Support Tasks for Pervasive Workflows

Srdjan Marinovic[1], Tobias Unger[2],
Naranker Dulay[1], and Frank Leymann[2]

[1] Department of Computing, Imperial College London, UK
[2] IAAS, University of Stuttgart, Germany
{srdjan,n.dulay}@imperial.ac.uk
{tobias.unger,leymann}@iaas.uni-stuttgart.de

Abstract. Pervasive workflows focus on encoding human-centric processes in pervasive domains, such as patient care. One of the main characteristics of these processes is that they have a number of support tasks, which are recommended but not mandatory. However support tasks' initiations are dependent not only on intra-workflow events, but mainly on inter-workflow and external contextual events. Current workflow models, however, do not address the issue of encoding task instantiations as a result of narrative of contextual events, nor do they support the notion of sharing a task between different processes.
This paper presents a language for encoding support tasks initiations based on $\mathcal{C}+$. Our approach is divorced from any particular workflow model and can be used to complement the current workflow models. Furthermore, we also introduce two support task modalities, should and must, to differentiate between their urgencies. However, based on the context these modalities can change dynamically and are not static like the concept of a task.

Keywords: Pervasive workflows, People-intensive processes, Adaptive and context-aware processes, Support Tasks, Event-based systems

1 Introduction

Workflow models and languages have been successfully applied in encoding and enacting business processes and service compositions. Their defining characteristic is that they tend to be well defined and understood. Workflow languages and models can be roughly split into two groups: (1) *structured workflows* where the structure of the workflow process is well defined and thus it is used to implicitly drive the execution, and (2) *constraint-based workflows* where the structure is not the dominant driving force but the inter-task constraints are well defined and understood and thus they are the driving force. Pervasive workflow models have simply adopted these two approaches to encode processes for pervasive domains [20].

Both structured and constraint-based workflows tend to view a workflow as set of tasks, such that task executions are not shared between a group of different workflows, in other words the task constraints are defined only over the

intra-workflow dependencies. These assumptions are well suited for the business and service domains, however in pervasive domains where workflows attempt to encode human-centric processes, these assumptions can rarely be made.

One of the main characteristics of human-centric processes is that they have a number of assistance tasks, which are recommended but not mandatory, whose initiation is dependent not only on intra-workflow events, but also on inter-workflow and external contextual events, and furthermore can depend *solely* on these external events. For example, in patient care tasks such as: administrating drugs and prescribed treatments, and obtaining consents are tasks that are strictly needed to ensure that the patient is taken care of. However, during the execution of these mandatory tasks, it is also desirable to change the bedding every so often, clean the medical instruments, check the nutrition intake and so forth. These tasks are not mandatory but overall their completion contributes towards a *better* care.

In this paper we introduce the term *support* task to refer to tasks that do not directly contribute to the workflow's goal but contribute towards a better quality of the service provided by that workflow or a group of workflows. Current workflow models do have a notion of an "optional" task which is a task that may be omitted due to different paths a workflow may take towards its completion. In this sense support tasks subsume this notion, and extend it further by allowing a support task to be shared between workflows, to be instantiated through a wider range of events (constraints) and finally insisting that every support task is instantiated with a certain modality. Where a modality indicates what the consequences of ignoring a support task are. Optional tasks do not have this notion since ignoring them means that the workflow will take another path to its completion, while in the case of a more general support task, a user needs to be aware of how much the quality of the achieved goal will suffer.

One candidate for encoding support tasks within workflows would be constraint-based workflows [13,16], where the constraints are specified over the intra-workflow task dependencies. However even if the constraints were extended, the underlying semantics of these languages, based on LTL, would still make encoding support tasks' constraints (such as the ones presented in Section II) much harder and less intuitive than the approach this paper advocates. Another appropriate candidate are Event-Condition-Action (ECA) rules-based workflows [10, 11]. These models are based on "traditional" ECA rules as found in active databases [14]. This paradigm considers each rule as an independent entity that can initiate an action as soon as the correct event narrative is recognised and the rules do not have a semantic model to encode how and which states are persisted in the absence of events. This makes traditional ECA rules a low-level enforcement tool rather than a high-level language which is needed to capture the support task executions.

This paper's contribution is to advocate the $\mathcal{C}+$ language [9] as a syntactic and semantic model for encoding conditional initiation and termination of support tasks. However instead of proposing yet another formalism for specifying workflows, we see this language as an add-on to the current workflow models thus supplementing them to in order to cater for the specific needs of support

tasks. The language $\mathcal{C}+$ is an AI formalism for reasoning about the causal effects of actions on the system's states. It has a very simple rule-based syntax that can easily and intuitively capture various instantiation conditions and modalities. We further propose two base modalities *should* and *must* for support tasks to encode their urgencies. $\mathcal{C}+$ is used to encode how various event narratives initiate and terminate support tasks and also how they influence their modalities.

