Process-oriented organisation modelling and analysis

VIARA POPOVA AND ALEXEI SHARPANSKYKH[*]
Vrije Universiteit Amsterdam, Department of Artificial Intelligence,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
{popova, sharp}@cs.vu.nl

This paper presents a formal framework for process-oriented modelling and analysis of organisations. The high expressivity of the sorted predicate logic language used for specification allows representing a wide range of process-related concepts (e.g. tasks, processes, resources), characteristics and relations, which are described in the paper. Furthermore, for every organisation, structural and behavioural constraints on process-related concepts can be identified. Some of them should always be fulfilled by the organisation (e.g. physical world constraints), whereas others allow some degree of organisational flexibility (e.g. some domain specific constraints). An organisational specification is correct if it satisfies a set of relevant organisational constraints. This paper describes automated formal techniques for establishing correctness of organisational specifications with respect to a set of diverse constraint types. The introduced framework is a part of a general framework for organisation modelling and analysis.

Keywords: enterprise modelling, process-oriented modelling, workflows, formal verification, constraints

1 Introduction

Every organisation achieves its goals by performing a set of tasks. Tasks are defined as organisational functions and their specification depends on the organisational type. For example, in mechanistic organisations (Scott 2001) each task is characterized by a detailed procedure that describes every aspect of the task execution, whereas in organic organisations (Scott 2001) a task specification may consist only of interface (i.e. input and output) characteristics and a general task description. Many modern organisations combine different features of both mechanistic and organic organisations. For example, management tasks of an organisation are usually specified at a high level of abstraction, whereas routine production tasks are often defined by detailed procedures.

The execution of tasks is often specified by dynamic structures called flows of control (or workflows), in which tasks are represented by processes. Usually control flows are based on a set of temporal ordering rules over processes.

Mechanistic organisations are often characterized by complex (hierarchical) structures of tasks and processes, whereas in organic organisations these structures are relatively simple, however, constantly varying. Furthermore, both types of organisations include a variety of relations between tasks and processes and other organisational concepts (e.g. resources, roles, agents). Therefore, to handle the high complexity of modern organisations that often possess features of both mechanistic and organic types, automated modelling and analysis techniques are required. Furthermore, to enable automation and guarantee the correctness of analysis, both modelling and analysis technique should be formal.

To this end, this paper introduces a formal framework for process-oriented modelling and analysis. In this framework, tasks, processes, resources and other related concepts are specified in the formal language L_{PR},

[*] Corresponding author

based on the sorted first-order predicate logic (Manzano 1996). The high expressivity of predicate logic allows including into L_{PR} a wide range of process-oriented concepts specified by sorts, sorted constants, variables, functions and predicates that represent relations on these concepts. The domains for sorts are considered to be finite, which allows performing effective reasoning and computational analysis on process-oriented specifications of organisations in L_{PR}. Such specifications correspond to structures over L_{PR} that are defined by the particular interpretations of sorts, constants, functions and predicates, and variable assignments.

For every organisation a set of structural and behavioural *constraints* expressed over its tasks and processes can de identified, which should be satisfied by the process-oriented specification. In this paper the set of constraints is represented by the *logical theory* T_{PR} in L_{PR}, i.e. a set of sentences expressed in L_{PR}. This means that all concepts and relations defined in L_{PR} may be used for the specification of constraints. A process-oriented specification in L_{PR} is *correct* if T_{PR} is satisfied by this specification, i.e. all sentences in theory T_{PR} are true in the logical structure corresponding to the specification.

The constraints in T_{PR} may be of different types: some are dictated by the restrictions of the physical world and should be satisfied by any process-oriented specification; others depend on the application domain and may be changed by the designer. The classification of constraints is described in this paper. Furthermore, this paper also introduces automated techniques for establishing the correctness of a process-oriented specification by verifying constraints. Interdependences that may exist in constraint sets are also handled by the proposed verification techniques. To our knowledge there exist no other frameworks that allow the simultaneous verification of different (interdependent) types of constraints based on the extensive set of concepts and relations as can be found in L_{PR}.

The framework introduced in this paper constitutes a part of a general formal framework for organisation modelling and analysis (Popova and Sharpanskykh 2007a) in which organisations are considered from other perspectives (or views) as well. In particular, *the performance-oriented view* (Popova and Sharpanskykh 2007c, Popova and Sharpanskykh 2007d) describes organisational goal structures, performance indicators structures, and relations between them. Within *the organisation-oriented view* (Broek et *al*. 2006, Sharpanskykh 2007a) organisational roles, their authority and interaction relations are defined. In *the agent-oriented view* (Popova and Sharpanskykh 2007a) different types of agents with their capabilities are identified and principles for allocating agents to roles are formulated. Concepts and relations within every view are formally described using dedicated languages based on the expressive order-sorted predicate logic. Furthermore, the views are connected to each other by means of sets of relations. This enables different types of analysis across different views. An example of such analysis involving the process- and performance-oriented views is the organisational performance evaluation considered in (Popova and Sharpanskykh 2007c). The relations between processes on the one hand and goals, performance indicators, roles and agents on the other hand are introduced in this paper.

The proposed views and concepts of the framework are similar to the ones defined in the Generalized Enterprise Reference Architecture and Methodology (GERAM) (Bernus *et al.* 1998) developed by the IFIP/IFAC Task Force, which forms a basis for comparison of the existing enterprise architectures and serves as a template for the development of new enterprise modelling frameworks. Although many enterprise architectures include a rich ontological basis for creating specifications of different views, most of them provide only a limited support for automated analysis of these specifications, addressed in the category *Enterprise Engineering Tools* of GERAM, primarily due to the lack of formal foundations in these frameworks. In contrast, the proposed framework enables different types of automated analysis both within particular views and across different views (Popova and Sharpanskykh 2007a).

The framework has been applied on a number of case studies from different domain areas. One such case study from the area of air-traffic control is described in (Sharpanskykh 2007b).

The paper is organized as follows. First, in Section 2, the running example for this paper is described. Section 3 introduces the language L_{PR}. Section 4 describes the classification of constraints. In Section 5, the methods for verification of constraints are given. In Section 6 the related work on process-oriented modelling is discussed. Section 7 concludes the paper.

2 The case study

To illustrate different aspects of our approach a running example is used that describes the operation of a 3PL (third-party logistics) provider. In general, 3PL companies provide logistics services to other companies. The considered operation cycle begins with the customer order intake process, after which the order is processed and depending on the customer (company) is scheduled for some delivery type (for different companies different delivery regulations may be applied). During the delivery the assigned driver is supervised by the assigned fleet manager. After the delivery is finished, the delivery summary report is provided to the customer.

In the context of this example consider a particular delivery scenario (see Fig. 1): The logistics company performs shipments of resources between three bases of some manufacturing enterprise (A, B and C), which are located in different regions. The base A has the storage facilities for both raw materials delivered by suppliers and for finished products of the enterprise prepared for further shipments to customers. The raw materials stored at A are required for the production processes of the enterprise's departments located at B and C (the resources of type rt1 - for B and the resources of type rt2 - for C) and are transported from A to these departments by trucks. Each delivery is preceded by the process of resource loading and is followed by the resource unloading process. The 3PL company owns trucks of two types: large trucks of the type tr1 with three capacity units each (a capacity unit is a relative spatial capacity measure of a truck) and small trucks of the type tr2 with one capacity unit each. All deliveries between the bases of the enterprise are performed by two trucks of the type tr1 and three trucks of the type tr2. The base C is geographically located between A and B, however also a more direct connection between A and B exists. At C two types of products are produced: the finished product of type rt4, which should be shipped to A, and the intermediate product of type rt3 that is used as an assembling part at B. Using raw materials of type rt1 and products of type rt3 the department at B produces finished products of type rt5, which should be subsequently shipped to the storage facilities at A.

In the considered scenario initially all trucks are located at the base A. The company assigns one truck of type tr1 to the direct delivery d1 of the materials of type rt1 from A to B. The delivery d2 of resources of type rt1 from A to B through C starts simultaneously with d1. The delivery d2 shares two trucks of the type tr2 with d3, the delivery of resources of type rt2 from A to C. The mode of sharing of each tr2 type truck is determined by particular weight and spatial restrictions. In particular, the weight limitations prescribe that at most 70% of the capacity unit of a tr2 truck can be filled with resources of type rt2, whereas the remaining 30% should be left empty. When a tr2 truck is shared between resources of types rt1 and rt2 (or of types rt1 and rt3), then the resources of type rt1 may occupy at most 50% of the truck's space. Moreover, one more tr2 type truck is assigned to d3. When d3 is finished and the resources of type rt2 are unloaded, this truck is scheduled to be loaded with resources of type rt4 and to return back to A (the delivery d6). After the resources of the type rt2 are unloaded at C from two trucks of the type tr2, the emptied space in these trucks will be promptly filled with products of type rt3, which are scheduled to be delivered to the base B (delivery d4). The deliveries d2 and d4 again share the tr2 trucks equally. After both d4 and d2 are finished, both tr2 trucks will be unloaded and loaded with finished products of the type rt5, which are scheduled to be delivered to A (the delivery d7). Similarly, the truck used for d1, after being unloaded at B, will be loaded with products of type rt5 that are required to be delivered to A (delivery d5).

3 Process-oriented modelling

Process-oriented specifications in the proposed framework are specified using the sorted predicate language L_{PR}. In this Section, a general overview of L_{PR} is given.

A *task* represents a function performed in the organisation and is characterized by a name and by a maximal and a minimal durations. The sort TASK contains the names of all tasks. The characteristics of the tasks are specified by the following predicate: task: TASK × TASK_PROPERTY × VALUE ∪ STRING. We sometimes use the following short notation for specifying these characteristics: t.p = v, where t∈TASK, p∈TASK_PROPERTY and v∈ VALUE ∪ STRING. Characteristics of other concepts defined below are formalized in a similar way. For example, the task Order_intake has minimal duration 1 hour and maximal duration 2 hours depending on the experience and the efficiency of the agent performing the task: Order_intake.min_duration=1h, Order_intake.max_duration=2h.

Tasks can range from very general to very specific. General tasks can be decomposed into more specific ones using AND- and OR-relations thus forming hierarchies. It is specified using the following predicates:

is_in_task_list: TASK × TASK_LIST specifies a task is a member of a task list

is_decomposed_to: TASK × TASK_LIST specifies that a task is decomposed into an AND-list of tasks meaning that all tasks in the task list together are necessary and sufficient in order to perform the decomposed task. Sometimes alternative decompositions of a task are possible which are connected by an OR-relation. They are specified as separate decompositions of the same task. For example the task DeliveryComp1 which encompasses all deliveries for Company 1, as described in Section 2, can be decomposed in several types of deliveries, modelled as more specific tasks in the task hierarchy of DeliveryComp1, which are all necessary for fulfilling the agreements with the client such as: delivery from A to B (DeliveryAB), from B to A (DeliveryBA), from A to C (DeliveryAC), from C to B (DeliveryCB) and from C to A (DeliveryCA). Therefore they should be modelled as an AND-decomposition list of task DeliveryComp1:

is_decomposed_to(DeliveryComp1, L)
is_in_task_list(DeliveryAB, L)
is_in_task_list(DeliveryBA, L)
is_in_task_list(DeliveryAC, L)
is_in_task_list(DeliveryCB, L)
is_in_task_list(DeliveryCA, L).

