ALLOW

Adaptable Pervasive Flows

213339

Deliverable D3.3

Flow Control Algorithms

Contributor.....: USTUTT (IAAS)
Author.....: Tobias Unger, Hanna Eberle

Reference Number....: ALLOW.USTUTT.D3.3.2011-08-3030
Version.....: 1.2
Date.....: 2011-08-30
Classification.....: Public
Circulation.....: Project Team
Contract Start Date...: 2008-02-01 - Duration: 42 Months
Project Coordinator...: USTUTT
Project Partner.....: USTUTT (IPVS), USTUTT (IAAS), UNI PASSAU, FBK, Imperial, ULANC

FET - Future and Emerging Technologies

Project funded by the European Community
under the FP7-Programme (2007 - 2013)

Copyright

Document History

Version	Issue Date	Author	Content and Changes
v 01	15.08.2011	Hanna Eberle and Tobias Unger	
V02	19.08.2011		Reviewed Version

Executive Summary

In the last period we deepened our understanding of the proposed concepts of person-centric flows and process fragments ,by elaborating on the semantics and architecture of person-centric flows and by investigation the composition options for process fragments and possible recovery behaviors for process fragments.

In this deliverable we will provide an overview of the workflow languages in ALLOW. We investigated these workflow languages with respect to flow flexibility and therefore for their applicability in pervasive scenarios. The result of this investigation is a classification scheme for the languages of the ALLOW project.

We analyzed and filtered the flow relevant requirements of the Healthcare scenario and designed a solution for the scenario integrating the concepts of Process Fragments and Person-Centric-Flows.

Table of Contents

Figures

1 Introduction

The purpose of the ALLOW project was to provide Flow Languages, their operational semantics, and their Flow Control Algorithms. The goal of the Allow project is to control or push actions in the real world using workflow technology. For this purpose it must be possible prescribe and describe flows that are happening in the real world. Mainly, the flows consist of actions and activities executed by people. A significant number of those activities are located in the real world i.e. there are hardly any activities that are only executed using a computer system.

As a first step the ALLOW project designed a flow language framework that allow to specify the descriptive and prescriptive real world flows. Since there are lots of different types of real world flows and their requirements sometimes do not fit together, the framework consist of different flow languages. Each of them is covering different aspects of the characteristics of real world flows. Subsequently, the operational semantics and control algorithms of the flow languages are defined.

However, the implementations of some scenarios require complex flow control algorithms, to address all the requirements of this scenario. They cannot be realized using one flow language only. Such scenarios require a composition of the ALLOW flow languages. We will show that complex scenarios can be implemented combining flow control algorithms of different languages.

This work consists of two major parts.

The first part provides an overview of the elaborated Flow Languages. This part concludes with an analysis of these Flow Languages with respect to their means of flow flexibility. We consider flexibility in workflows to be spanned between the two dimensions, level of structure of the workflow language the workflow model is designed in and the degree of changeability the workflow is exposed to.

The second part provides an integrated view on the flow control algorithms of Process Fragments and Person-centric Flows. We will discuss the control dependencies of Person-centric Flows and Process Fragments on the basis of the realization of the Healthcare Scenario with Process Fragments and Person-centric Flows. We show that the two concepts complement each other and do not overlap, since each concept deals with different organizational issues and therefore can be combined to a complex flow control algorithm. The elaborated flow languages can be com-

posed in different ways, which is determined by the problem domain of a particular scenario.

2 Flow Control Algorithms

Over the last 3 years it turned out, that we need more than a single flow language, but rather a framework of flow languages. Three flow languages were developed. Each of these languages is addressing different needs and deals with different characteristics of pervasive application scenarios. The developed flow languages are APFL, Process Fragments and Person-Centric Flows. In this deliverable we shortly outline APFL and Process Fragments, since these concepts have been discussed in previous deliverables. Person-centric Flows are presented in more detail, since this concepts hasn't been described in any deliverable, yet.We conclude this chapter providing a classification of the different workflow languages due to their means of flexibility.

For further details we refer to the concepts to the published papers discussing certain aspects of the flow languages. All the papers can be found in the appendix of the document.

2.1 APFL

Adaptable Pervasive Flows is a workflow-based paradigm for the design and execution of pervasive applications, where dynamic workflows situated in the real world are able to modify their execution in order to adapt to changes in their environment. This requires on the one hand that a flow must be context-aware and on the other hand that it must be flexible enough to allow an easy and continuous adaptation.

Therefore, APFL contains a set of constructs and principles for embedding the adaptation logic within the specification of a flow and it also contains means to model the adaptation trigger as part of the workflow model, where the adaptation triggers are represented as constraints. If the constraints are violated the modeled adaptation logic is executed.

All knowledge about the business case must be known at designtime.

The language provides different modeling constructs for the normal and positively expected business logic and the cases that are not desired, but one has to deal with anyways if they occur.

APFL was specified in Deliverable 5.1 [5].

2.2 Process Fragments

Process fragments are used to represent fragmentary process knowledge. Process knowledge is assumed to be 'local'. This means that process fragments represent fragmentary business knowledge for specific occasions.

Process fragments can be integrated at design or runtime using scenario adequate integration technics.

Runtime decisions can be implemented by the process fragment model itself or be implemented by the integration technic by choosing a certain process fragment to be composed with the running process fragment.

Hence, the process fragment approach is a hybrid approach, where runtime decisions are made either within the process execution context in an imperative way or are declared by the integration technic.

The contributions we provided in this area are the following:

- Process Fragment concept
- Integration technics for Process Fragments
- Recovery concepts for Process Fragments
- Execution architecture for Process Fragments

The related papers are [7][8][9].

2.3 Person-centric Flows

Task Management is a core component within Workflow Management Systems from the outset. The primary objective of Task Management is the creation and management of tasks carried out by people, where those tasks are part of business processes. In the context of the ALLOW project these processes are modeled and executed using the flow languages APFL and Process Fragments. However, today's Task Management Systems focus on effective work distribution. The goal of this work is to develop concepts helping people organizing their work, since they have not found their way into state of the art workflow technology. Our approach

is based on the idea of a Person-centric Flow. The basic idea behind a Person-centric Flow is to understand a person's task list as a flow of tasks. A Person-centric Flow tries to simulate the behavior of people by linearizing and refining their tasks into an appropriate work plan. This is achieved by two major concepts: a concept for refinement of tasks and a concept of a task recommendation and control system.

A Person-centric Flow is an IT-representation of the flow of activities an individual person is performing [13]. For example the daily care schedule of a nurse can be understood as the person-centric flow of the nurse. Mostly, these activities are planned and created by the hospital's healthcare documentation and planning system. The IT representation of such an activity is called human task. People are informed about their tasks using task lists. Additionally, the person-centric flow must consider the fact that people divide tasks into smaller pieces which are more manageable for them or combine the execution of two or more tasks, because they are very similar.

Person-centric flows are created by enriching the task list with flow information, which can be recommendations or obligations. The person-centric flow can be synchronized with what the person is doing either explicitly by ticking the tasks on the task list or by observing the person using activity sensing [15]. The other way round, persons can be provided with recommendations and instructions how to execute their tasks based on the flow information which help people e.g. to save resources or time. This can be done e.g. by calling their attention to possible errors or to the fact, that the chosen ordering of the tasks is prohibited. Ambient guidance strategies provide means to direct and correct humans if necessary, e.g. if a nurse forgets an important procedure [16] . To be able to implement ambient guidance the ambient guidance system must be aware of the work a person has to perform and how this work should be done. Therefore, the person-centric flow opens great opportunities to support people in doing their work and to ensure compliance to certain regulations. The person-centric flow can be utilized to check whether a person deviates from her person-centric flow and whether this deviation is tolerable or not. Applications can be informed about the violation of the person's behavior from her person-centric flow using events. Events are also generated in case a person deviates from her flow. Imagine a nurse has a person-centric flow which prescribes as first step to measure the blood pressure of patient *A*, then to disinfect her hands, and, finally, to measure the blood pressure of patient *B*. If the nurse decides to take the blood pressure of person *B* directly after measuring blood pressure of person *A* without disinfecting her hands between these two tasks, the nurse deviates from her person-centric flow in an intolerable way and the ambient

guidance system is informed by a *violation event*. The violation information can be used to inform the nurse that she has to disinfect her hands. Additionally, also tolerable deviations can be detected. A tolerable deviation would be to measure blood pressure of patient *B*, then to disinfect her hands, and, finally, to measure the blood pressure of person *A*. Also in this case, the ambient guidance system can be informed about the deviation by a *deviation event*. Such an information can be used e.g. for providing the nurse with the health record of person *B* instead of providing her with the health record of person *A*.