In summary the contributions of this paper are primarily twofold: **(1)** Defining an extended notion of a "non-compulsory" task called "support" tasks for use in pervasive workflows, and **(2)** Proposing a formal model using the language $\mathcal{C}+$ to encode the enactment of support task in a more flexible way than current workflow models provide.

2 Case Study: Pervasive Patient Care

Before we give a more detailed definition of a support task, we would like to illustrate their defining characteristics in pervasive domains with focus on patient care in a hospital's surgical ward and in particular we draw upon the investigations done within the ALLOW project (funded by the EU FP7 programme) for the Surgical Hospital in Hannover. As a running example, we shall use the workflow depicted in the Figure 1.

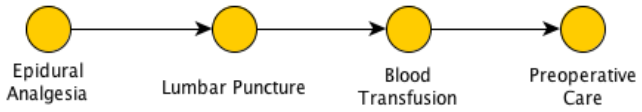

Fig. 1. Patient's Workflow of Nursing Procedures for a Day Shift

The presented workflow is done either by the patient's primary nurse or an assigned nurse and it schedules four medical procedures to be done within the current shift:

1. *Epidural Analgesia* – is a pain management procedure which consists of administrating analgesics into the epidural space of the spinal cord. A nurse's role is to administrate the drug, such as diamorphine, and monitor for any allergic or unexpected effects.
2. *Lumbar Puncture* – a procedure which involves withdrawing cerebrospinal fluid. The nurse's role includes preparing the patient, and doing the follow-up monitoring.
3. *Blood Transfusion* – a nurse's role consists of checking that the patient's record and the blood are in accord, and setting up the apparatus.
4. *Preoperative Care* – a nurse's role in this procedure consists of skin preparation, checks of various forms and consents that need to be signed.

This is an example of one workflow focused on a particular patient, but the patient can have many other similar workflows given to patient's doctor or other care givers. However simply carrying out these medical procedures (i.e. executing the given workflow) without recognising the "humanity" of a patient is not nursing [8]. It is additional *little* things, such as: cleaning and helping with the patient's hygiene, that have been described as having a profound effect for recovery and, indeed, for cure. In addition to these the Hannover hospital also insists on the following normative regulations regarding the patient care:

1. *A nurse **should** take a photo after every procedure on patients with flesh wounds. This **must** be done at least once per day for patients with open flesh wounds.*
2. *A nurse **must** use appropriate masks, gloves and gowns if dealing with patients with highly-infectious diseases.*
3. *A nurse **should** check daily the patient's weight and food in-take.*
4. *A nurse **should** update a patient's record after all drug injections and blood transfusions, recording any side effects. This **must** be done before the start of a new shift.*

We hope that this subset of regulations gives a feel for the kind of tasks that pervade every patient-care workflow. It is precisely these kind of tasks that we consider as *support* tasks since their execution is not tied into a particular *goal* oriented workflow such as the one presented in the Figure 1. or any other patient care workflows. But clearly there is a strong dependency between these tasks and all other patient care workflows.

3 Support Tasks and Norms

The previous section attempted to descriptively introduce a notion a support task and this section gives a more precise definition of such tasks and their modalities. We also provide an overview of how support tasks can be used with existing workflow engines. First we introduce some preliminary definitions regarding workflows and their goals that will help us clearly mark the difference between support tasks and typical workflow tasks.

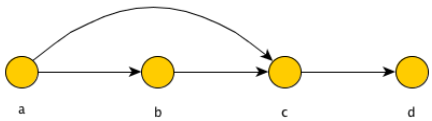

Fig. 2. A simple workflow

Definition 1 *A workflow instance w is represented by a set S_P^w of workflow paths P_w where each path is a finite sequence of tasks.*

Definition 2 *A workflow path P_w is defined by a workflow goal G_w that it leads to. It follows that a workflow instance has a set of goals S_G^w that it can achieve.*

Clearly different paths may share some of their tasks and they may also lead to a same goal.

Definition 3 *A workflow w's task is a goal task if its removal from all the paths that contain it, prevents the workflow from reaching the goals defined by those paths.*

Definition 4 *A support task is a workflow task such that its removal from all the paths that contain it, does not prevent the workflow from reaching the goals defined by those paths.*

The workflow depicted in Figure 2 consists of two paths namely $\{a, b, c, d\}$ and $\{a, c, d\}$ and let us take the execution of the task d as the goal of the workflow. In this case skipping the task b does not influence the reachability of the goal but it can influence the way the goal has been reached and thus the overall quality of the service provided by the workflow. Hence in this example b is considered to be a support task. However if the two paths had different goals then b would be a goal task instead. Viewed in this light support tasks share a lot of with *optional* or *non-compulsory* tasks as found in both structured and constrained-based workflows.