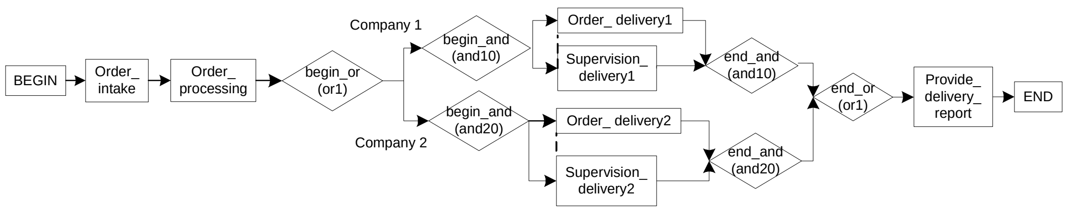
These deliveries can be executed in different ways. For example DeliveryAB, which should transport 4 capacity units of resource type rt1, can be performed by two big trucks (DeliveryAB20) or by one big and two small trucks (DeliveryAB12) or by four small trucks (DeliveryAB04). Such information can be represented as alternative decompositions of the task DeliveryAB:

is_decomposed_to(DeliveryAB, L1)
is_in_task_list(DeliveryAB20, L1)
is_decomposed_to(DeliveryAB, L2)
is_in_task_list(DeliveryAB12, L2)
is_decomposed_to(DeliveryAB, L3)
is_in_task_list(DeliveryAB04, L3).

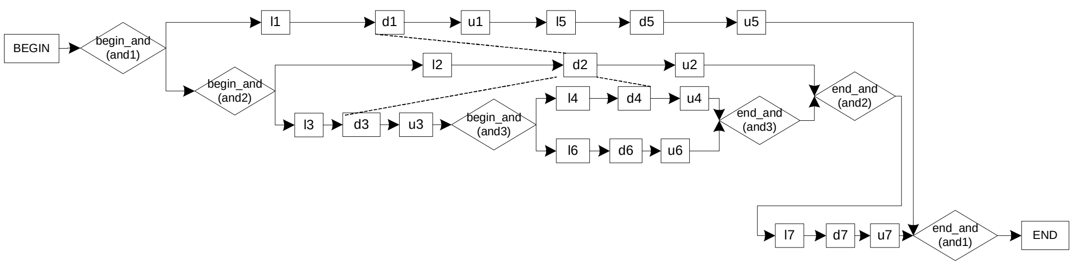
These alternative subtasks can be decomposed further using the information that there are two possible routes between A and B – direct and via C. For example, for DeliveryAB12 it was decided to split the trucks so that one big truck travels directly from A to B and two small trucks travel via C.

A *workflow* is defined by a set of (partially) temporally ordered *processes*. Each process, except for the special ones with zero duration introduced below, is defined using a task as a template and all characteristics of the task are inherited by the process. This is specified using the predicate: is_instance_of: PROCESS × TASK, e.g. is_instance_of(d1, DeliveryAB). Decisions are also treated as processes that are associated with decision variables taking as possible values the possible decision outcomes.

Definition 1 (A workflow): A workflow with the name w is defined by a tuple <w, P, C> with a set of processes P and a set of ordering relations C on processes from P.



(a)



(b)

Fig. 1. The generalized workflow that illustrates the operation of a 3PL delivery company (a) and the detailed workflow for the deliveries for a specific company (b).

Fig.1 is a graphical representation of the workflow built for the running example. It shows the processes at two different aggregation levels. Fig.1a shows the overall workflow for processing and executing an incoming order. Fig.1b is a more detailed description of the workflow corresponding to process Order_delivery1 for the orders of Company 1. Analogous detailed workflow can be built for the deliveries to Company 2 using relevant information (left out of the case study for simplicity).

A workflow starts with the process BEGIN and ends with the process END; both have zero duration. The (partial) order of execution of processes in the workflow is defined by sequencing, branching, loop and synchronization relations (referred to as ordering relations) specified by the designer.

A *sequencing relation* is specified by the predicate starts_after: PROCESS $\times$ PROCESS $\times$ VALUE expressing that the process specified by the first argument starts after the process specified by the second argument with the delay expressed by the third argument, e.g. starts_after(Order_processing, Order_intake, 0) represented graphically by solid arrows between the processes. For each process p, different from BEGIN and END at least two sequencing constraints are defined, which specify the process that precedes p and the process which follows after p.

By specifying sequencing constraints different paths of a workflow are formed. A *path* is a sequence of processes (p_1, p_2, ..., p_{n-1}, p_n), where n>1 and starts_after(p_2, p_1) $\wedge$ starts_after(p_3, p_2) $\wedge$ ... $\wedge$ starts_after(p_n, p_{n-1}). A path in a workflow, in which $p_1 = p_n$ is called a *cycle*. No cycles are allowed in the workflow structures.

Synchronization relations define temporal relations between processes that are executed in parallel (e.g. starts_with, finishes_with, starts_during: PROCESS $\times$ PROCESS). An example of such a relation is shown by a dashed line between the beginnings or endings of the processes in Fig. 1, meaning that the connected processes should start or finish simultaneously. For example: starts_with(d2,d3), finishes_with(d2,d4) which comes from the fact that these deliveries share trucks and therefore it is impossible to have them start / finish at different times. Taken together, synchronization and sequencing relations allow specifying all cases of interval relations defined in (Allen 1983).

Branching relations are defined over and- and or-structures. An and(or)-structure with name id, starts with the zero-duration process begin_and(id) (begin_or(id)) and finishes by the zero-duration process end_and(id) (end_or(id)). These special processes are represented graphically by rhombuses. Our treatment of AND-structures is similar to the parallel split pattern combined with all types of the merge pattern from (van der Aalst *et al*. 2003), represented in our case by an and-condition. The first processes in every branch of an and-structure start at the same time. For each and-structure a condition is defined (and_cond: AND_STRUCTURE × CONDITION_EXPRESSION), which determines when the process p following after the and-structure may start. The following types of conditions may be used: (1) constant any: meaning that as soon as all processes of one of the branches finish, the process p starts; (2) constant all: meaning that as soon as all processes of all branches finish, the process p starts; (3) a condition expressed by a logical formula constructed from the functions finished, not_finished: PROCESS → {true, false} using Boolean connectives ∨ and ∧. In Fig.1a the and-structures contain the condition value all, meaning that only when all processes in the and-structure are finished, the process specified after the end of the and-structure is allowed to start. This is represented formally as: and_cond(and0, all). In Fig.1b AND-structure and3 contains condition finished(u4) therefore the execution of the workflow will continue only when process u4 finishes irrespective of whether the last process of the other branch (u6) is finished or not. The reason for this is that the branch of d6 (corresponding to the delivery from C to A) does not influence the processes after the AND-structure, thus, they can start theirs execution without waiting.

For every or-structure a condition is defined (or_cond: OR_STRUCTURE × CONDITION_EXPRESSION), based on which it is determined which branches of the or-structure will start. The condition may consist only of a condition variable or it may be a disjunction of conjunctions of expressions in the form condition_variable [OP value], where OP∈{=, ≠, <, >}, and value belongs to the domain of the condition variable. The following types of condition variables can be used: (1) a decision variable; (2) a variable over the sort that includes all states of a certain object in the environment, e.g. market conditions, customer demand, taxes, weather conditions, etc.; (3) a variable over the sort that includes all values of certain characteristic of an object in the environment. For an or-structure branches are specified that correspond to all possible values of the condition expression, using the predicate or_branch: OR_COND_VALUE × PROCESS, which expresses that the branch of the or-structure that begins with the process specified in the second argument corresponds to the value of the condition expression specified in the first argument. An or-branch may correspond to the constant other, which should be interpreted as all other values from the domain of the condition variable. Our treatment of or-structures allows realizing both exclusive and multiple-choice patterns from (van der Aalst *et al*. 2003). The or-structure in the running example, Fig.1a specifies the exclusive choice between two types of delivery depending on the company name. Here the company name is the characteristic company of the environmental object incoming_order and takes two possible values, Company1 and Company2. Therefore the condition of the OR-structure can be represented in the following way:

or_cond(or1, incoming_order.company)
or_branch(Company1, begin_and(and1))
or_branch(Company2, begin_and(and2))

Loop relations are defined over *loop*-structures with conditions that realize cycle patterns from (van der Aalst *et al*. 2003). A loop-structure with name id, starts with the zero-duration process begin_loop(id) and finishes by the zero-duration process end_loop(id). For every loop-structure a Boolean condition (loop_cond: LOOP × CONDITION_EXPRESSION) and the maximal number of times of the loop execution (loop_max: LOOP × VALUE) are specified. No loops were identified for the running example.

Tasks use, consume and produce resources of different types. Resource types include tools, supplies, components and other material or digital artefacts. Also data are considered as a special resource type. The predicates task_uses, task_consumes, task_produces: TASK × RESOURCE_TYPE × VALUE indicate resource types that are input or output of tasks with their prescribed amounts. Resource types are characterized by: *name*; *category* – discrete or continuous; *measurement_unit*; *expiration_duration* – the length d of the time interval for which a resource type can be used. Specific resources represent instances of particular resource types and inherit their characteristics. The resources have, in addition to the inherited characteristics, also *name* and *amount*. Every resource in the workflow has to be produced by a process of this workflow or be available in the organisation before the beginning of the workflow execution.

For the running example, 5 resource types were identified rt1 to rt5 corresponding to the raw materials, components and products described in Section 2 as well as tr1 and tr2 corresponding to the two types of trucks available for the deliveries. For each task which uses a resource type, the corresponding necessary amount can be specified e.g. task_uses(deliveryAB12, tr1, 1), task_uses(deliveryAB12, tr2, 2), task_uses(deliveryAB12, rt1, 3). Furthermore no resource type is consumed or produced by a task in the example since the operations of the client company are considered out of the scope and are not part of this workflow.

Resource types can sometimes be functionally divisible, i.e. they can be divided in parts in such a way that a part can have a different purpose, therefore is a different resource type. For example a car can be decomposed to its components which do not have the same purpose as the car. This is specified by the following predicate:

is_func_part_of: RESOURCE_TYPE × RESOURCE_TYPE where the second resource type is functionally divisible and the first resource type is its functional part. For example computer might be defined as a functionally divisible resource since its parts have different purpose than the whole computer.

Some resources can be shared (used simultaneously) by a set of processes (e.g. storage facilities, transportation vehicles, some computers). The predicate resource_sharable: RESOURCE_TYPE × PROCESS_LIST defines that the resource type can be used (but not consumed!) by the processes in the list (or a sub-list of this list) at the same time. The shared amount of the resource type should be sufficient for the execution of every process in the process list. Alternative sharing of the same resource type can be specified as well. Our representation of shared resources is different from (Barkaoui and Petrucci 1998) in several aspects: (1) the shared resource amount is used by processes simultaneously; (2) alternative sets of processes that are allowed to share a resource can be defined; (3) different amounts of a resource can be shared simultaneously; (4) specific conditions (requirements) for resource sharing can be defined. In the running example, trucks may be considered as shared resources. Delivery processes d2, d3 and d2, d4 are required to share trucks of type tr2:

```
resource_sharable(tr2, L1)
resource_sharable(tr2, L2)
is_in_process_list(d2, L1)
is_in_process_list(d3, L1)
is_in_process_list(d2, L2)
is_in_process_list(d4, L2)
```

Every resource in the workflow has to be produced by a process of this workflow. This is specified using the predicate: process_output: PROCESS × RESOURCE. In this way the end time point of the process producing the resource is taken as creation time for this resource and the time when it will expire is calculated with respect to this creation time. A process can produce only one resource instance of the same resource type. In some situations, resources could be available in the organisation before the beginning of the workflow execution, for example machines and other durable tools, materials already purchased, etc. In these cases such resources that will be used in the workflow are specified as output of its process BEGIN. For the running example, the used resources are considered available at the beginning of the workflow i.e. produced by the BEGIN process:

```
process_output(BEGIN, r1)
r1.amount = 4
is_instance_of(r1, rt1)
etc.
```

For some application areas it is important to keep track of where the resources are at certain time points. For modelling such information the concept of a *location* is used which for example can represent the available storage facilities. Processes can add or remove resource types from locations which can be specified using the predicates: process_adds_resource_type_to: PROCESS × RESOURCE_TYPE × LOCATION × VALUE and process_rem_resource_type_from: PROCESS × RESOURCE_TYPE × LOCATION × VALUE where the last argument specifies the amount of the added or removed resource type. Resources are considered removed at the starting time point of the corresponding process and are added at the ending time point of the process.

