Lightweight Approach for Seamless Modeling of Process Flows in Case Management Models

Christoph Czepa, Huy Tran, Uwe Zdun
Research Group Software Architecture
Faculty of Computer Science
University of Vienna
Währingerstraße 29
1090 Vienna, Austria
{firstname.lastname}@univie.ac.at

Thanh Tran, Erhard Weiss, Christoph Ruhsam
ISIS Papyrus Europe AG
Alter Wienerweg 12
2344 Maria Enzersdorf, Austria
{firstname.lastname}@isis-papyrus.com

ABSTRACT

Case management models are business process models that allow a great degree of flexibility at runtime by design. In contrast to flow-driven business processes (e.g., BPMN, EPC, UML activity diagrams), case management models primarily describe a business process by tasks, goals (i.e., milestones), stages, and dependencies between them. However, flow-driven processes are still often required and relevant in practice. In the recent case management standard CMMN (Case Management Model and Notation), support for process flows is offered by referencing BPMN processes. This results in a conceptual break between case elements and those in such subprocesses, so that dependencies from and to elements contained in flow-driven processes are unsupported. Moreover, case designers and other involved stakeholders are required to have substantial knowledge of not only case modeling but also of flow-driven business process modeling, which makes it overly complex. To counteract these current limitations, this paper proposes a lightweight and seamless integration of process flows in case management modeling as a first class citizen. Just a single new element, the Flow Dependency, in combination with existing case elements, enables support for fully integrated process flows in case models. Although the approach is that lightweight, an evaluation based on workflow patterns shows its high degree of expressiveness.

CCS Concepts

•**Applied computing → Business process modeling;**

Keywords

Business Process Modeling, Case Management, Workflow Patterns, Case Management Model and Notation, Process Flows

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC 2017,April 03-07, 2017, Marrakech, Morocco
Copyright 2017 ACM 978-1-4503-4486-9/17/04. . . $15.00
http://dx.doi.org/10.1145/3019612.3019616

1. INTRODUCTION

Nowadays, case management is the business process management solution of choice of many software vendors [4]. Although vendors seem to often pursue proprietary solutions, there exist fundamental common concepts such as goals and dependencies (cf. [1]). There are also ongoing standardization efforts by the Object Management Group (OMG), which aim to cover those recurring fundamental concepts. Their goal is to make case models interchangeable and representable by common graphical notations. As a result, a first version of the Case Management Model and Notation (CMMN) [10] standard was released in 2014. The CMMN standard includes many conceptual elements, such as stages and dependencies, that make it distinct from flow-driven business processes, such as BPMN [9]. However, since the case management modeling elements are not specifically designed for the modeling of flow-driven aspects of a case, the CMMN standard still allows to reference flow-driven BPMN subprocesses. Many of today's case management solutions seem to follow the same pattern.

One of the downsides of the current situation is the conceptual break in the presence of such flow-driven subprocesses in case models. The elements of a BPMN subprocess are isolated from all other elements of the case which are not part of the same subprocess. As a result, it is impossible to model dependencies to or from elements of the subprocess, and it is not possible to use case modeling elements inside of the subprocess because they are not part of the BPMN standard. Trying to model process flows with existing case elements only is not a viable option as it might result in complicated constructs (cf. [2]), and some behavior cannot be modeled at all. For example, it is not possible to model the synchronizing merge workflow pattern with case management elements.

Moreover, to be able to model and understand flow-driven aspects, a case designer or different kind of involved stakeholder must be at home with both case modeling and the modeling of flow-driven processes (e.g., BPMN). As a result of combining case modeling and the modeling of flow-driven business processes, the overall modeling complexity is drastically increased (cf. [6]).

This paper proposes a lightweight approach for the seamless modeling of flow-driven aspects in case management models.

Process flows are seamlessly integrated and its elements are inter-operable with other case elements. The approach is lightweight because merely a single additional element, the *Flow Dependency* is introduced. Nevertheless, this single new element has a huge impact on the execution semantics of a case, and makes it possible to model various standard workflow patterns in non-complicated manner, as opposed to using the existing set of CMMN elements. Several workflow patterns, like the synchronizing merge workflow pattern, which cannot be described using pure CMMN, are representable using our approach. The proposed approach does not require gateways as it makes use of many existing case modeling elements instead. An evaluation of the approach on basis of workflow patterns (cf. [14]) shows the high degree of expressiveness of the proposed approach.

2. BACKGROUND: CASE MODELING

Before going into details of the proposed approach, we would like to quickly introduce existing established case modeling concepts. To be independent from vendor-specific proprietary notations, we adopt the CMMN standard as the foundation of our approach. Figure 1 shows in (a) the shape of a task, in (b) the shape of a goal (also called milestone). The shape of the stage element is shown in Figure 1 (c). A stage may contain tasks, goals, or other stages, so they can be used for structuring a case, and it is possible to nest them. Figure 1 (d) shows the shape of a dependency, which symbolizes a precedence constraint of a criterion (e.g., a task must be completed before another task is allowed to be executed). Figure 1 (e) shows the shape of an entry criterion, which limits the access to a task, goal, or stage if it is attached to it. A criterion consists in CMMN of if-parts and on-parts. If-parts are rules expressed as boolean formulas that must all be satisfied whereas on-parts are dependencies that must all be satisfied in order to satisfy the criterion. Figure 1 (f) shows the exit criterion shape. An exit criterion is evaluated at the completion of a task, stage or goal.