However as presented in the case study the support tasks are instantiated based on the narrative of both contextual events and workflow events (describing a workflow's execution progress), and they are tasks that can be executed as part of any workflow whose goal is "enhanced" through the execution of that support task. It is important to notice that once executed, a particular support task may not have to to be re-executed in all the workflows that share the same goal. In other words one instance of a task can be shared by a group of workflows. This is most clearly seen in the patient-care domain where the goal is to treat the patient and as such the patient is the focus of many workflows, therefore changing the bed could be done in any workflow instance focused on that patient. Thus the main difference between optional tasks and support tasks lies not in their "non-compulsory" nature which they share, but in the way they are instantiated and attached to workflows. We can summarise these differences as:

1. ***Instantiation*** – Optional tasks are not instantiated by constraint-based workflow models. In structured workflow models they are instantiated only when a particular workflow execution point has been reached (e.g. an *if* or *parallel* construct. On the other hand, support tasks are instantiated by a narrative of context and workflow events, not necessarily tied to a fixed execution point.
2. ***Attachment*** – Optional tasks are treated as typical workflow tasks belonging to a particular workflow instance. On the other hand support tasks are usually not confined to a workflow but to a set of workflows.

3. ***Modality*** – Each support task has an associated modality indicating how it effects the quality of the goal. There is no such notion for optional tasks. We expand on the modalities in the next sub-section.

For these three reasons we have chosen to introduce the term "support" task to highlight the differences with the established notions underpinning the "non-compulsory" tasks found in structured and constraint-based workflows. We further introduce a notion of a *support norm* which, intuitively, represents a policy that specifies under which conditions a support task is either instantiated or terminated and what modality the support task has once instantiated. For example, in the presented use-case, each regulation can be considered a support norm.

3.1 Support Task Modalities

This paper uses the term modality as an indicator of what the effects are of ignoring a particular instantiated support task. It then follows that whenever a support task is instantiated it must always be done so with a certain modality. In this paper, we have identified two distinct modalities for support tasks' executions, namely: *should*, and *must*. The Table 1 summarises the differences between their potential effects and thus it gives semantic interpretations of the modalities with respect to the quality of the provided service.

Table 1. *Task modalities and the effects of delaying/ignoring those support tasks.*

Modality	**Ignoring/Delaying**
Should	No effect
Must	Negative effect

Ignoring or delaying a *should* task will have no effect on the quality of the service, but ignoring the *must* task has a negative effect on the quality. The negative effect means that the quality achieved solely by the goal tasks will be further devalued by this ignorance. Please note that the goal can still be achieved even if all the *must* and *should* tasks are ignored. Clearly the number of modalities and their quantification is highly domain specific, but we feel that the mentioned two modalities can be taken as core modalities which can be further specialised and their effects quantified in a particular domain.

4 Architecture Overview

As it has been aforementioned, support norms are not meant to replace a particular workflow language but rather to supplement them by providing a way to encode their initiations which, as argued, are not easily captured with the workflow languages. Following this ethos we propose that this separation is extended into the implemented system's architecture, as depicted in Fig. 3.

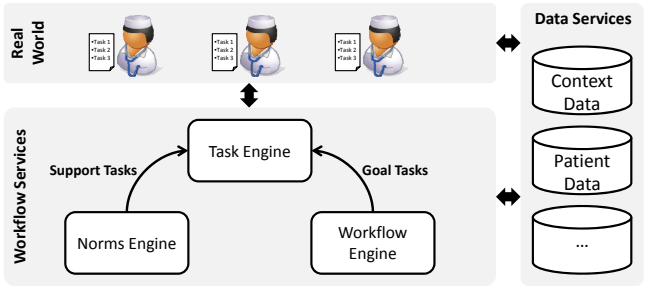

Fig. 3. System Architecture

The Workflow Services in Figure 3 essentially partitions support tasks' and goal tasks' initiations between the Norms engine and the Workflow engine respectively. The Workflow Engine drives the execution of workflows defined through a workflow language, while the Norms Engine collects the events from the environment (as well as from the Workflow Engine) and initiates or terminates support tasks. Both of these engines notify the Task Engine about which tasks have been initiated. The Task Engine is a separate, standalone component supporting creation and management of tasks carried out by a person. Therefore the Task Engine serves as a broker between tasks emerging from workflows and users performing these tasks. The Task Engine is provided with task execution context which further contains the workflow identifier (of which the task is part of), the role or the user for which the task is intended and possibly other information. This contextual information is used by the Task Engine to schedule and notify users about the new tasks that have become available.

Following the presented case study, the Data Services in the Figure 3 represent knowledge bases containing patient data, hospital records and environment data. The environment data is the contextual data containing information such as physical actions that nurses are doing and what the state of the environment is. The physical actions are either recorded manually for example by ticking the actions on a mobile device or recorded automatically by a context recognition system (cf. [20]). Beside the nurses' actions the system can also capture additional information such as heart monitoring and so forth.

In our current prototype implementation we have used the iCCalc tool[3] together with Yap Prolog version 6[4] (both of which are publicly available) as our Norms Engine. The evaluation of a particular norm can result in different events in the system. If a nurse has checked the patient's weight only once, a support task is instantiated at the Task Engine telling that nurse that the patient's weight should be checked at a later time as well. The evaluation can also result in a change of a task's modality. For example whenever the nurse conducts a procedure on a patient with a flesh wound, the support task "take a photo" is instantiated with modality *should*. If the system detects that the nurse conduct's

[3] http://www.doc.ic.ac.uk/~rac101/iccalc/
[4] http://www.dcc.fc.up.pt/~vsc/Yap/

the last procedure on that patient on this day and still has not taken a photo of the wound, the modality of the task is changed to *must*. The next two sections will detail how the norms are encoded and evaluated.