Definition 1 (Person-centric flow): *A person-centric flow defines a partial ordering over a set of tasks which have to be performed by one single person. Tasks of a person-centric flow can be classified in three tenses: past tasks, present tasks, and future tasks. Past tasks are either completed correctly or incorrectly and their ordering is known. Present tasks are tasks that are currently presented in a person's task list. Their ordering is planned by the person but can change dynamically. Future tasks are tasks which are assumed to be executed in the future. Both the set of future task as well as their ordering may change dynamically. The tasks of a person-centric flow derive from a set of business processes in which the person participates and/or from standalone tasks. Furthermore, the tasks may be refined in order to operate on the person's activity level.*

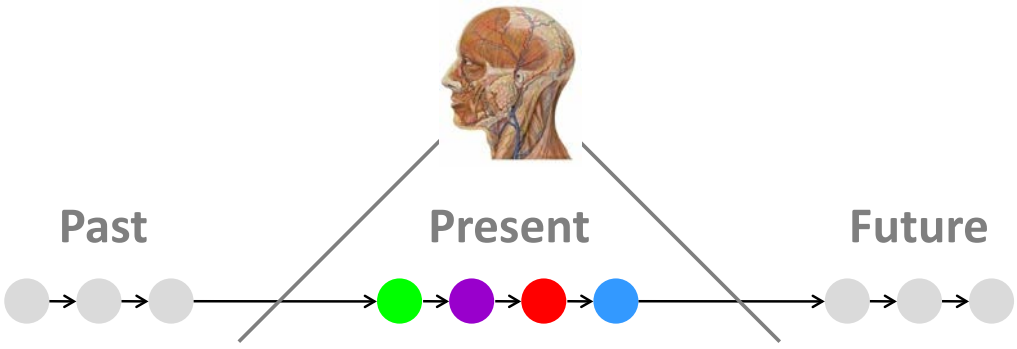

Figure 1: tenses of a person-centric flow

Figure 1 shows a simple graphical representation of a person-centric flow. Please note that present and future tasks can be mixed up within a person-centric flow. Only the task that is actually executed must be included in the set of present tasks. Since a person-centric flow is formed implicitly in a person's mind it must

be predicted by the WfMS in order to utilize the ordering information. A person cannot be demanded to tell the WfMS its actual flow. Since tasks appear and disappear in a high frequency, this would be an additional stress factor. As a consequence predictions may be wrong. Furthermore, the person-centric flow paradigm is partly contrary to the existing workflow paradigm. For example a person-centric flow has no prescribed flow model as the set of tasks changes dynamically and so does the ordering of the tasks. Many control flow patterns like loops are not needed in person-centric flows since each task is executed once. A single instance of a person-centric flow is associated to one person. In ALLOW we rely on declarative workflows in order to describe a person-centric flow of a person [14].

2.3.1 Refinement Stage

Tasks within business processes are often modeled with a granularity which differs significantly from the granularity the tasks are actually executed by people. Hence, very important characteristic of people is that they refine their tasks into a granularity which corresponds to their activity level. Investigations in the field of human computer interaction (HCI) show that people usually structure the working of a task by splitting it into a set of smaller subtasks, so that the overall task can be managed. Additionally, our studies showed that people also merge a set of tasks into a more coarse-grained task. A nurse, for example, wakes up all patients within a room altogether instead of waking up each patient separately. Workflow research in the past has not paid much attention to this approach. We developed a concept for supporting this work approach by giving the people the ability to arrange their tasks as a set of subtasks (cf. e.g.[11]). In particular, we have identified two rearrangement patterns, namely split and merge, which helps people to manage their individual work approaches. Therefore, a system is introduced enabling the dynamic refinement of tasks. The system provides the following refinement patterns:

- Split: splits one task into a set of sub-tasks.
- Merge: combines one or more tasks into a single task.

Merge and split operations are executed either manually or automatically (scenario depended). At runtime, origins of sub-tasks or merge-tasks are completed on completion of (all) of their successors. The result of the refinement is a task list consisting of tasks on leaf level.

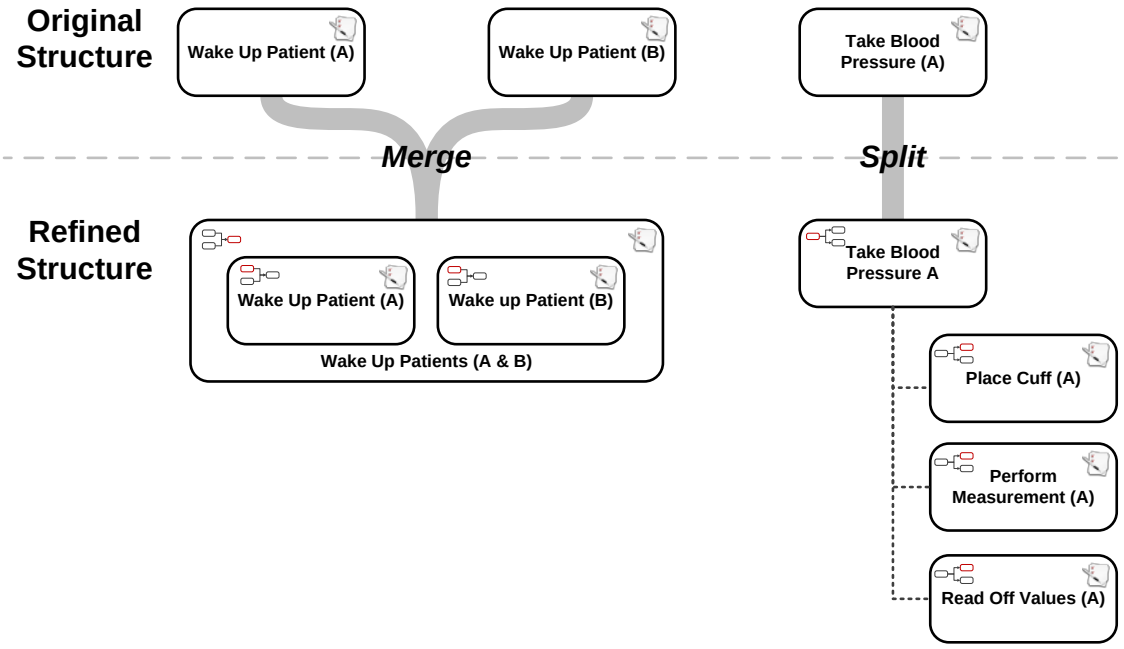

Figure 2: Merging and splitting tasks

Figure 2 shows the exemplary the merge and the split patterns. A nurse has decided to wake up the patients together. After merging the two tasks the nurse's task list only contains the merged task. Same applies to the task which the nurse splits up into subtasks. The nurse's task list only contains the subtasks instead of the root task.

2.3.2 Recommendation & Control Stage

It is known from appropriate studies, that people typically have a hard time to carry out several tasks in parallel, so they start linearizing their work. One of the problems associated with the linear execution of tasks is the considerable cognitive load when switching from one task to another. Our approach extends the prior art by introducing concepts to task management that reduce this cognitive load. Based on the person-centric flow we introduce a concept that helps people finding an optimal schedule for their tasks by recommending an optimal order. The recommendation system considers the contextual proximity of a performer to a task so that a context switch produces a lower cognitive load that means less setup time is needed. Users obviously can ignore the recommendation to provide them with the flexibility to react to unforeseen situations. On the other hand the system can be used to control the execution order of a person and provide him/her with corrective information.

Figure 3 shows an example task list containing 6 tasks 4 of them refined.

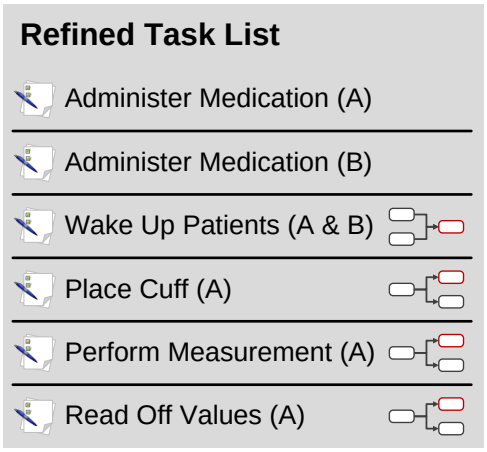

Figure 3: initial worklist

There might exist a plan for the execution order of the tasks planned by the person but the ordering can be changed dynamically. People are very good in scheduling their tasks. A growing number of tasks, however, lead to scheduling errors and an increasing cognitive load. A system that provides recommendations and controls the execution to notify a person about an error can counteract this. Thus, the (refined) task list is enriched with recommendations and obligations which are generated by domain-specific algorithms and/or extracted from top-level business processes. The recommendations can be used to guide a person. On the other hand, the obligations can be used to control the execution.

