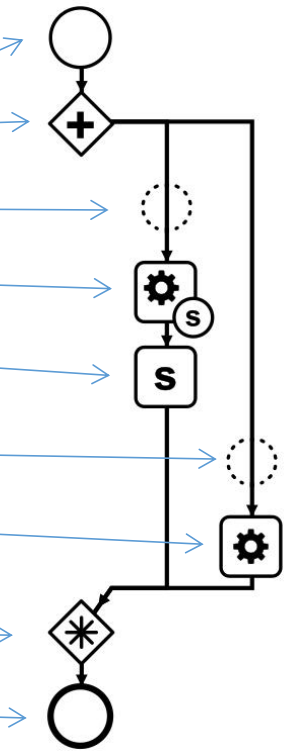


CPEE Trees - **Parallel - Special Join**

depth-first + breadth, pre-order traversal; potential traversals:

- ABCFDEG, ABCFDGE, ABCFGD
- ABCFDE, ABCFDG, ABCFG (the G,E,D have not yet been started)
- tasks may have irreversible consequences (i.e. sending an email); for ABCFG, ignoring E can be problematic.

```
A <description xmlns="http://cpee.org/ns/description/1.0">
B   <parallel wait="1" cancel="last">
C     <parallel_branch>
D       <call id="a1" endpoint=""/>
E       <manipulate id="a3" label=""/>
      </parallel_branch>
F     <parallel_branch>
G       <call id="a2" endpoint=""/>
      </parallel_branch>
    </parallel>
  </description>
```

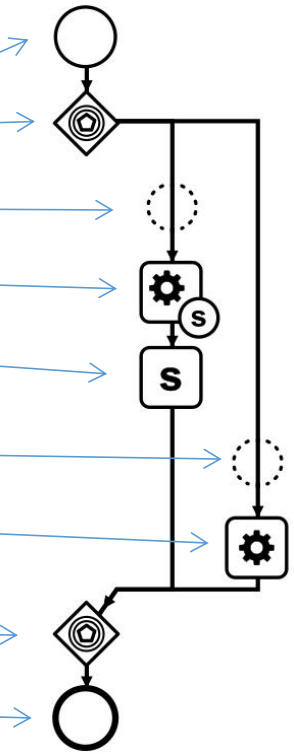


CPEE Trees - **Parallel - Event-Based Gateway**

depth-first + breadth, pre-order traversal; potential traversals:

- ABCFG, ABCFDE
- if D and G are considered events that are waited for, these tasks are non-consequential. Ignoring either D or G will have no consequences.

```
A <description xmlns="http://cpee.org/ns/description/1.0">
B   <parallel wait="1" cancel="first">
C     <parallel_branch>
D       <call id="a1" endpoint=""/>
E       <manipulate id="a3" label=""/>
      </parallel_branch>
F     <parallel_branch>
G       <call id="a2" endpoint=""/>
      </parallel_branch>
    </parallel>
  </description>
```

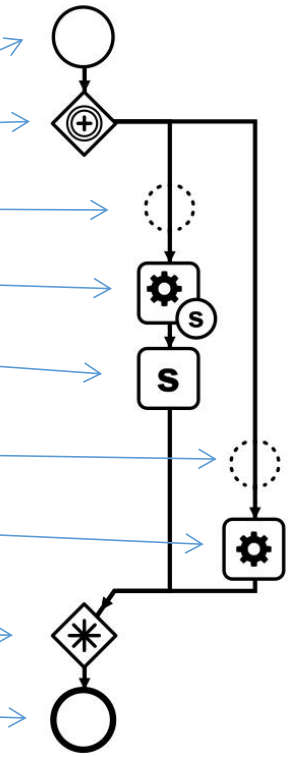


CPEE Trees - **Parallel - Special Cancelation**

depth-first + breadth, pre-order traversal; potential traversals:

- ABCFDEG, ABCFDGE, ABCFGDE
- in this case behaves like a standard parallel. If more branches existed, this would realize a special case of the Event-Based Gateway

```
A <description xmlns="http://cpee.org/ns/description/1.0">
B   <parallel wait="3" cancel="first">
C     <parallel_branch>
D       <call id="a1" endpoint=""/>
E       <manipulate id="a3" label=""/>
      </parallel_branch>
F     <parallel_branch>
G       <call id="a2" endpoint=""/>
      </parallel_branch>
    </parallel>
  </description>
```

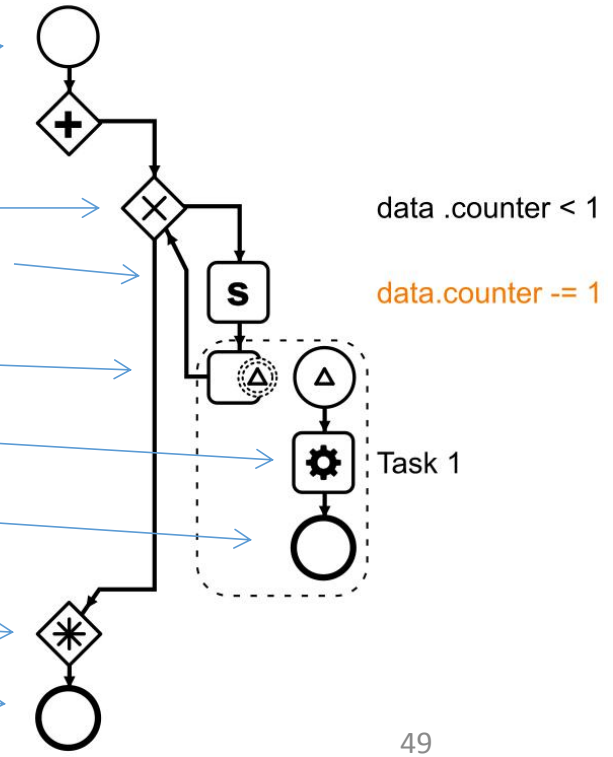


CPEE Trees - **Variable Parallel**

depth-first + breadth, pre-order traversal; potential traversals:

- AB(CDE)*C(F)*
- e.g. for counter = 3: ABCDECDECDECFFF
- arbitrary elements might exist as children of parallel; they determine the number of branches.

```
A <description xmlns="http://cpee.org/ns/description/1.0">
B  <parallel wait="-1" cancel="last">
C   <loop mode="pre_test" condition="data .counter &lt; 1">
D    <manipulate id="a2" label="data.counter -= 1">data.counter -= 1</manipulate>
E    <parallel_branch>
F     <call id="a1" endpoint=""/>
     </parallel_branch>
    </loop>
   </parallel>
  </description>
```

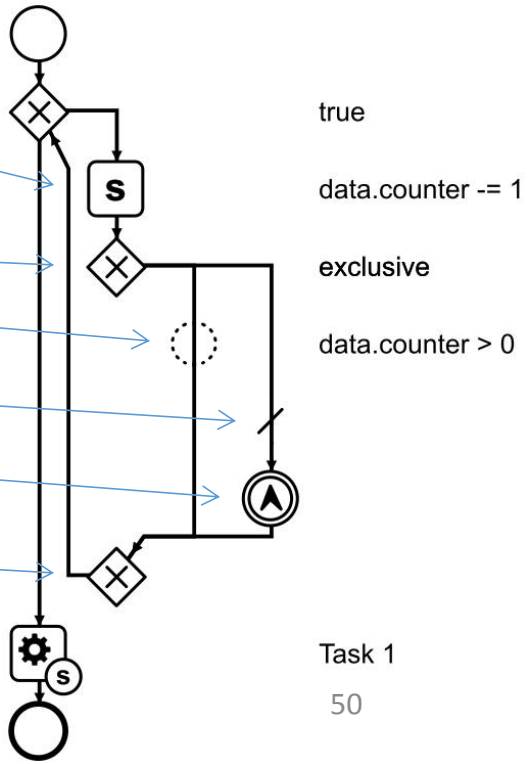


CPEE Trees - **Escape (BPMN Escalate)**

depth-first, pre-order traversal; only one possible traversal:

- A(BCDE)*FGH
- e.g. for counter = 3: ABCDEBCDEBCDEFGH

```
A <description xmlns="http://cpee.org/ns/description/1.0">
B  <loop mode="pre_test" condition="true">
C   <manipulate id="a1" label="data.counter -= 1">data.counter -= 1</manipulate>
D   <choose mode="exclusive">
E    <alternative condition="data.counter &gt; 0"/>
F    <otherwise>
G     <escape/>
     </otherwise>
    </choose>
   </loop>
H  <call id="a1" endpoint=""/>
  </description>
```

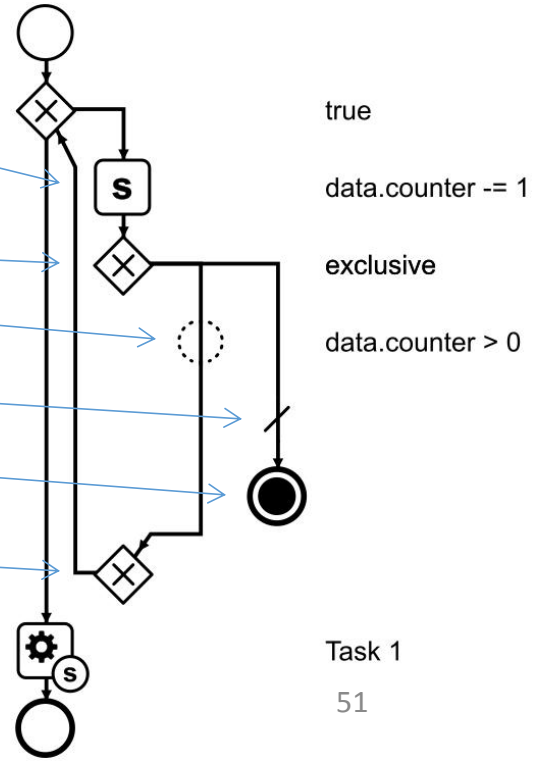


CPEE Trees - **Terminate**

depth-first, pre-order traversal; only one possible traversal:

- A(BCDE)*FG
- e.g. for counter = 3: ABCDEBCDEBCDEFG

```
A <description xmlns="http://cpee.org/ns/description/1.0">
B  <loop mode="pre_test" condition="true">
C   <manipulate id="a1" label="data.counter -= 1">data.counter -= 1</manipulate>
D   <choose mode="exclusive">
E    <alternative condition="data.counter &gt; 0"/>
F    <otherwise>
G     <terminate/>
     </otherwise>
    </choose>
   </loop>
H  <call id="a1" endpoint=""/>
  </description>
```



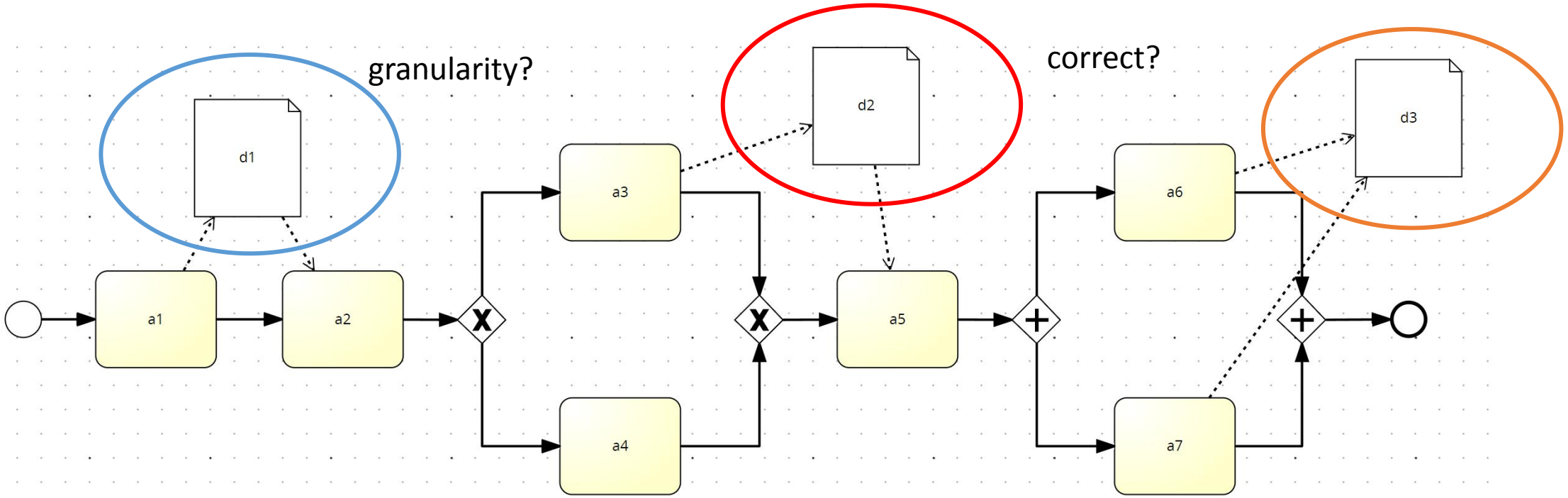
Developing Executable Processes...

... requires the definition and specification of the following aspect (selection):

- Control Flow (Behavioral Aspect)
- Data Flow (Information Aspect)
- Activities (Functional Aspect)
- Deadlines (Temporal Aspect)
- Actor Assignments (Organizational Aspect)
- Application Services (Operational Aspect)

Data Flow

- Data flow modeling in BPMN



Data flow in WSM Nets

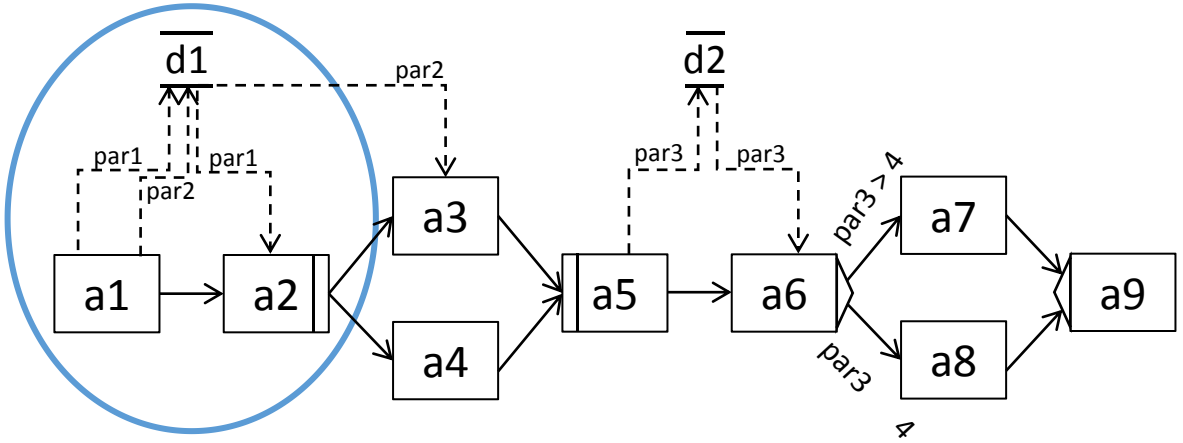
Definition 4.3 (Data flow schema, simplified from [ReDa98]). Let S = (N, D, NT, ...) be the control flow schema (based on WSM Nets) and let D denote a finite set of data elements associated with S. Let further PARS(X) denote the set of parameters associated with the task X [PARS(X) := InPARS(X) ∪ OutPARS(X)]. A data link df between a parameter par^{df} and a data element d^{df} is then described by the 4-tuple $df = (d^{df}, n^{df}, par^{df}, access\_mode^{df})$ with

- $d^{df} \in D$, $n^{df} \in N$, $par^{df} \in PARS(n^{df})$, $access\_mode^{df} \in \{read, write\}$

The set of all data links DF, connecting task or service parameters with global data elements from D, is called the data flow schema of S.

[ReDa98] M. Reichert, P. Dadam: ADEPTflex-Supporting Dynamic Changes of Workflows Without Losing Control. J. Intell. Inf. Syst. 10(2): 93-129 (1998). DOI: 10.1023/A:1008604709862

Data flow in WSM Nets



granularity at parameter level

- DF = {(d1, a1, par1, write), (d1, a1, par2, write), (d1, a2, par1, read), (d1, a3, par2, read), (d2, a5, par3, write), (d2, a6, par3, read)}
- Execution: read access when activity is started

Data flow in WSM Nets

Definition 4.2 (Process Instance Based on a WSM-Net). A process instance I is defined by a tuple (S, M^S, Val^S, $trace^I$) where

– S = (N, D, NT, CtrlE, SyncE,...) denotes the process schema the execution of I is based on.

– MS = (NS^S, ES^S) describes node and edge markings of I with:

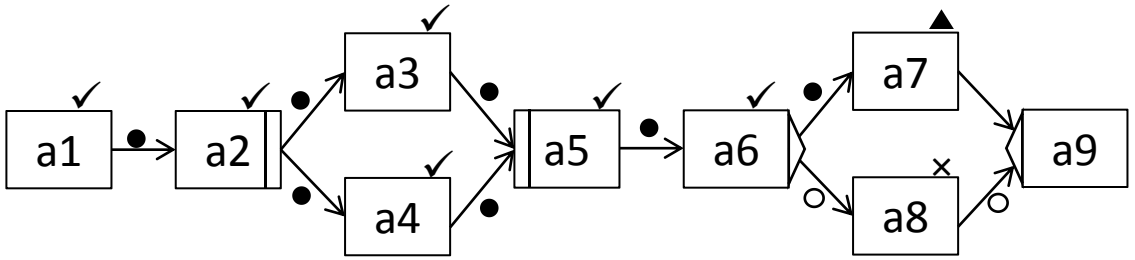
NS^S: N → {NotActivated, Activated, Running, Completed, Skipped}

ES^S: (CtrlE ∪ SyncE ∪ LoopE) → {NotSignaled, TrueSignaled, FalseSignaled}

– Val^S is a function on D. It reflects for each data element d ∈ D either its current value or the value UNDEFINED (if d has not been written yet).

– $trace^I$ = $e_0,..., e_k$ is the execution history of I whereby $e_0,..., e_k$ denote the start and end events of activity executions. For each started activity X the values of data elements read by X and for each completed activity Y the values of data elements written by Y are logged.

Data flow in WSM Nets



- DF = {(d1, a1, par1, write), (d1, a2, par2, read), (d1, a3, par1, read), (d2, a5, par3, write), (d2, a6, par4, read)}
- $trace^{I}$ =<start(a1), end(a1, write{(d1, par1, „Smith“), (d1, par2, 34)}), start(a2, read(d1, par1, „Smith“)), end(a2), start(a3, read(d1, par2, 34)), start(a4), end(a4), end(a3), start(a5), end(a5, write(d2, par3, 5)), start(a6, read(d2, par3, 5))>

Data flow Correctness

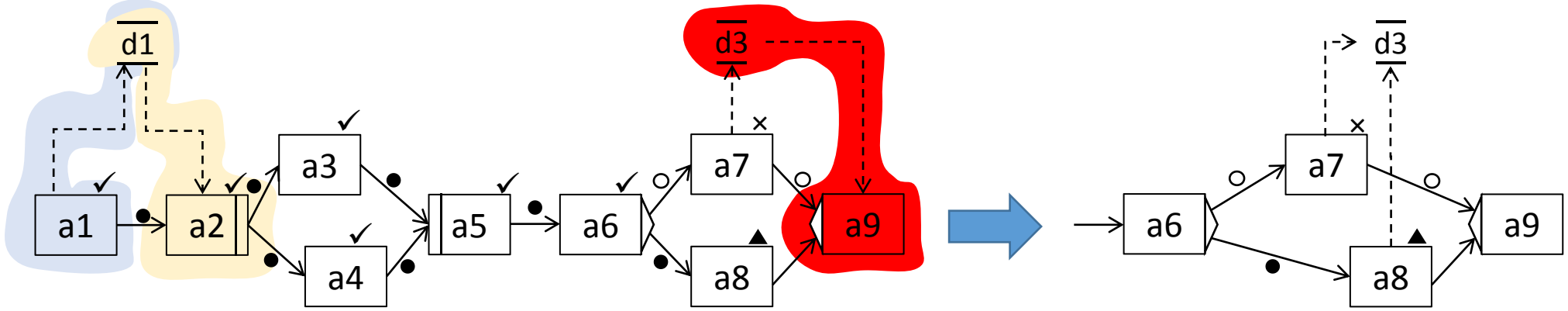
Rule 1: For every read access to a data element d, more specifically to a parameter par^d, by activity a, there must exist a mandatory write access to d, pard by a predecessor activity a‘ of a.

Rule 1 guarantees the supply of data elements for read accesses and hence prevents that data elements are missing, resulting in, for example, failure of activity invocations during runtime.

Particularly „dangerous“ are read accesses from branches of alternative branches (see next slide). Requirement: the data element must be written from each branch.

Data flow Correctness

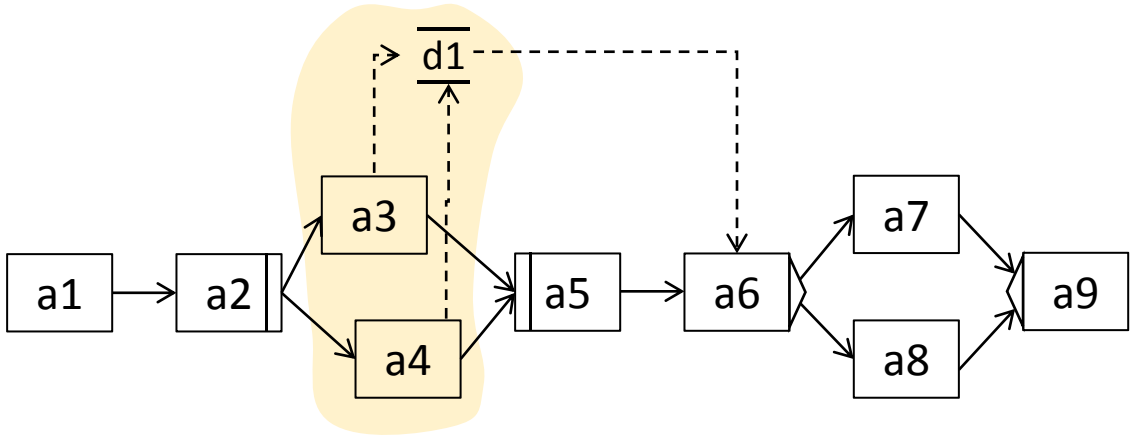
Rule 1 illustrated based on the example WSM Net



- a2 reads d1, par1; a1 is predecessor of a2 with mandatory write access
- a9 reads d3, but there is no mandatory write access, i.e., d3 might be written (if a7 is activated and executed), but could by also not written in case a7 is skipped.
- Hence, for alternative branchings it is required that the data element is written by activities from all branches.