For the implementation of the Workflow engine, we use the Apache ODE[5] open source engine. In our current prototype all contextual events are simulated and in for our future work within the Allow project, we are currently assessing how our current prototype implementation can be deployed and tested in the *real world* environment.

5 Specifying Support Norms: Syntax and Semantics

Management of support tasks' executions is encoded as a set of support norms. In essence the norms are declarative rules that govern the Norms Engine's interaction with the Task Engine and this interaction can be summarised as two operations: add a support task, and remove a support task. However, it is often useful to specify integrity rules as well to indicate which support tasks (and with which modality) cannot be initiated at the same time. Therefore we define three types of support norms, namely:

- ***Initiation Norms*** – Specify under which conditions a support task is initiated, i.e. added to the Task Engine.
- ***Termination Norms*** – Specify under which conditions a support task is terminated, i.e. removed from the Task Engine.

Clearly support norms will be based on the context information and workflows of the application. Therefore we need a notion of a ***system state***, which intuitively, represents a snapshot of the context and workflow execution information.

To formally capture a system state, we use a *multi-valued propositional signature* σ, which is a set of *constants*, where each constant has a non-empty set $dom(c)$ of values called the *domain* of c. An atom of a signature σ is an expression of a form $c = v$ (c *has value* v) such that $c \in \sigma$ and $v \in dom(c)$. A boolean constant has $\{true, false\}$ as its domain. We write $\neg c$ for $c = false$, and c for $c = true$. A *formula* ψ of σ is any propositional formula of atoms of σ. An interpretation of a σ is a function that maps every constant c to a value in $dom(c)$. An interpretation, $\mathcal{I}$ satisfies an atom $c = v$ if $\mathcal{I}(c) = v$, written as $\mathcal{I} \models c = v$. The satisfaction relation is extended to formulae of σ according to the usual truth tables for propositional connectives. $\mathcal{I}(\sigma)$ stands for the set of all interpretations of σ. Please note that two interpretations that assign the same values for all σ's constants are indistinguishable and in fact represent the same state.

It follows that the Norms Engine needs to determine which state the system is in (i.e. form an interpretation) and simply evaluate norms against it. We will freely employ variables which have a finite domain when specifying constants. So

[5] http://ode.apache.org/

for example to say that a system state contains constants that describe locations of all nurses we can simply say:

$$X \in \{nurseA, nurseB\}$$
$$dom(location(X)) = \{ward, reception, operating\_theatre\}$$

Furthermore, in order to enable the Norms Engine to correctly relate event occurrences in the environment to the new system state (that the system may have moved to), we need to specify ***Domain rules***. Intuitively a domain rule specifies how system constants change, from one state to another, in response to events and actions in the environment. These three aspects: support norms, system states' signature, and domain rules define all the necessary information that the Norms Engine needs in order to dynamically initiate and terminate support tasks and to this end we define a support policy as:

Definition 5 *A support policy P is a triple $(\pi_P, \sigma_P, \delta_P)$ where π_P is a finite set of support norms, σ_P is a multi-valued propositional signature, and δ_P is a finite set of domain rules.*

Definition 6 *Support policy P's signature σ_P is defined as: $\sigma_P^s \cup \sigma_P^d \cup \sigma_P^a$ where all three subsets are disjoint.*

σ_P^a contains constants that describe various actions and events that the Norms Engine is subscribed to. For example constant $start(Ag, T, W)$ says that an event has been fired to say that agent Ag has started executing task T within the workflow W. σ_P^d contains constants that capture values of the domain-specific properties, such as locations of nurses, or historic information regarding workflow executions such as $done(T, W)$ which specifies which tasks have been executed within workflow W. Both of these sets are defined by the policy writer and are quite domain specific.

On the other hand the constant set σ_P^s is a fixed set consisting of two constants: $should(Ag, T_s, W)$ and $must(Ag, T_s, W)$. These two constants are used to convey which support tasks T_s are initiated/terminated for an agent Ag, and to which workflow they are attached to. The notion of an "agent" is not necessarily linked to a particular person, it can represent a role or a group.

The syntax for support norms π and domain rules δ, as well as the semantics for automated evaluation of these norms and rules over the system's states (given by σ) will be underpinned by the $\mathcal{C}+$ language [9]. $\mathcal{C}+$ falls within the umbrella of action languages and is used to formulate and describe effects of actions and events on fluents, where a fluent represents a state of a particular property in the system that is being modelled. An appealing aspect of $\mathcal{C}+$ is that it has a very intuitive syntax to specify *causal laws* which describe how fluents are changed by a narrative of events and actions. Its semantics can be given in two equivalent ways: 1) *Descriptive* – by defining a labelled transition system (LTS) to capture how the fluents change [4] , and 2) *Executable* – by translating the language into a set of propositional sentences [9]. We also refer a reader to a very good and succinct introduction to $\mathcal{C}+$ as given in [5].