For the running example we define three locations corresponding to the storage facilities of A, B and C. Only delivery processes are expected to add or remove resource types from locations (during loading and unloading the resources are considered to be still at the same location). Therefore we can specify that:

process_rem_resource_type_from(d2, rt1, A, 2)
process_adds_resource_type_to(d2, rt1, B, 2)
etc.

The process-oriented view is related to the *organisation-oriented* and the *agent-oriented views* through the sorts ROLE and AGENT. Each object of the sort ROLE describes a set of functionalities realized by organisational processes in a certain specification, which are assigned together to individuals who will be performing them. These individuals are objects of the sort AGENT. An agent can be allocated to one or more roles if it satisfies the requirements for performing these roles. For example, role_performs_process(Driver,d1) and agent_plays_role(Allan, Driver). For more details the reader is referred to (Popova and Sharpanskykh 2007e).

The process-oriented view is also related to the *performance-oriented view* through the sorts GOAL and PI (performance indicator). Objects of sort GOAL are organisational objectives and are defined as expressions based on performance indicators (objects of sort PI). The performance-oriented view is discussed in detail in (Popova and Sharpanskykh 2007c, Popova and Sharpanskykh 2007d). It relates to the process-oriented view through the following predicates:

is_realizable_by: GOAL × TASK_LIST – the goal can be realized by the tasks in the list

measures: PI × PROCESS – the performance indicator expresses an aspect of the performance of the process.

4 Constraints

Constraints are expressed as formulae in theory T_{PR} that are constructed from terms of L_{PR} in a standard way (Manzano 1996) using Boolean connectives and quantifiers over variables. The constraints are divided in two groups: (1) *generic constraints* need to be satisfied by any specification built using this framework; (2) *domain-specific constraints* are dictated by the application domain of the specification. Two types of generic constraints are considered: (1) structural constraints used to ensure correctness of the workflow, task and resource hierarchies; (2) constraints imposed by the physical world. Both types of generic constraints are described in Sections 4.1. Section 4.2 discusses the domain-specific constraints.

4.1 Generic constraints

The language allows building three types of structures: the workflow, the task hierarchy and the resource hierarchy. For each of them structural constraints are defined.

Workflow structural constraints

With respect to the workflow we define a set of structural constraints: structural correctness, temporal correctness and condition correctness constraints.

Structural correctness of the workflow

First let us introduce *reachability* and *complete reachablitity* relations.

Definition 2 (Reachability relation): The process p2 is reachable from the process p1 in the workflow w (reachable_from_in(p2, p1, w)) if there exists a sequence of processes constructed using the sequencing relations that starts at p1 and includes p2.

On Fig.1b, for example, process l7 is reachable from process u3 through two sequences of processes, one of which is: u3, begin_and(and3), l4, d4, u4, end_and(and3), end_and(and2), l7. However process u3 is not reachable from process l7. Also, for example, process u3 is not reachable from process u2 and process u2 is not reachable from process u3.

The truth value of the relation reachable_from_in(p2, p1, w) is determined as follows:

```
1. Initial settings: Let L be an empty FIFO queue and A be an empty set. Put p1 into L.
2. Until L becomes empty perform steps 3-6.
3. Dequeue L and assign the obtained process name to the variable curr_process.
4. Identify the set A={a | starts_after(a, curr_process)}
5. If p2∈A, then return true.
6. Enqueue all elements of A in L.
7. Return false.
```

To investigate the correctness and the computational properties of this algorithm, a workflow <w, P, C> is represented as a unidirected graph $G = (V, E)$, in which each vertex $v \in V$ represents some organizational process $p \in P$, and each edge $e \in E$ with the initial vertex representing the process x and the terminal vertex representing the process y corresponds to the relation starts_after(y, x). Note that loops represented in the process-oriented language by sequences of processes do not introduce cycles in the corresponding graph representation. The algorithm considers gradually all vertices in the graph that belong to the paths beginning from the vertex that represents the process p1, until it finds the process p2. Since each path in a workflow finishes with the vertex corresponding to the process END, any execution of the algorithm eventually terminates. Such type of algorithms is called breadth-first search and its correctness and computational properties are well established (cf., Cormen et al, 2001).

The time complexity of the proposed algorithm is estimated for the worst case under the assumption that each primitive operation (e.g., assignment of a value, extracting/placing from/into a queue) takes one time unit: The step 1 of the algorithm is performed only once and takes 1 time unit. The steps 2-6 in the worst case can be repeated $|P|-1$. The step 2 takes 1 time unit each time it is executed. The step 3 takes 2 time units each time when it is executed. All executions of the step 4 taken together in the worst case may take $|P-1|\cdot(|E|+1)$ time units. The step 5 does not take more than $|P|-2$ time units. The step 6 for all its executions taken together in the worst case may take $|P|-2$ time units. Thus, the overall time complexity of the algorithm for the worst case is $O(|P|*(|E|+|P|)$. However, in most practical cases the time complexity is less than $O(|P|*(|E|+|P|)$, since often subsets of $|P|$ are considered.

Definition 3 (Complete reachability relation): The process p2 is completely reachable from the process p1 in the workflow w (completely_reachable_from_in(p2, p1, w)) if all process sequences built using the sequencing relations that start at p1 include p2.

For example, the process Provide_delivery_report from the running example, Fig.1a, is completely reachable from the process Order_intake but process Order_delivery1 is not completely reachable from the process Order_intake.

The truth value of the complete reachability relation completely_reachable_from_in(p2, p1, w) is determined by the following algorithm:

```
1. Initial settings: Let L be an empty queue and A be an empty set. Put p1 into L.
2. Until L is empty perform steps 3-5.
3. Dequeue L and assign the obtained process name to the variable curr_process.
4. Identify the set A={a | starts_after(a, curr_process)}
5. If A=∅, then return false.
   else if A≠{p2}, then enqueue all elements of A in L.
6. Return true.
```

Using the same type of representation of a workflow as in the previous algorithm (i.e., as a graph), it is easy to see that this algorithm is also a variation of breadth-first search. In contrast to the previous algorithm, this algorithm checks by the gradual consideration of the vertices, if each path that begins with the vertex corresponding to p1 also includes the vertex corresponding to p2. If one (or more) of the paths does not include p2, the condition in the line 5 of the algorithm will eventually become true (i.e., the vertex corresponding to the process END will eventually be reached, for which $\neg\exists a$ starts_after(a, END), and therefore $A = \emptyset$).

The time complexity of this algorithm is calculated in a similar way as for the previous algorithm and is estimated for the worst case as $O(|P|*(|E|+|P|)$.

Definition 4 (A well-formed and-structure): An and-structure with the name and_id defined in the workflow <w, P, C> is well-formed if the following constraints hold:

(1) $\exists p\in P\ \exists p1\in P$ such that p = begin_and(and_id) $\wedge$ p1 = end_and(and_id);

(2) completely_reachable_from_in(end_and(and_id), begin_and(and_id), w) is true.

(3) each path of the and-structure is free of cycles.

Well-formed or- and loop-structures are defined similarly. Both and- and or-structures defined in the running example in Section 2 are well-formed.

Set of processes P(id) of an and-structure with the name id is constructed by the following procedure:

```
1. Initial settings: Let L be an empty queue and P(id) be an empty set. Put begin_and(id) into L.
2. Until L is empty perform steps 3-5.
3. Dequeue L and assign the obtained process name to the variable curr_process.
4. Identify the set A = {a | starts_after(a, curr_process)}
5. If A ≠ {end_and(p2)}, then put all the elements of A into P(id) and enqueue them in L.
6. Return P(id).
```

If the graph representation of a workflow is used as in the algorithms above, then the set of processes P(and1) of the and-structure and1 corresponds to the set of all vertices of the corresponding graph that belong to the paths beginning with the vertex representing the process begin_and(and1) and finishing with the vertex representing the process end_and(and1). The vertices that belong to the considered paths are processed in the order as in the breadth-first search algorithm, therefore the time complexity is estimated for the worst case as $O(|P|*(|E|+|P|)$.

Sets of processes for or- and loop-structures are defined in a similar way.

Now, the structural correctness property for a workflow can be introduced.

Definition 5 (A structurally correct workflow): A workflow <w, P, C> is structurally correct if the following constraints are satisfied:

(1) A workflow contains only one BEGIN (the first process), followed by one process and only one END (the last process) preceded by one process. Formally:

(a) $\exists s \in P\ \exists s1 \in P\ s=\text{BEGIN} \wedge \text{starts\_after}(s1, s) \wedge (\forall s2 \in P\ \text{starts\_after}(s2, s) \Rightarrow s2=s1)$

(b) $\exists s \in P\ \exists s1 \in P\ s=\text{END} \wedge \text{starts\_after}(s, s1) \wedge (\forall s2 \in P\ \text{starts\_after}(s, s2) \Rightarrow s2=s1)$

(c) $\forall s \in P\ (\neg \text{starts\_after}(\text{BEGIN}, s)) \wedge (\neg \text{starts\_after}(s, \text{END}))$

(2) For every process p, different from BEGIN, END, and the starting and ending processes for and- and or-structures, exactly two sequencing relations should be defined that identify the process that precedes p and the process that follows after p:

(a) $\exists s \in P\ \text{starts\_after}(p, s) \wedge (\forall s1 \in P\ \text{starts\_after}(p, s1) \Rightarrow s1=s)$

(b) $\exists s \in P\ \text{starts\_after}(s, p) \wedge (\forall s1 \in P\ \text{starts\_after}(s1, p) \Rightarrow s1=s)$

(3) Loops should be introduced only by loop-structures, no other cycles are allowed:

$\forall p1, p2 \in P\ \text{reachable\_from\_in}(p1, p2, w) \Rightarrow \neg \text{reachable\_from\_in}(p2, p1, w)$

(4) Processes, over which a synchronization constraint is specified, should not belong to the same or-structure. Formally: for each constraint from C in the form starts_with(p1, p2), finishes_with(p1, p2) or starts_during(p1, p2):

$\neg \exists id{:}\text{OR\_STRUCTURE}\ p1 \in P(id) \wedge p2 \in P(id)$

(5) All and-, or- and loop-structures in w are well-formed and each process $p \in P$ can be reached from the BEGIN, and the END can be reached from each process:

$\forall p \in P\ \text{reachable\_from\_in}(p, \text{BEGIN}, w) \wedge \text{reachable\_from\_in}(\text{END}, p, w)$

The workflow defined by the running example in Section 2 is structurally correct.

(6) Each path of the workflow is free of cycles.