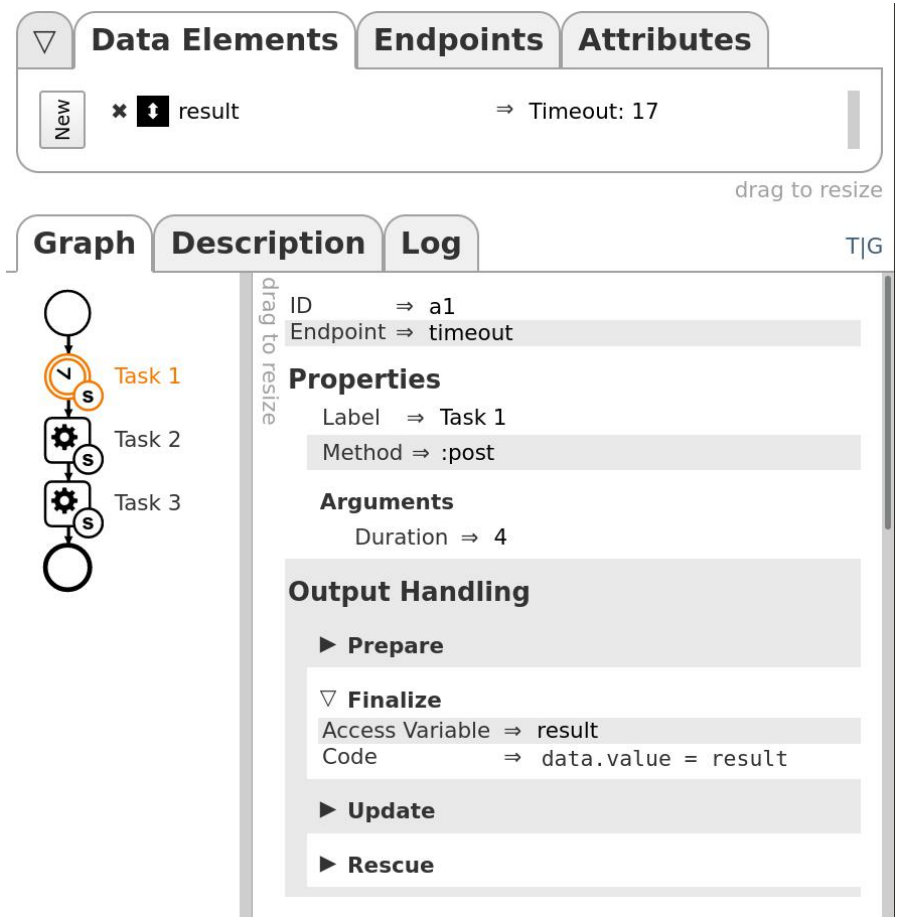
Data flow Correctness

Rule 2: no concurrent write accesses to data elements/parameters by activities in parallel branches.



- a3 and a4 write d1 (in abritrary order); hence one write access is always overwritten by the other one.
- This could happen also to subsequent write accesses, if the data element is not read by an activity in between.

CPEE Trees - **Data**



All input to a task must come from the process context.

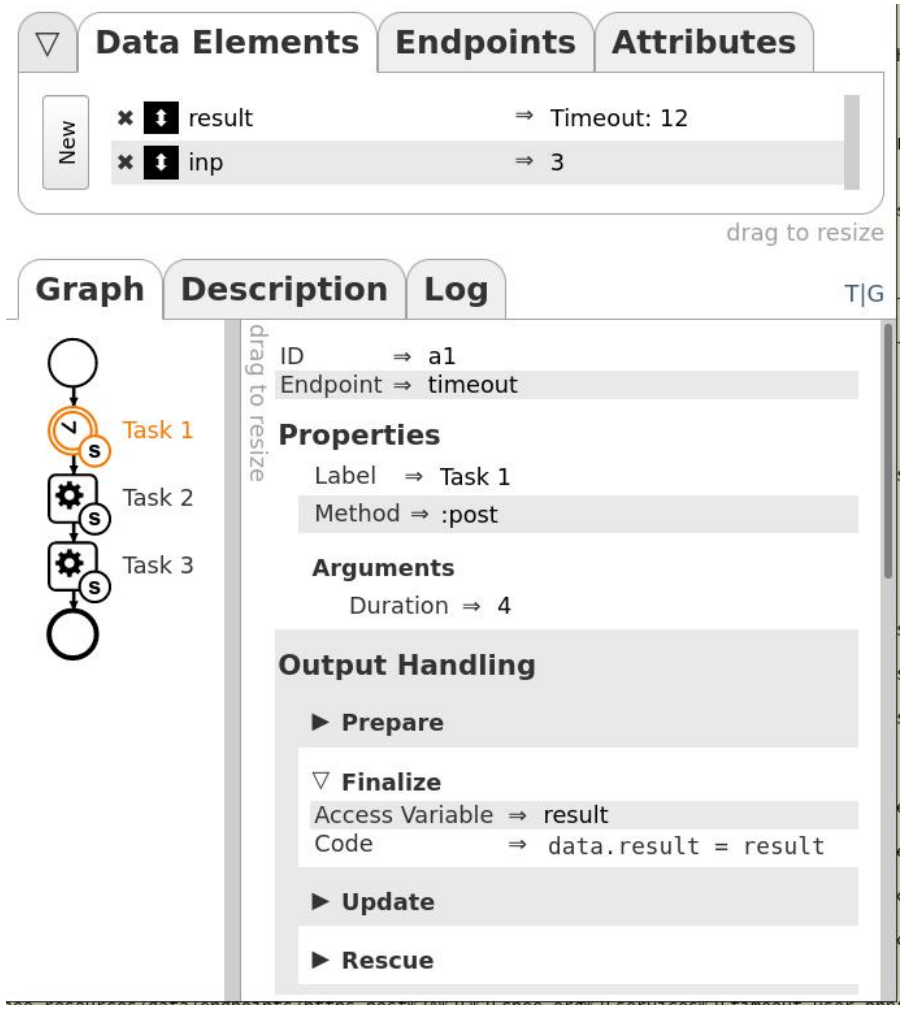
Output from a task can be stored in the process context.

The process context consists of a list of variables with a name and a value. For cpee.org the values can be:

- String
- Integer
- Float
- JSON

Everything else will be represented as an IETF RFC 2397 data-url.

CPEE Trees - **Data**



Example on the left:

- One data element named “result” exists
- The data element has the string value “Timeout: 12”
- Task 1 is an event which is implemented through the endpoint “timeout”
- Task 1 is implemented through a REST service
  - the call method is “post”
  - the static argument is “4”

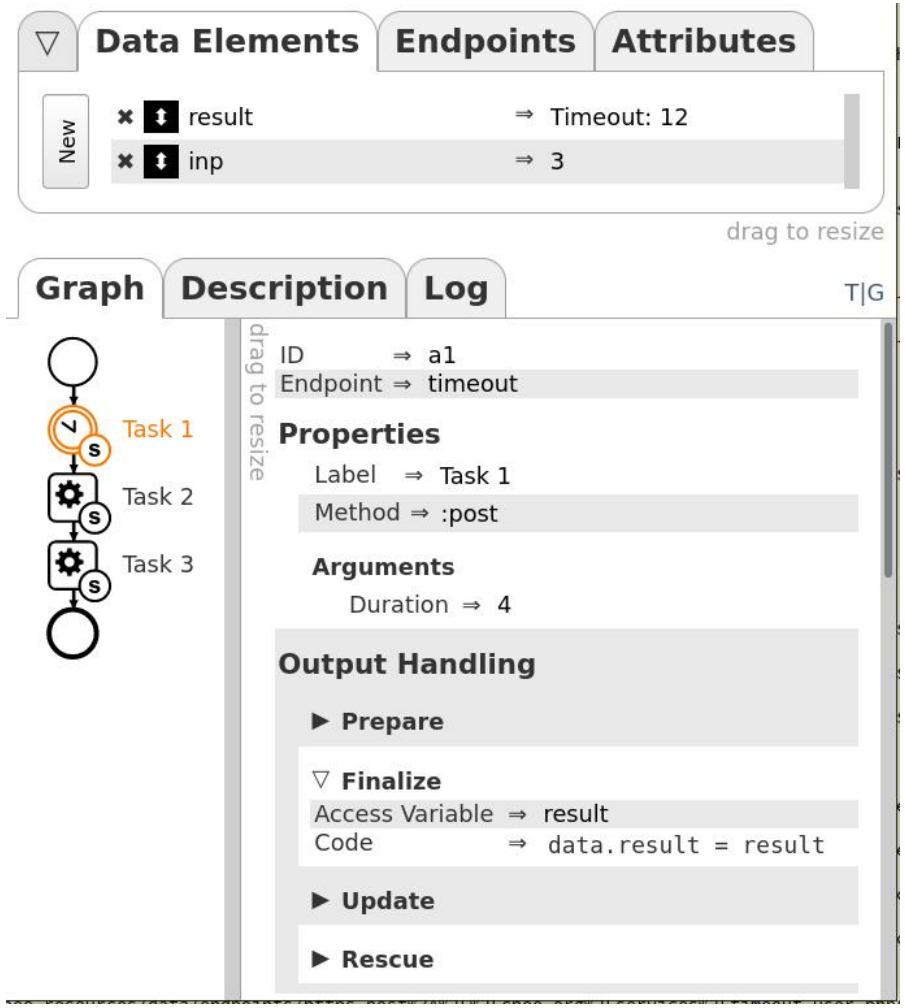
For a process to be dynamic, i.e. read data in one task, and pass it as an argument to the next task, it is necessary to:

- use data elements as argument values
- write data elements when a task returns data

Example on the left:

- To pass the data element “inp” as an argument
  Duration => !data.inp
- Each argument value that starts with a **!** can access all data elements, all endpoints and all arguments with **data.name, endpoints.name or arguments.name**
- The output of the task/event is handled through a code snippet in “Finalize”:
  - Everything that is returned is stored in “result”
  - Everything that should be kept in the process context can be assigned to a data element, i.e.: data.name = result
  - If result is a json, keys can accessed, i.e.: data.name = result[“key”]

CPEE Trees - **Data**



Data flow is neither graphically represented nor represented by special elements in CPEE trees.

- Arguments describe input to the implementation
- Output handling dynamically filters the received data and moves it to the process context (data elements).

Developing Executable Processes...

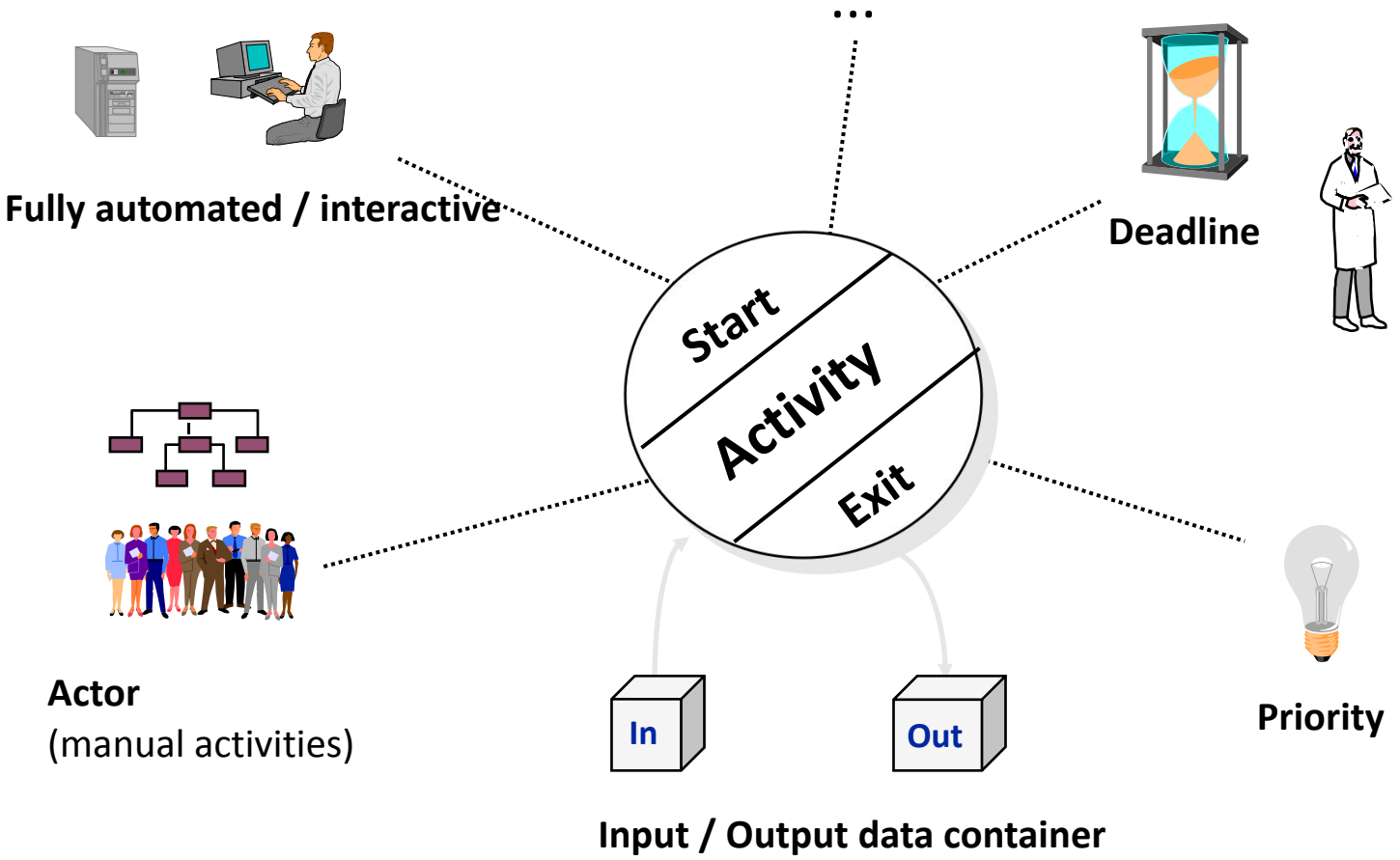
... requires the definition and specification of the following aspect (selection):

- Control Flow (Behavioral Aspect)
- Data Flow (Information Aspect)
- Activities (Functional Aspect)
- Deadlines (Temporal Aspect)
- Actor Assignments (Organizational Aspect)
- Application Services (Operational Aspect)

Functional Aspect: Activities

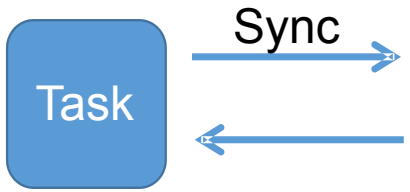
- Activity: a description of a piece of work that forms one logical step in a process model. Activities are also called tasks.
- Requires human and/or machine resource(s) for its execution
- During runtime activity executions are controlled by the process management system/process engine. This may result in:
  - executed application services being automatically invoked by the process management system/process engine, i.e., with no process participant being involved (*fully automated*)
  - one or more work items being assigned to actors who may work on the respective activity (*interactive*)

Functional Aspect: Activities



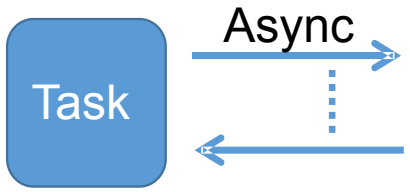
Functional Aspect: Interaction Patterns

Tasks communicate with functionalities (built-in, webservices, ...) that implement them. Tasks send input (optional), and receive output that can be merged with the process context. The following basic interaction patterns between tasks and their implementations exist:



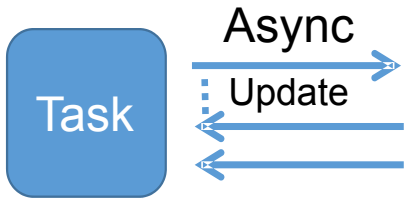
Answer a request immediately. It is not important for implementations to know details about the tasks.

- For requests where the implementation can return data immediately.
- A process can only be stopped when upon return.



Answer a request later. An implementation has to know instance and task id, in order to sucessfully address the task upon return.

- For long-running requests (can be months or years).
- A process engine can cancel such as task if it is e.g., stopping. Explanation: the implentation actively returns data, thus can get information if data is no longer need, and therefor can react accordingly.



Answer a request multiple times. Special form of Async.

- For long-running requests (can be months or years).
- For informing the process engine about progress, or sending partial results.

Functional Aspect: Interaction Patterns

Camunda: implementations are external services which utilize a library/API

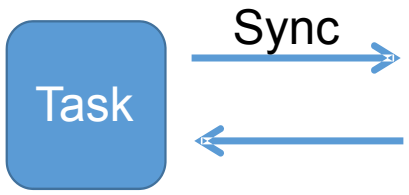
- Java through org.camunda.bpm.client.ExternalTaskClient
- NodeJS ycamunda-external-task-client-js
- Python camunda-external-task-client-python3
- Each implementation utilizes a camunda/community provided API. The features of the API controls all interaction with the engine.

cpee.org

- Utilizes plain HTTP with special HTTP headers instead of a custom API.
- Fully language agnostic, all programming languages with support for HTTP can be used.
- Headers sent with each HTTP call to an implementation:
  - CPEE-BASE - base location of the engine where the instance is running on (e.g. cpee.org/flow/engine/)
  - CPEE-INSTANCE - instance number (e.g., 123)
  - CPEE-INSTANCE-URL - url pointing to the instance (e.g., cpee.org/flow/engine/123)
  - CPEE-INSTANCE-UUID - unique identifier of the instance
  - CPEE-CALLBACK - url to send any information to, should the implmentation decide to answer asynchronously
  - CPEE-CALLBACK-ID - unique identifier for the answer
  - CPEE-ACTIVITY - id of the task calling the implementation (e.g. a1)
  - CPEE-LABEL - label of the task calling the implementation (e.g. Query Production Schedule)

Functional Aspect: Interaction Patterns

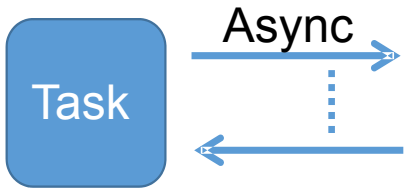
Each task implementation for cpee.org can utilize the following HTTP headers in their response.



CPEE-SALVAGE response header - implementation signals it internally failed but wants to retry.

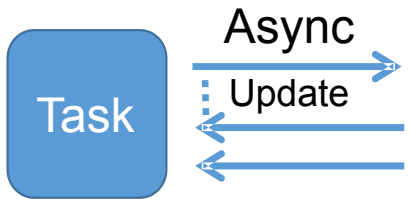
CPEE-INSTANTIATON response header - the implementation create a sub-process.

CPEE-EVENT - tell the engine to send a custom event. Used e.g. by worklists to commuicate that a user has taken a task.



CPEE-CALLBACK: true response header and value - implementation signals that it wants to send data later, to the provided CPEE-CALLBACK request header.

All the headers above.



CPEE-UPDATE: true response header and value - implementation signals that it wants to send data more data after this response.

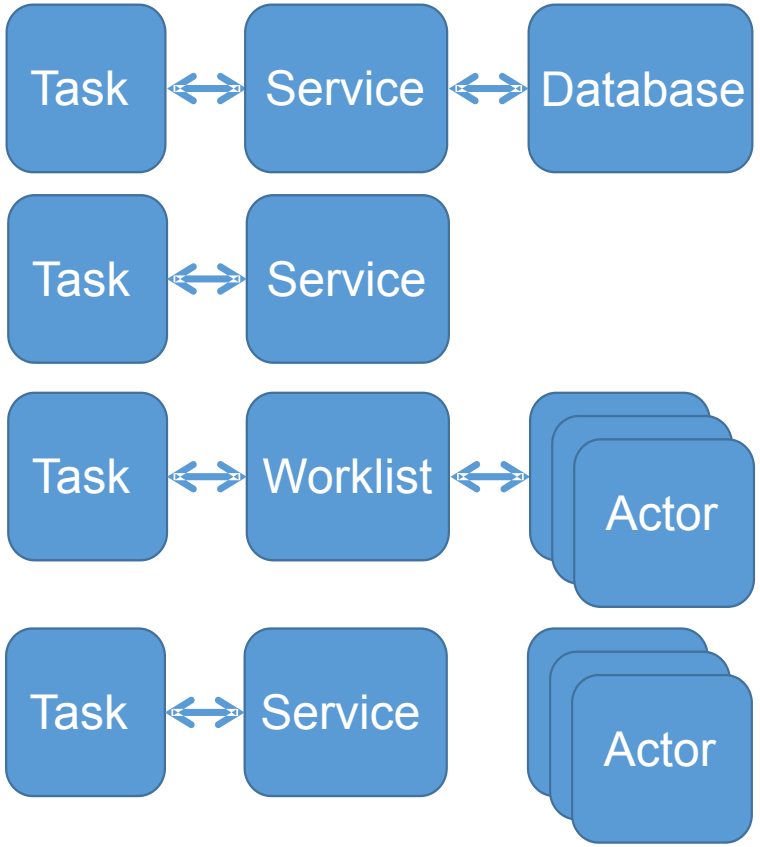
CPEE-UPDATE-STATUS: set the engine status with the response.