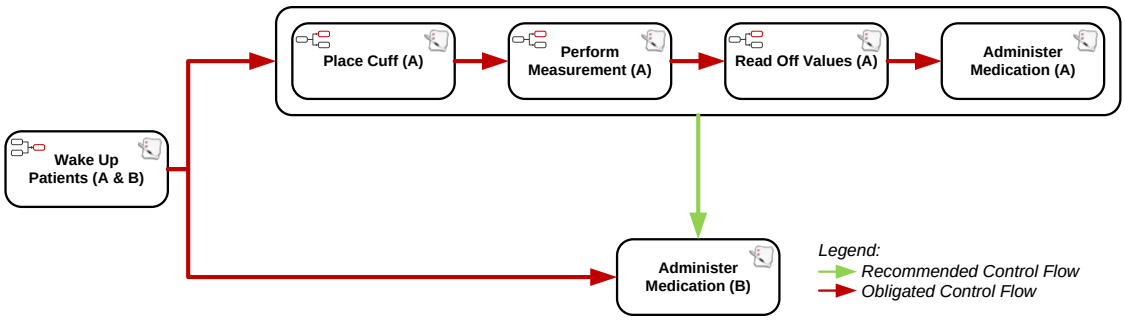

Figure 4: Recommendations and obligations represented as graph

Figure 4 shows the recommendation and control graph which is automatically generated by a set of domain-specific algorithms. Possible algorithms are schedul-

ing algorithms, scoring algorithms, etc. Constraints originating from the original task list or the refined task list are propagated to the recommendation and control graph.

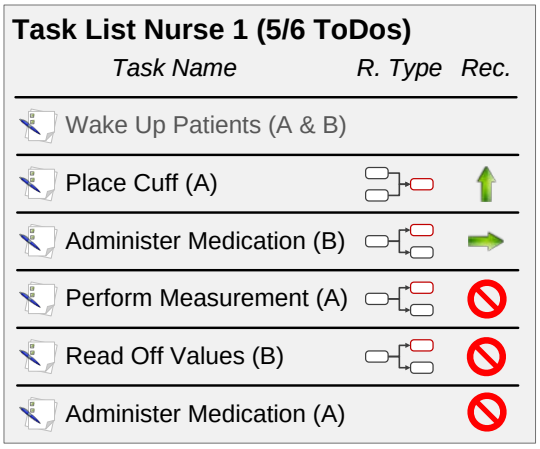

Figure 5: Enriched worklist

Figure 5 shows a task list visualization of the recommendation and control graph which can be presented to the user. In this example arrows and prohibition signs are used to provide the nurse with recommendations.

2.3.2.1 Person-centric Flow Language

This section gives a brief overview about control-flow constraints that can be used along with task models to design a person-centric flow.

- *Unary constraints:* In [17] several constraints were introduced that can be defined on a single task model; they are called *unary constraints* here. Namely, these are the *existence*, *init,* and *last* constraint. The *existence* constraint defines a lower and/or upper bound concerning the number of instances of a task model that must be executed. If the *init* constraint was defined on a task model, an instance of this task model must be the first task that is executed during workflow execution. The *last* constraint on the other hand defines that an instance of the task model where the constraint was defined on must be the last task that is executed during workflow execution.
- *Choice constraints:* Another class of constraints suggested in [17] is the *choice constraints*. These constraints are used to specify that from a given set of task models a certain subset has to be chosen for instantia-

tion and execution. If the constraint defines for instance that *2* instances from the set of task models *{A, B, C}* have to be executed, one of the subsets *{A, B}*, *{A, C}*, or *{B, C}* of task models must be chosen for execution. The choice constraint defines only a lower bound concerning the size of the subsets, i.e. the execution of the task models *{A, B, C}* is also valid.

- *Relation constraints:* The class of *relation constraints* defines the execution order of the instances of two task models. In [17] several sequential relation constraints are described. They can be used to restrict the sequential execution order of two tasks. The *A precedence B* constraint is an example for these kind of constraints. It imposes the restriction that an instance of a task model *A* has always to be executed before an instance of task model *B*. The *A response B* constraint on the other hand defines that an instance of task model *A* has always to be followed by an instance of task model *B*.

However, with sequential relation constraints it is not possible to model the requirement that two tasks must be performed exactly or at least partly at the same time. These kinds of requirements usually emerge through collaborative work, i.e. for achieving a certain goal several tasks have to be executed simultaneously and each of them is performed by a different person. For instance when the pair programming technique is applied to develop software, one programmer writes the code (task *A*) and the other one has to review the typed code (task *B*) simultaneously. We introduce parallel relation constraints to model these kinds of restrictions. Each of these constraints can be used to determine the degree of simultaneous execution (e.g. partly or completely). The constraints base on the interval relations of Allen's interval algebra [18] (briefly called IA). The IA was chosen because firstly, the interval relations proposed there cover all possible parallel relations between two intervals and secondly, algorithms exist for this algebra to verify that interval relations are consistent. An example for these kinds of constraints is the *A during B* constraint which defines that task *A* has to be started and completed during the execution of task *B*. To realize the pair programming example an *equals* constraint has to be defined between the task models *Review* and *Write Code*.

In [14] the relation constraints are extended by time parameters. These time parameters can be used to reflect temporal restrictions between the tasks (e.g. that a task has to be executed within a certain time after its predecessor task was completed). Furthermore, for each relation constraint also a corresponding *negation*

constraint exists. These constraints provide the capability to prohibit a certain execution order. The negation constraints are also described in[14].

2.3.2.2 Formal Definition

In this Section we present our formalization of person-centric flows as presented in [10]. The formalization bases on the formalization presented in [19]. Let $\mathscr{P}$ be the set of all persons. $\mathscr{TM}$ denotes the universe of all available task models. The runtime instances of task models are denoted as the set $\mathscr{TI} = \{\mathscr{TM} \times \mathbb{N}\}$. Each task instance has a state, where the set of possible states is denoted as $taskState : \mathscr{TI} \rightarrow \{\perp, activated, running, completed\}$. Task instances with a task state $\perp$ denote virtual task instances, which are going to be created in the future, but nevertheless they are an important part of the person-centric flow determination. Each task instance has exactly one person assigned to it, which is defined by the map: $staffAssignment : \mathscr{TI} \rightarrow \mathscr{P}$. If a task instance execution is started and the task state changes from activated to running the starting time for that task instance is set. The starting time of a task instance is later retrieved by the map: $startingTime : \mathscr{TI} \rightarrow (\mathbb{N} \cup \{\perp\})$. If the task instance changes to the state completed the completion time is set, which can also be retrieved by a map: $completionTime : \mathscr{TI} \rightarrow (\mathbb{N} \cup \{\perp\})$. The set of all constraints is denoted by the set $\mathscr{C}$. Constraints are defined on task instances, which are assigned to a constraint by the map: $tasksOfConstraint : \mathscr{C} \rightarrow 2^{\mathscr{TI}}$. There exist several types of constraints, which can be classified in three different classes. Unary constraint types define restrictions on one single task instance, e.g. how many times this instance has to be executed. If the *init* constraint was defined on a task model, an instance of this task model must be the first task that is executed during workflow execution. The *last* constraint on the other hand defines that an instance of the task model where the constraint was defined on must be the last task that is executed during workflow execution. Constraints types of the choice constraint class are used to specify, that a subset of a set of task instances has to be chosen for execution. Relation constraints types define the execution order of the instances of two task instances. The *A precedence B* constraint is an example for this constraint class. A more detailed description of the constraint types can be found in [12][14].