Temporal correctness of the workflow

The duration of each process in a workflow may vary in actual executions and, because of the temporal ordering of processes in the workflow, each process may have different starting points in different executions. Among all starting points the earliest (est_p) and the latest starting time (lst_p) for each process p can be identified. Before describing a procedure for calculation these time parameters, let us introduce sets of relevant processes and relevant ordering relations, and an algorithm for their construction.

Let PR(p) be a set of relevant processes with respect to $p \in P$ in the workflow <w, P, C>, i.e. $PR(p) \subset P$, such that each process in PR(p) influences the starting time of p.

Let CR(p) be a set of relevant ordering relations with respect to $p \in P$ in the workflow <w, P, C>, i.e. $CR(p) \subset C$, such that each relation in CR(p) influences the starting time of p.

Sets of relevant processes and relevant ordering relations with respect to $p \in P$ in the workflow <w, P, C> are constructed by the following procedure:

```
1. Initial settings: Let L be an empty stack, and PR(p) and CR(p) be empty sets. Put p into L.
2. Repeat the steps 3-8 until L is empty, then exit.
3. Remove the element from the top of L and assign its name to the variable current_name.
4. Identify the set $C' = \{c \in C \mid \exists s \in P\ [\ c=\text{starts\_after}(\text{current\_name}, s) \vee c=\text{starts\_with}(\text{current\_name}, s) \vee c=\text{starts\_with}(s, \text{current\_name}) \vee c=\text{finishes\_with}(\text{current\_name}, s) \vee c=\text{finishes\_with}(s, \text{current\_name}) \vee c=\text{starts\_during}(\text{current\_name}, s)]\}$
5. $CR(p) = CR(p) \cup C'$
6. Identify the set $P' = \{p \in P \mid \exists c \in C'\ c= \text{starts\_after}(\text{current\_name}, p) \vee c=\text{starts\_with}(\text{current\_name}, p) \vee c=\text{starts\_with}(p, \text{current\_name}) \vee c=\text{finishes\_with}(\text{current\_name}, p) \vee c=\text{finishes\_with}(p, \text{current\_name}) \vee c=\text{starts\_during}(\text{current\_name}, p)]\}$
7. $P' = P' \setminus \{p \mid p \in PR(p) \wedge p \in P'\}$
8. If $P' \neq \{\text{BEGIN}\}$, then $PR(p) = PR(p) \cup P'$ and add all elements of P' to L in any order.
```

In the graph representation of a workflow, this procedure traces back all paths that include the vertex corresponding to the process p, starting from p. The processes represented by the vertices that belong to the considered paths are included into the set PR(p). Furthermore, the set PR(p) also includes the processes that are not represented by vertices on the considered paths, however are related to these processes by synchronisation relations. The set CR(p) consists of the ordering constraints, formulated over the processes from PR(p). Each execution of this procedure will eventually terminate, since each considered path begins with the vertex representing the process BEGIN. When BEGIN is encountered on some path, no further vertices along this path will be added to the stack L.

The time complexity of this procedure is estimated for the worst case as follows: Given the workflow <w, P, C>, the internal cycle (steps 3-8) may be repeated |P| times at most. The steps 4 and 6 take $|C|^2 \cdot |P|$ and $|C| \cdot |P|^2$ time units respectively for all executions of the cycle. The step 5 can be performed |C| time units for all executions of the cycle taken together in the worst case. The execution of the step 7 takes $|P|^2 + 2 \cdot |P|$ time units for all executions of the cycle taken together. The execution of the step 8 takes $3 \cdot |P|$ time units for all executions of the cycle taken together. The overall time complexity of the procedure is estimated for the worst case as: $|C|^2 \cdot |P| + |C| \cdot |P|^2 + |C| + |P|^2 + 8 \cdot |P| + 1$.

The procedure of calculation of est_p and lst_p

The earliest (latest) starting time of a process p in the workflow <w, P, C> is calculated under the assumption that all relevant processes in PR(p) have minimal (maximal) durations specified in their characteristics. Furthermore, $est_{BEGIN} = lst_{BEGIN} = 0$.

To calculate est_p and lst_p in the workflow <w, P, C>:

1. Identify the relevant sets PR(p) and CR(p).
2. Assume that the duration of every $p \in P$ is defined by its property min_duration (max_duration).
3. The duration of every or-structure in PR(p) is equal to the duration of its shortest (longest) branch, calculated as the sum of minimum durations of processes that belong to the branch as follows. For the or-structure with the name or_struct:

```
a) Initialization: Let ST be an empty stack, and min_duration=0, curr_duration=0.
b) Put all elements of the set {a | starts_after(a, begin_or(or_struct))} into ST in any order.
c) Until ST is not empty perform steps d)-h).
d) Remove the element from the top of ST and assign its value to the variable curr_process.
e) Until curr_process ≠ end_or(or_struct), perform steps f) and g).
f) curr_duration= curr_duration + curr_process.min_duration
g) curr_process= {a | starts_after(a, curr_process) }
h) If min_duration=0 ∨ curr_duration < min_duration, then min_duration= curr_duration
i) Return min_duration
```

4. The duration of every and-structure in PR(p) depends on its end-condition:
   a) in case the condition is 'any' the duration is determined by the shortest branch of the and-structure;
   b) in case the condition is 'all' the duration is determined by the longest branch of the and-structure;
   c) in case of a more complex condition defined by an and/or-list of processes, the duration is determined by processing the list recursively starting from the most nested parts of a condition:
      – the or-list is replaced by the process with the shortest duration in this list;
      – the and-list is replaced by the process with the longest duration in this list.

   The duration of the process obtained in the end is the duration of the and-structure.
5. To calculate the earliest starting point of the process p the duration of every loop-structure in PR(p) is counted as the sum of minimum durations of processes that belong to the loop. To calculate the latest starting point of the process p the duration of every loop-structure in PR(p) is counted as the product of the sum of maximum durations of processes that belong to the loop and the maximum amount of loop execution times.

The time complexity of this procedure can be estimated as follows: As it has been shown above, the execution of the step 1 takes $|C|^2 \cdot |P| + |C| \cdot |P|^2 + |C| + |P|^2 + 8 \cdot |P| + 1$ in the worst case. The complexity of processing of every or-structure is calculated under the assumption that each or-structure contains $m << |P(or1)|$ branches, where $P(or1)$ is the set of processes of the or-structure calculated as shown above. Then, the execution of the step 3a for an or-structure with the id $or1$ takes 2 time units. The execution of the step 3b does not take more than $m \cdot |C|$ time units. The internal cycle 3d)-h) can be repeated m times at most. The step 3d) takes 2 time units. The execution of the steps 3f)-g) takes in the worst case $O(|P(or1)|^2 \cdot |C|)$ time units. The overall time complexity for processing the or-structure with the id $or1$ is estimated as $O(|P(or1)|^2 \cdot |C|)$. The processing of an and-structure is performed using a similar sequence of steps, as for an or-structure, therefore the complexity of processing of an and-structure with the id $and1$ under the assumption that each and-structure contains $m << |P(and1)|$ branches, is also $O(|P(and1)|^2 \cdot |C|)$.

At step 5 each loop of a workflow can be considered as a sequence of processes, in which each and- and or-structure is replaced by a single composite process. When the earliest and latest starting and ending time points of the and- and or-structures included into a loop have been calculated, the corresponding temporal parameters of the whole loop structure can be identified by gradual processing of the sequence of the processes of the loop. Thus, the time complexity of processing a loop with the id $l1$ is $O(|P(l1)| \cdot |C|)$.

As one can see, the time required for the execution of each step of the procedure is polynomial in number of the processes of corresponding structures of a workflow and in the number of constraints. The time complexity of the procedure for the whole workflow $<w, P, C>$ for the worst case is not greater than $O(|P|^2 C)$.

The earliest (latest) ending time point of the process p (eet_p (let_p)) is calculated as est_p + p.min_duration (lst_p + p.max_duration). Then, the earliest (latest) creation time of the resource r (ect_r (lct_r)) produced by p are defined as: $ect_r = eet_p$ and $lct_r = let_p$, and the earliest (latest) expiration time of r (eet_r (let_r)) is calculated as: $eet_r = ect_r$ + r.expiration_duration ($let_r = lct_r$ + r.expiration_duration).

For the example on Fig.1a process Order_intake starts immediately after the zero-time process BEGIN therefore $est_{Order\_intake} = lst_{Order\_intake} = 0$ (the first time point of the execution of the workflow). As it was mentioned earlier Order_intake.min_duration = 1, Order_intake.max_duration = 2. Therefore $eet_{Order\_intake} = 1$ and $let_{Order\_intake} = 2$. If we model the new order as a resource (data) produced by this process, then $ect_{order} = eet_{Order\_intake} = 1$ and $lct_{order} = let_{Order\_intake} = 2$. If, for simplicity, we assume that the order will only be valid for 8 hours then order.expiration_duration = 8h, therefore $eet_{order} = 9$ and $let_{order} = 10$.

Synchronization relations defined in a specification may influence the starting time of processes in this specification. Moreover, some sequencing/branching/cycle relations of the specification may be in conflict with synchronization relations introduced by the designer. Let us define $[t1, t2] = [est_p, lst_p] \cap [est_s, lst_s]$ and $[t3, t4] = [eet_p, let_p] \cap [eet_s, let_s]$ for processes p and s. A *conflict* occurs in following cases: (a) if starts_with(p, s) is introduced and $[t1, t2] = \varnothing$; (b) if starts_during(p, s) is introduced and $[t1, t2] = \varnothing$; (c) if finishes_with(p, s) is introduced and $[t3, t4] = \varnothing$.

In the workflow in Fig.1b process d1 should start at the same time as process d2, starts_with(d1, d2). Let us assume that l1.min_duration = 1h, l1.max_duration = 2h, l2.min_duration = 3h, l2.max_duration = 4h and no delays are modelled in the relevant part of this workflow. Then, for d1 and d2, $[t1, t2] = [est_{d1}, lst_{d1}] \cap [est_{d2}, lst_{d2}] = [1, 2] \cap [3, 4] = \varnothing$ which indicates a conflict. If however it was modelled that l2.min_duration = 2h then $[t1, t2] = [1, 2] \cap [2, 4] \neq \varnothing$ which is a temporally correct assignment.

Definition 6 (A temporally correct workflow): A workflow $<w, P, C>$ is temporally correct in the specification M if the set of ordering relations in M is not conflicting.

If a workflow is temporally correct, starting points of processes influenced by the introduced synchronization relation are updated, using the values t1, t2, t3 and t4 defined above: (a) in case of starts_with(p, s) assign $est_p=t1$; $est_s=t1$; $lst_p=t2$; $lst_s=t2$; (b) in case of starts_during(p, s) assign $est_p=t1$ and $lst_p=t2$; (c) in case of finishes_with(p, s) assign eet_p=t3; eet_s=t3; let_s=t4; let_s=t4. Then, update the values of the earliest (latest) starting points for the processes reachable from p and for the processes reachable from s.

Condition correctness

This property concerns conditions specified for or- and loop-structures.

Definition 7 (A correct condition): A condition of the or-/loop-structure is *correct* iff the following two constraints are satisfied:

a) all values of condition variable(s) considered in the structure belong to the domain of this (these) variable(s);

b) all elements from the domain of condition variable(s) are taken into consideration in the structure;

c) the condition expression is not always true and is not always false.