All the headers above.

Functional Aspect: Interaction Patterns

For monolithic process engines, worklists or database query functionality might be built into the engine.

For modular process engines, everything is implemented as external services.

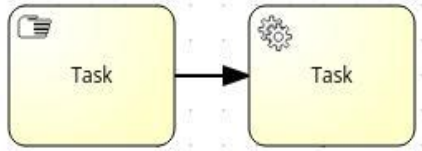


Engine communicates with an external service which in turn implements database access.

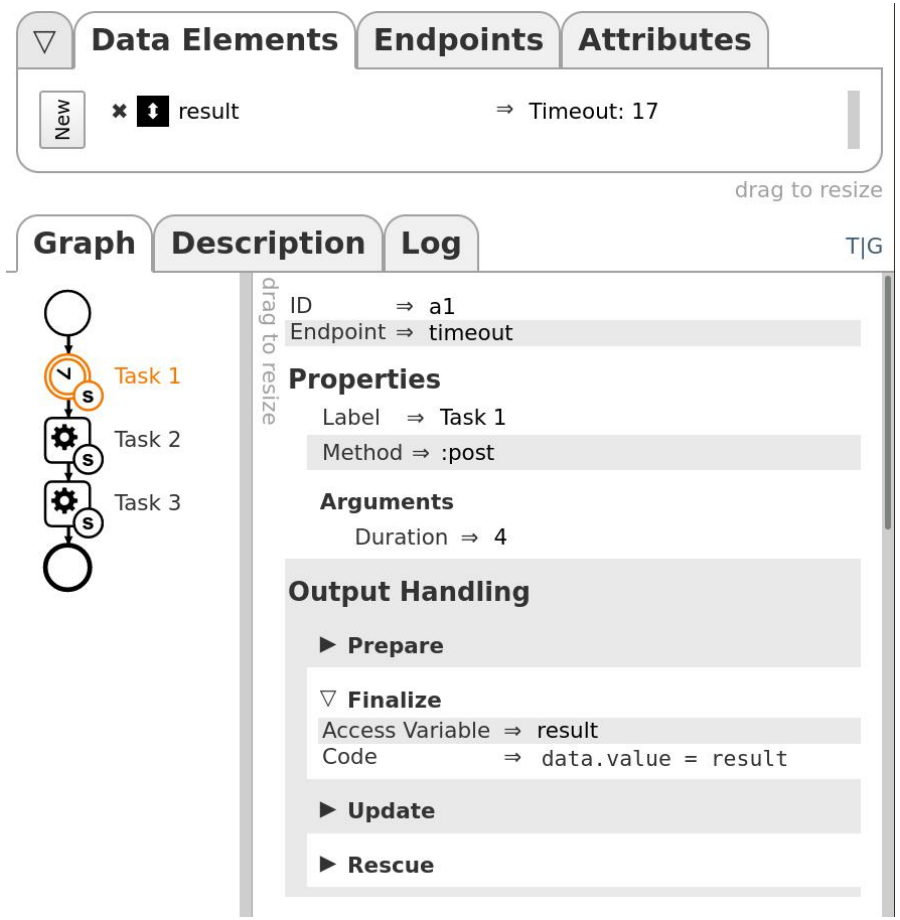
Engine communicates with an external service which receives input and generates output.

Engine communicates with a special external service which implements a UI for actors to interact with a process task.

Engine communicates with an external service which passively tracks the work done by humans



Functional Aspect: CPEE Trees



Tasks is the term that is used for an activity in BPMN

Tasks/Events have associated attributes:

- ID - uniquely identifies the task/event in a process
- Endpoint - link to the functionality that implements the task/event
- Label - a text that describes the task
- Method & Arguments - see slide Endpoints
- Output Handling - see slide Data

Important: from the POV of the process model, a task/event has an external implementation which is a **black box**. It is passed some input ("Arguments") and returns some output. The output is processed by using code snippets stored in "Output Handling".

Developing Executable Processes...

... requires the definition and specification of the following aspect (selection):

- Control Flow (Behavioral Aspect)
- Data Flow (Information Aspect)
- Activities (Functional Aspect)
- Deadlines (Temporal Aspect)
- Actor Assignments (Organizational Aspect)
- Application Services (Operational Aspect)

Temporal Aspect

Deadline

- A time-based scheduling constraint which requires that a certain activity (or work item) shall be completed within a certain period of time → deadline)
- Activity scheduling by a WfMS attempts to meet deadline constraints set against particular tasks
- The deadline may be expressed as an attribute of the workflow model or within workflow relevant data
- Escalation / Notification procedures may be involved if deadlines are not met

Developing Executable Processes...

... requires the definition and specification of the following aspect (selection):

- Control Flow (Behavioral Aspect)
- Data Flow (Information Aspect)
- Activities (Functional Aspect)
- Deadlines (Temporal Aspect)
- Actor Assignments (Organizational Aspect)
- Application Services (Operational Aspect)

Task authorization

- Definition of organizational models (organigram)
- Defines organizational units, roles, and actors
- Established: role-based access control (RBAC) → abstraction from actors who might change, i.e., authorization is defined based on roles and during runtime the roles are resolved to actors that are currently present in the organizational model
- Example: access_rule (task) ← Role=`clerk‘ AND OrgUnit= `Loan‘

Task authorization

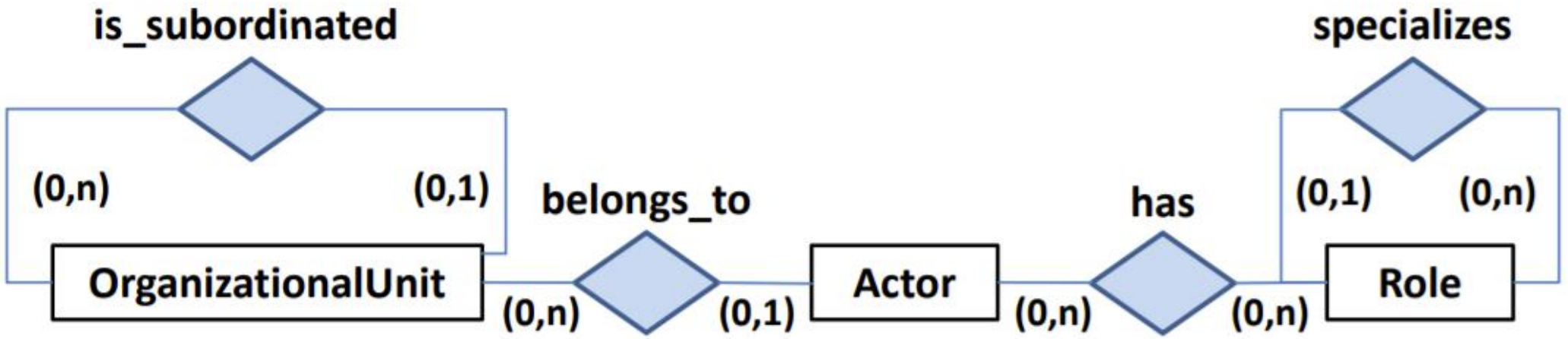
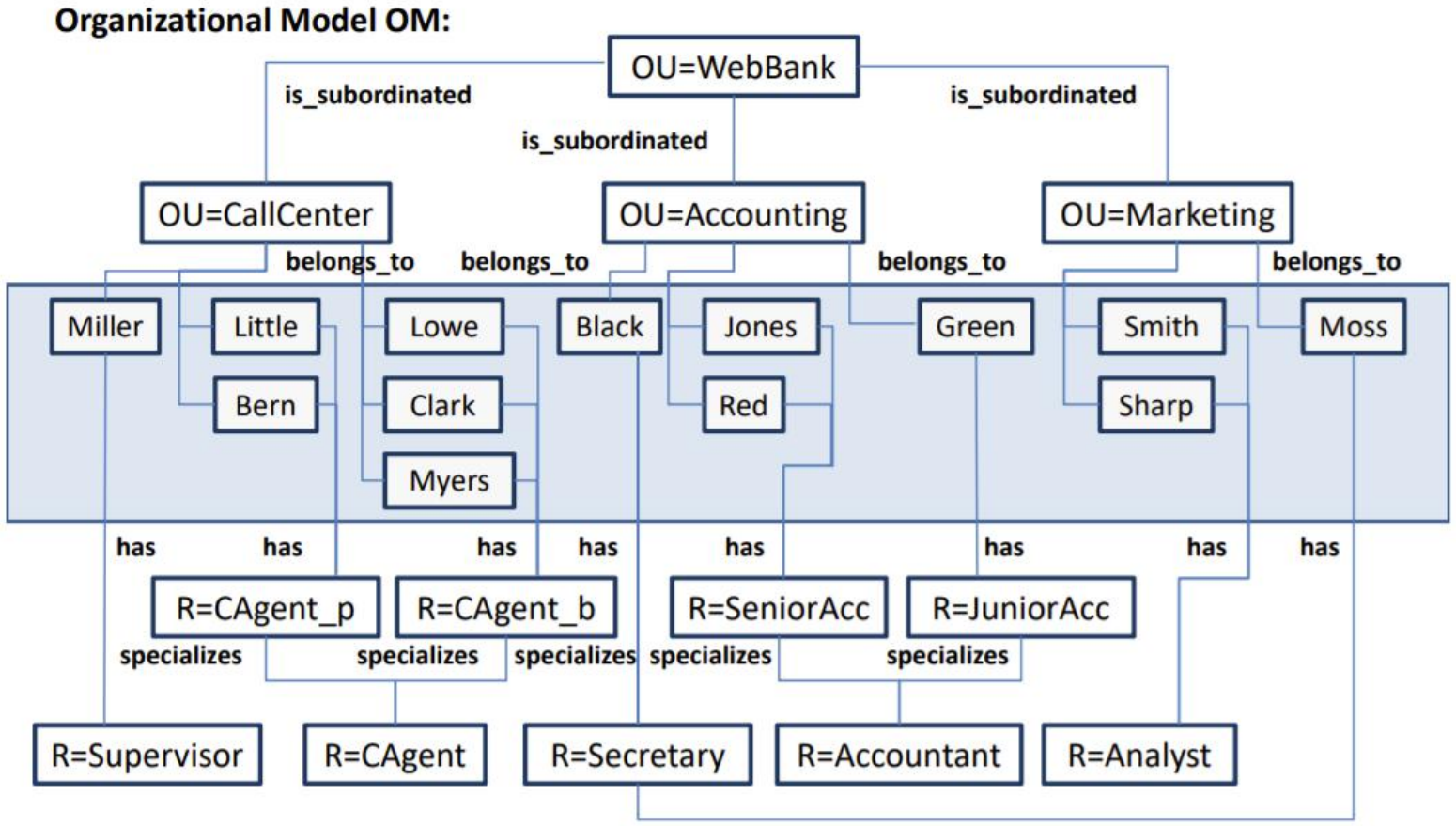


Figure 3. Organizational meta model (in ER notation)

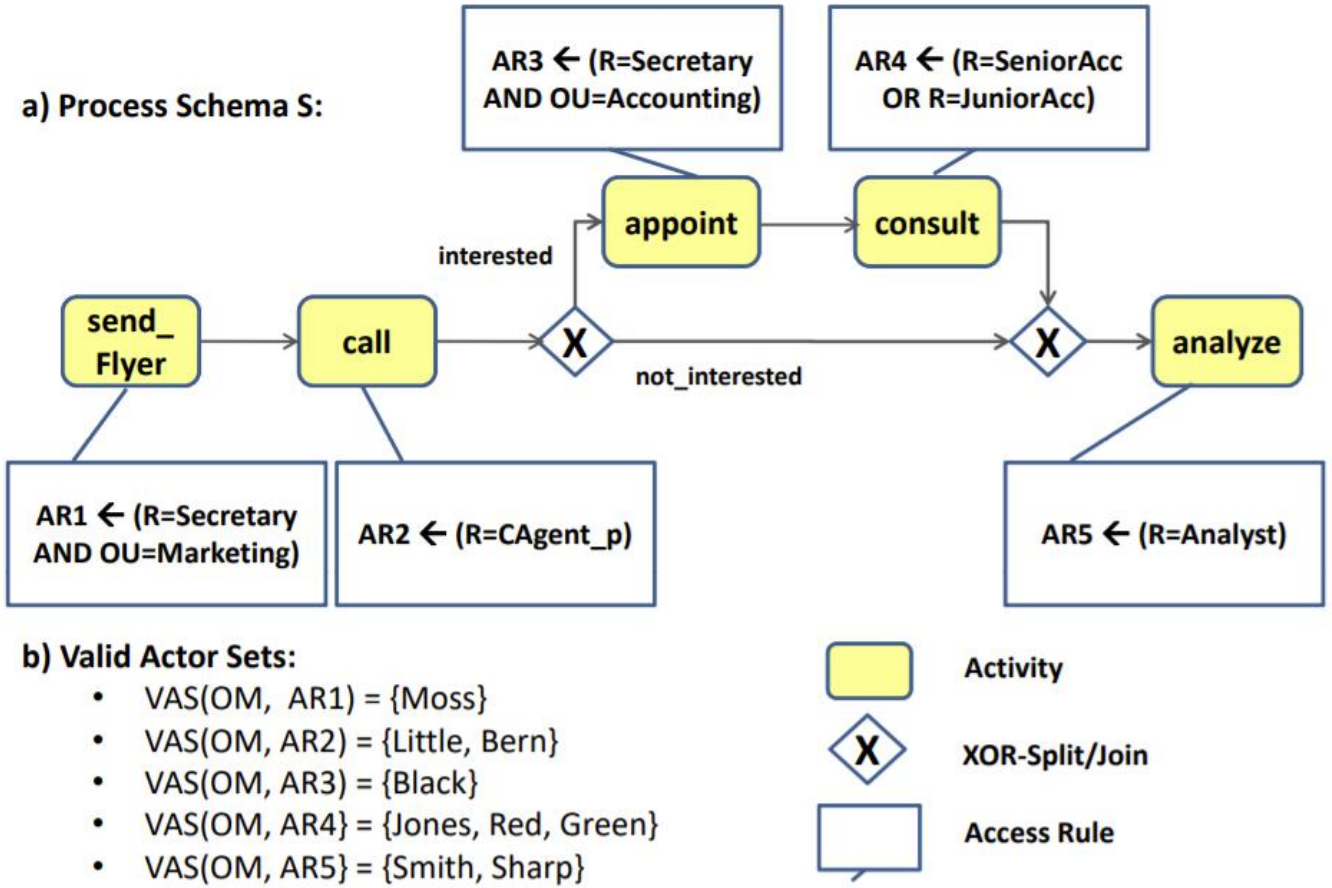
Stefanie Rinderle-Ma, Manfred Reichert:
Comprehensive life cycle support for access rules in information systems: the CEOSIS project. Enterp. Inf. Syst. 3(3): 219-251 (2009)

Task authorization



Stefanie Rinderle-Ma, Manfred Reichert:
Comprehensive life cycle support for access rules in information systems: the CEOSIS project. Enterp. Inf. Syst. 3(3): 219-251 (2009)

Task authorization

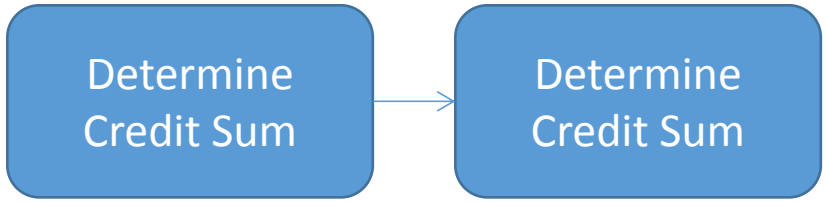


Stefanie Rinderle-Ma, Manfred Reichert:
Comprehensive life cycle support for access rules in information systems: the CEOSIS project. Enterp. Inf. Syst. 3(3): 219-251 (2009)

Task authorization

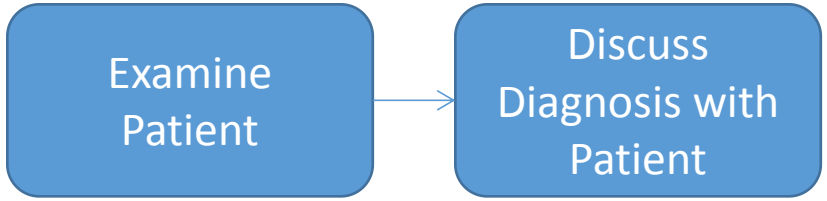
Separation of Duty

- A task is only authorized for an actor that was not involved in a different task
- Example: Controlling, Four Eyes Principle



Binding of Duty

- A task is only authorizes for an actor that was also authorized for a different task
- Example: Confidentiality, Privacy

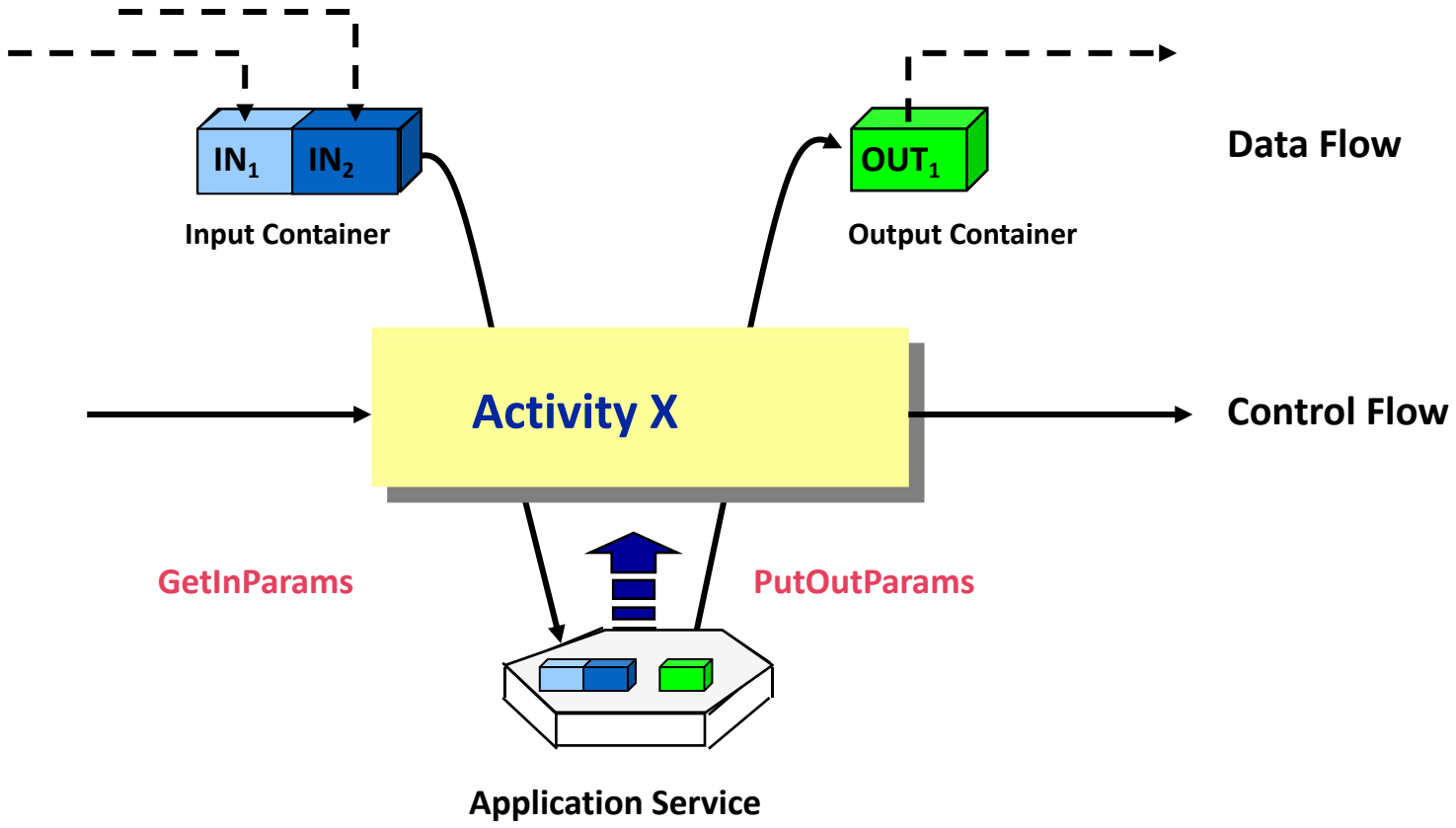


Developing Executable Processes...