A person-centric flow is denoted as follows: A person-centric flow of a person $p \in \mathscr{P}$ at an observable point in time $i \in \mathbb{N}$ is a tuple $PCF_{p,i} = (PTI_{p,i}, HTI_{p,i}, C_i, c_{strength}, c_{type}, m)$, where $PTI_p = \{ti | staffAssignment(ti) = p \wedge taskState(ti) \in \{activated, running\}\}$ denotes the present task instances of a person and $HTI_p = \{ti | staffAssignment(ti) = p \wedge taskState(ti) = completed\}$ the already completed task in-

stances or history task instances. $C_i \subseteq \mathscr{C}$ denotes the set of the currently existing and to be satisfied constraints. The $C_i$evolves over time, always depending on the present task instances and the history task instances, since the tasks involved in $C_i$must be of the present task instances or history task instances of the $PCF_{p,i}$. For each constraint in C_i a strength and a type is specified. The strength notes whether a constraint is optional or mandatory. Optional constraints are usually used for guidance and mandatory constraints are needed to ensure certain qualities and requirements: $c_{strength}: C_i \rightarrow \{optional, mandatory\}$. The type of a constraint obtains one of the three options intention, guidance or enforcement. Intentional constraints are constraints that are used to be able to follow the flow a person has in mind. Guidance constraints are constraints that can be used to include special guidance constraints (e.g. a recommendation to measure the heart rate before measuring the blood pressure) within the person-centric flow. Enforcement constraints are used e.g. to support a proactive ambient guidance for constraints which have to be satisfied (e.g. that a nurse hast to disinfect her hand after washing a patient). Therefore, the map $c_{type}: C_i \rightarrow \{intention, guidance, enforcement\}$ is defined in order to assign a constraint with a strength. The constraint set $C_i$of a $PCF_{p,i}$must hold, that if a constraint c in $C_i$is a constraint of the intensional type, the strength of the same constraint must be optional, since intensional constraints are supporting constraints but not constraints that are enforced. On the other hand, if a constraint c in C_i is an enforcement constraint the strength of the same constraint must be mandatory. We define $C_i = \{c \in \mathscr{C} | (tasksOfConstraint(c) \in PTI_{p,i} \cup HTI_{p,i})$ and $(c_{strength}(c) = optional$ if $c_{type}(c) = intention)$ and $(c_{strength}(c) = mandatory$ if $c_{type}(c) = enforcement)\}$ The mode of the person-centric flow is set by $m \in \mathscr{M}$.

2.3.2.3 Person-centric Flow Execution

A person-centric flow gets executed by the nurse performing her tasks. If the nurse starts a task the task in the person-centric flow gets triggered and set to state running. The work a nurse is performing is evaluated against the constraint set of the intended person-centric flow. Depending on that evaluation, the person-centric flow must be either adapted to the real world, or if the nurse is violating mandatory constraints, the system generates corrective guidance events. The evaluation of the person-centric flow is defined as follows. Since the $PCF_{p,i}$ is executed and its definition changes over time, the constraint set is evaluated against a time depending constraint set with a time depending execution state. The execution state of a $PCF_{p,i}$ for a person $p \in \mathscr{P}$ and a point in time $i \in \mathscr{N}$ is defined by the task states. The history task instance set $HTI_{p,i}$ and the running task instances of $PTI_{p,i}$are evaluated against the constraints. This joined task instances set is called

satisfaction task instances set $STI_{p,i}$, with $STI_{p,i} = HTI_{p,i} \cup \{ti | ti \in PTI_{p,i} \wedge taskState(ti) = running\}$. All the tasks of the $STI_{p,i}$define are running or completed and therefore the task starting time is specified. Therefore the $STI_{p,i}$ can be used for constraint evaluation, hence there already exists a ordering between the tasks this set defined by the task instances starting time. The ordering of the executed task instances denotes the execution trace of the $PCF_{p,i}$. The remaining task instances are the activated or open task instances of $PCF_{p,i}$, which are denoted as $OTI_{p,i} = \{ti | ti \in PTI_p \wedge taskState(ti) = activated\}$. Based on the $OTI_{p,i}$we are able to determine, whether it will be possible to satisfy C_i, if $C_i$is violated by the $STI_{p,i}$. A violated $C_i$is can be satisfied, if there exists a partial execution trace or a partial ordering of $OTI_{p,i} \cup STI_{p,i}$, which satisfies C_i. Therefore we define the constraint evaluation procedure for a $PCF_{p,i}$ at a certain point in time $i$as follows:

$$eval(STI, OTI, C) = \begin{cases} satisfied & if STI_P \models C \\ temporarily violated & if (STI \not\models C) \wedge (\exists t \in 2^{OTI} : STI \cup t \models C) \\ violated & otherwise \end{cases}$$

2.3.2.4 Person-centric Flow Generation Algorithms

Since the person-centric flow may change dynamically according to the current situation, the constraints must be re-generated continuously. The prediction algorithms create a constraint set based on the present tasks $PTI_p$and past tasks HTI_p. Formally, a prediction algorithm is a function $prediction(PTI_h, PTI_p) \subseteq \mathscr{C}$ returning the actual valid constraint set for the set of tasks.

Since we are not able to capture the person-centric flow a person has in mind, we need to find algorithms to determine the task orderings. Especially history based algorithms are promising since people often have behavior patterns which can be detected by history-based algorithms. Also algorithms operating e.g. on task deadlines without considering the history (e.g. scheduling algorithms) are appropriate (c.f. [20][21]). Furthermore, we developed a context-based scoring algorithm for generating recommendations.

2.3.2.5 Architecture

The major idea of our approach is to utilize the knowledge about the person-centric flow and its state in order to improve applications like ambient guidance. Figure 6 shows our overall architecture. The central element is the person-centric flow manager, which receives the constraints from the plugged in constraint gen-

eration algorithms. We argue that according to the scenario different constraint generation algorithms will be necessary, because every scenario deals with different context and focuses on different optimization goals and therefore with different optimization criteria that define the input for the constraint generation algorithm. Tasks are managed in a task manger which executes the business process in cooperation with the process engine.

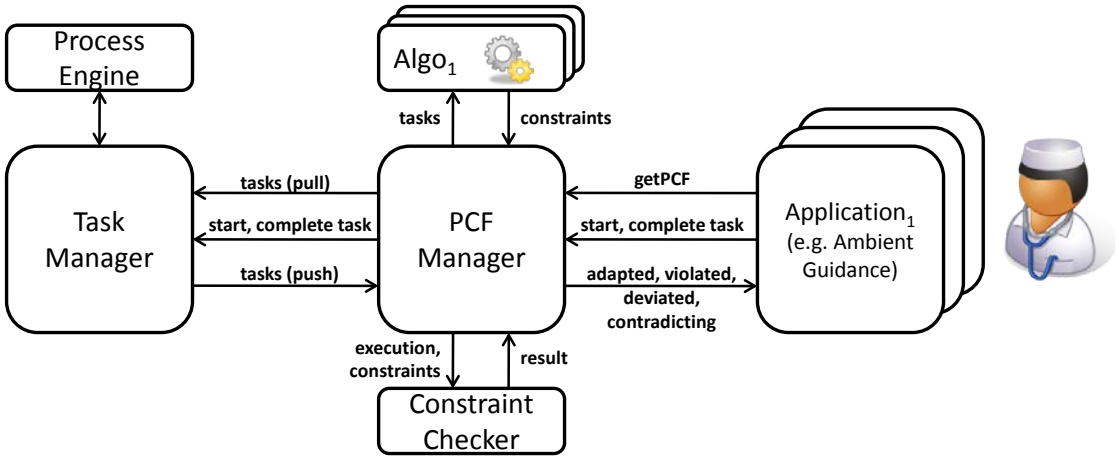

Figure 6: PCF execution architecture

A constraint checker component evaluates the execution against the current person-centric flow as well as the person-centric flow model against itself. The events that are produced by the constraint checker component are utilized by the registered ambient guidance components. For example an ambient guidance system might use the person-centric flow events to provide the person with directions and guiding the person pro-actively through all her work. If a person deviates from the person-centric flow or if she even violates the person-centric flow, an event is thrown by the constraint checker component. The information contained in this event can be utilized by the ambient guidance to adapt the guidance. For example if a person starts another task than predicted, the ambient guidance attune to the new situation by updating the guidance information presented to the user. Additionally, after an execution violation a new generation of the person-centric flow is triggered. In the case that the person-centric flow is adapted or the model is self-contradicting the constraint checker throws either an adaptation event or a contradiction event. The adaptation event indicates that the plans of the person have changed, which facilitates the reconfiguration of the ambient guidance component in order to provide the person with information about tasks according to the changed plans.

2.3.2.6 Event Model

In order to guide people effectively we need information about the intended person-centric flow of a person. This encompasses the list of task instances that have to be performed by a person and the predicted constraints that have to be met by these task instances. Moreover, information about the execution state of the person-centric flow have to be gathered and published (refer to *Execution-related Events* below), e.g. whether the execution is deviating from the predicted person-centric flow or even violating the constraints. In order to avoid that applications have to periodically request for state changes in the person-centric flow (e.g. whether the flow was violated), the person-centric flow manager (PCF Manager) sends events to the applications if the state of the flow has changed. In addition, there are also events sent to the PCF Manager. These are real world events (e.g. if a nurse starts blood pressure measurement) that are captured using activity recognition. After the PCF Manager has received such events it triggers the Constraint Checker that checks that the person behaves as expected by verifying that the actual valid constraints are met. If she does not behave as expected an event is sent to the application by the PCF Manager. The advantages of using events are that we can evaluate the constraints in a centralized way and that we are able to inform other components (e.g. the application) that are interested in these events almost in real time. Furthermore, it prevents the person-centric flow manager from being overburdened by answering continuous polls from the applications.