Values of condition variables considered in or- and loop-structures are identified by the following procedure:

a) for or-structures if a condition is expressed by one condition variable, then the condition values are obtained from the corresponding or_branch predicates; if one of the identified values is equal to OTHER, then the domain of the condition variable should contain at least one more element different from other extracted values.

b) if the condition expression is complex, i.e. built using the Boolean connectives, then it is divided into basic expressions in form *condition variable OP value*, where OP$\in$ {=, $\neq$, <, >} and each basic expression is analysed as follows:

1. If OP is '=' or '$\neq$', then the domain of the condition variable should contain both the *value* and at least one more value different from *value*.
2. If OP is '>' or '<', then the domain of the condition variable should contain at least one element greater than *value* and at least one element smaller than *value*.

If these conditions are satisfied then the whole complex expression is checked whether it is not always true and not always false. This is performed as follows:

1. For each conjunction: if more than one expression on the same variable is present then the intersection of the sets of domain values defined by these expressions contains at least one value;
2. For each disjunction of conjunctions containing expressions over the same variables: at least one of the intersections of the sets of values defined by the expressions on the same variables should be non-empty.

Tasks and resource inter-level consistency constraints

Tasks form hierarchies based on decomposition relations between them. When building such hierarchies, consistency should be maintained by making sure the set of inter-level constraints is satisfied. Here only some examples of inter-level constraints are given (for the complete list of these and other types of constraints (the reader is referred to Popova and Sharpanskykh 2007e):

'For every and-decomposition of a task, the minimal duration of the task is at least the maximal of all minimal durations of its subtasks'.
Formally, $\forall$ t:TASK, t1:TASK, L:TASK_LIST is_decomposed_to(t, L) $\wedge$ is_in_task_list(t1, L) $\Rightarrow$ t.min_duration $\geq$ t1.min_duration

'If a task uses certain resource type as input then there exists at least one subtask in at least one and-decomposition of this task that uses this resource type.'

'For every and-decomposition of a task, if one subtask uses a resource type as input which in not an input for the composite task then there exists another subtask that produces such resource type.'

'If a task produces certain resource type as output then there exists at least one subtask in at least one and-decomposition of this task that produces this resource type.'

'For every and-decomposition of a task, if one subtask produces a resource type as output which in not an output for the composite task then there exists another subtask that uses such resource type as input.'

Functionally divisible resource or data types also form hierarchies. Due to the wide variety of possible situations, only one consistency constraint can be formulated, which should be satisfied for data types:

'If data type dt2 is a functional part of data type dt1, then the expiration duration of dt1 is at most the expiration duration of dt2.'

Physical world generic constraints

Generic constraints come from the physical world irrespective of the application domain. Here several examples of such constraints are given.

GC1: 'No role executes more than one process at the same time'

Formally:

$\forall$ r:ROLE, p1, p2:PROCESS, tp1, tp2, tp3, tp4:TIME_POINT role_performs_process(r, p1) $\wedge$ role_performs_process(r, p2) $\wedge$ est_{p1} = tp1 $\wedge$ let_{p1} = tp2 $\wedge$ est_{p2} = tp3 $\wedge$ let_{p2} = tp4
$\Rightarrow$ ((tp2 < tp3) $\vee$ (tp4 < tp1))

GC2: 'Not consumed resources become available after all processes are finished'

GC3: 'For every process that uses certain amount of a resource of some type as input, without consuming it, either at least that amount of resource of this type is available or can be shared with another process at every time point during the possible execution of the process'

GC4: 'Non-sharable resources cannot be used by more than one task at the same time'

GC5: 'For all resource types, if a resource of this type is used by a process then the process starts before all resource instances of this type expire'

4.2 Domain-specific constraints

Domain-specific constraints are imposed by the application domain in which the specific specification will be used and can be classified according to their sources.

Constraints imposed by the organisation have been chosen (e.g. by the management of the company) as necessary and need to be satisfied by any specification for the particular organisation. Such constraints can often be found in company policy documents, internal procedures descriptions, etc. For example:

'At every time point, the amount of available resources is at least a pre-specified minimum amount and/or is at most a pre-specified maximum amount.' (company policy on minimal and maximal amount of resource necessary to store)

'The duration of the execution of the workflow should not exceed a specified maximum duration.'

'Certain information types cannot be used by certain tasks.' (security/privacy)

'If at some time point the amount of a resource at a certain location is below a pre-specified minimal amount then within a pre-specified time interval this amount will become more than the minimal amount'. (company policy on replenishing a resource on time)

Constraints coming from external parties are enforced by an external party such as the society or the government and can contain rules about working hours, safety procedures, emissions, and so on. Sources for such constraints can be laws, regulations, agreements, etc. Examples of specific constraints of this group that might be relevant in some situations can be:

'Specific type of information should be used within a pre-specified time interval after it is created.' (e.g. news items should be communicated only when they are recent)

'A driver should not drive more than 6 hours per day.'

Constraints of the physical world come from the physical world with respect to the specific application domain and should be satisfied by any specification in this domain. This is in contrast to the generic physical constraints which should be satisfied by any specification irrespective of the application domain. For example, 'there is always a break of at least 15 minutes between two consecutive lectures' (follows from the limitation of most humans to stay concentrated on a lecture for a very long time) or 'a location cannot store more than a certain prespecified amount of resource' (storage capacity).

For all these types of constraints there are predefined templates, which can be selected and customized by the designer by assigning specific values to the parameters of the template without the need to express logical formulae. Examples of such templates with their parameters in brackets are:

DC1(p1:PROCESS, p2:PROCESS, d:VALUE): 'If the same agent executes both processes, then there is a delay of duration at least d between the end of the first process and the beginning of the second one' (can also be formulated for a specific agent as additional parameter)

Formally:

$\forall$ r1, r2:ROLE, tp1, tp2, tp3, tp4:TIME_POINT: ($\exists$a:AGENT: (agent_plays_role(a,r1) $\wedge$ agent_plays_role(a,r2) $\wedge$ role_performs_process(r1,p1) $\wedge$ role_performs_process(r2,p2) $\wedge$ est_{p1} = tp1 $\wedge$ let_{p1} = tp2 $\wedge$ est_{p2} = tp3 $\wedge$ let_{p2} = tp4)
$\Rightarrow$ ((tp2 < tp3) $\wedge$ (tp3 – tp2 $\geq$d)) $\vee$ ((tp4 < tp1) $\wedge$ (tp1 – tp4 $\geq$ d)))

DC2(rt:RESOURCE_TYPE, min_am:VALUE): 'At every time point the amount of resource of type rt available is at least min_am amount'

DC3(t:TASK, dr:VALUE): 'For every agent performing processes of this task the sum of the durations of these processes should not exceed dr'

DC4(rt:RESOURCE_TYPE, min_am:VALUE): 'At the end of the workflow there is at least min_am amount of resource of type rt'

DC5(d:VALUE): 'The duration between the first and the last tasks is at most d'

DC6(rt:RESOURCE_TYPE): 'Data of type rt is created by a task during the workflow'

DC7(rt:RESOURCE_TYPE, t:TASK): 'Resource of type rt cannot be used by task t'

DC8(p1, p2:PROCESS): 'No agent can play roles that perform p1 and p2'

DC9(a:AGENT, rt:RESOURCE_TYPE): 'Agent a has no access to resource of type rt'

DC10(r:ROLE, rt:RESOURCE_TYPE): 'Role r has no access to resource of type rt'

DC11(rt1, rt2:RESOURCE_TYPE): 'Resources of types rt1 and rt2 can not be used together for the same task'

DC12(t:TASK, dr:VALUE): 'For every agent performing processes of this task the sum of the durations of these processes should not exceed dr'

DC13(l:LOCATION, rt:RESOURCE_TYPE, m:VALUE): 'At every time point the amount of resource of type rt at location l is at most m'

DC14(l:LOCATION, rt:RESOURCE_TYPE, m:VALUE, dr:VALUE): At every time point t, if the amount of rt at location l is less than m then there exists a future time point t1$\leq$ t + dr at which the amount of rt at l is at least m'

DC15(ag_list: AGENT_LIST, dr:VALUE): 'The amount of working hours of each agent from the ag_list should not exceed dr'

DC16(a: AGENT, t: TASK): 'Agent a should not be allocated to task t'

For the case study, a number of relevant domain-specific constraints can be formulated. The first example is a special case of the constraint DC15, which is imposed by the labour legislation (a constraint coming from an external party):

DC15(DRIVERS, 6): The amount of driving hours for each agent driver should not exceed 6 hours per day.

Another agent-related constraint is a special case of the constraint DC16, comes from the organisation and prevents assigning some drivers for certain types of deliveries (e.g. because of the employee's preferences or some geographical factors):

DC16(ag5, DeliveryAB): The agent ag5 should not be allocated to any delivery of type deliveryAB.

One more constraint imposed by the organisation describes the resource integrity before and after each delivery:

DC17: For each delivery the amount of resource being loaded on a truck should be equal to the resource amount being unloaded from the truck after the delivery has finished.

Another constraint comes from the physical world and describes the conditions of loading of resources of type rt2 on a truck of type tr2 determined by the weight limitations:

DC18(tr2, rt2, 0.7): Maximum 0.7 capacity units of a truck of type tr2 can be used for a delivery of the resource type rt2, whereas 0.3 remaining units should be left empty.

For shared deliveries of resources of type rt2 with some other resource types, special regulations are formulated as constraints.

The following constraint comes from the customer - Company 1 and is determined by its technological process:

DC19: The difference between the finishing time points of the deliveries d1 and d4 should be less than 2 hours.

The last example identifies the constraint that comes from the external parties (Company 1 and its customers):

DC20: For all time points the amount of resources of type rt5 at the base A should be greater than 25.

5 Correctness verification of a process-oriented specification

The verification of the correctness of a process-oriented specification is performed during or at the end of the design of the specification, depending on the verified types of constraints. In particular, some domain-specific constraints might not (yet) be satisfied for incomplete specifications. The designer can choose the moment when they should be checked. The syntactical check of the specification for a specification and the verification of generic constraints are performed at every design step. Note that often only the set of relevant generic constraints is verified. This set is identified based on the type of the change made by the designer in the specification. For example, if the minimal or maximal duration of a task or a decomposition relation between tasks is changed, then the corresponding task inter-level consistency constraint(s) expressed over these tasks should be checked. Changes in resource structures are dealt with similarly. If the designer changes the set of ordering relations or the (minimum or maximum) duration of a task of which an existing process is an instance, then first the structural constraints of the workflow are checked, and after that physical world and domain specific constraints.

For all other types of changes, the set of constraints that should be checked is formed from physical world and domain specific constrains from T_{PR} expressed over objects involved into relations affected by the change. For the checking, it is assumed that each process p in the workflow <w, P, C> can be active (executed) at any time point during the interval $[est_p, let_p]$. Therefore, it will be checked with respect to the whole interval $[est_p, let_p]$, even though the actual execution of p may take less time. Thus all possible intervals of the execution of p are taken into account. If the constraint is not satisfied in some possible execution, it can be discovered without checking all executions separately. This dedicated verification is computationally much cheaper than the general-purpose state-based analysis of all possible executions of a specification, i.e., by trying one by one all possible combinations of durations of processes (e.g. by model checking (Clarke, Grumberg and Peled. 2000)) however still allows establishing correctness of the model.

Here three example algorithms are given for checking the satisfaction of constraints GC1, DC1 and GC3 defined in Section 4. The first and second algorithms are typical for the verification of constraints over processes, roles and agents, and the third one illustrates the verification of constraints over resource amounts related to processes.