Definition 7 *A $\langle S, A, R\rangle$ is said to be an interpretation of P's σ_P iff:*

- *S – a set of system states, where each state is an interpretation of $\sigma^s \cup \sigma^d$.*
- *A – a set of transition events where $A = \mathcal{I}(\sigma^a)$.*
- *R – a set of labelled transitions; $R \subseteq S \times A \times S$.*

This description of an LTS over σ_P matches the definition of an LTS given by $\mathcal{C}+$ over the same multi-valued signature (as presented in [4]).

An event ϵ is executable in the state s when there is a transition (s_i, ϵ, s_{i+1}). A *path/run* of length m of the $\langle S, A, R\rangle$ is a sequence $s_0\epsilon_0 s_1 ... s_{m-1}\epsilon_{m-1}s_m$, such that $m \geq 0$ and $(s_i, \epsilon, s_{i+1}) \in R$ for $i \in 1..m$. Thus given a current systen state s the Norms Engine semantically interprets the σ^s constants in the following way:

- $s \models should(Ag, T_s, W)$ – the support task T_s (attached to W) is initiated for an agent Ag with the *should* modality.
- $s \models must(Ag, T_s, W)$ – the support task T_s (attached to W) is initiated for the agent Ag with the *must* modality.
- $s \models \neg(should(Ag, T_s, W) \vee must(Ag, T_s, W))$ – the support task T_s (attached to W) is *not* initiated, in other words it is terminated.

The Norms Engine needs to be given one LTS defined over σ_P, which will provide a formal semantics (both descriptive and executable) for a support policy P. We refer to this preferred LTS as a model of P. However before formally defining a model for a support policy P, we introduce the syntax and the semantics behind the support norms and domain rules.

Definition 8 *An initiation/termination support norm is defined as:*

$$\textbf{\textit{caused}}\ [\neg][should/must](Ag, T_s, W)\ \textbf{\textit{if}}\ \psi\ \textbf{\textit{after}}\ \phi$$

where ψ is a formula of signature $\sigma^d \cup \sigma^s$; ϕ is a formula of signature $\sigma^d \cup \sigma^s \cup \sigma^a$.

If the σ^s's atom is preceded by a $\neg$ sign we refer to it as a termination norm, and if not then it is an initiation norm. Intuitively, the constant *should/must* is caused to be set/unset in a state s_{i+1}, after any transition (s_i, α, s_{i+1}) such that $s_{i+1} \models \psi$ and $s_i \cup \alpha \models \phi$. If ϕ is omitted, T (truth constant) is added implicitly to indicate that the norm is applied after any transition.

Support norms have a 1-to-1 correspondence with $\mathcal{C}+$'s dynamic laws in both the syntax and the semantics. Furthermore all support constants are *inertial* which means that after following domain rules are automatically added to every support policy:

caused $should(Ag, T_s, W)$ **if** $should(Ag, T_s, W)$ **after** $should(Ag, T_s, W)$

caused $\neg should(Ag, T_s, W)$ **if** $\neg should(Ag, T_s, W)$ **after** $\neg should(Ag, T_s, W)$

and similarly for the *must* constant. Intuitively these laws indirectly persist an initiated or terminated task unless they are explicitly changed by some transition. This represents a form of non-monotonic reasoning since *by default* all tasks' initiations/terminations continue persisting until known otherwise. This is clearly

a very useful property to have since this means that the policy writer does not have to burden himself of making sure that initiated tasks remain initiated until terminated, this is now implicit.

Given this syntax we can encode the Regulation 4 from our Case Study through the following norms:

caused $should(N, update\_record(P), W_P)$ **if** $done(N, T, W_P) \wedge drug\_based(T)$
caused $must(N, update\_record(P), W_P)$ **if** $\neg inShift(N)$ **after** $should(N, update\_record(P), W_P)$
caused $\neg[should/must](N, update\_record(P), W_P)$ **if** $done(N, update\_record(P), W_P)$

The first rule can be read as: a support task *update_record* for a patient P is initiated for a nurse N and is attached to W_P if drug based task T is done within that patient's workflow or workflow groups. The second rule initiates the must modality for the *update_record* task if the nurse has finished her shift and she still hasn't competed the support task. The readings for other rules follow the similar interpretations. W_P can represent either a specific workflow or, as we have implemented it, a group of workflows that have the patient P as their focus. In this way any workflow that N is doing for P will have *update_record* as one of its tasks. Thus a support task does not have to have a "fixed" workflow attachment.