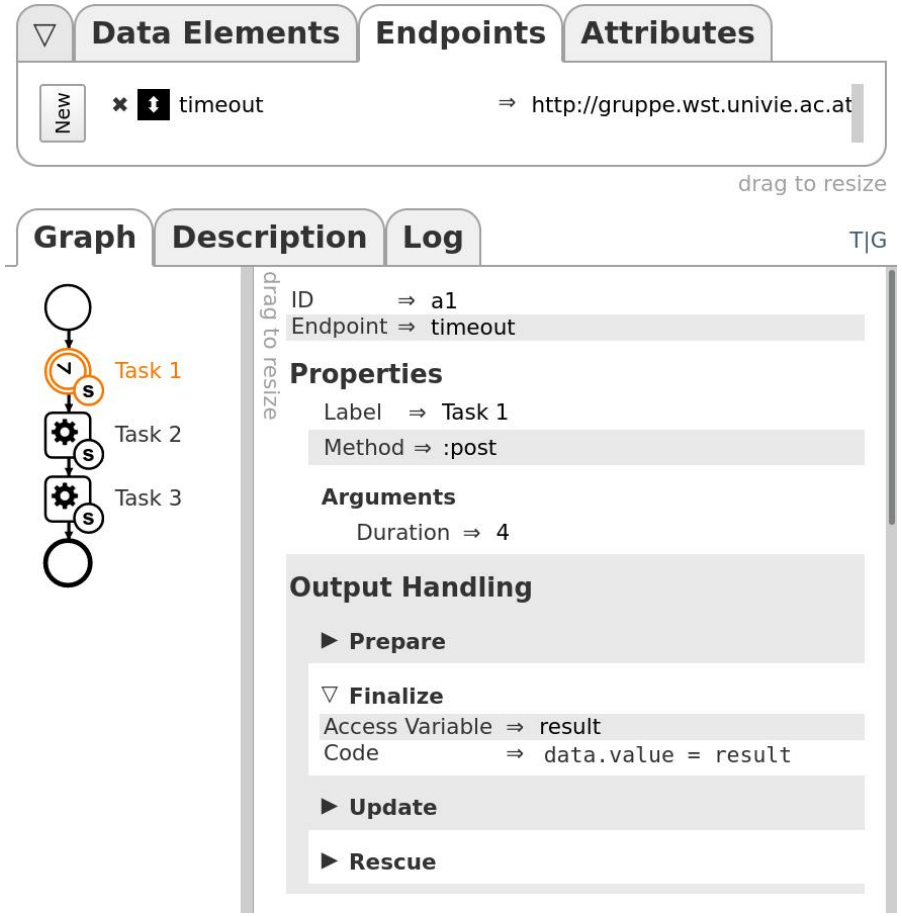
... requires the definition and specification of the following aspect (selection):

- Control Flow (Behavioral Aspect)
- Data Flow (Information Aspect)
- Activities (Functional Aspect)
- Deadlines (Temporal Aspect)
- Actor Assignments (Organizational Aspect)
- Application Services (Operational Aspect)

Application Integration



CPEE Trees - **Endpoints**



Endpoints are links to the implementation of a task/event.

Each task/event has to point to **at least one endpoint** that implements it.

Endpoints can be addressed through a variety of different protocols, such as SOAP, REST, OPC-UA, ...

Standard endpoints for CPEE trees are assumend to point to REST services:

- They have a method: GET, POST, PUT, DELETE
- They have 1..n arguments. Arguments are always name/value pairs. The same name may occur more than once.

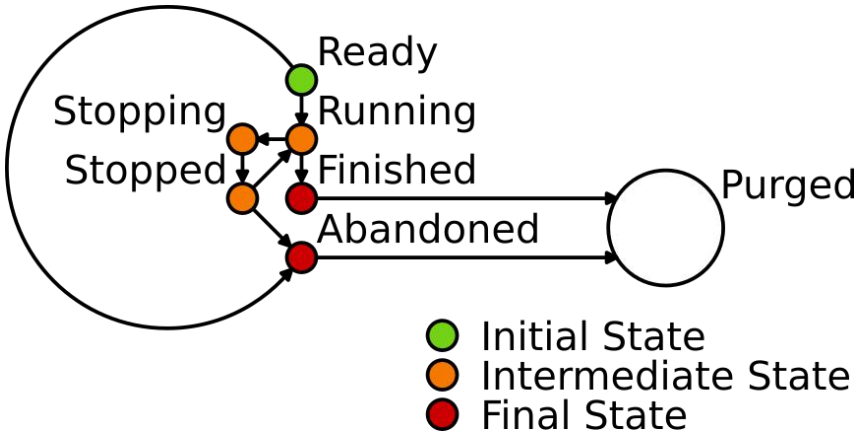
By default all endpoints are considered to be automatic tasks. If special annotations for a rest service exist, the visual representation of the task changes to represent its purpose. Examples:

- Sub-process
- Send message
- Receive message
- Wait

Nonetheless - even the creation of a sub-process is implemented as a REST service with specific parameters:

- The path to process model (which represents the sub-process)
- Data-elements passed to the sub-process upon creation (BPMN start-event with envelope).

CPEE Trees - **Instance Lifecycle**



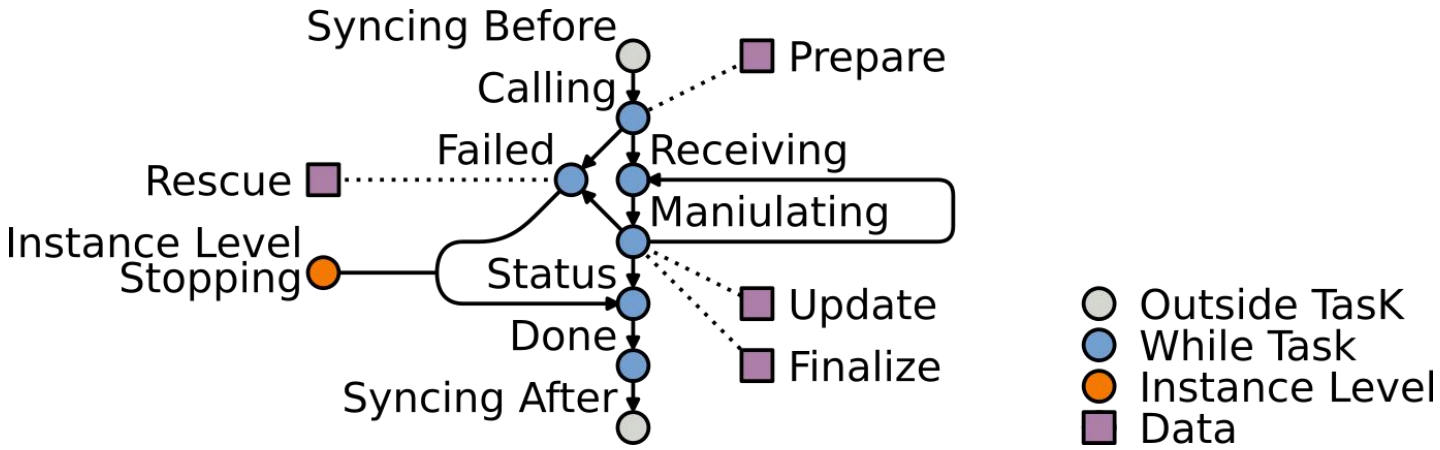
Each process instance goes through a series of states while it is executed:

- Upon creation it is ready
- When started it is running
- If an error occurs it is stopping
- When all parallel running tasks (even the ones that had no errors) are finished, it is stopped
- A user can set the instance back to running (after potentially changing the data elements or the control flow)
- After an end event is found it is finished
- A finished instance can **NOT** be restarted again.
- If an instance is ready or stopped, it can be set to state abandoned by a user.

The data-elements of all instances, even finished and abandoned ones can be inspected in the process engine.

- A user can decide to fully remove the instance from the process engine, in which case it is purged.
- Purged instances can only be inspected in a log.

CPEE Trees - **Task Lifecycle**



Each task goes through a series of states when it is executed:

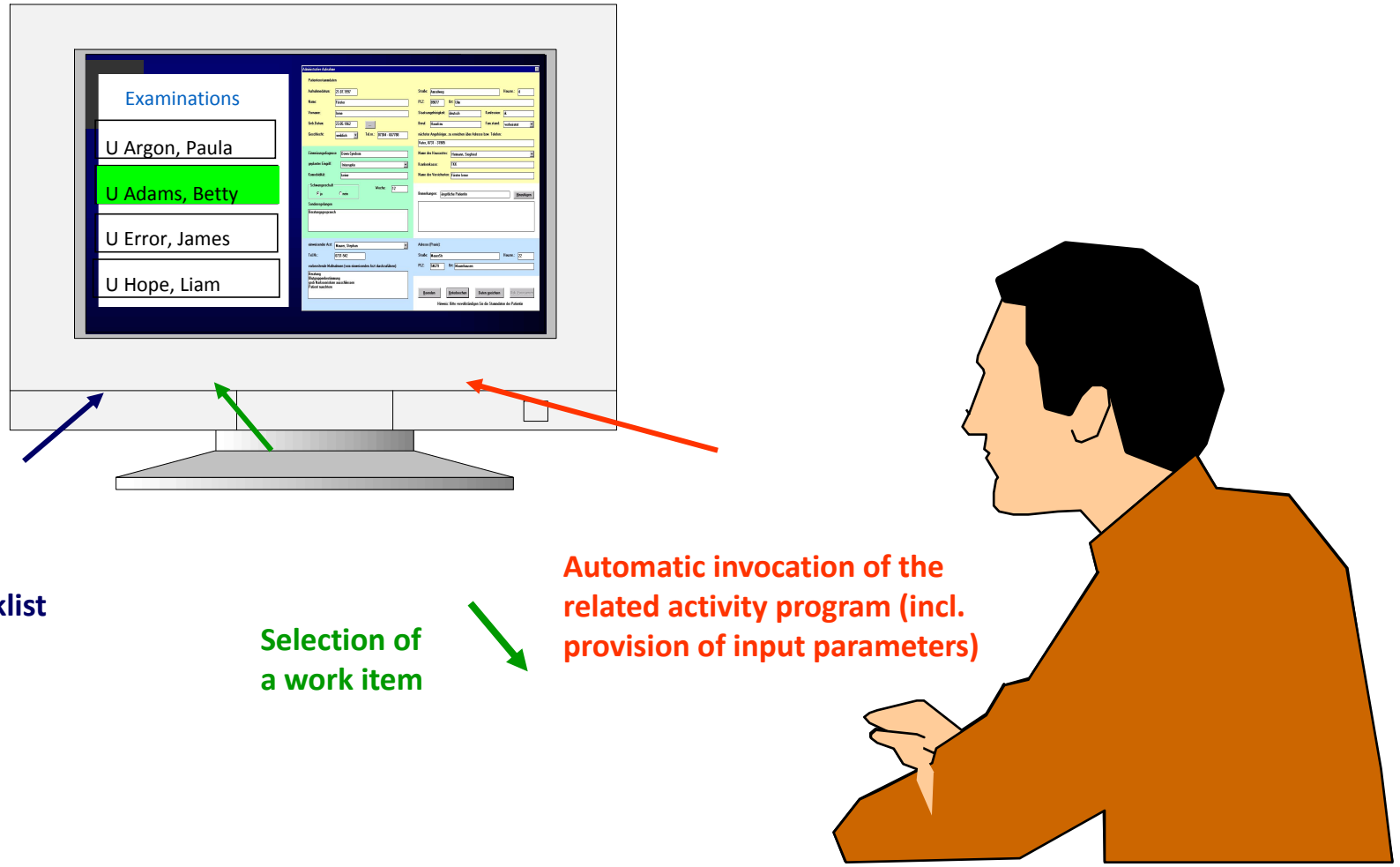
- **Calling:** the REST service is called with a specific method and the arguments.
- **Receiving:** the REST service sends back raw data.
- **Manipulating:** the "Finalized" code snippet is executed and is changing the process context.
- **Status:** the status of the process instance is set, e.g. to the string "task x sent back 12 bytes". Setting the status is optional, and is determined through code in "Finalize".
- **Done**

- **Failed:** if an error occurs when calling the REST service, the process goes to state failed. In failed the optional "Rescue" code snippet is called. This code can determine, if the instance is stopped or the instance can continue normally. This code snipped can change the process context (i.e. write data elements) as well as the status).
- **Manipulating ctd:** REST services can send back data in multiple small packages. If this is detected, "Receiving" might occur again. As long as additional data packages are expected, the "Update" code snippet is called for each package.
- **Calling ctd:** the "Prepare" code snippet can be utilized for changing the process context before a particular task is called.

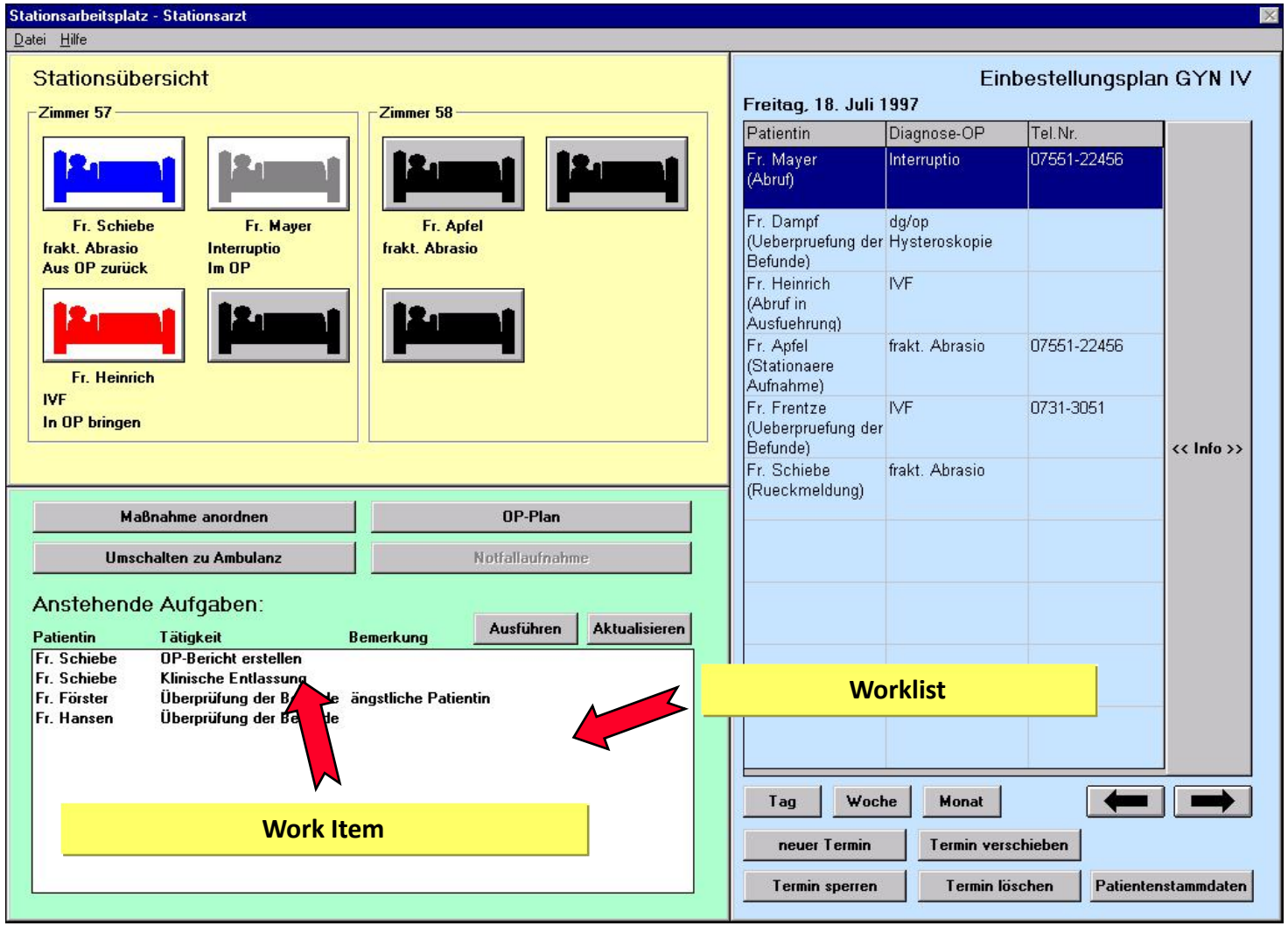
Refined lifecycles might exists for particular task implementations: e.g. the XES IEEE 1849-2016 standard defines states which are especially suitable for humans carrying out individual tasks such as schedule, assign, reassign, start, suspend, resume, complete, withdraw, ... **Important:** from the POV of the process engine, a task carried out by a human is still a black box:

- The task has input - information which is passed on to the human.
- The task has output - some artefacts (e.g. documents) created by the human.
- The task waits until the output is reveived.

Worklists


Examinations
U Argon, Paula
U Adams, Betty
U Error, James
U Hope, Liam
User
Worklist
Selection of
a work item
Automatic invocation of the
related activity program (incl.
provision of input parameters)

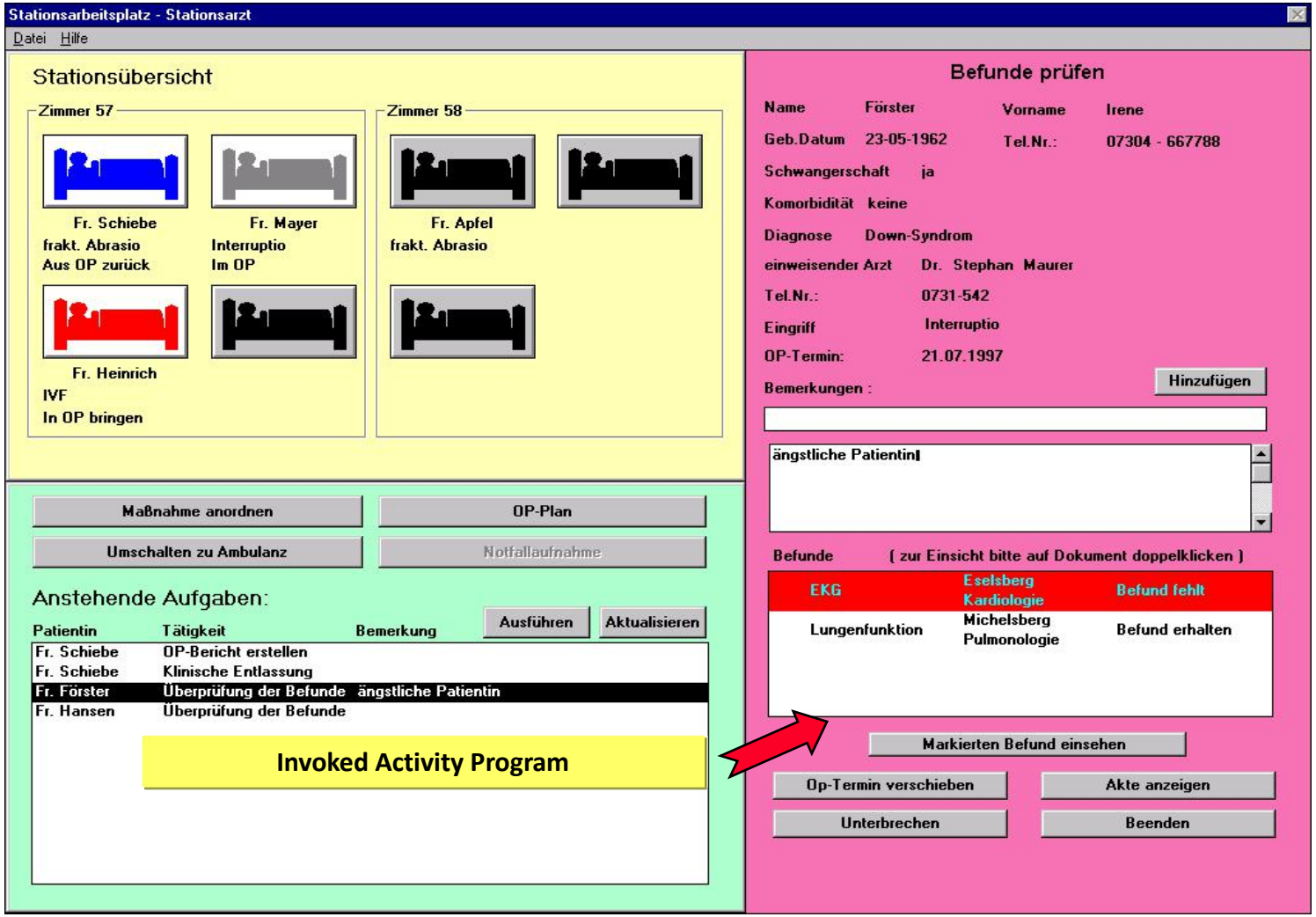
Worklists



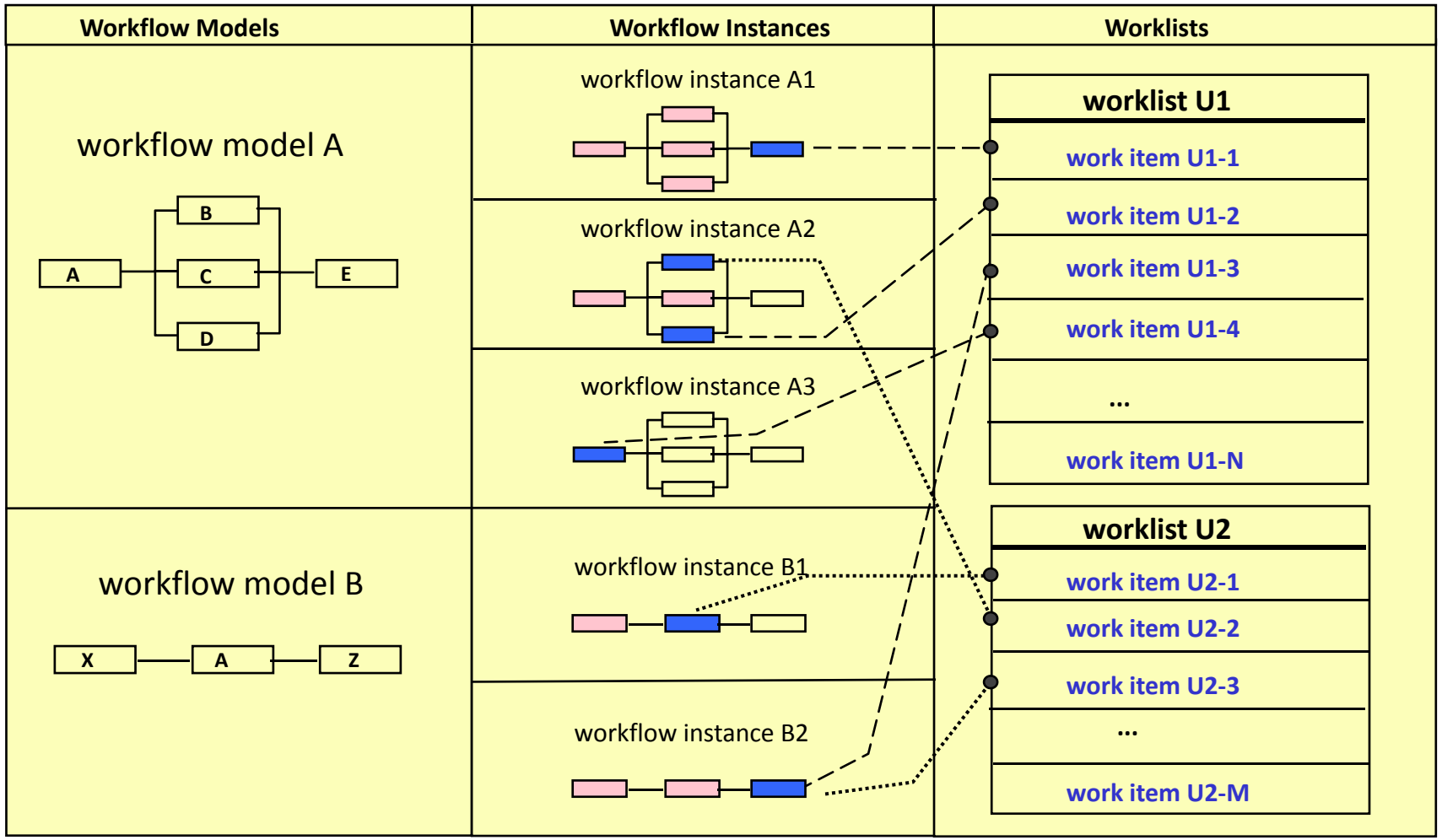
End User View (1)

Worklists

End User View (2)



Worklists


Workflow Models
Workflow Instances
Worklists
workflow model A
A
B
C
D
E
workflow model B
X
A
Z
workflow instance A1
workflow instance A2
workflow instance A3
workflow instance B1
workflow instance B2
worklist U1
work item U1-1
work item U1-2
work item U1-3
work item U1-4
...
work item U1-N
worklist U2
work item U2-1
work item U2-2
work item U2-3
...
work item U2-M

Worklists

Vendors often provide simple generic worklist generic clients.