Deviations and violations are detected by the Constraint Checker component. It verifies that no inconsistencies emerge between the constraints and it also ensures that the constraints are met during the execution of the person-centric flow. The Constraint Checker is always triggered when tasks change their state to *running* or *completed* and when the constraint or task model set has changed. Depending on the outcome of the verifications the Constraint Checker can raise the following events:

Model-related Events

- *Adapted*: A person-centric flow is adapted, if the set of task and/or the set of constraint changes.

- *Contradicted*: It may happen that new constraints contradict with already existing constraints. If this is the case the flow is in “contradicted” state. For example if the intention of a nurse is to wash a patient before taking the heart rate this may contradict to a rule saying that the tasks have to be

done the other way round. In this case for example an intention constraint would contradict to an enforcement constraint.

Execution-related Events

- *Satisfied*: A person-centric flow is satisfied, if all constraints evaluate to true. In other words the person behaves as predicted and her behavior causes no violation of e.g. a security rule.

- *Deviated*: A person-centric flow is in state deviated, if the person deviated from the predicted behavior. For example tasks are executed in a different order than predicted.

- *Violated*: A person-centric flow is in state violated, if the person violates for example a security constraint. Technically spoken this means that at least one of the mandatory constraints is violated.

It has to be clearly stated that our approach only informs the applications about deviations or violations of the person from the person-centric flow. We do not prevent a person from starting tasks which cause a violation of the person-centric flow. This is due to several reasons: Actions are executed in the real world. Mostly, if a violation of the person-centric flow is detected, the person has already started working on a task. Generally, two reactions are possible. On the one hand the person is informed about the violation and can decide whether to stop or continue the work. On the other hand, due to changing situations, the constraints are changing so that working on the task would no longer cause a violation of the person-centric flow. In the latter case, the violation disappears automatically as soon as the constraints are evaluated the next time.

Model-related Events Generation Implementation

In order to generate model-related events the Constraint Checker has to validate the consistency of the constraints models to ensure that there exist no contradictions **Fehler! Verweisquelle konnte nicht gefunden werden.** between them. In the following an overview is given that describes how the Constraint Checker detects contradictions. A more detailed description of these steps can be found in [12]. If any inconsistency between two or more constraints was detected the Constraint Checker sends a “Contradicted” event. Since tasks and constraints are generated dynamically, for each task a task model is created at runtime. These task models are required by the constraint validation mechanisms that are de-

scribed here. To validate relation, *init*, *last* and negation constraints [12] a constraint network is created and validated. In the following it is described how this is done.

The relation constraints are transformed to interval relations of the interval algebra **Fehler! Verweisquelle konnte nicht gefunden werden.** where each task model is represented by an interval. For instance if a *response* constraint is defined between task models `A` and `B` it is transformed to the interval relation $A\{before, meets\}B$. In the terms of interval algebra this means, that interval (task) *A* has to appear either immediately (indicated by the *meets* relation) or any time before *B* (indicated by the *before* relation). The interval relations that are created from the relation constraints form a constraint network like the one that is illustrated in Figure 7.

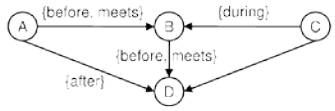

Figure 7: Example Constraints

In [18] and [22] reasoning algorithms were introduced that can be applied on constraint networks. These reasoning algorithms infer transitively the relations between all intervals and discover contradictions between them. For instance in Figure 7, it can be inferred from the relations "`C` has to be executed during the execution of `B`" and "`B` has to be executed before `D`" that `C` must be executed before `D`. Moreover, it can be also inferred that a contradiction exists in the network since `A` cannot be performed after `D` but before `B`. If such contradictions are detected we can conclude that there exist inconsistencies between the relation constraints (including the *init* and *last* constraint) were the constraint network was created from. In [9] a comprehensive overview is provided how to detect inconsistencies between constraints.

Execution-related Events Generation Implementation

To emit execution-related events the Constraint Checker verifies during the execution of $PCF_{p,i}$ if the constraints are met. If all constraints are met $PCF_{p,i}$ is in the state *satisfied* and a "Satisfied" event is generated. On the other hand, if one or more of the constraints are violated $PCF_{p,i}$ is in the *temporarily violated* or in the *violated* state. If the violation of an optional constraint was determined (temporarily or permanent) the "Deviated" event is emitted. If a mandatory constraint was

violated the "Violated" event is generated. To perform the verification the Constraint Checker uses the procedure $eval(STI, OTI, C)$. In the following it is described for each constraint how the procedure determines if the constraint was met.

***Init* constraint:** An init constraint is

- *satisfied*, if an instance of the task model, where the constraint was defined on, is the first task of the $PCF_{p,i}$ that was executed;
- *violated*, if an instance of another task model, i.e. a task model where the constraint is not defined on is the first task of the $PCF_{p,i}$ that was started;
- *temporarily violated*, if $PCF_{p,i}$ was started but no task was executed yet, i.e. an instance of the task model where the *init* constraint was defined on can still be the first task that is executed.

***Last* constraint:** A last constraint evaluates to

- *satisfied*: If an instance of the task model that is associated with this constraint is executed and no other task of the $PCF_{p,i}$ is in the state *running* anymore.
- *violated*: This constraint cannot cause the $PCF_{p,i}$ to become violated. Even if the user executes an instance of a task model that is not associated with this constraint she has always the possibility to start the task that is associated to the *last* constraint when no other task is in the state *running*.
- *temporarily violated*: If an instance of the task model was not started as last task of the $PCF_{p,i}$.

***Existence* constraints:** It has to be simply counted in the $STI_{p,i}$ how many instances of the task models, where this constraint was defined on, are in the *running* or *completed* state.

- *satisfied*: If the number meets the lower and upper bound that was defined by the constraint.
- *violated*: If the upper bound concerning the allowed number of *running* or *completed* instances of the task model was exceeded in $STI_{p,i}$.

- *temporarily violated*: If the lower bound concerning the minimum required instances of *running* or *completed* instances of the task model was not met in $STI_{p,i}$ yet.

***Choice* constraints:** The *choice* constraint defines that from a set of task models a certain number K of them has to be instantiated and executed. It can be simply checked in $STI_{p,i}$ if instances of the K different task models appear there.

- *satisfied*: If K or more instances of different task models from the set are a members of $STI_{p,i}$.

- *violated*: The *choice* constraint cannot violate $PCF_{p,i}$.

- *temporarily violated*: If less than K instances of different task models from the set are members of $STI_{p,i}$.

***Relation* constraints:** To verify that the relation constraints were met we are utilizing again the constraint network that was introduced above. The Constraint Checker determines for each task that is put into the *running* and *completed* state if the relations of this task to the other tasks in $STI_{p,i}$ corresponds to the interval relations between the task model where the task was created from and all other task models in the constraint network. In order to determine the partial order between the task that was started or completed and the other tasks we express the relation between the tasks with the point algebra[22][23] (this algebra defines a partial order between the start and end points of a task pair). This enables us to determine the interval relations between the task and the other tasks. Then it is simply checked if the determined interval relations match with the interval relations in the constraint network. The constraint network in Figure 7 defines for instance that a *during* interval relation must hold between the tasks *C* and *B*. If $STI_{p,i}$ would contain the information that task *B* was completed before *D* the *during* constraint that is represented by the interval relation would have been violated. However, this approach can only determine if permanent violations have occurred.

- *satisfied*: If the relations of the task that was started or completed and the other tasks in $STI_{p,i}$ match with the interval relations that were defined in the constraint network.

- *violated*: If there are any relations between the task that was started or completed and the other tasks in $STI_{p,i}$ that does not correspond to the interval relations defined by the constraint network.

- *temporarily violated*: If there is a sequential relation constraint defined between the task models A and B and an instance of A is in the *completed* state but an instance of B has not been executed yet. Or if a parallel relation constraint was defined between A and B and the instances of these models have not been completed yet. For instance if a *during* constraint was defined between A and B and an instance of A was started after an instance of B but both tasks were not completed yet.

***Negation* constraints:** As mentioned before negation constraints are transformed to the interval relations that represent the constraint that is negated. If a task is put into *running* or *completed* state it is checked if $STI_{p,i}$ contains the forbidden relation.

- *satisfied*: If there exists no relation in $STI_{p,i}$ between the task and the other tasks that is forbidden by the negation constraint.

- *violated*: If there exists any relation in $STI_{p,i}$ between the task and the other tasks that is forbidden by the negation constraint.

- *temporarily violated*: Negation constraints cannot violate $PCF_{p,i}$ temporarily.

2.4 Classification of Flow Languages

In this section we discuss the flow languages. We argue that each of the presented flow languages serves a certain purpose. We investigate these languages focusing on the level of structure of each of the language. We consider therefore the degree of the imperative parts of a flow description and the declarative parts.