Notice that this encoding can leave the task $update\_record(P)$ initiated with both modalities since once the nurse is not in her shift the must modality is initiated but the should modality is not terminated. We could attempt to say that should modality is terminated by the nurse not being in her shift, but this would be a burden on the policy writer as he has to be careful when specifying conditions and especially when changing them. What the policy writer would really like to say is that the must modality overrides the should modality and we can simply add additional termination norm such as:

caused $\neg should(N, update\_record(P), W_P)$ **if** $must(N, update\_record(P), W_P)$

to capture this requirement. Notice that using the same norms we have added "integrity" constraints that prevent dual modality being initiated and we also specify which one is preferred. We can use a similar strategy to describe overriding between support tasks, which can be used to implement priority ordering of tasks. For example let us imagine that in the presented case study we would like to instruct the nurse to take photos of a patient's flesh wounds (constant a) before the checking of the patient's food in-take (constant b). This can be encoded using:

caused $should(N, a(P), W_P)$ **if** $hasFleshWound(P) \wedge \neg done(a(P), W_P)$
caused $should(N, b(P), W_P)$ **if** $\neg done(b(P), W_P) \wedge \neg overriden(b(P), W_P)$
caused $overriden(b(P), W_P)$ **if** $should(N, a(P), W_P)$
caused $\neg overriden(b(P), W_P)$ **if** $\neg should(N, a(P), W_P)$

Also notice that in these rules even though a support task is initiated for all nurses, if only one of them needs to do it, as opposed to the *update_record* task. However we can change this to say that two nurses need to check the food in-take. To do this we will use domain rules to further refine the $done(b(P), W_P$ constant.

Definition 9 *A domain rule is defined as:*

$$\textbf{\textit{caused}}\ c = v\ \textbf{\textit{if}}\ \psi\ \textbf{\textit{after}}\ \phi$$

where $c \in \sigma^d$ *and* $v \in dom(c)$*;* ψ *is a formula of signature* $\sigma^d \cup \sigma^s$*;* ϕ *is a formula of signature* $\sigma^d \cup \sigma^s \cup \sigma^a$*.*

The semantics follow the same idea as the ones for support norms and we also make all domain constants inertial as well. And returning to our example we simply say:

caused $finished(N_1, b(P), W_P)$ **if** $\top$ **after** $finishes(N_1, b(P), W_P)$

caused $done(b(P), W_P)$ **if** $finished(N_1, b(P), W_P)$ **after** $finishes(N_2, t) \wedge N_2 \neq N_1$

A policy writer may wish add additional "meta" norms, that manage global aspects of how tasks are terminated or initiated. For example it may be the case that a *should* support task is terminated after a workflow to which it is attached has been finished. These kind of concerns can also be captured with a terminate support norm in the following way:

$$\textbf{caused}\ \neg should(Ag, T, W)\ \textbf{if}\ finished(W) \wedge attached(T, W)$$

But this will persist any support tasks initiated with the must modality if they are still waiting execution.

As we have noted at the beginning of this section many LTSs can be defined over a support policy P's signature σ_P. However, we would like to define an LTS where all transitions follow the described meaning of support norms π and domain rules δ. We refer to such an LTS as the *model* M_P of the support policy P. For a support policy P the following sets can be defined:

$$E(s, \epsilon, s') =_{def} \{c = v \mid c = v \textit{ is a head in } \pi_P \cup \delta_P, s' \models \psi, s \cup \epsilon \models \phi\}$$

Definition 10 *Given a support policy* P*, an interpretation* s *is a* state *of* M_P *iff* s *is an interpretation of* $\sigma_P^{s \cup d}$*.*

Definition 11 *Given a support policy* P*,* (s, ϵ, s') *is a* transition *of* M_P*, where* s *and* s' *are interpretations of* $\sigma^{s \cup d}$ *and* ϵ *of* σ^a*, iff* $s' \models E(s, \epsilon, s')$*.*

Lemma 1. *A support policy* P *has exactly one model* M_P*.*

Proof. First we observe all potential models must agree on all their states by definition 10. If there are no valid transitions defined by P, then it follows that there is only one M_P. However if there are valid transitions, then let us consider the case where there are two models M'_P and M''_P. By definition 11, any transition of M'_P must be part of M''_P and vice-versa, and as they both share all the states and transitions then it follows that $M'_P = M''_P$.

We take P's M_P to denote P's semantics. Furthermore we have noted that the support norms and domain rules are syntactically and semantically based $\mathcal{C}+$'s definite dynamic causal laws. A set of finite causal laws defined over some fixed σ_D is an action description D. Therefore a support policy P defines a unique definite action description D_P over its σ_P. As demonstrated in [4], every D defines an LTS over σ_D.

Proposition 1. *Given a support policy P and its definite action description D_P, then M_P and the LTS defined by the D_P are equivalent.*

We give a proof sketch due to the space constraints. The proof follows by showing that the definitions of states and transitions of M_P exactly match the definitions of transitions and states of the LTS defined by the D_P as given in [4], taking into account that D_P does not contain any static laws but only dynamic ones and where all fluents are inertial. iCCalc uses $\mathcal{C}+$ executional semantics, which are equivalent to $\mathcal{C}+$'s LTS semantics (and thus equivalent to M_P), thus allowing us to use this tool to drive the initiations and terminations of the support tasks.