Purpose: presentation of generic, application-independent workflow / activity attributes:

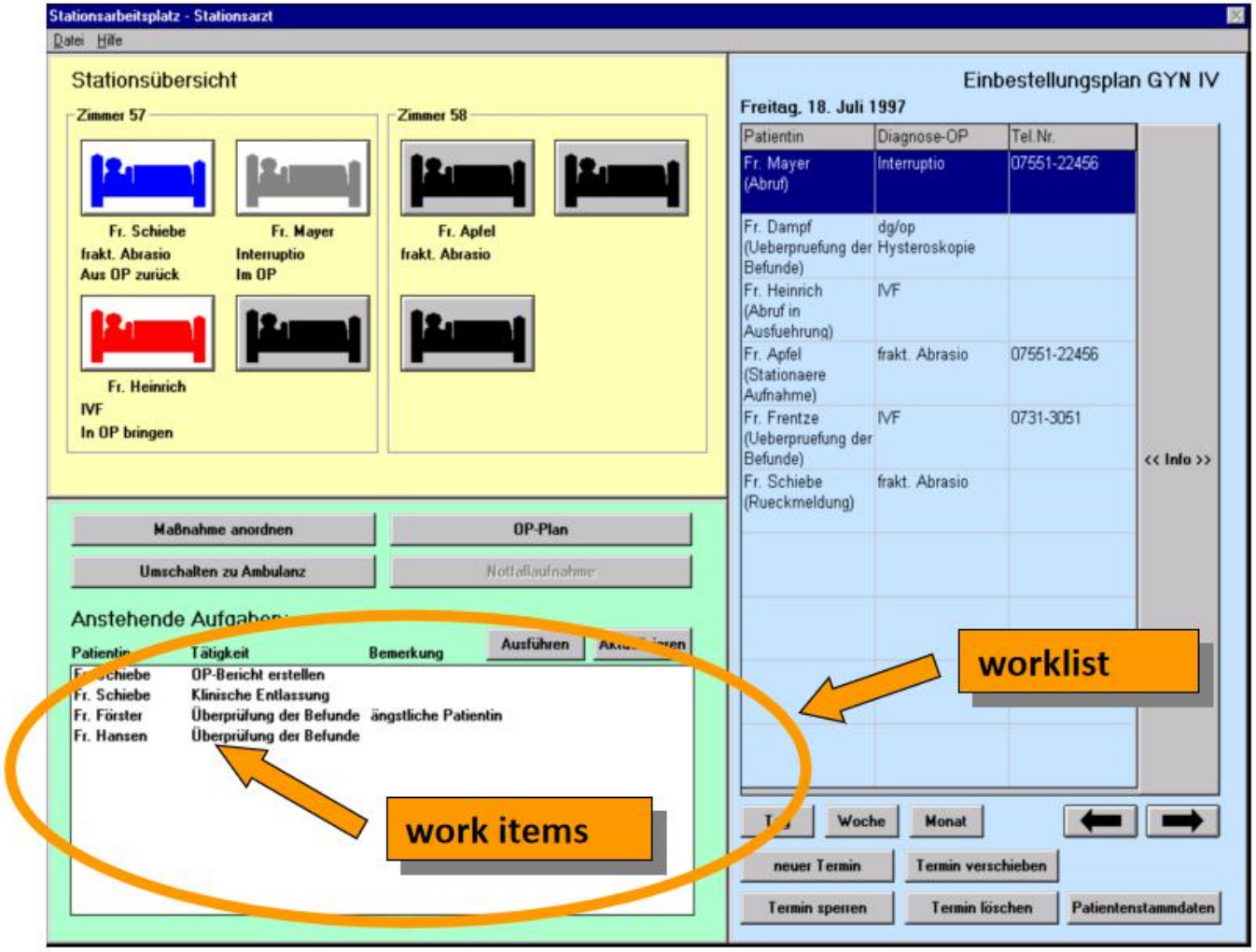
- Activity status
- Activity name
- Related application program / service
- Activation time

Worklists

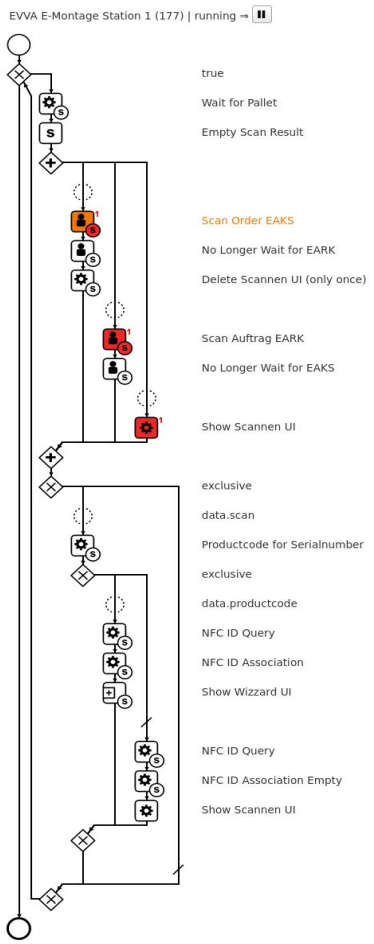
Characteristic properties of a worklist client:

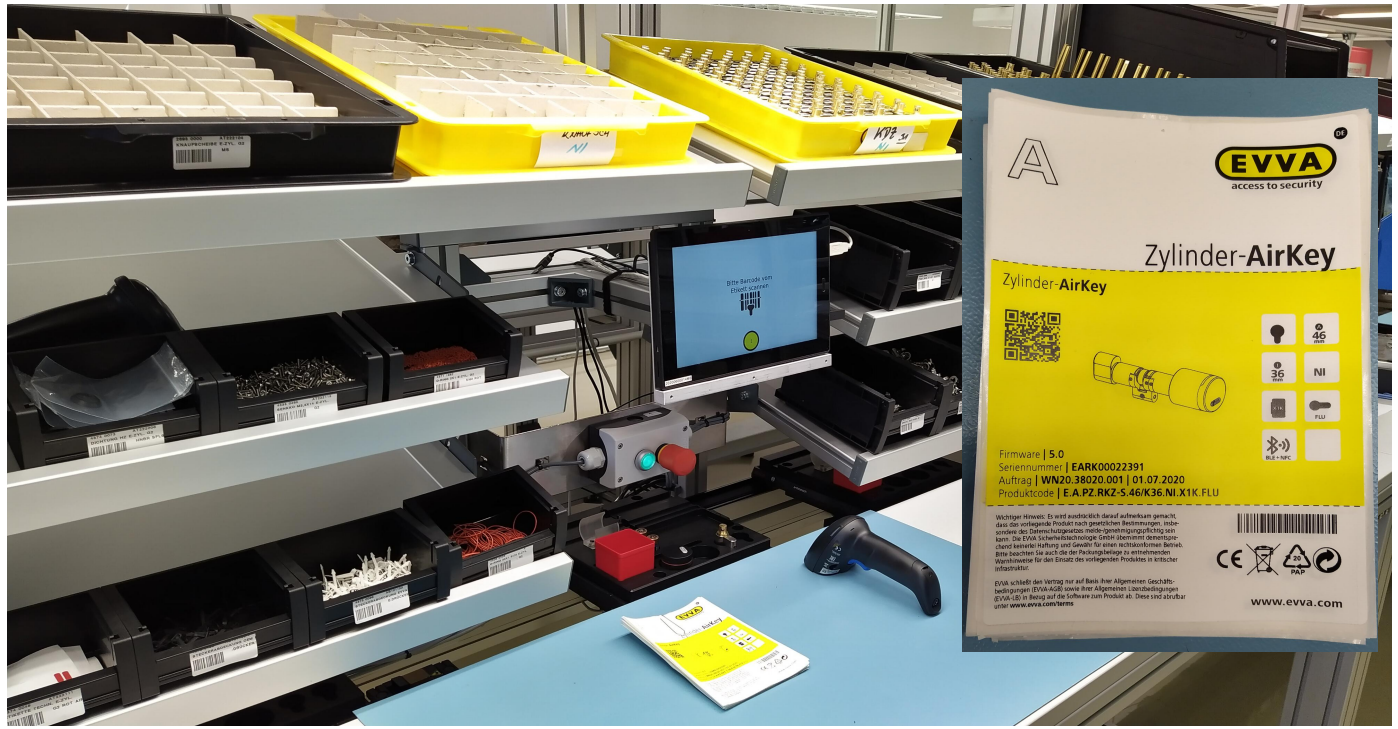
- Parallel connection to multiple process management systems, process engines (→ transparent for the end user)
- Control and monitoring of ***process instances***
  - Querying the status of process instances
  - Creating, starting, terminating, suspending, ... process instances
- Control and monitoring of ***activity instances*** and ***work items*** respectively
  - Aggregating *work items* in ***worklists*** (→ application-specific criteria: priority, kind of task, process type, ...
  - Delegating, starting, suspending, terminating … tasks
  - Monitoring deadlines (→ escalation & notication mechanisms)
- Access to application data.

Domain-specific Worklists and UIs

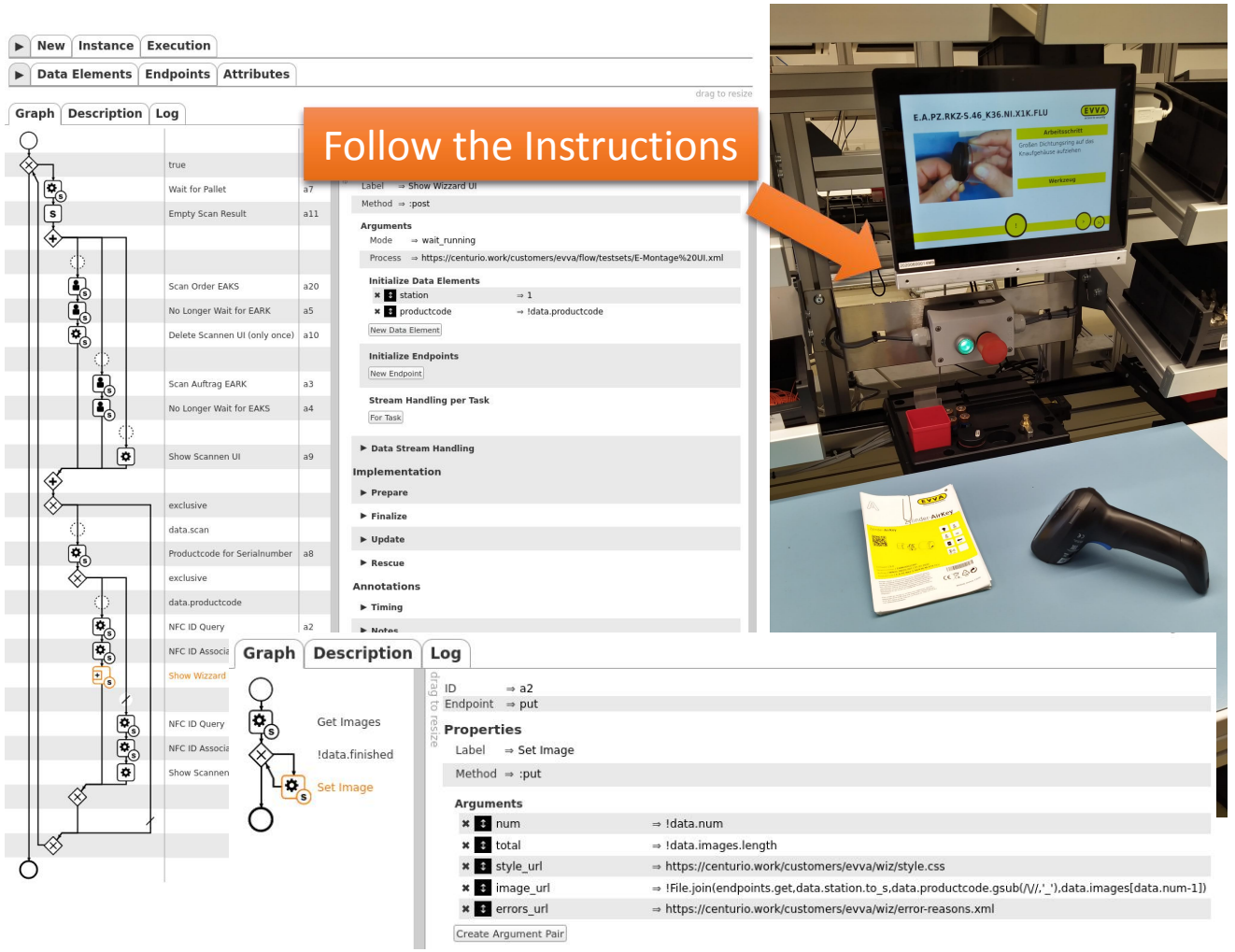


Station 1: Scan Packing Label & Equipping Work Piece Carriers

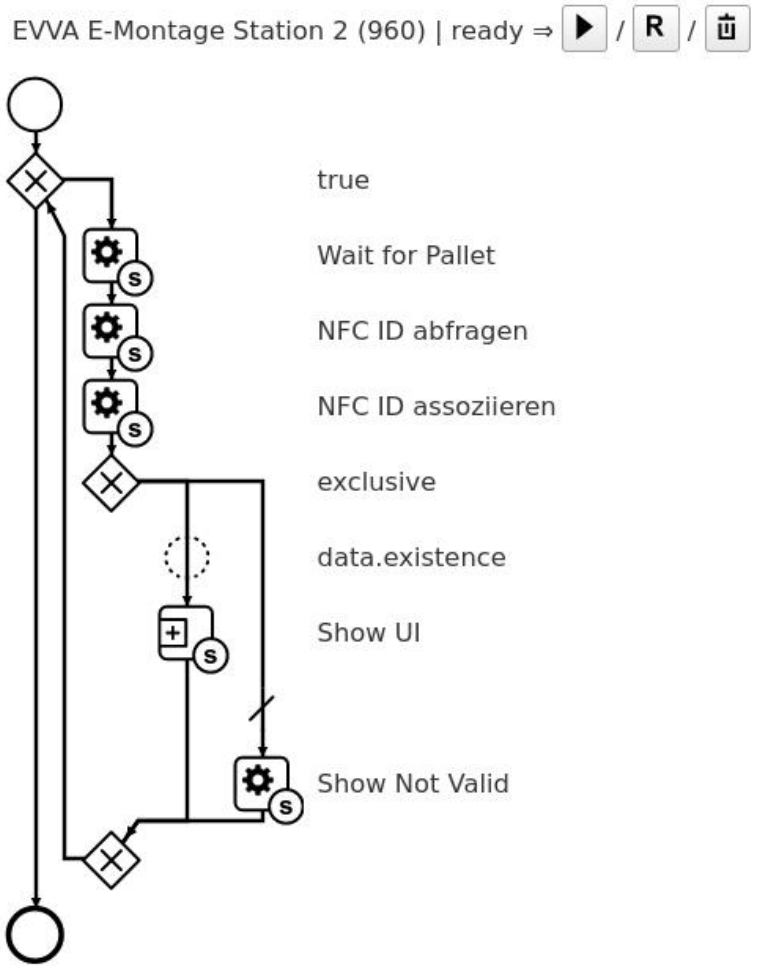




Station 1: Scan Packing Label & Equipping Work Piece Carriers

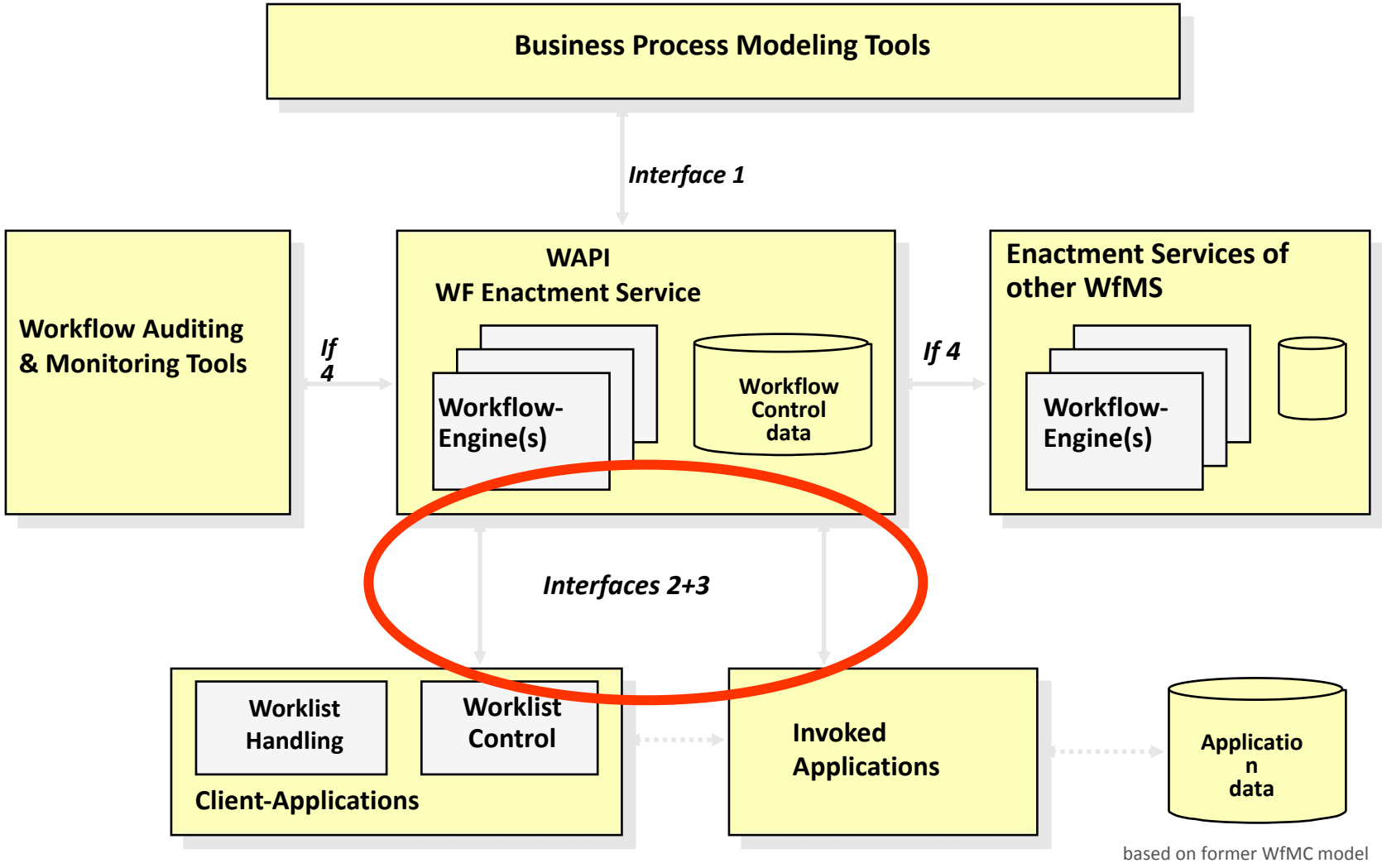


Station 2: Detect Work Piece Carrier, Show Associated Instructions



Follow the Instructions

Workflow Reference Model



based on former WfMC model

Worklist Implementation

A Worklist consists of two parts

- Worklist control: manages available tasks for all process instances, knows.
- Worklist handling: user interface for actors to interact with information about available tasks, and for working on individual tasks.

Worklist control can be

- built into a process engine - monolothic process engines.
- be independent of a process engine - modular process engines.