Algorithm for verification of GC1

```
1. Let L be an empty queue and N be an empty set. Enqueue in L all roles defined in the specification.
2. Until L is empty perform steps 3-5.
3. Dequeue L and assign the obtained value to the variable curr_role.
4. Put into N all processes assigned to curr_role in the specification.
5. For each processes $p1,p2 \in N$ determine if they can be executed at the same time:
   if $\neg((let_{p1} < est_{p2}) \vee (let_{p2} < est_{p1}))$, then GC1 is not satisfied, exit; else empty N
6. GC1 is satisfied.
```

The proof of the correctness of this algorithm is straightforward. The algorithm processes one by one all pairs of the processes allocated to a role and identifies if the processes in each pair can be executed at the same time. Two arbitrary chosen processes p1 and p2 cannot be executed simultaneously when either $let_{p1} < est_{p2}$ or $let_{p2} < est_{p1}$. Both these conditions are checked at step 5. Thus, if there is a possibility for some of the processes allocated to a role can be executed at the same time, it will be discovered by the algorithm.

The time complexity of this algorithm is estimated as follows: The internal cycle that includes the steps 3-5 is performed $|ROLE|$ times, where ROLE is the set of all roles defined in the organizational specification. In the worst case, all executions of the step 4 together take $|CS|\cdot|ROLE| + |P|$ time units, assuming that each process of the workflow $<w, P, C>$ is allocated to one role and CS is the complete set of constraints defined for the organization. The execution time of the step 5 is calculated as $|P|!/2\cdot(|P|-2)!$ or $(|P|^2-|P|)/2$. Thus, the overall time complexity of the algorithm is estimated as $O(|ROLE|\cdot(|P|^2 + |CS|))$.

Algorithm for verification of DC1

```
1. Determine the roles r1 and r2 that perform the processes p1 and p2:
     role_performs_process(r1,p1) $\wedge$ role_performs_process(r2,p2)
2. If the identified roles are allocated to different agents ($\neg\exists$a:AGENT agent_plays_role(a,r1) $\wedge$
     agent_plays_role(a,r2)), then DC1 is satisfied, exit.
3. Determine if there is a delay of duration at least d between the processes p1 and p2:
   if (p1.est>p2.est $\wedge$ p1.let+d $\leq$ p2.est) $\vee$ (p2.est>p1.est $\wedge$ p2.let+d $\leq$ p1.est), then DC1 is satisfied;
.  otherwise DC1 is not satisfied.
```

The proof of the correctness of the algorithm is straightforward and follows directly from the structure of the formula representing the constraint.

The time complexity of the algorithms is estimated as $O(|CS|)$, where CS is the complete set of constraints defined for the organization.

Before describing an algorithm for checking GC3 let us introduce a definition of a workflow segment and a labelling procedure for workflow segments.

Definition 7 (A workflow segment): A segment SG of the workflow $<w, P, C>$ is a set of processes from P ordered by C that are executed under the same set of values of or-conditions from w. This set of values is dynamically formed from the values of conditions of or-structures, from which the processes of SG can be reached. The set SEGMENTS contains all segments of the workflow.

Each segment has a label, assigned according to the following rules:

- the segment that contains processes that are executed independent of any condition values has the label ‘1’.
- the label for a segment that corresponds to a branch of a certain or-structure is formed from three parts that follow each other:
  (1) the prefix defined by the label L of the segment, to which the beginning process of the or-structure belongs;
  (2) the index of the branch in the or-structure obtained incrementally starting from 1;

(3) the sequential index of the or-structure in the segment with the label L, put in square brackets.

For example, the process Order_intake from the example introduced in Section 2 belongs to the segment labelled by '1', whereas the process Order_delivery2 belongs to the segment labelled by 1.2[1].

Further the algorithm is given for checking the satisfaction of GC3 with respect to the process p and the resource r. In this algorithm the following notations are used:

res_produced_by(r, p, am) for is_instance_of(p, t) $\wedge$ task_produces(t, r, am)
res_used_by(r, p, am) for is_instance_of(p, t) $\wedge$ task_uses(t, r, am)
res_consumed_by(r, p, am) for is_instance_of(p, t) $\wedge$ task_consumes(t, r, am)

Algorithm for verification of GC3

1. Identify the set of time points TP within the duration of p ($est_p \le t < let_p$), at which the amount of some resource(s) of type r changes (i.e. time points at which other processes that use/consume/produce a resource of type r may start or finish). For every time point $t \in TP$ perform steps 2-7.

2. Determine the set RS of segments that contain finished before or executed simultaneously with p processes, which execution may influence the amount of resources of type r:

$RS = \{s \in SEGMENTS \mid \exists a\ a \in s \wedge [let_a < t \wedge [\ am1 > 0 \vee am3 > 0] \ ] \vee [\ let_a > t \wedge est_a \le t \wedge [\ am1 > 0 \vee am2 > 0 \vee am3 > 0\ ]$,

where am1, am2, and am3 are specified in res_consumed_by(r, a, am1), res_used_by(r, a, am2) and res_produced_by(r, a, am3).

3. The labels of segments in RS that correspond to the branches belonging to the same or-structure are grouped.

4. The n-ary Cartesian product of all obtained groups is generated (n is the number of groups): $g_1 \times \ldots \times g_n$. Each tuple in the obtained product set corresponds to a possible combination of segments in the workflow. In such a way all possible execution of processes in the workflow, which use/produce/consume r and have latest ending time $\le let_p$ are considered.

5. For every tuple in the product set identify the set of processes PS that corresponds to the tuple. If two or more processes from the same segment related by a sequencing relation may be executed at the same time in different instances of the workflow, replace PS by a number of sets, each of which will contain only one from these processes.

6. For every set of processes PS corresponding to the tuple, identify the set of resources RPS of type r produced by processes in PS.

7. For every process $a \in PS$ that consumes some amount of the resource of type r identify if this amount of not expired resource(s) from RPS is available. Update RPS after every iteration:

7.1 Initial settings: Let temp_amount = am, where am is defined by res_consumed_by(r, a, am)

7.2 Until temp_amount > 0 and RPS is not empty perform 7.3 and 7.4

7.3 Identify resource $res \in RPS$ with the smallest earliest expiration time, which did not expire yet. It is assumed that such resource will be used first by a.

7.4 If res.amount $\ge$ temp_amount,
  then update the amount of the resource as res.amount = res.amount - temp_amount and set temp_amount = 0.
  else update temp_amount = temp_amount - res.amount and delete res from the RPS.

7.5 If temp_amount > 0,
  then **GC3 is not satisfied** with respect to the process p and resource type r, exit.

8. For every process $a \in PS$ that uses a certain amount of the resource of type r at time point t identify, if this amount of not expired resource(s) from RPS is available. Update RPS after every iteration:

8.1 Initial settings: Let temp_amount= 0.

8.2 From the specification identify process lists that may share a resource of type r and that contain as least one process from PS. For each process $a \in PS$ at most one list will be chosen from the identified lists. Furthermore, any list that contains a may be selected, since the choice of the list does not influence the amount of available resources in RPS.

8.3 For every chosen process list L update temp_amount = temp_amount + am, where am is defined in res_used_by(r, s, am) and s is some process from L. Delete all processes that belong to L from PS.
8.4 For every process $a \in PS$ update temp_amount = temp_amount + am, where am is defined in res_used(r, a, am). Then perform the same calculations as on steps 7.2-7.5.
9. **GP3 is satisfied** with respect to the process p and resource type r.

The informal explanation and the proof of correctness of the algorithm

For the following explanation the notation avail_res_amount (t, r, am) will be used, meaning that at the time point t the total amount of available resources (i.e., not used and not consumed) of the type r equals am.

To check the satisfaction of the constraint GP3 for process p that requires the amount am_req of the resource type r, it is needed to verify that:

1) avail_res_amount (est_p, r, am) $\wedge$ am $\geq$ am_req (i.e., the available resource amount of type r is sufficient for the execution of the process p) or $\exists$p1:PROCESS $\exists$l:PROCESS_LIST $est_p > est_{p1} \wedge let_{p1} \geq est_p \wedge$ resource_sharable(r, l) $\wedge$ is_in_list(p1, l) $\wedge$ is_in_list(p, l) (meaning that there exists another process p1 being executed at est_p, with which p may share the resource type r)
2) for all $t \in (est_p, let_p]$, at which the amount of available resources of the type r changes: avail_res_amount (t, r, am) $\wedge$ am $\geq$ am_req or $\exists$p1:PROCESS $\exists$l:PROCESS_LIST $t > est_{p1} \wedge let_{p1} \geq t \wedge$ resource_sharable(r, l) $\wedge$ is_in_list(p1, l) $\wedge$ is_in_list(p, l)

In the following also this notation is used:

res_produced_by(r, p, am) for is_instance_of(p, t) $\wedge$ task_produces(t, r, am)
res_used_by(r, p, am) for is_instance_of(p, t) $\wedge$ task_uses(t, r, am)
res_consumed_by(r, p, am) for is_instance_of(p, t) $\wedge$ task_consumes(t, r, am)

The amount of the resource type r in (2) may change due to the following events: (a) beginning of some process that uses/consumes the resource amount amt of type r, in this case if no sharing possibilities exist for the process, then avail_res_amount (t_b, r, am - amt), where t_b is the beginning time point of the process and am is the available resource amount of type r at the time point before t_b; (b) finishing of some process that uses the resource amount amt1 or produces the resource amount amt2 of the type r, in this case avail_res_amount (t_e, r, am + amt1) if no other processes share the resource being used or avail_res_amount (t_b, r, am + amt2); here t_e is the finishing time point of the process and am is the available resource amount of type r at the time point before t_e. In the proposed modelling framework it is assumed that during the execution of a process the amount of the resources that it uses does not change. Therefore, the set of time points TP identified at the first step of the algorithm is defined as: {t $\in$ TIME | $\exists$p1: PROCESS $\exists$am1, am2, am3:VALUE (res_produced_by(r, p1, am1) $\vee$ res_produced_by(r, p1, am2) $\vee$ res_produced_by(r, p1, am3)) $\wedge$ $est_p \leq est_{p1} \leq let_p \wedge t = est_{p1}$} $\cup$ {t $\in$ TIME | $\exists$p1: PROCESS $\exists$am1, am2, am3:VALUE (res_produced_by(r, p1, am1) $\vee$ res_produced_by(r, p1, am2) $\vee$ res_produced_by(r, p1, am3)) $\wedge$ $est_p \leq let_{p1} \leq let_p$ $\wedge$ $t = let_{p1}$}.

For all $t \in TP$ the value am in avail_res_amount (t, r, am) is calculated based on the amounts of resources of the type r produced and consumed before t, and based on the resource amounts of type r consumed, produced and used at t. The sets of processes that produce, consume and use these resources may be different for different executions of the workflow. Alternative execution paths of the workflow are formed from different executions of the or-structures of the workflow. Furthermore, processes that form these paths may have different duration in different executions. To guarantee the satisfaction of the constraints (1) and (2) for every possible execution of the process p, all execution paths of the workflow that differ in the resource amount am in avail_res_amount (t, r, am) at least at one time point $t \in TP$, should be checked.

First such paths should be identified. This is performed by the steps 2-6 of the algorithm. An important, though obvious, observation here is that different branches of the same or-structure cannot be executed at the same time, whereas different branches of different or-structures often can be executed simultaneously in various combinations, thus forming different execution paths of the workflow. To identify these paths all branches of the or-structures of the workflow that form segments containing finished before or executed simultaneously with p processes, which execution may influence the amount of resources of type r, are labelled using the procedure described above. Further, the introduced labels of the segments are put into the set RS (step 2). Note that a segment may contain (nested) and-structures. The processes of segments, with labels that have the same prefix and the same sequential index cannot be executed simultaneously. Such segments are combined into groups at step 3. Thus, each group contains alternative partial execution paths. Then, the Cartesian product of all obtained groups is determined at step 4, thus, defining all possible execution paths of the workflow for the time period $[0, let_p]$ that contain processes that use and/or consume and/or produce resources of type r.