Definition 12 *Given an M_P, a start state s_0, a narrative of events of type $\alpha_0, ..., \alpha_n$ a prediction query answers whether for every path $\omega = s_0\epsilon_0 s_1 ... s_n\epsilon_n s_{n+1}$ such that $\epsilon_i \models \alpha_i$ (for all $i \in \{0...n\}$) it follows that $s_{n+1} \models G$, where G is a formula over σ_P.*

This is a typical deduction query which is used by the Norms Engine (iCCalc) to manage the support tasks at run-time. After receiving an event or a set of concurrent events these events are added to the narrative and the prediction query determines which tasks are to be initiated and terminated in the new state.

5.1 Comparison With Rule-based and Constraint-based Workflows

Rule-based workflows [10,11] encode a workflow execution as a set of "traditional" ECA rules, as used in active databases [14], and they concentrate on encoding imperative workflows which have a well-defined structure given as an acyclic graph. However, the initiation and termination of support tasks does not have a well-defined structure and thus the ECA rules would need to be used to provide overriding of different task modalities and different support tasks, and overriding between concurrently initiated actions is precisely a feature that traditional ECA rules do not cater for. Furthermore correctly initiating and terminating inertial constants, like the example of multiple executions of the same task before it can be considered as finished, is hard and error-prone to encode with these ECA rules. Notice that a support policy's M_P, once constructed, can be encoded as a set of ECA rules. The ECAs would only have one action $Constant.set(Value)$ and for each one of M_P's transitions, there would be one rule with multiple parallel actions, where each action sets one value for a different constant which that constant is supposed to hold after the transition, in other words in a new state. Viewed in this light support norms and domain rules can be seen as a more suitable high-level language to express the interactions between events, actions and system states, and can be used to generate low-level ECA rules. Much as [10,11] use imperative structures to generate ECA rules as well.

Recent complex event processing languages, such as RuleML [3], extend the ECA concept with inference over a narrative of events and determine persistence of situations, and in this they share the same motivation as that underpinning $\mathcal{C}+$. To some extent they could also be used to encode support norms, but they

lack a clear way of encoding overriding between concurrently initiated actions, nor do they have well-defined declarative semantics like $\mathcal{C}+$.

Constraint-based workflows [13,16] use a set of constraints to express whether a certain narrative of task executions can be considered as acceptable to recognise a workflow as successfully completed. However, they do not initiate and terminate tasks, in other words they are not intended to drive the execution like support norms do. Also there does not seem to be a clear way to express overriding of support tasks through constraints and also to express that tasks are inertial. In other words once initiated they remain initiated until terminated or overridden. Similarly encoding the example domain rule would not be straight-forward if possible at all. But given that a policy's M_P is a standard LTS model these two approaches could be combined for the benefit of both by using the higher-level LTL-based language to check whether an M_P satisfies certain constraints.

6 Background Work

Constraint-based [13, 16] workflow systems share similar motivations to the work presented in this paper as they provide a declarative way to specify a workflow process. However, they are focused on specifying constraints over task dependencies rather than proactively initiating tasks and for this reason it is difficult to encode support norms like those presented in this paper. In our work there is no limitation that rules and constraints are only related to activities and activity states of a single workflow process. Furthermore [16] allows optional constraints to be violated without any consequences except the generation of a warning. From our perspective modality cannot be emulated by optional constraints. As discussed earlier we require that a task is instantiated and presented at the appropriate person's tasklist in order to provide them with freedom to chose whether to do it or not. In case of an optional constraint no task instance would be generated. Furthermore, our concept allows the change of the modality during the execution of a task. If mapping the modality to optional constraints, an adaptation of the workflow would be required. However, these two concepts can complement each other, and priorities can be used to further subdivide modalities.

Rule-based workflow systems [10, 11] appear syntactically to be very similar to the presented $\mathcal{C}+$ formalism, however semantically they are quite different. $\mathcal{C}+$ provides a well-defined semantics on how system states are persisted or changed in response to either other system states or various actions. ECA rules are traditionally focused on actions fired by independent events and thus expressing inter-dependencies between parallel actions requires additional semantic constructs. In this way $\mathcal{C}+$ can be seen as a formalism to capture ECA rules and provide a formal way to extend them with integrity rules and conflict-resolution rules (firing multiple conflicting actions).

Adaptive workflow models [2,6] conceptually are similar to the work presented here, but are focused on capturing exceptional situations under which the whole process is adapted, our work emphasises the capture of support tasks and their

modalities and initiations rather than adaptive workflow processes. However, even though the concepts appear orthogonal, $\mathcal{C}+$ language could be used to describe adaptation constraints as well, we leave this investigation as future work.