Basically, two types of workflow languages exist: imperative languages (cf. e.g. **Fehler! Verweisquelle konnte nicht gefunden werden.**) and declarative languages (cf. e.g. **Fehler! Verweisquelle konnte nicht gefunden werden.**). Imperative languages are assumed to be easier to understand and thus process modeling also is assumed to be easier. Imperative languages describe exactly the way how a goal can be achieved where the goal itself is defined by the outcome of the business process. On the other hand business processes defined in a declarative way define the goal explicitly that needs to be achieved, but don't specify the way how exactly this goal can be achieved (e.g. **Fehler! Verweisquelle konnte nicht gefunden werden.**). Therefore declarative languages promise a higher degree of flexibility, since it is not defined the way how to achieve a goal but the boundaries

an execution trace needs to comply to. While imperative languages are designed to be executed completely automatically in stable environments declarative languages are more suitable for processes, where the exact execution order is disposed to a user and which are running in permanently changing environments.

Traditional workflow technology uses workflow languages of the imperative type (e.g. graph-based workflow languages). The models cannot be changed during runtime, therefore the complete decision space needs to be known at design time.

The APFL is a graph-based workflow language and therefore can be categorized as imperative language. It introduces a set of flexibility concepts to overcome the rigid structure of traditional workflow languages.

Process Fragments is a hybrid approach. The process fragments are modeled in an imperative way, but can be selected and composed based on a declarative process description. A process fragment itself is not changed while it is executed.

Constraint-based workflows are designed in a declarative way, but the declarative workflow specification is not allowed to be changed during runtime. Also Person Centric Flows are designed in a declarative way, defining some constraints on the activity set. However Person Centric Flows allow the constraint set to be during runtime.

In Figure 8 we provide an overview of the presented results.

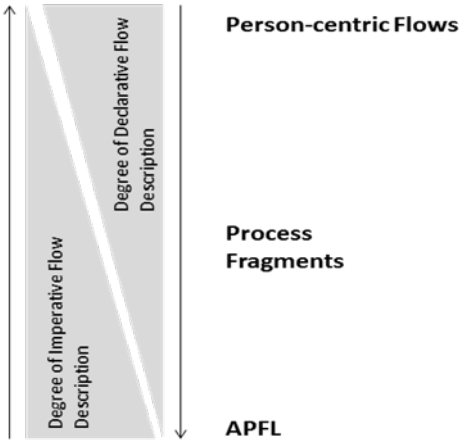

Figure 8: Degree of Structuredness

Which language should be used for which purpose can be figured out identifying the drivers of the execution of the flow, that ought to be implemented. The drivers of the execution of an imperatively modelled flow are mainly navigation events. If one activity completes another one gets activated. Usually, imperative languages allow reacting also to external events defining event-type activities. However, these activities get activated by a navigation event.

Drivers of the execution of a declaratively modelled flow are events that were created by other systems, e.g. observing a nurse during work. The execution of a declarative flow is to react to such an event. A transition of the internal representation of the flow is taken.

Again the drivers of the execution of process fragments are a hybrid of imperatively drivers and declaratively drivers. If flexibility is defined as the ability to react to external events, declarative flows provide the best means to implement flexible flows. However, if flows are used to drive a process APFL is the language to choose.

3 The Healthcare Scenario- Process Fragments and Person-centric flows applied

3.1 Scenario Analysis

The overall **business goal** in care processes of the Healthcare Scenario is to heal the patient. Therefore the main perspective for modeling is the view, which activities need to be performed by whom using what resources to achieve this business goal. There exist also processes for the individual resources, the staff and so on, but all these resources and staff need to be coordinated or orchestrated to achieve the business goal. Therefore, the integrating views of the activities to be performed are modeled from a patient's perspective. However, there are other non-functional goals to be encompassed, e.g. an optimal workflow for each single nurse.

Therefore, we focus on two types of flows in the Healthcare scenario, the patient flow, which implements an orchestrating perspective of the scenario and the flows performed by the nurses, that represents the actual flow performed by the nurse, and helps her to organize the tasks she has to perform.

In the Healthcare Scenario proposed by UPassau we can identify different types of flows.

- The care plan for each patient, which is determined
    - statically by the defining the care-packages the patient needs. All information about the patient and the necessary treatments is stored in the care plan.
    - and needs to be planned dynamically per day, since the care-packages, their orderings and their timing depending on **the daily setting**, which depends on the nurses that are available at certain times and the restrained resources that are needed to perform the procedures.
- The flow the nurse is executed
    - Highly dynamic and driven by a nurse herself, however there are some constraints the work of the nurse needs to comply to.

  - Determined by the set of tasks, that need to be done and their timing, when the things need to be done

- The administrative flows, that are not described here, but are triggered, when a new patient is registered, e.g. insurance flows, and documentation flows, ordering of medicine the patient is needing a.s.o. These are supporting flows for the patient flow and implement the logistics in the background.

The Patient Flows consist of tasks that need to be performed by a nurse. When this task is started, the task is added to a task list of a nurse. All the tasks in a task list define the tasks of the person-centric flow of the nurse.

However, a person-centric flow is more than just a set of tasks. It also specifies some constraints over the task execution by restraining the possible orderings of the tasks.

In Figure 9 we present an overall view of the processes and how they are related. In the middle the hospitals world knowledge is represented. The knowledge is used by the planning component, which makes use of this data to plan the patient, nurse and administrative flows or deadlines.

The patient flow is made of process fragments. A composition step occurs every day and is triggered by the decisions of the doctors. The patient flows get executed and generate tasks for the nurse flows. The constraints of the nurse flows are generated by the planning component, which is scheduling all the resources. The executions of both types of flows change the state of the world.

The planning needs to be done in an iterative way, since the knowledge of the world changes due to some unexpected events. Unexpected events are very common in the hospital domain and appear quite often.

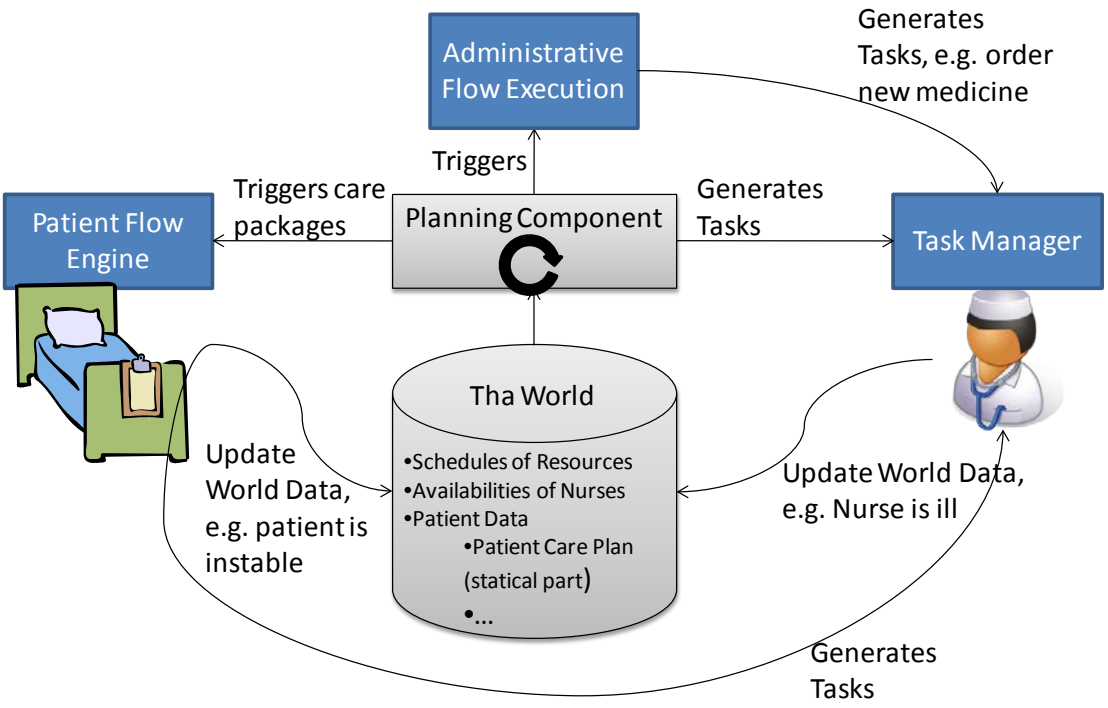

Figure 9: Overview of Healthcare Flows and Data

3.2 Patient Flows

The patient flow should document all the activities that are executed in favor of the patients healing process. The patient flow is made up of some treatment procedures, called care packages. The care packages prescribe a certain ordering of activities for each care package. The patient flow for each patient is computed every day and defines a composition of some care-package procedures and some resting periods for the patient.