At step 5 of the algorithm for each tuple in the product the set of processes PS is identified that belong to the segments in the tuple and that use and/or consume and/or produce resources of type r. If two or more processes from the same segment related by a sequencing relation may be executed at the same time in different instances of the workflow, i.e., for processes p1 and p2: $[est_{p1}, let_{p1}] \cap [est_{p2}, let_{p2}] \neq \varnothing$ PS is replaced by a number of sets, each of which will contain only one from these processes. This is needed because at most one from the processes related by a sequencing relation can be executed at any time point in any instance of the workflow.

Then, each execution path is processed separately. For each path the set of resources of the type r produced by the processes of the path during the interval $[0, let_p]$ is determined (step 6). Then, for each process of the path that consumes some amount amt of the resource type r, the value of the available resources(s) of the type r with the earliest expiration time decreases by amt at the earliest starting time point of the process (step 7). If no such resource(s) is (are) available, the constraint is not satisfied. After that it is checked if the remaining resource amounts related to time points suffice for the execution of the processes of the path that use some amounts of the resource type r and which execution interval has a non-empty intersection with the interval $[est_p, let_p]$ (step 8). Now let us consider the steps 6-8 in detail.

First, at step 6 the set RPS of resources of the type r produced by the processes of the path is identified. These resources are used and consumed by other processes of the path. Each resource is allowed to be used or consumed before its expiration. Furthermore, it is assumed that resources with the earliest expiration time will be used or consumed first by processes of the path.

Then, at step 7 for each process of the path that consumes some amount of the resource type r it is identified if this amount of not expired resource(s) from RPS is available. Each time when a process that consumes the amount amt of the resource type r is identified, the amount of the resource r1 with the earliest expiration duration decreases: r1.amount = r1.amount - amt. In case amt > r1.amount, then the difference amount amt-r1.amount is taken from another resource(s) that has (have) the earliest expiration time after r. If no such resource is available, the constraint GP3 is not satisfied.

Finally, at step 8 for all time points $t \in TP$ for every process of the path that uses a certain amount of the resource type r it is identified if this amount of not expired resource(s) from RPS is available. Note that only the processes being executed at time points $t \in TP$ should be checked, since all processes that finished before est_p also released the resources that they had used. Note that some processes from PS may share the same resource amount of the type r if they belong to the same process list l such that resource_sharable(r, l). For each process $a \in PS$ at most one list is chosen. Furthermore, any list that contains a may be selected, since the choice of the list does not influence the amount of available resources in RPS. When the total used resource amount is calculated at each time point $t \in TP$, the resource amount of the type r shared by a list of processes being executed at t is taken to be equal to the resource amount of the type r used by any process from this list (step 8.3). To the obtained amount is added the sum of all amounts of resources of type r used by the processes from PS being executed at t that cannot share resource type r (step 8.4). Then, it is determined if the obtained total used resource amount does not exceed the total available amount of resources of type r at the time point t calculated by the execution of the steps 7 and 8. If it does not exceed, the constraint GC3 is satisfied, otherwise, GC3 is not satisfied.

In order to reduce the number of tuples generated at step 4 and to improve the computational properties, the introduced verification algorithm is extended with tuple reduction steps (the extended version of the algorithm can be found in (Popova and Sharpanskykh 2007e)). This algorithm performs the local elimination of segments within each group that do not contribute to the worst case situation. More specifically, in each group all segments are eliminated, except for the segment that uses the largest amount of resources of type r. This allows reducing the number of execution paths that have to be checked significantly.

Some discussions about the time complexity for the considered algorithm are given below.

The execution of the step 1 takes $O(|P|)$, where P is the set of processes of the workflow $<w, P, C>$. The execution time of the step 2 depends on the number of or-structures of the workflow and the number of branches in each or-structure and takes $O(b)$, b is the overall number of all branches in all or-structures of the workflow. The execution of the step 3 takes also not more than $O(b)$. The time complexity of the execution of the step 4 is dependant on the number of or-structures of the workflow, on the number of branches in each or-structure and on the level of nesting of the or-structures. For the worst case, in a workflow that contains k or-structures with the overall number of branches b, the execution of step 4 takes $(b/k)^k$ time points. For workflows that have nested or-structures, the execution time required for the step 4 is less than $(b/k)^k$, because of workflows with such structures have a smaller number of segments. The steps 5-9 may be repeated $(b/k)^k$ times in the worst case. Each execution of the step 5 in the worst case may take $O(|P|\cdot|C|)$. Each execution of the step 6 in the worst case take not more than $O(|P|)$. Each execution of the steps 7 and 8 may take not more than $O(|P|^2)$. Thus, from the performed complexity analysis it follows that the time complexity of the proposed algorithm is polynomial in the number of the processes and the ordering constraints of a workflow, however exponential in the number of or-structures and segments formed based on these or-structures. An approach to decrease the number of segments, and thus, to decrease the complexity of the proposed algorithm significantly, is briefly described above; more specific details of this extension can be found in (Popova and Sharpanskykh 2007e).

The proposed verification algorithm is computationally much cheaper than standard model checking procedures, which time complexity for analysing specifications in the proposed process-oriented language would be exponential, not only in the number of or-structures and in the number of branches of these structures, but also in the number of the processes in these branches. Furthermore, during model checking, the set of time points TP' for which the satisfaction of the constraint GC3 for the process p should be checked always includes every time point from the interval $[est_p, let_p]$. The set TP' contains in most cases much more elements in comparison to the set TP, obtained by an execution of the proposed algorithm.

6 Related literature

Different aspects of process-oriented modelling and analysis have been investigated in different areas, such as enterprise modelling, artificial intelligence, operations research and others. The following aspects of the process-oriented modelling are usually considered in these areas: functional, behavioural, information-, resource- and organisation-related. Let us briefly discuss each of these aspects.

The functional aspect is usually represented by static task structures (Fox 1992, Menzel and Mayer 1998, Malone, Crowston and Herman 2003), in which characteristics of tasks (activities) (such as input, output and function) and relations between them are defined. Task structures usually serve as templates for process execution structures. To reduce the complexity and to provide means for process-oriented modelling at different levels of abstraction, tasks are structured in hierarchies built on refinement relations as in (Fox 1992, Malone, Crowston and Herman 2003) and in the framework proposed here. Furthermore, in the approach proposed here, special verification means based on constraints are provided, in order to guarantee the correctness of built hierarchical structures, which are absent in other mentioned frameworks.

The behavioural aspect is realized by process execution structures, which are often called control flows. Currently a great variety of languages and frameworks for process-oriented modelling exist. Some of the proposed languages are purely graphical (Menzel and Mayer 1998, Yang and Zhang 2003), whereas others have formal foundations (Fox 1992, van der Aalst 1998). Although process-oriented languages differ in their specification means and expressivity, many of them realize similar control patterns of process execution (or of workflows). In (van der Aalst *et al*. 2003) an extensive overview and a classification of different types of workflow patterns is presented. The graphical process specification languages such as BPMN, BPML, UEML, YAWL (van der Aalst and ter Hofstede 2005) realize these templates to a different extent. Also the process-oriented language presented in this paper supports the most essential and commonly used templates, which are identified in the introduction of the language. Furthermore, more specific templates described in (van der Aalst *et al*. 2003) not addressed in this paper can easily be implemented by an extension of the introduced language. The proposed language allows temporal numerical expressivity for the specification of control flows (e.g. durations of processes and delays, real time constraints), which is deficient in a number of other frameworks (e.g. BPMN, CIMOSA (1993), IDEF3 (Menzel and Mayer 1998)). Furthermore, many of the existing process modelling languages are not formally grounded, and, therefore, can not be used for formal analysis. Although some frameworks propose automated techniques for analysis of process-oriented specification even without properly defined semantics of the modelling language, still the results of such analysis are not completely reliable. Furthermore, the behaviour of such process-oriented specifications may be unpredictable. For example, in the ARIS framework (Scheer and Nuettgens 2000) the control flows are modelled using informal Event-driven Process Chains (EPCs), which limits the possibilities for analysis and its reliability. Similar observations can be made with respect to the frameworks described in (CIMOSA 1993, Yang and Zhang 2003).

However, also a number of formal methods have been applied for modelling and analysing of control flows: process algebra, Petri nets and their extensions and modifications (such as Workflow Nets), and different types of logics.

In (Singh 1996) process algebra is used to represent ordering constraints on processes, however it lacks the expressivity to represent global constraints on processes and (real) numbers (i.e. durations).

In (Arbab 2004) a language for the composition of software components is described, which can be used for coordination of concurrent processes. This language is a channel-based coordination model in which complex coordinators (or connectors) are built compositionally from simpler coordinators. Verification possibilities of models in this language are limited to ordering constraints considered in this paper.

Petri-nets and their modifications (Peterson 1981) have been extensively used for formal modelling and analysis of workflows (van der Aalst 1997, van der Aalst 1998). This formalism is useful for specifying ordering constraints, however it is difficult to express global constraints over multiple objects, characteristics and relations of the organisation (e.g. many physical world and domain-specific constraints considered in this paper) using Petri Nets. One such example would be the constraint ‘the breaks between the processes allocated to some role should be at least one hour’. In (Adam, Atluri and Huang 1998) it is shown that one can manually construct a Petri net that satisfies a certain set of global constraints, however, the resulting representation does not include the information about the constraints themselves. Furthermore, Petri Nets are difficult to use by non-professionals, whereas the introduced approach proposes an intuitive, close to the natural, predicate language, which can be represented graphically.

Different types of logics have been used for modelling and analysis of control flows. One of them is the propositional temporal logic (Attie *et al.* 1993). Although temporal logic is highly suitable for specifying ordering constraints, it has a number of expressivity limitations, e.g. numbers cannot be expressed, in most cases variables and composite structures (such as predicates) cannot be used. Furthermore, most of the existing general-purpose algorithms for checking properties expressed as temporal logic formulae on flow specifications (e.g. model checking (Clarke, Grumberg and Peled 2000)) have a high computational cost.

The first-order predicate logic has been used for designing ontologically rich process-oriented specifications in (Fox 1992). However, analysis issues of such specifications are not addressed. Different variations of transaction logics (Bonner and Kifer 1998) have been applied for modelling, executing (scheduling) and analysing control flows. Originally, the transition logic has been developed as an extension of the first-order logic for the representation of state changes in databases and logical programs. Therefore, although it allows designing correct flows and performing effective analysis, it still lacks the ontological expressivity to represent the variety of objects and relations that exist in organisations.

In (Bi and Zhao 2004) propositional logic is applied for workflow verification. The proposed verification approach is based on process inference, which reduces the logical representation of a workflow model to a simpler logic-based representation. Using this approach the following types of workflow abnormalities can be identified: deadlocks, lack of synchronization, activities without termination or without activation and infinite cycles. However, because of the limited expressivity of propositional logic only ordering constraints considered in this paper can be expressed and verified using this method.

A number of dedicated formal techniques have been developed for checking temporal constraints on processes in workflows (Bettini, Wang, and Jajodia 2002, Lu *et al.* 2006).