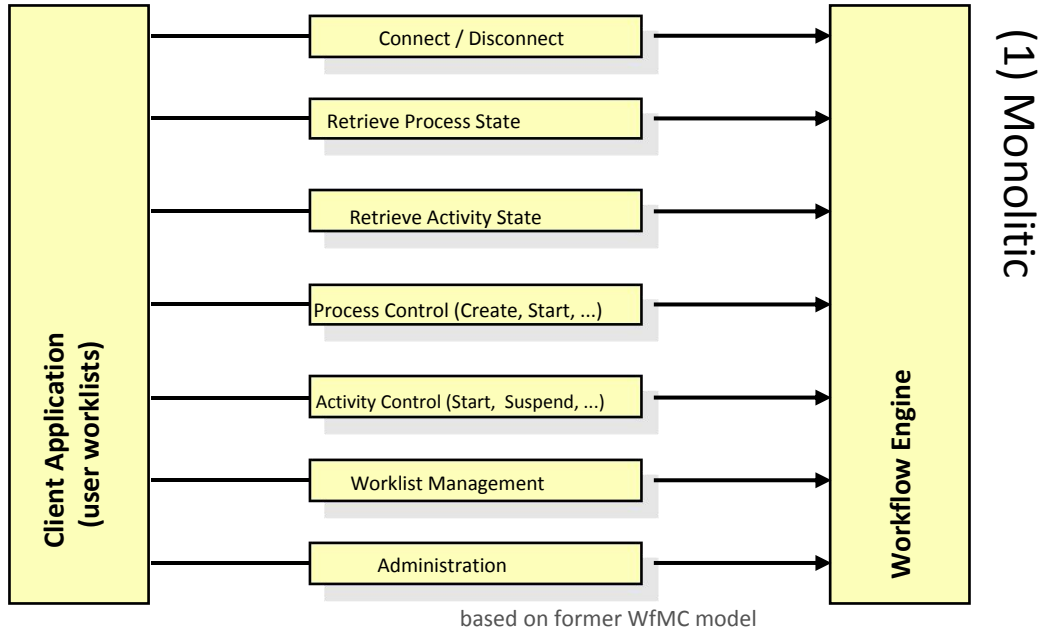
Worklist Implementation

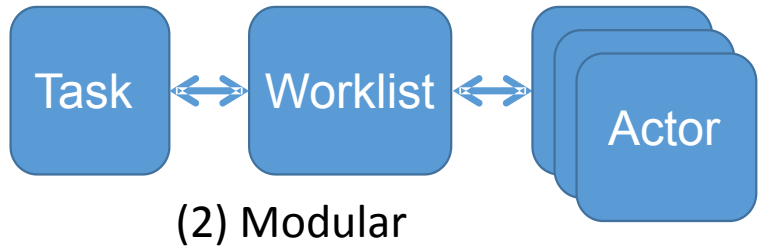
Monolithic Process Engines with integrated worklist functionality:

- Vendors provide **Application Programming Interface (API)** for implementing worklist clients (i.e., for accessing worklists)
- In most cases communication between client and server is **unilateral**; i.e., clients themselves are responsible for updating their worklists implementation efforts.
- Available APIs differ from vendor to vendor

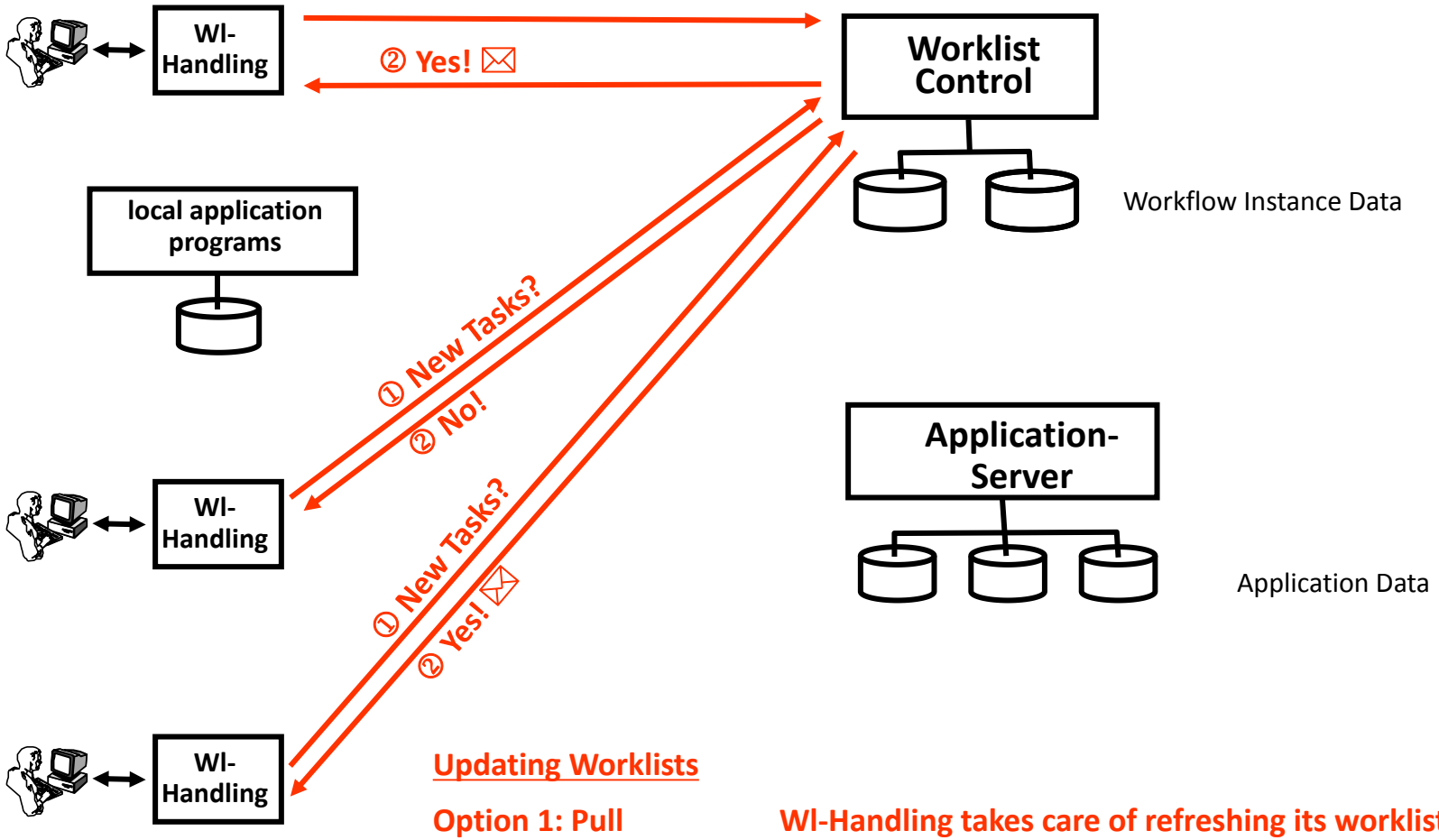
Modular Process Engines:

- Worklist is an external service like any other.
- Form the POV of the Process Engine, input data is sent, output data is received.
- All communication topics depicted in (1) are handled internally in the worklist.
- The worklist becomes vendor independent.





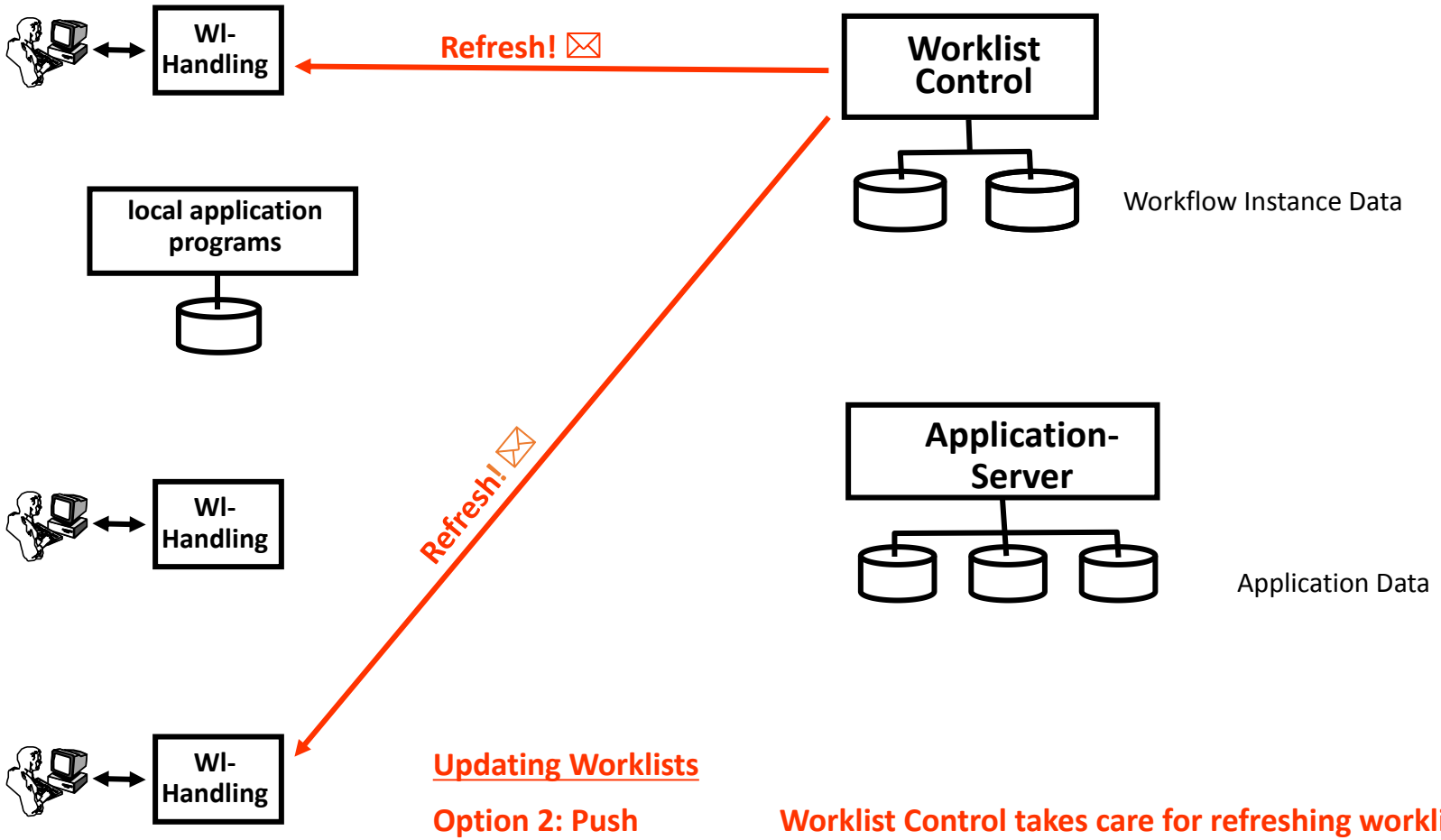
Worklist Implementation Variants



Updating Worklists

Option 1: Pull **WI-Handling takes care of refreshing its worklists**

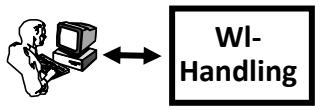
Worklist Implementation Variants

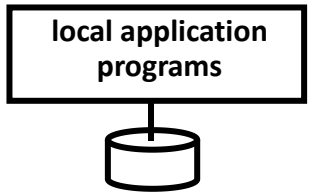


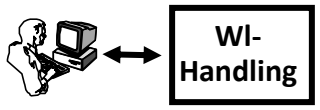
Updating Worklists

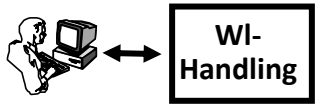
Option 2: Push — **Worklist Control takes care for refreshing worklists**

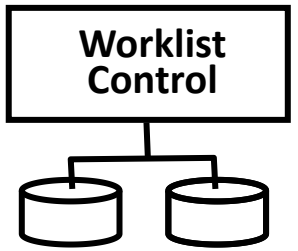
Worklist Implementation Variants



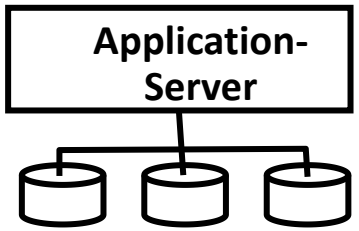








Workflow Instance Data



Application Data

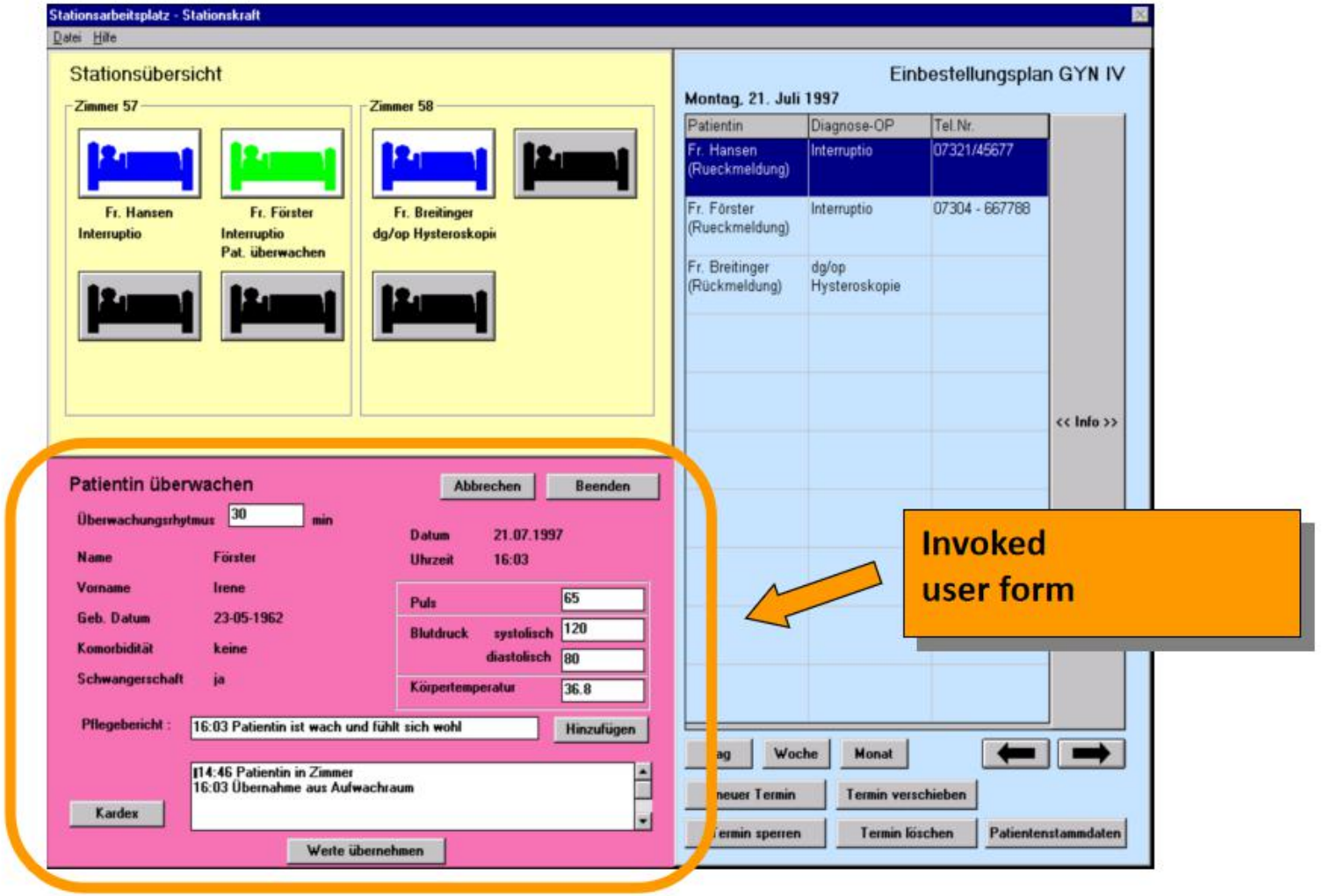
<u>Updating Worklists (Evaluation)</u>

Classical Pull: outdated worklists + unnecessary communication

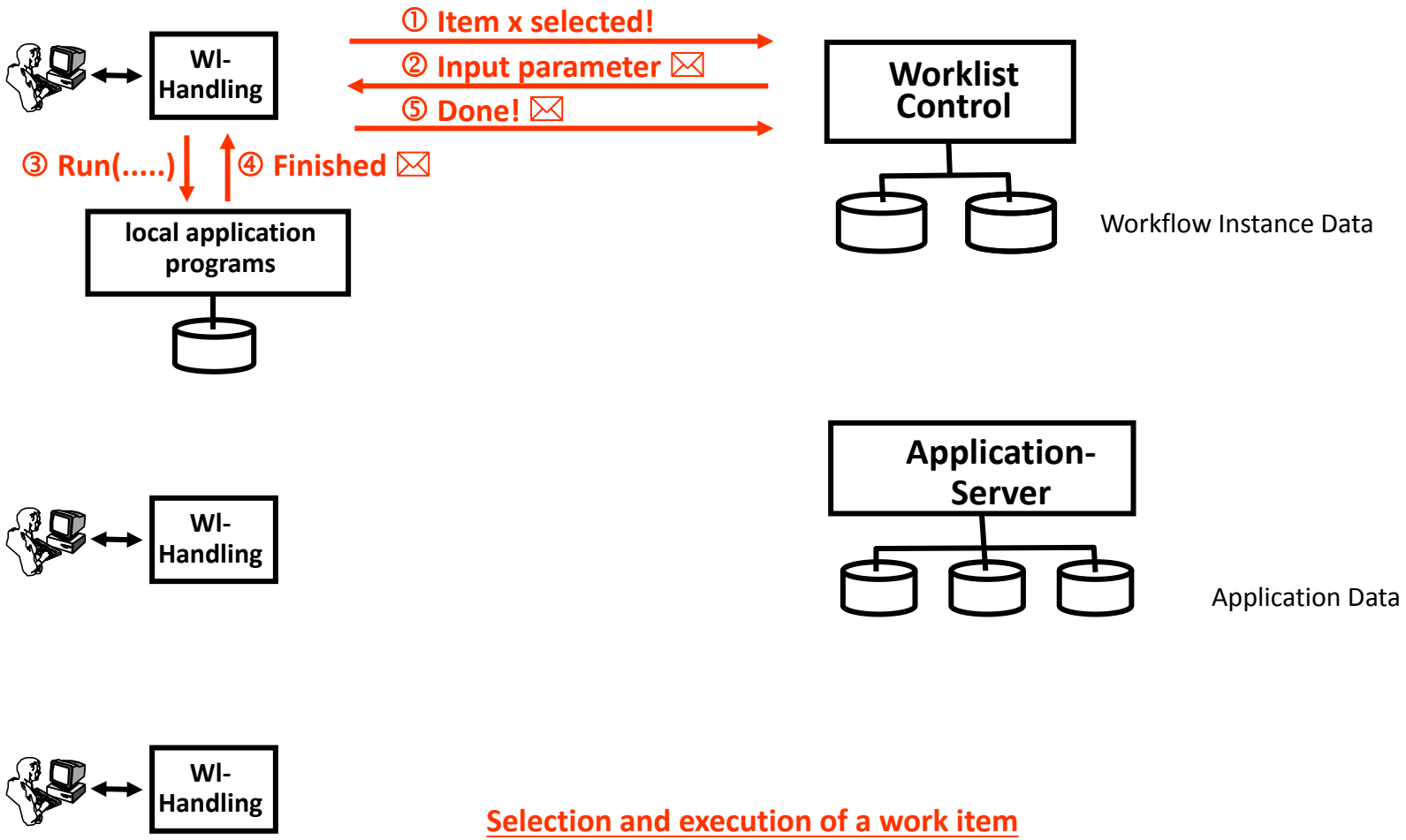
Classical Push: potentially too much communication

Good compromise: time-driven push

Selection and Execution of a Work Item



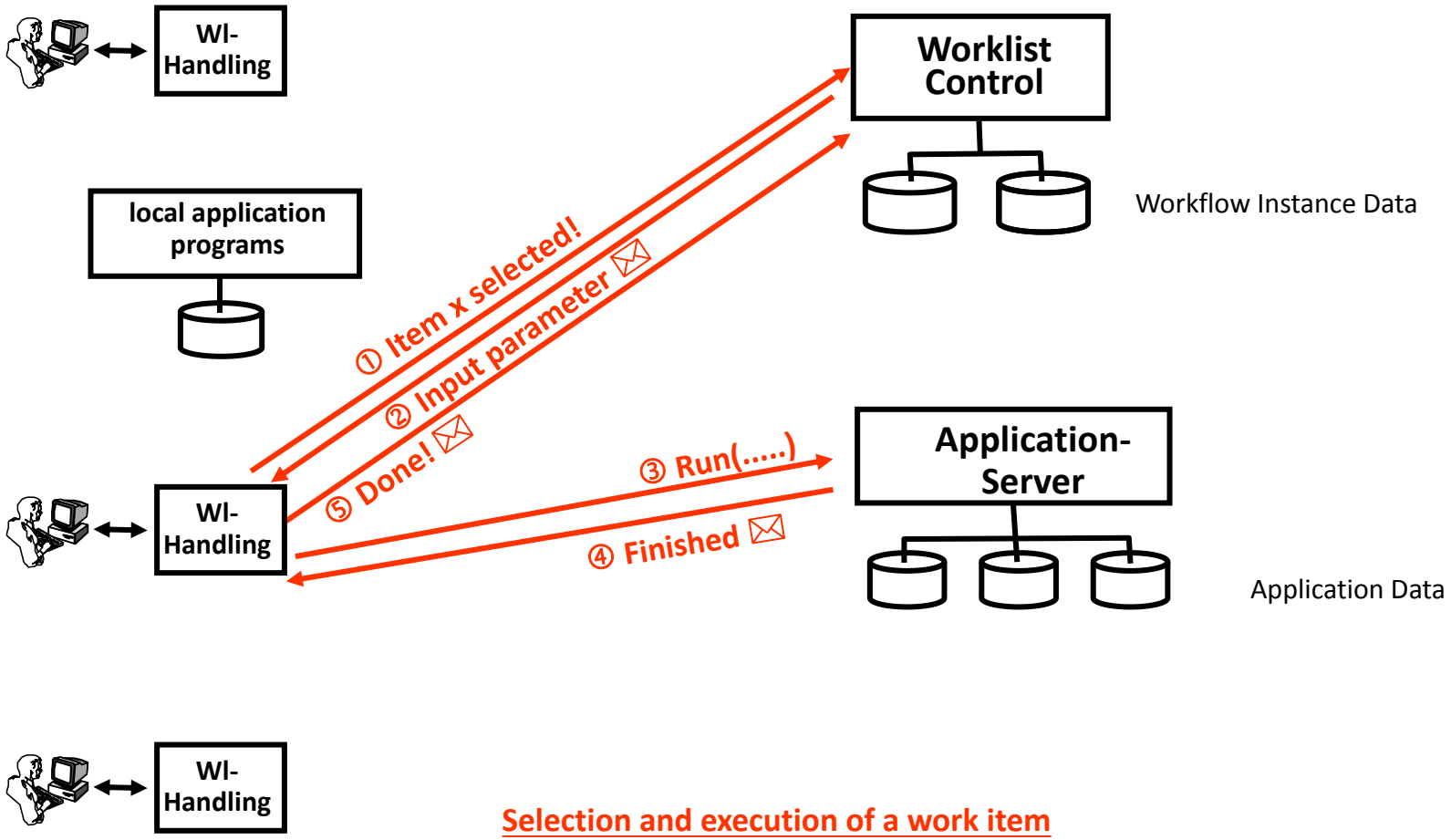
Selection and Execution of a Work Item



Selection and execution of a work item

Option 1: A local application component is invoked and executed!