[12, 15, 19] detail the basic principles of task management but lack important concepts like modality, inter-workflow dependencies and instantiations. These approaches also assign tasks with priorities which in principle are related to *static* modalities, but do not consider dynamic changes to priorities at runtime. Similarly to the constraint-based workflows they can be used to complement our current approach. Decker et al. [7] describes patterns on process instantiation rather than tasks. However, the way we instantiate tasks can be considered as implementation of the creation patterns. In our work tasks are instantiated as a response to an event occurred in the environment which is similar to the patterns C-2 to C-5. Our approach allows controlling the complete task life-cycle. In particular, we allow the termination and completion of a task as a result of a rule evaluation. Therefore, our work also implements termination patterns described by Aalst et al. [1, 17]. However, in our case, a task can be terminated, even if it is not completed. The evaluation of rules including environmental data can be considerd as an implementation of the external data interaction patterns described by Russel et al. [18].

7 Conclusion and Future Work

Pervasive workflows usually have a number of non-compulsory assistance tasks, whose executions contribute towards a better service, but are not necessary for achieving a workflow's goal. In the presented case study, tasks such as: changing the beds, checking the nutrition in-take are not strictly necessary for the treatment procedures but contribute to the well-being of the patient. We have further identified that these tasks are typically initiated by the contextual and inter-workflows events and not just by the intra-workflow events. And they are initiated with a modality which indicates how much the quality of service can suffer if the support tasks are ignored. It appears that constraint-based and imperative workflow models are not well-suited for encoding support tasks' initiations and terminations. To this end, we use the $\mathcal{C}+$ language to provide a straight-forward way for capturing support norms which govern the execution of support tasks. $\mathcal{C}+$'s semantics are defined over a labelled transition system and thus offer an interesting future research area for combing this formalism with current constraint-based workflow models. This paper also described our current architecture which demonstrates how support tasks and their management can be easily made complimentary to current workflow management systems. In our future work we will investigate in greater detail how this integration can add more flexibility to the current pervasive workflow models and engines.

Acknowledgments. We thank Robert Craven for his help with the iCCalc tool and semantics of $\mathcal{C}+$ language. We also thank Ms Regina Schmeer, Director of Nursing Science at Hannover Medical School, for her insight into the nursing and

patient-care processes. This research was supported by EU FP7 research grant 213339 (ALLOW).

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases 14, 5–51 (2003)
2. van der Aalst, W.M.P., Weske, M., Grünbauer, D.: Case handling: a new paradigm for business process support. Data Knowl. Eng. 53, 129–162 (2005)
3. Adi, A., Etzion, O.: Amit - the situation manager. The VLDB Journal 13(2), 177–203 (2004)
4. Artikis, A., Sergot, M.J., Pitt, J.V.: Specifying norm-governed computational societies. ACM Trans. Comput. Log. 10(1) (2009)
5. Craven, R., Sergot, M.: Agent strands in the action language. Journal of Applied Logic 6, 172 – 191 (2008)
6. Dadam, P., Reichert, M.: The adept project: a decade of research and development for robust and flexible process support. Computer Science - R&D 23, 81–97 (2009)
7. Decker, G., Mendling, J.: Process instantiation. Data Knowl. Eng. 68, 777–792 (2009)
8. Dougherty, L., Lister, S.: The Royal Marsden Hospital Manual of Clinical Nursing Procedures. John Wiley & Sons, 7. auflage edn. (Jul 2008)
9. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. Artificial Intelligence 153, 49 – 104 (2004)
10. Goh, A., Koh, Y.K., Domazet, D.S.: Eca rule-based support for workflows. Artificial Intelligence in Engineering 15, 37 – 46 (2001)
11. Jung, J.Y.: Generating rule-based executable process models for service outsourcing. In: Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations. pp. 114–118 (2009)
12. Leymann, F., Roller, D.: Production Workflow - Concepts and Techniques. PTR Prentice Hall (January 2000)
13. Leymann, F., Unger, T., Wagner, S.: On Designing a People-oriented Constraint-based Workflow Language. In: ZEUS (2010)
14. Li, X., Medina Marń, J., Chapa, S.: A structural model of eca rules in active database. In: MICAI 2002: Advances in Artificial Intelligence, vol. 2313, pp. 73–87. Springer Berlin / Heidelberg (2002)
15. OASIS: Web Services Human Task Specification Version 1.1, Committee Draft 06 (2009)
16. Pesic, M., Schonenberg, M.H., Sidorova, N., van der Aalst, W.M.P.: Constraint-based workflow models: Change made easy. In: OTM Conferences (1). pp. 77–94 (2007)
17. Russell, N., Arthur, van der Aalst, W.M.P., Mulyar, N.: Workflow Control-Flow Patterns: A Revised View. Tech. rep., BPMcenter.org (2006)
18. Russell, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Workflow data patterns: Identification, representation and tool support. In: ER. pp. 353–368 (2005)
19. Unger, T., Bauer, T.: Towards a standardized task management. In: Multikonferenz Wirtschaftsinformatik (2008)
20. Wieland, M., Kaczmarczyk, P., Nicklas, D.: Context integration for smart workflows. In: PerCom. pp. 239–242 (2008)