Information- and resource-related aspects are modelled in a number of informal and semi-formal frameworks as separate flows and in relation to processes (BPML, BPMN, Fox 1992, Barkaoui and Petrucci 1998, Menzel and Mayer 1998, Yang and Chen 2004). In particular, a number of workflow resource patterns are introduced in (Russell *et al.* 2004) that aim to capture the various ways in which resources are represented and utilized in workflows. Whereas these patterns provide an aggregated view on the resource allocation that includes authority-related aspects and the characteristics of roles, the framework proposed in this paper distinguishes different types of organisational aspects into separate views and establishes relations between these views. Thus, the models of resources and their relations to tasks and processes are specified separately from the role- and authority-related aspects. However, if needed, particular domain-specific constraints can be specified that are based on information from different organisational views to describe different modes of creation/use of resources.

Furthermore, not many frameworks address the verification aspects of resource-based models. Often in formal analysis only a very limited number of aspects of resources and information related to process-oriented models are addressed (Barkaoui and Petrucci 1998, Li, Yang and Chen 2004). In the proposed framework resources are characterized by a type, an expiration time, an amount that may be used, consumed or produced by a process. Furthermore, a process may share a certain amount of some resource with other process(es) and a physical replacement of resources can be specified. Our representation of shared resources is different from (Barkaoui and Petrucci 1998) in three aspects: (1) a certain specified amount of the resource can be shared among processes at the same time; (2) sets of processes that are allowed to share a certain resource can be predefined; (3) different amounts of the same resource can be shared (at the same time). Information is treated as a special kind of a resource. The algorithm for verification of the resource related constraints takes into account all characteristics and modes of use of resources at the same time. To our knowledge there exist no other frameworks that represent and verify all the specified resource characteristics and dependences simultaneously.

Organisational aspects are modelled in many frameworks from the area of artificial intelligence (Horling and Lesser 2005) and enterprise systems (CIMOSA 1993, Bernus et al. 1998, Scheer and Nuettgens 2000). Often such models specify (different types of) relations between tasks (processes) and agents (roles, actors). However, a specification of processes (tasks) and relations between them is often kept simple to enable computationally effective agent-(role-) oriented analysis. The proposed framework establishes relations between concepts from the process-oriented view and the concepts form the organisation-oriented view (e.g. roles and agents), as well as the concepts from the performance-oriented view (e.g. goals, performance-indicators), while keeping the complete ontological expressiveness of each of the views. By doing this different sophisticated methods of analysis across views can be performed.

7 Conclusions

This paper introduces a formal framework for process-oriented modelling and analysis. The framework is based on an expressive sorted predicate logic language L_{PR}, which allows specifying a wide range of concepts and relations of the process-oriented view on organisations. In particular, L_{PR} provides means for the detailed modelling of resources, including different modes of sharing that are distinct from other existing modelling frameworks. Moreover, since the process-oriented view is related to other organisational views, process-oriented specifications may include relations between tasks, processes, resources and other organisational concepts (e.g. roles, goals, agents). Furthermore, L_{PR} is used for the specification of different types of organisational constraints that should be satisfied by process-oriented specifications. These constraints may express both local (i.e. related to individual objects) and global (i.e. related to multiple objects) properties of an organisation. Also, the paper proposes efficient dedicated analysis techniques for checking the correctness of process-oriented specifications with respect to different sets of constraints, all of which are implemented. The proposed verification algorithms are more (time- and resource-) efficient than general-purpose logical analysis techniques (e.g. model checking and theorem proving), as they do not require checking of properties along all the possible execution paths of process-oriented specifications. To our knowledge there exist no other frameworks that allow the simultaneous verification of different (interdependent) types of constraints based on the extensive set of concepts and relations as can be found in L_{PR}.

The proposed approach differs from constraint satisfaction methods developed in (Tsang 1993). Whereas the main focus of the latter techniques is on finding (optimal) solutions given a consistent and stable set of constraints, our approach addresses both design of a specification and of constraints that should be satisfied by the specification. The designer is free to vary both the specification and the constraint specifications. The designer is supported by the automated tool that allows identifying sources of inconsistencies and mistakes both in the specification and the constraint specifications.

The developed approach allows scalability by performing compositional design of specifications. Using task hierarchies specifications can be built at different levels of abstraction. General constraints defined for high level processes are refined into more specific ones that should be satisfied by processes of lower levels. In such a way, to decrease complexity, specifications of different abstraction levels can be analysed separately keeping relations with each other through task hierarchies and the constraint refinement.

Furthermore, although the introduced predicate language is very intuitive, still a graphical interface for creating and changing specifications would be of help. Such an interface is currently being developed. However, graphics would provide only a little help in the specification of constraints. For this property templates can be used as shown in this paper. The current version of the software related to this paper can be downloaded from http://www.few.vu.nl/~sharp/workflow_checker.rar.

The formal methods discussed in the paper are dedicated for the verification of process-oriented specifications, however, also a number of formal techniques for the analysis of actual execution based on the introduced process-oriented specification, have been developed. These techniques are discussed in (Popova and Sharpanskykh 2007b).

References

Arbab, F., Reo: a Channel- based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 2004, 14(3).

Allen, J.F., Maintaining knowledge about temporal intervals. *Communications of the ACM*, 1983, 26, pp. 832–843.

Adam, N.R., Atluri, V., Huang, W.-K., Modeling and analysis of workflows using Petri Nets. *Journal of Intelligent Information Systems*, 1998, 10, pp. 131–158.

Attie, P., Singh, M., Sheth, A., Rusinkiewicz, M., Specifying and enforcing intertask dependencies. In *Proceedings of the 19th VLDB Conference*, 1993.

Barkaoui, K., Petrucci L., Structural analysis of workflow nets with shared resources. In *Workflow Management: Net-based Concepts, Models, Techniques and Tools*, edited by W.M.P. van der Aalst, G. De Michelis, C. A. Ellis, 1998, 98, pp. 82–95.

Bernus, et al. (eds.), *Handbook on architectures of information systems*, Heidelberg, 1998 (Springer-Verlag).

Bettini, C., Wang, X., Jajodia, S., Temporal reasoning in workflow systems. *Distributed and Parallel Databases,* 2002, 11(3), pp. 269–306.

Bi, H.H., Zhao, J.L., Applying Propositional Logic to Workflow Verification. In Information Technology and Management, 2004, 5, pp.293-318.

Bonner, A. J., Kifer, M., A logic for programming database transactions. In *Logics for Databases and Information Systems*, edited by J. Chomicki, G. Saake, pp. 117–166, 1998 (Kluwer).

Broek, E., Jonker, C., Sharpanskykh, A., Treur, J., and Yolum, P., Formal modeling and analysis of organizations. In *Coordination, Organization, Institutions and Norms in Agent Systems I*, LNAI 3913, 2006 (Springer).

Business Process Modeling Language (BPML).http://www.bpmi.org.

Business Process Modeling Notation (BPMN) http://www.bpmn.org/

Cormen, T.H., Leiserson, C. E., Rivest, R. L., Stein, C., Introduction to Algorithms, 2001 (MIT Press)

CIMOSA – Open system architecture for CIM, ESPRIT Consortium AMICE, 1993 (Springer-Verlag, Berlin).

Clarke, E.M., Grumberg, O., Peled, D.A., *Model checking*, 2000 (MIT Press).

Fox, M.S., The TOVE project: towards a common-sense model of the enterprise. In *Proceedings of ICIEMT'92*, edited by C.J. Petrie Jr., pp. 310–319, 1992 (MIT Press).

Horling, B., and Lesser, V., A Survey of multi-agent organizational paradigms. The *Knowledge engineering review*, 19(4), pp. 281–316, 2005 (Cambridge University Press).

Li, H., Yang, Y., Chen, T.Y., Resource constraints analysis of workflow specifications. *Journal of Systems and Software,* 2004, 73(2), pp. 271–285.

Lu, R., Sadiq, S., Padmanabhan, V., Governatori, G., Using a temporal constraint network for business process execution. In *Proceedings of 17th Australasian Database Conference,* Australian Computer Science Association, ACS, pp. 157-166, 2006.

Malone, T., Crowston, K., Herman, G. (eds.): *Organizing business knowledge: The MIT Process Handbook,* 2003 (MIT Press, Cambridge, MA).

Manzano, M., *Extensions of First Order Logic*, 1996 (Cambridge University Press).

Menzel, C., Mayer, R.J., The IDEF family of languages. In *Handbook on Architectures of Information Systems*, edited by P. Bernus et al., pp. 209–241, 1998 (Springer-Verlag, Heidelberg).

Peterson, J.L., *Petri net theory and the modeling of systems*. 1981 (Prentice-Hall, Englewood Cliffs)

Popova, V. and Sharpanskykh, A. (2007a), A Formal framework for modeling and analysis of organizations. In *Proceedings of the Situational Method Engineering Conference, ME'07*, 2007 (Springer Verlag).

Popova V., and Sharpanskykh, A. (2007b). Formal analysis of executions of organizational scenarios based on process-oriented models. In *Proceedings of 21st European Conference on Modelling and Simulation ECMS*'07, 2007 (SCS Press) pp. 36-44.

Popova, V., and Sharpanskykh, A. (2007c), Formal modelling of goals in agent organizations. In Proceedings of AOMS workshop joint with IJCAI'07, 2007, pp.74-86.

Popova, V., and Sharpanskykh, A. (2007d), Modelling organizational performance indicators, In *Proc. of IMSM'07 conference*, edited by F. Barros et al., 2007, pp. 165–170.

Popova, V., and Sharpanskykh, A. (2007e), Process-oriented organization modeling and analysis based on constraints, Technical Report 062911AI, VUA, http://hdl.handle.net/1871/10545.

Russell, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P. Workflow Resource Patterns. BETA Working Paper Series, WP 127, Eindhoven University of Technology, Eindhoven, 2004.

Scheer, A.-W., Nuettgens. M., ARIS Architecture and reference models for business process management. In *LNCS* 1806, edited by W.M.P. van der Aalst et al., pp. 366–389, 2000 (Springer-Verlag, Berlin).

Scott, W.R., *Institutions and organizations*. 2nd edn, 2001 (SAGE Publications, Thousand Oaks London New Delhi).

Sharpanskykh, A., (2007a) Authority and its implementation in enterprise information systems. In *Proceeding of the 1st International Workshop on Management of Enterprise Information Systems, MEIS 2007*, 2007 (INSTICC Press) pp. 33-43.

Sharpanskykh, A. (2007b), Modeling and Analysis of Organizations from the Air Traffic Management Domain, Technical Report 170707TR, VUA, http://few.vu.nl/~sharp/tr170707.pdf

Singh, M. P., Synthesizing distributed constrained events from transactional workflow specifications. In *Proc. of the 12th IEEE Intl. Conf. on Data Engineering*, 1996, pp. 616–623.

Tsang, E., *Foundations of Constraint Satisfaction*. 1993 (Academic Press).

Van der Aalst, W.M.P., Verification of Workflow Nets. In *LNCS* 1248, edited by P. Azema and G. Balbo, Application and Theory of Petri Nets, pp. 407-426, 1997 (Springer-Verlag, Berlin).

Van der Aalst, W. M. P., The application of Petri Nets to workflow management, *The Journal of Circuits, Systems and Computers,* 1998, 8(1), pp. 21–66.

Van der Aalst, W.M.P., and Ter Hofstede, A.H.M., YAWL: Yet another workflow language, *Information Systems* , 2005, 30(4), pp. 245-275.

Van der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B., and Barros, A.P., Workflow patterns, *Distributed and Parallel Databases,* 2003, 14(3), pp. 5–51.

Yang, D., and Zhang, S., Modeling workflow process models with statechart. In *Proc. of ECBS*, 2003, pp. 55–61.