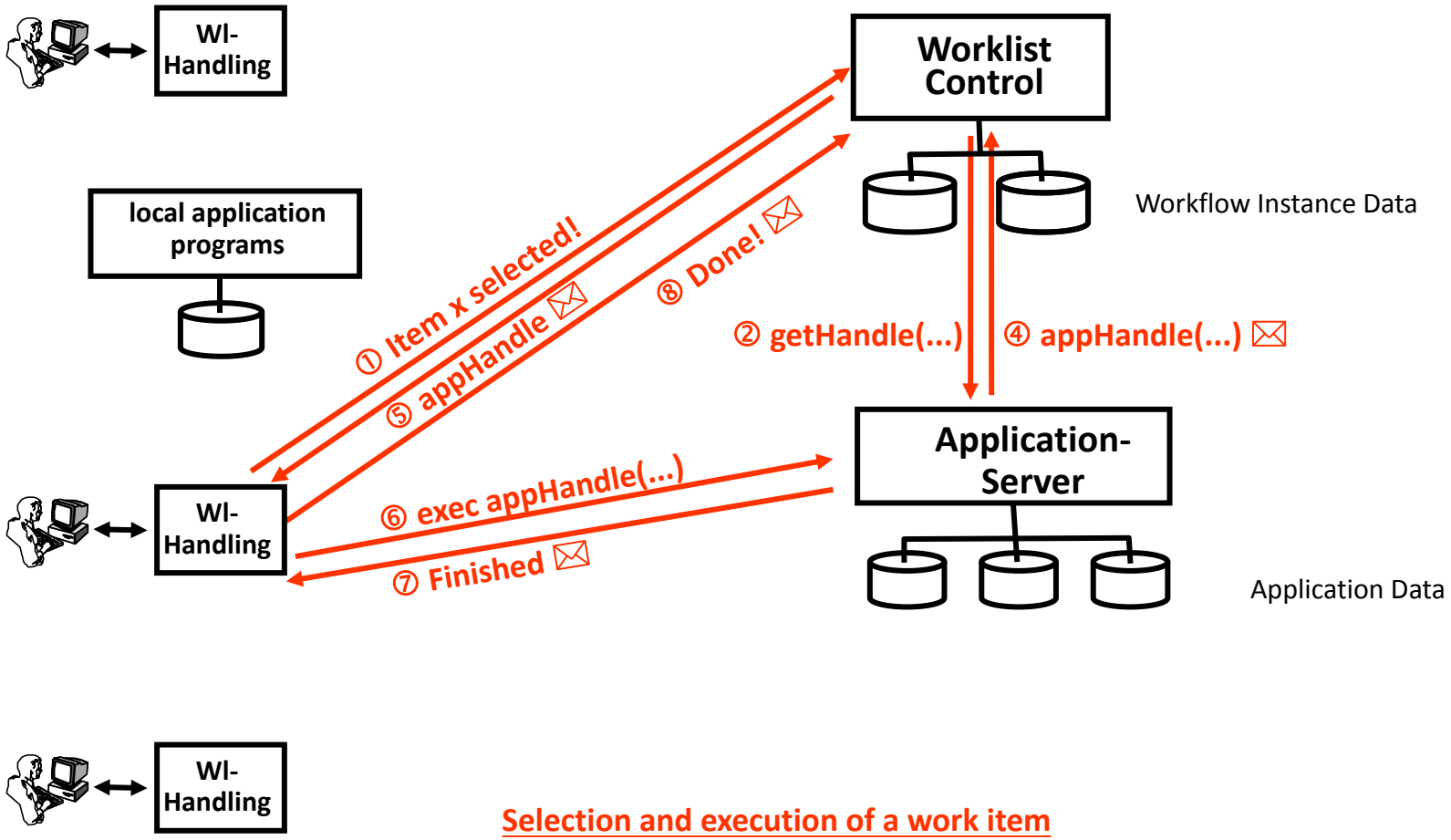
Selection and Execution of a Work Item



Selection and execution of a work item

Option 2: App.-Server – direct invocation by Wl-Handling

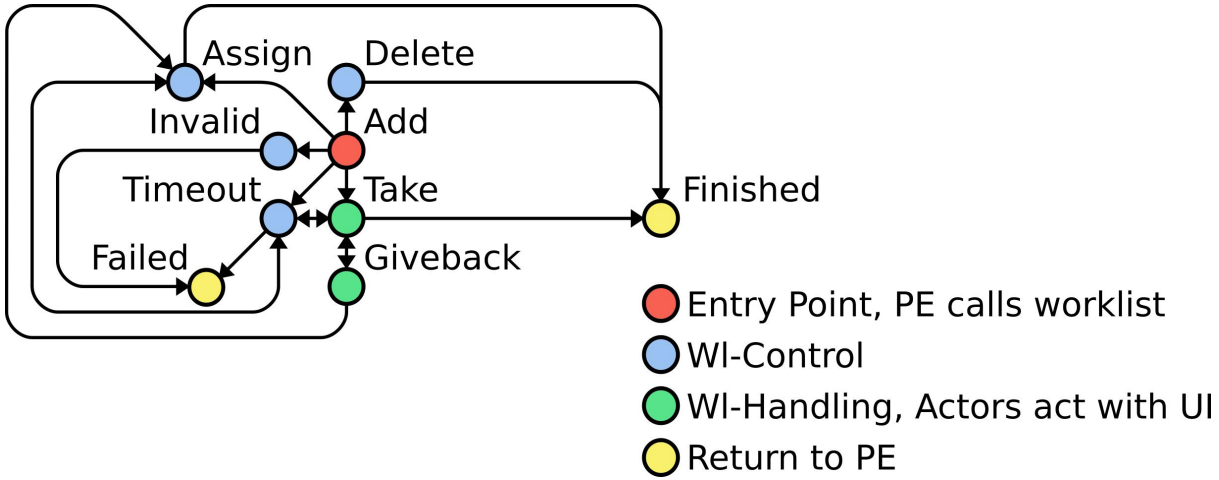
Selection and Execution of a Work Item



Selection and execution of a work item

Option 3: App.-Server – invocation prepared by WL-Handling

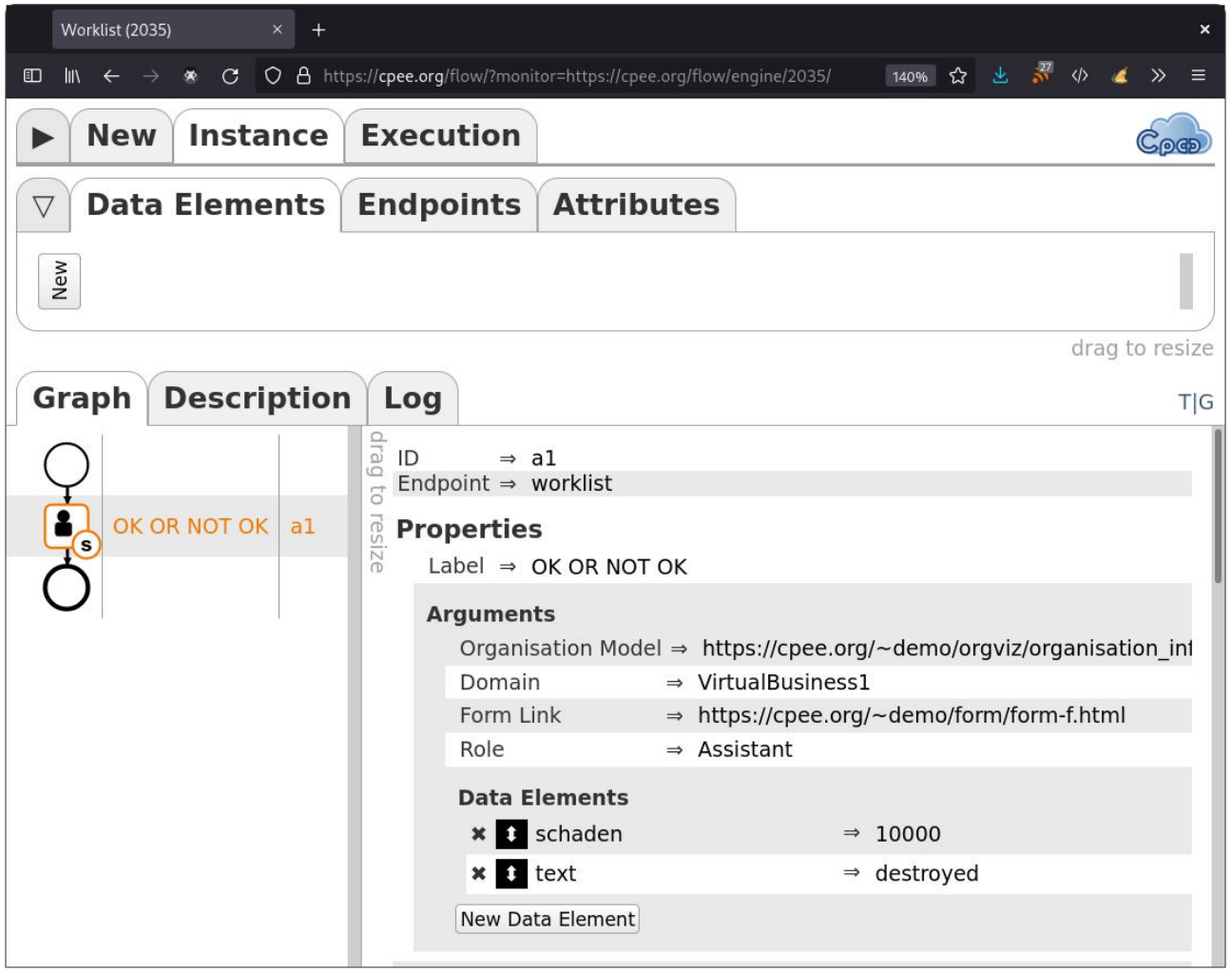
CPEE Worklist Lifecyle



- Invalid: the task is missing parameters. This leads to a failed state.
- Delete: the task is no longer necessary and thus deleted. Noetheless a regular finish.
- Failed: control is returned to the PE.
- Finished: control is returned to the PE. Actor-generated data might be returned.

- Add: a process engine (PE) delegates work on a task - typical parameters include:
  - Organization model
  - Role
  - Subject - HTML Form, Word Document, Application to call, ...
  - Data elements (process context)
- Take: an actor can actively request the task
- Giveback: an actor can return the task to be available for other actors to be taken
- Timeout: a deadline was violated (see temporal aspect)
- Assign: the task is automatically assigned to an actor. Assign might use the following algorithms:
  - Round Robin
  - Lowest workload
  - Skill based
  - Customer based
  - ...

CPEE Worklist



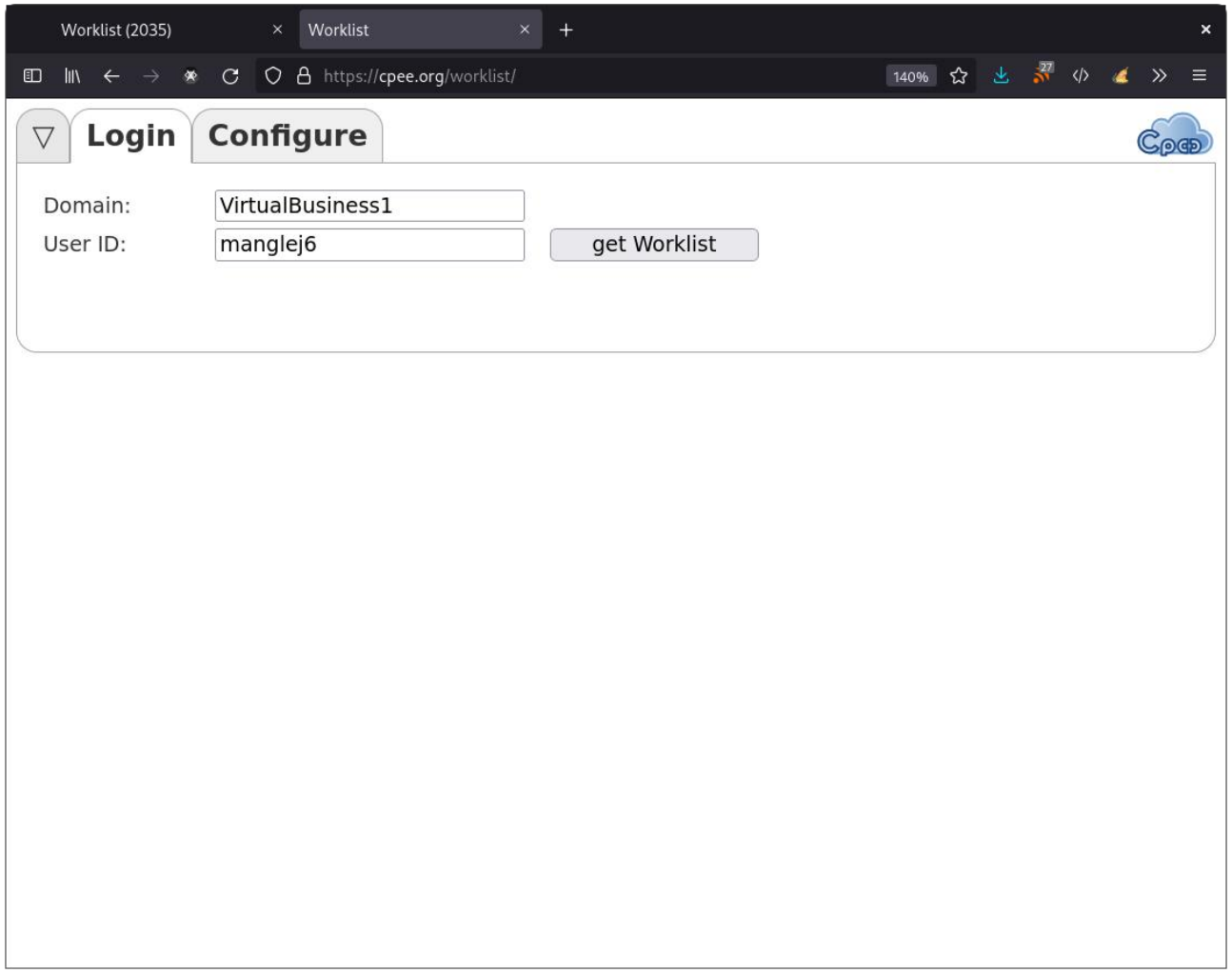
Simple sample process.

Task has parameters (see above):

- Endpoint: location of the worklist
- Organisation model
- Form Link: HTML form to be shown
- Role: group of actors that are authorized to work on the task
- Data elements to be passed to the worklist

The data elements should all information required by actors to work on the task. The actors will get not other information to work on the task (although they can use backend systems that are not known to the process engine).

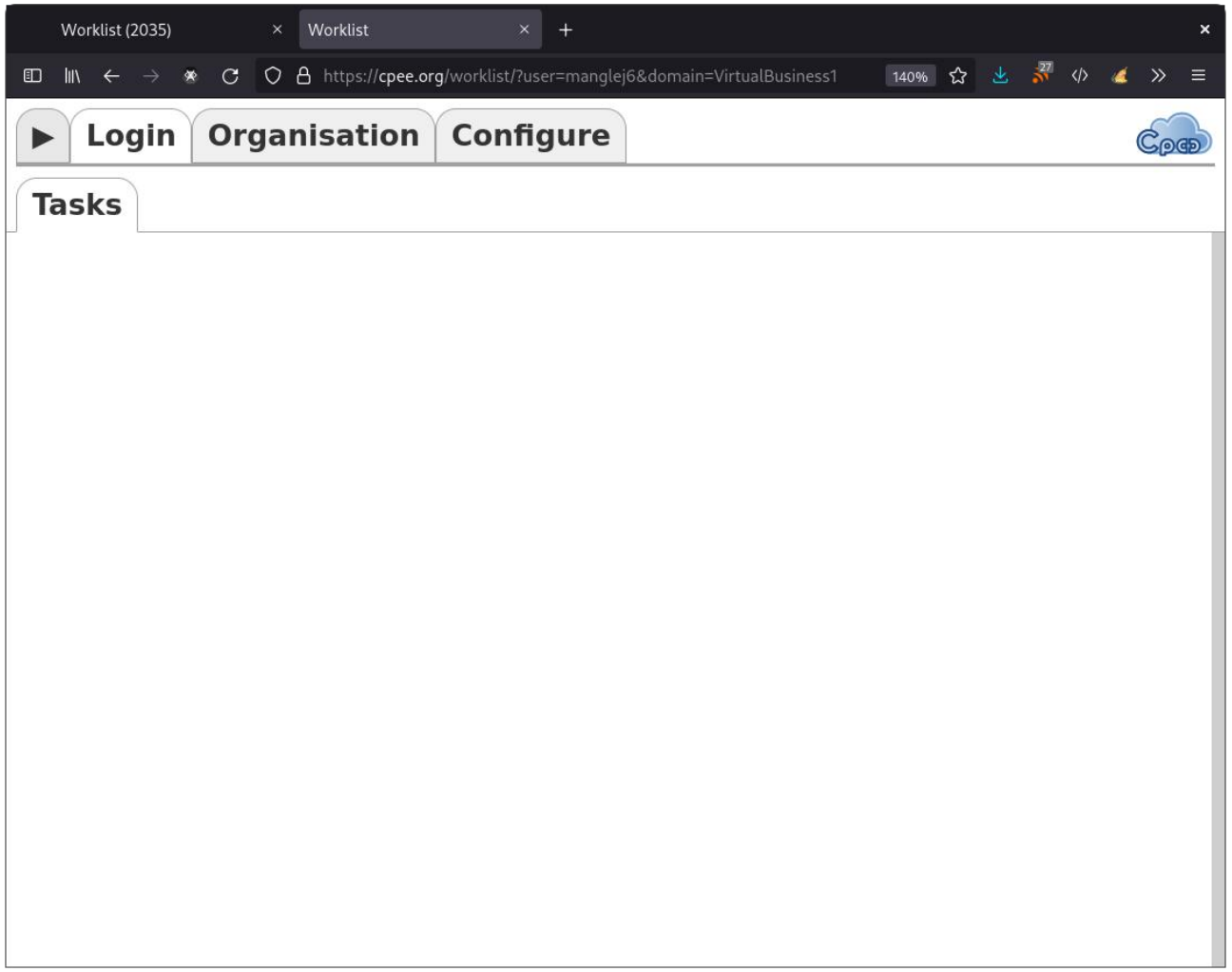
CPEE Worklist



Actors have a special UI. They never see process engine UI.

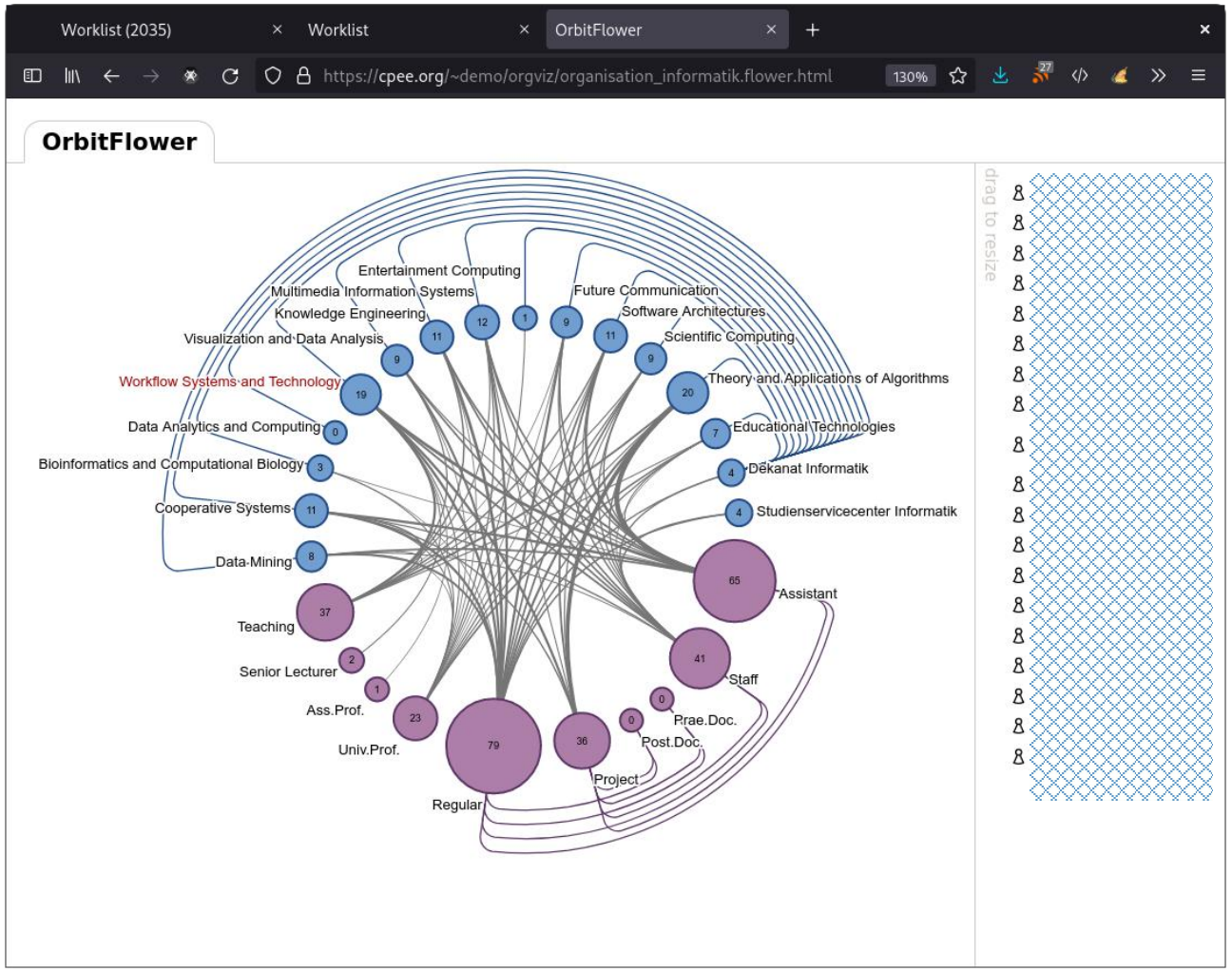
The have to log in (user manglej6).