The patient flow does not represent the activities a patient has to perform, but rather the activities that need to be done by the hospital staff to foster a optimal healing process.

In Figure 10 a process fragment for the daily weak-up routine is depicted. These activities should be performed by one nurse, but can also be distributed among the nurses. The overall morning routine should not take more than one hour, but some of the activities, e.g. update curve with vital signs do not have a high priority.

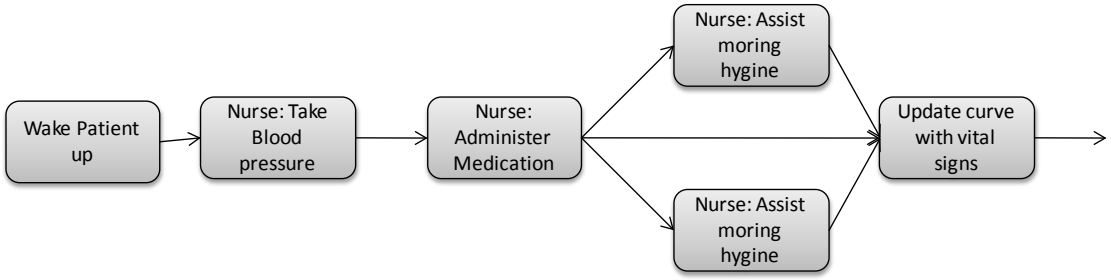

Figure 10: Daily Routine – Care Package

In Figure 11 a process fragment is depicted, which is needed to document the resting time for a patient in between the care package executions. Therefore the process fragment consists of one activity, representing the resting of the patient. Resting for the patient might either be explicitly planned by the planning component or it might be used to tamp the times in between care packages, which may vary in their time needed for their execution.

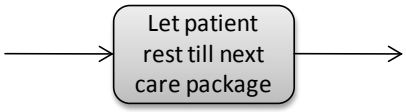

Figure 11: Intermediate Process Fragment to Represent the Resting Time of a Patient

Figure 12 depicts a small process fragment, which allows the patient to be prepared by a nurse for the operation. This action is modelled independently from the actual process fragment for the operation, because the preparation needs not do be done directly before the actual operation procedure, e.g. there might be some waiting time for the patient in-between.

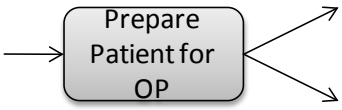

Figure 12: Process Fragment. Prepare Patient for OP

The actual operation procedure is represented in Figure 13. Two nurses, a doctor and an anaesthetist are orchestrated by this process fragment. Each of the in-

volved persons knows what they need to do. The exact interactions between the collaborators and their exact actions are not depicted here, since they are highly context depending and therefore vary in the way they are combined. But, the doctors and nurses none the less will follow some routines and protocols, which are always necessary, if people work together. The detailed operation routines could also be represented as process fragments and selected and composed depending on context information.

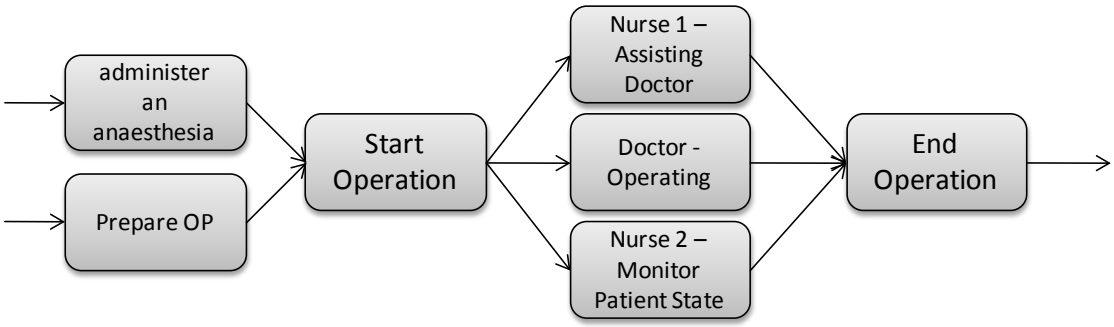

Figure 13: Operation Care Package

The process fragment depicted in Figure 14 depicts the care package for screening. This procedure should be performed within one hour, to enable an optimal capacity utilization of the laboratory.

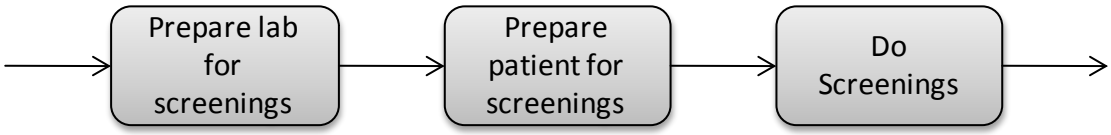

Figure 14: Screenings Care Package

The process fragments are selected and composed according to the care packages that are proposed by the doctor.

3.3 Nurse Flows

In this example the nurse flows represent a view of the patient flows filtered for a particular nurse i.e. the nurse flow contains all task which have to be executed by the nurse at a specific point in time and additional ordering information. This corresponds to the concept of person-centric flows presented above. In the following

we show a brief example how the person-centric flow is generated for a particular nurse.

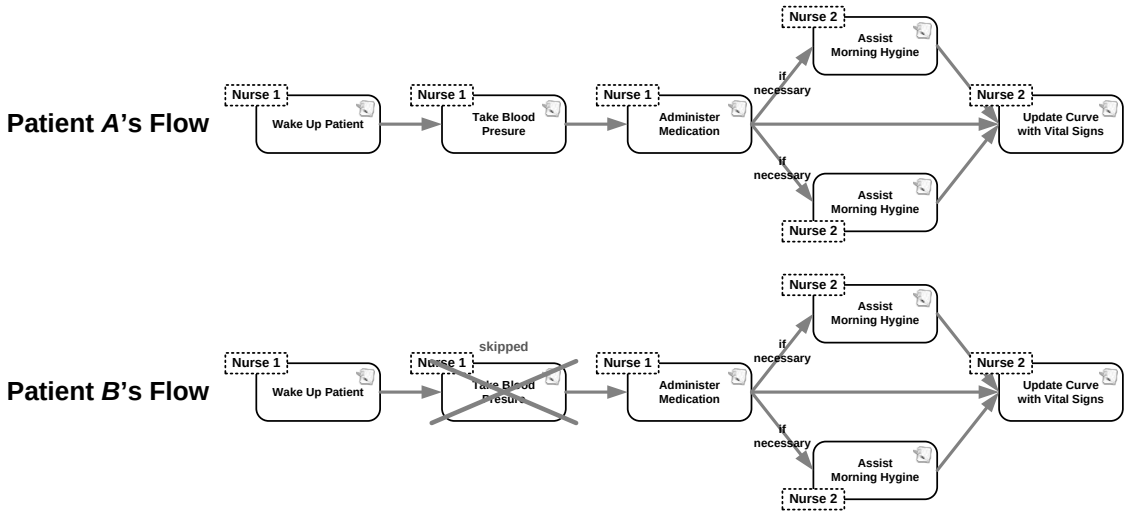

Figure 15: Instances of two patient flows

Figure 15 shows to instances of the daily care patient flow presented in Figure 10. The two instances involves to patients (A and B) and to nurses (1 and 2). The dashed boxes represent the staff assignment i.e. the respective tasks have to be executed by the nurse or the nurse has to support the patient.

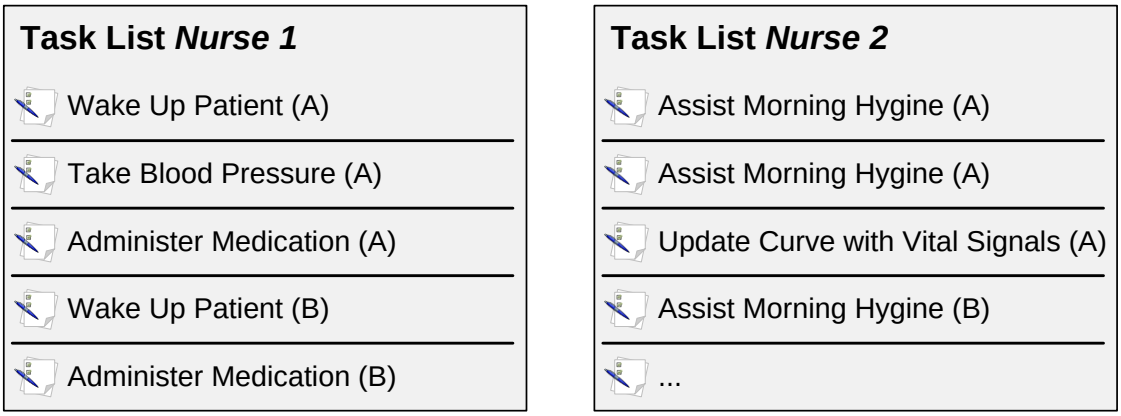

Figure 16: Task lists of nurse 1 and nurse 2

Figure 16 shows the task lists of nurse 1 and nurse 2 which originate from the execution of the patient flows. In the following we will focus on the tasks and person-centric flow of nurse 1. Figure 17 shows the whole process of generating the person-centric flow. Nurse 1 has decided to wake up the patients together. After

merging the two tasks the refined task list only contains the merged task. Same applies to the task which the nurse splits up into subtasks. Here again, the refined task list only contains the subtasks instead of the root task. The lower part of Figure 17 shows the recommendation and control graph which is automatically generated by a set of domain-specific algorithms. Afterwards, a task list visualization of the recommendation and control graph is presented to the nurse. In this example arrows and prohibition signs are used to provide the nurse with recommendations.