CPEE Worklist



The worklist is empty.

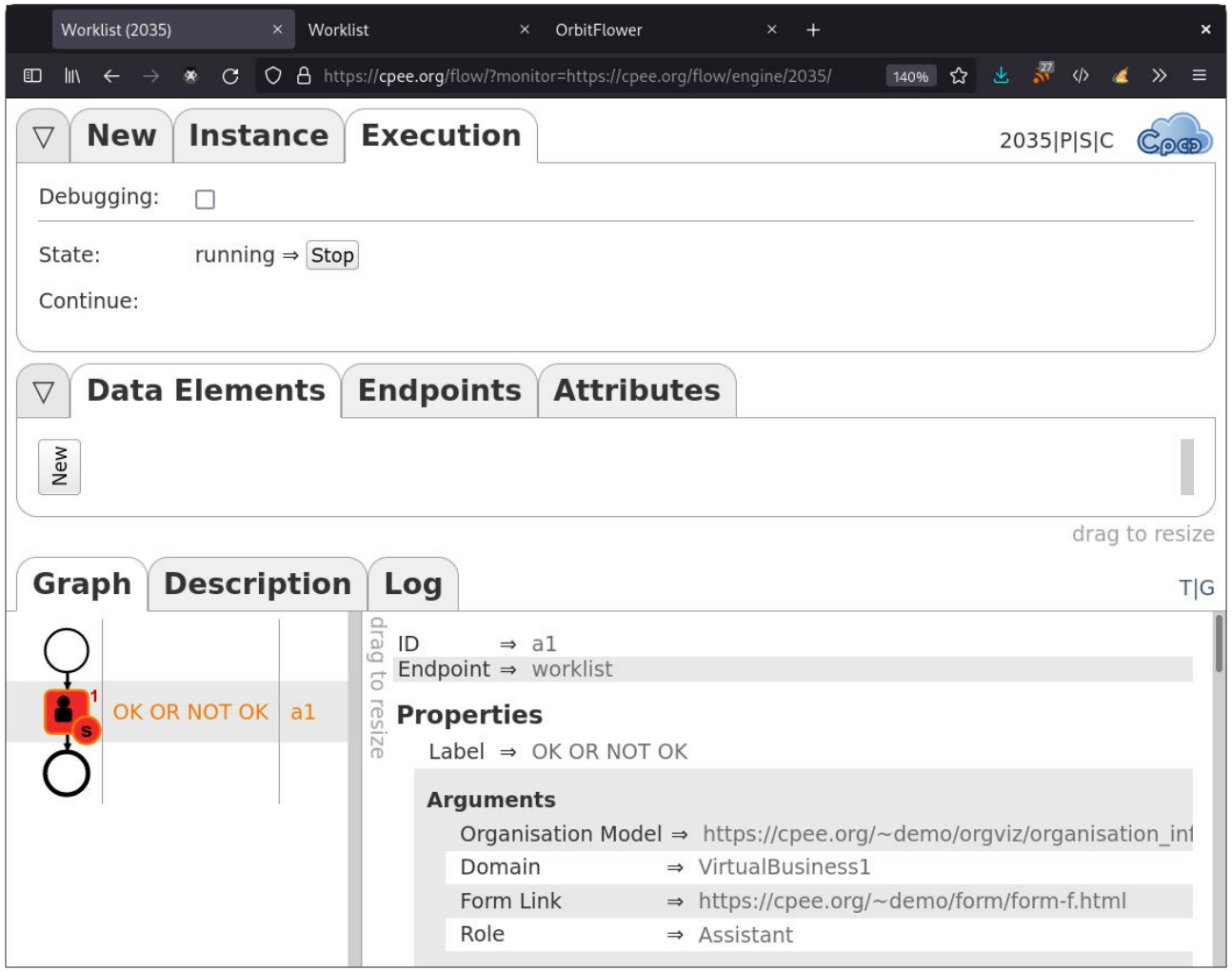
CPEE Worklist



The actor is part of an organisation, is assigned to a unit and has a role.

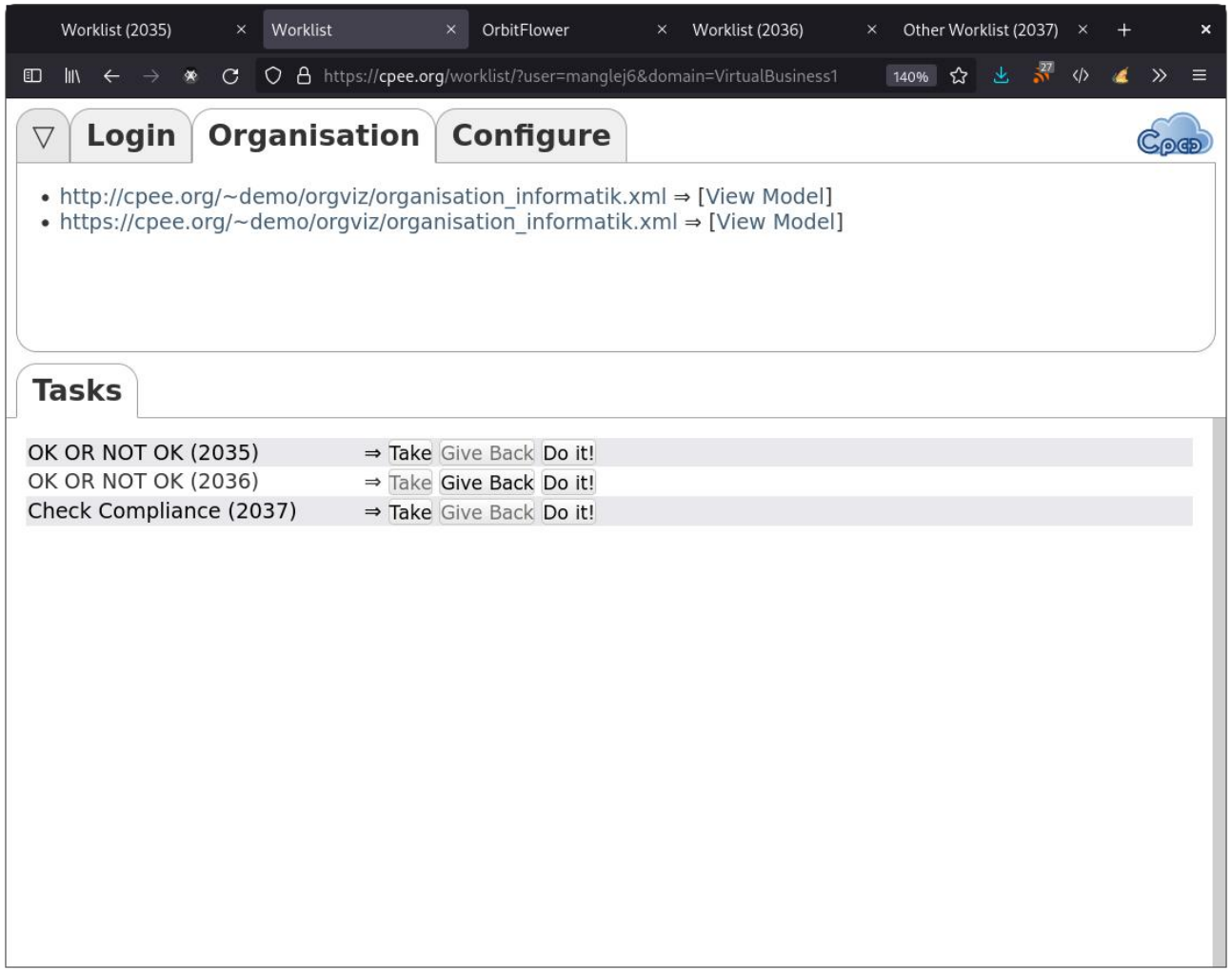
further reading OrbitFlower:
Simone Kriglstein, Juergen Mangler, Stefanie Rinderle-Ma: Who is who: On visualizing organizational models in Collaborative Systems. CollaborateCom 2012: 279-288

CPEE Worklist



The process instance is running.

CPEE Worklist



The task shows up in the actors UI.

Also:

- A second instance has been started.
- A task from a very different process also shows up.

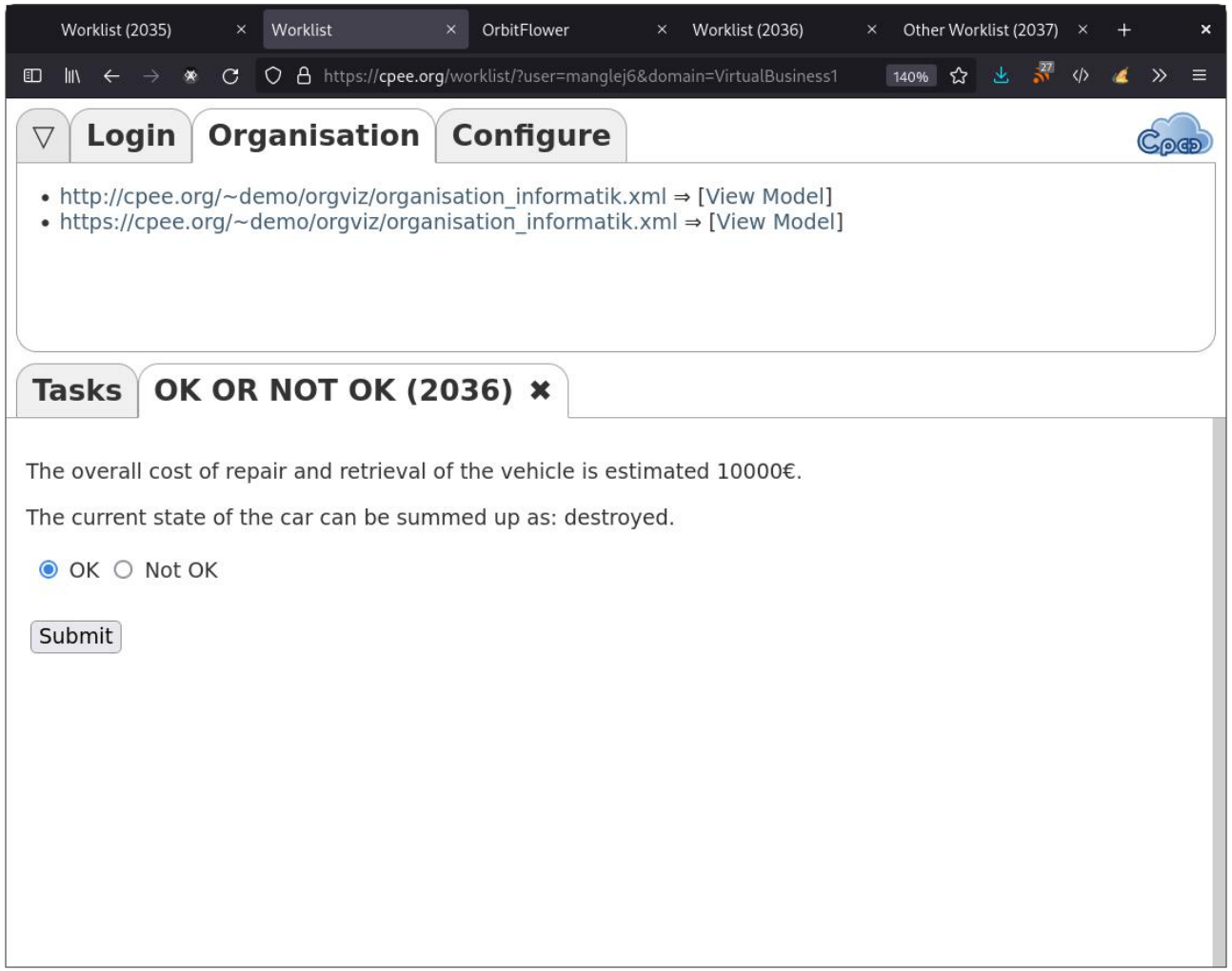
All tasks have in common: the actor role is authorized to work on them.

Tasks can:

- Taken
- Given back
- Done (which automatically takes them).

If an actor takes a task, it becomes invisible for all other actors. If it is given back it becomes visible again.

CPEE Worklist



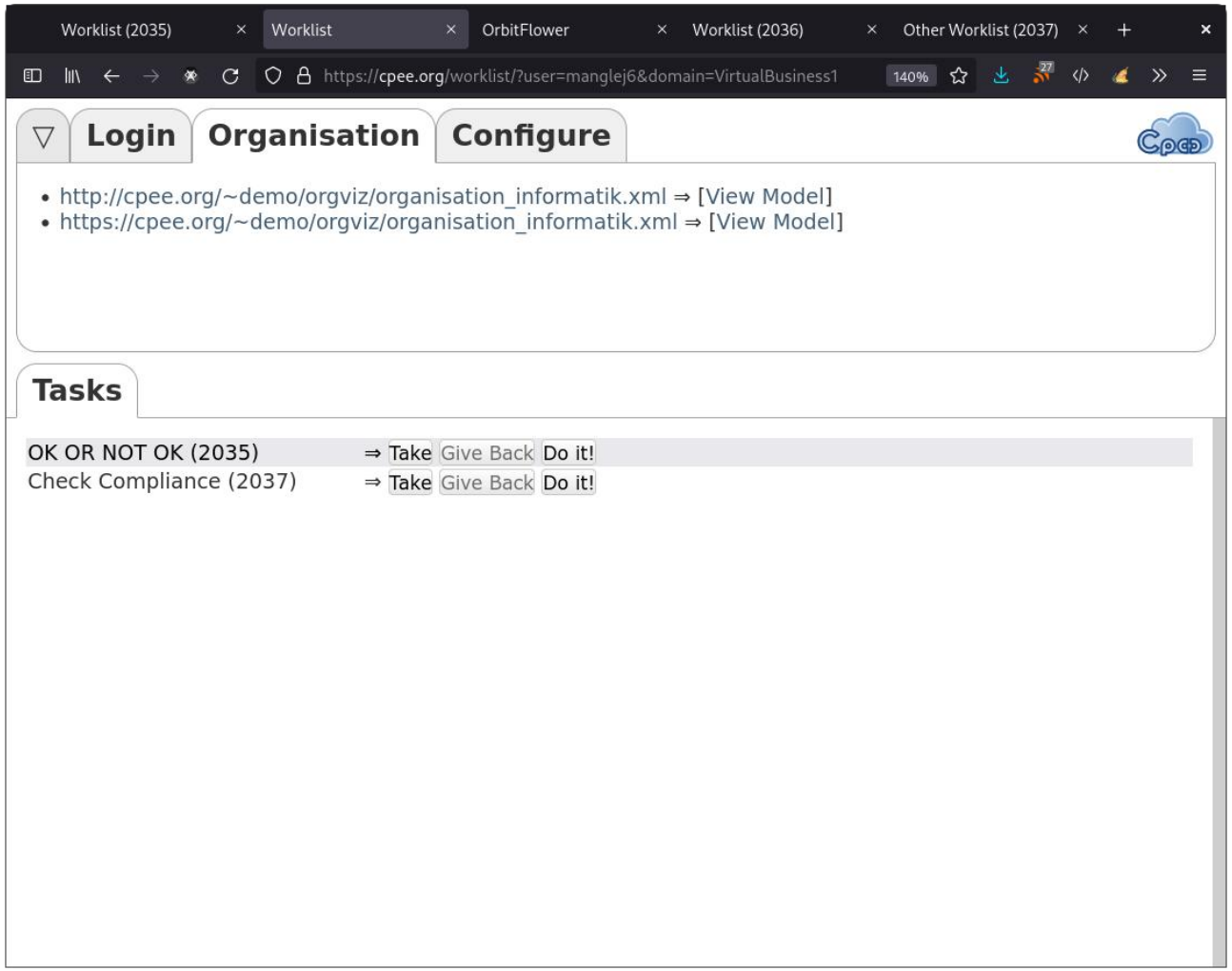
The actor works on a taken task.

The UI passed by the PE is shown inline.

The data elements passed by the PE are used to customize the UI:

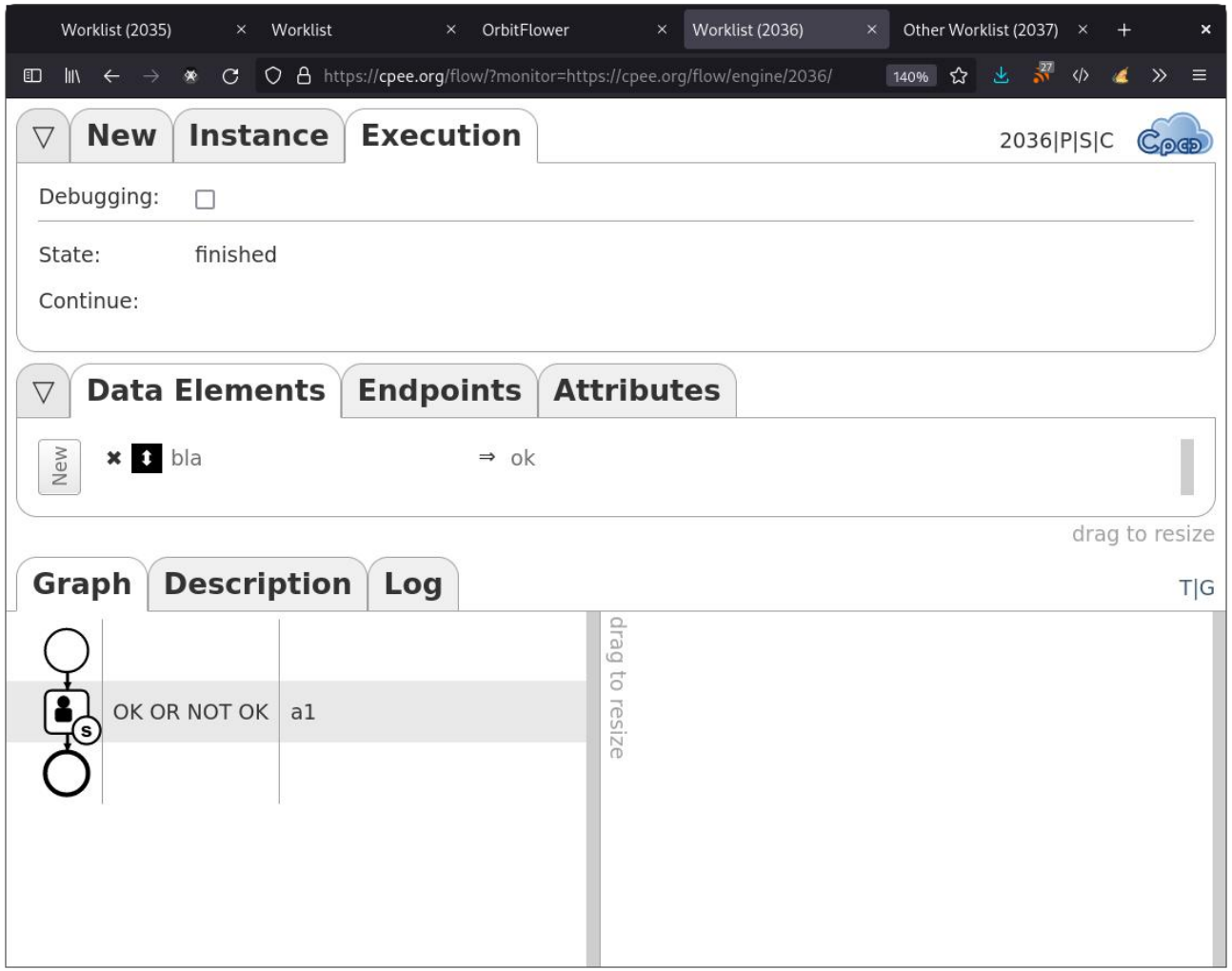
- The value in the data element “schaden” is copied into the UI.
- The value in the data element “text” is copied into the UI.

CPEE Worklist



The actor finished work on the task, it vanishes from the UI.

CPEE Worklist



The corresponding process instance is finished. The task has received the data from the worklist, and saved it into a data element “bla”.

Summary

- Process implementation means to transform your process model into executable code.
- Hence, all execution aspects have to be taken care of.
- More aspects:
  - Process evolution and change
  - Interorganizational process execution