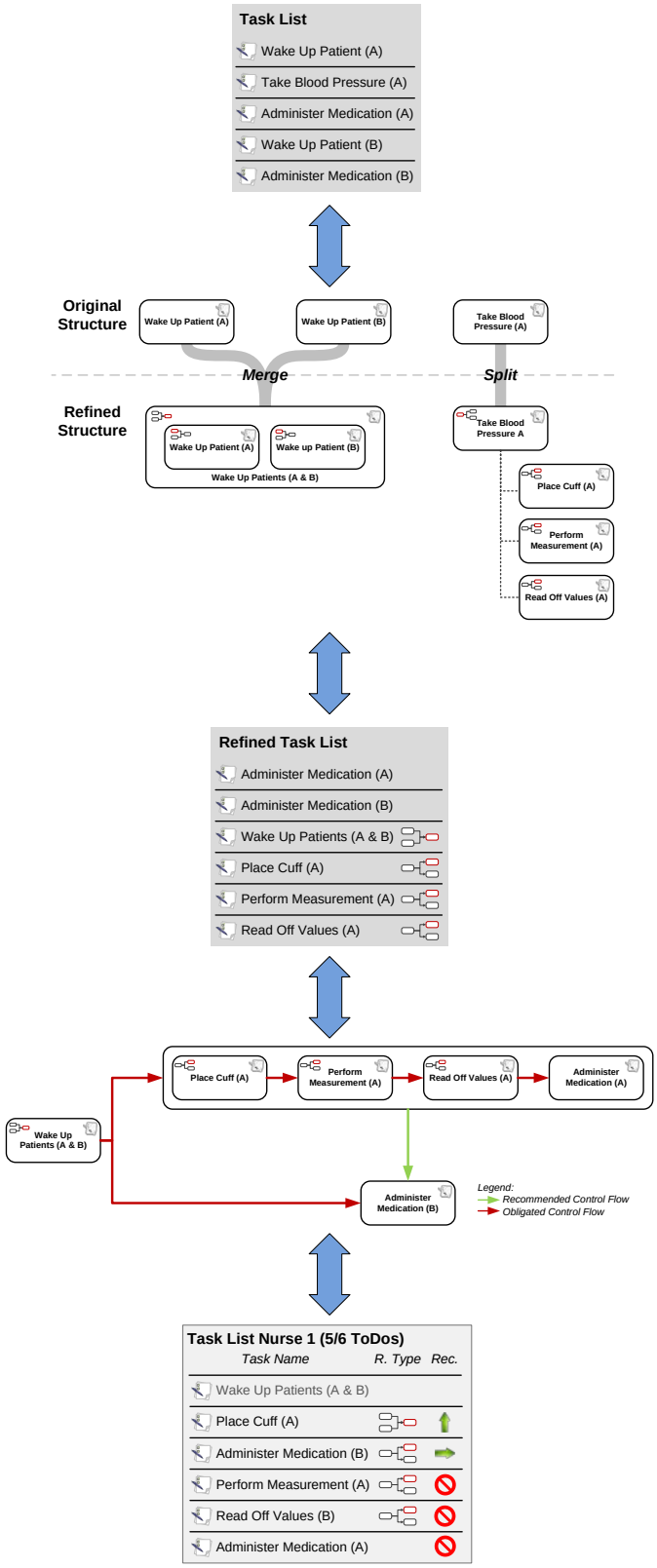

Figure 17: Generation of nurse 1's person-centric flow

4 Conclusion

In this deliverable provided a summary of the elaborated flow languages and presented the new concept of person-centric flows. We identified classification criteria for the flow languages with regard to the flexibility means of each language.

Based on the healthcare scenario we showed how the two approaches of process fragments and person-centric flows can be applied in an integrated way, each language solving different functional aspects of the scenario.

5 References

[1] ALLOW Consortium. Grant Agreement Annex I - Description of Work. (2007)

[2] ALLOW Consortium. Deliverable AD.1 (2008), http://www.allow-project.eu/

[3] ALLOW Consortium. Deliverable D 6.1 (2008), http://www.allow-project.eu/

[4] ALLOW Consortium. Deliverable D2.1 (2008), http://www.allow-project.eu/

[5] ALLOW Consortium. Deliverable D5.1 (2008), http://www.allow-project.eu/

[6] ALLOW Consortium. Deliverable D5.2 (2009), http://www.allow-project.eu/

[7] Eberle, Hanna; Leymann, Frank; Schleicher, Daniel; Schumm, David; Unger, Tobias: Process Fragment Composition Operations. In: Proceedings of APSCC 2010.

[8] Eberle, Hanna; Leymann, Frank; Unger, Tobias: Transactional Process Fragments - Recovery Strategies for Flexible Workflows with Process Fragments. In: Proceedings of APSCC 2010.

[9] Eberle, Hanna; Unger, Tobias; Leymann, Frank: Process Fragments. In: Meersman, R. (Hrsg); Dillon, T. (Hrsg); Herrero, P. (Hrsg): On the Move to Meaningful Internet Systems: OTM 2009, Part I.

[10] Unger, Tobias; Eberle, Hanna; Leymann, Frank; Wagner, Sebastian: An Event-model for Constraint-based Person-centric Flows. In: Proceedings of the 2010 International Conference on Progress in Informatics and Computing (PIC-2010).

[11] Unger, Tobias; Roller, Dieter: Applying Processes for User-driven Refinement of People Activities. In: Proceedings of the 14th IEEE International EDOC Conference (EDOC 2010).

[12] Leymann, Frank; Unger, Tobias; Wagner, Sebastian: On designing a people-oriented constraint-based workflow language. In: Gierds, Christian (Hrsg); Sürmeli, Jan (Hrsg): Proceedings of the 2nd Central-European Workshop on Services and their Composition, ZEUS 2010, Berlin, Germany, February 25--26, 2010.

[13] Unger, Tobias; Eberle, Hanna; Leymann, Frank: Research challenges on person-centric flows. In: Gierds, Christian (Hrsg); Sürmeli, Jan (Hrsg): Proceed-

ings of the 2nd Central-European Workshop on Services and their Composition, ZEUS 2010, Berlin, Germany, February 25--26, 2010.

[14] Wagner, Sebastian: A Concept of Human-oriented Workflows. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Diploma Thesis No. 2987 (2010).

[15] Kai S. Kunze, Florian Wagner, Ersun Kartal, Ernesto Morales Kluge, Paul Lukowicz: Does Context Matter? - A Quantitative Evaluation in a Real World Maintenance Scenario. Pervasive 2009: 372-389

[16] Gerd Kortuem, Fahim Kawsar, Bashar Altakrouri: Flow-driven ambient guidance. PerCom Workshops 2010: 796-799

[17] M. Pesic, "Constraint-based workflow management systems: Shifting control to users." Ph.D. dissertation, Eindhoven University of Technology, 2008.

[18] J. F. Allen, "Maintaining knowledge about temporal intervals," Commun. ACM, vol. 26, no. 11, pp. 832–843, 1983.

[19] M. Pesic, M. H. Schonenberg, N. Sidorova, and W. M. P. van der Aalst, "Constraint-based workflow models: Change made easy," in OTM Conferences (1), 2007.

[20] R. Han, Y. Liu, L. Wen, and J. Wang, "A Two-Stage Probabilistic Approach to Manage Personal Worklist in Workflow Management Systems," in OTM Conferences (1), 2009, pp. 24–41.

[21] J. Petzold, F. Bagci, W. Trumler, and T. Ungerer, "Comparison of different methods for next location prediction," in Euro-Par, 2006.

[22] M. B. Vilain and H. A. Kautz, "Constraint propagation algorithms for temporal reasoning," in AAAI, 1986, pp. 377–382.

[23] P. van Beek, "Exact and approximate reasoning about qualitative temporal relations," Ph.D. dissertation, 1990.