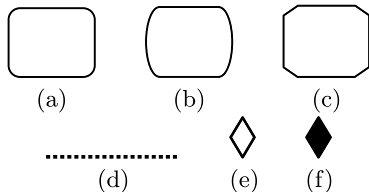

Figure 1: Case element notations in CMMN: (a) Task shape, (b) Goal/milestone shape, (c) Stage shape, (d) Dependency shape, (e) Entry criterion shape, and (f) Exit criterion shape

3. MOTIVATIONAL EXAMPLE

A situation demanding a synchronizing merge is caused by one or more multiple choice decisions, and the merge must wait for one or more incoming paths, dependent on the decisions made. That is, it is generally unknown at design time of the model, how many incoming paths there will be at runtime. Figure 2 shows this scenario modeled in BPMN and the same multiple choice in CMMN, but there exists no possible way for modeling a synchronizing merge in CMMN. Moreover, there is no guarantee that the conditions of the entry criteria (diamond shapes) of `B`, `C` and `D` are only evaluated at the time of completion of `A`, so they can become satisfied at a later time. Clearly, this contradicts the behavior that is usually expected from process flows.

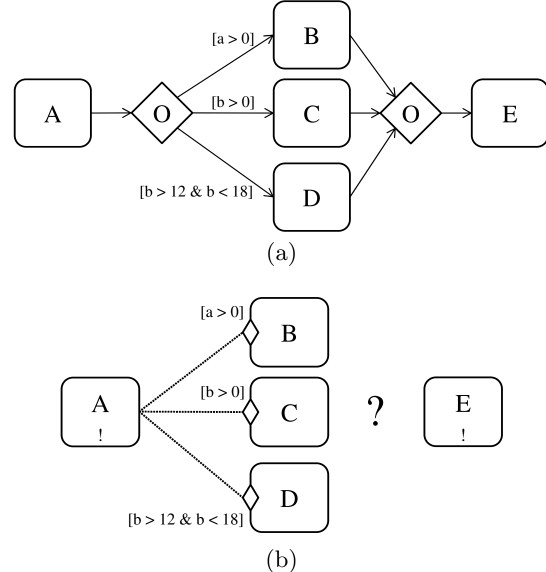

Figure 2: Synchronizing merge not representable in CMMN: (a) Multiple choice and synchronizing merge in BPMN, and (b) Multiple choice in CMMN

4. APPROACH

For enabling the integrated modeling of flow-driven aspects in case models, we propose a new kind of dependency, the *Flow Dependency*. Figure 3 shows the shape of this new concept, and the subsequent sections will introduce its semantics.

Figure 3: Flow dependency shape

4.1 Non-Branching Flows

Figure 4 shows the usage of this new connector to define flow dependencies between tasks, stages and goals (i.e., milestones). In those usage scenarios (which are not intended to be exhaustive), the completion of a case element from which the dotted arrow originates causes the automatic enabling or activation of the subsequent case element. Flow dependencies demand the eventual activation of subsequent elements, so if in (a) `Task 1` is completed, the flow dependency activates `Task 2`. As long as `Task 2` is not complete, the parent element of it, in which it is contained, namely its stage, cannot be completed. By this, it is guaranteed that the flow is not interrupted prematurely.

Entry and exit criteria can be used as guard conditions in conjunction with flow dependencies. In contrast to classical

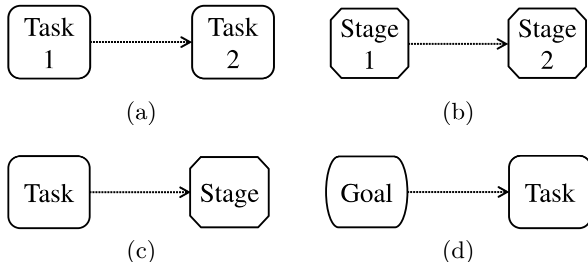

Figure 4: Example usage scenarios for flow dependencies between tasks, stages and goals (i.e., milestones)

flow-driven languages, such as BPMN, that usually define guard conditions on transitions (i.e., sequence flows), our approach does not allow to define guard conditions on flow dependencies. Instead, it makes use of the existing notions of exit and entry criteria. Figure 5 shows usage scenarios of flow dependencies with criteria. In Figure 5 (a), `Task 2` has an entry criterion, which guards the access to it. At the moment `Task 1` completes, the condition of this entry criterion is evaluated either to true or false, depending on the current state of the involved data attributes of the condition. If the condition is evaluated to false, then `Task 2` cannot be reached in this scenario. That is, the guard condition prevents its enabling. On the other hand, if the condition is evaluated to true, then `Task 2` is activated and must reach a terminal state eventually. In Figure 5 (b), `Stage 1` has an exit criterion with a flow dependency to `Stage 2`. The exit criterion is evaluated at the completion of `Stage 1`, and in case of a positive evaluation, `Stage 2` becomes active. Again, if a case element has been enabled or activated by a flow dependency, it must eventually be completed or reach another terminal state. In Figure 5 (c), there is a flow dependency from an exit criterion to an entry criterion. First, the exit criterion of `Task` is evaluated. If its condition turns out to be false, the flow dependency is not triggered, otherwise it leads to the evaluation of the entry criterion of `Stage`, which leads to the activation of that stage if the condition is evaluated to true.

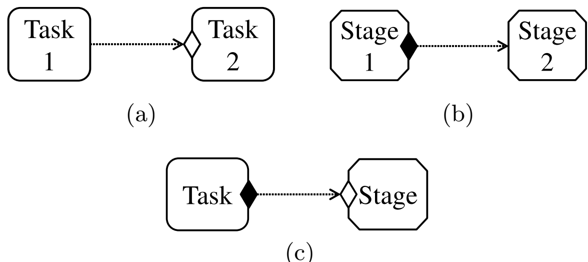

Figure 5: Example usage scenarios for sequential dependencies in conjunction with criteria

4.2 Branching Flows

In this section, we discuss the branching behavior of process flows that are formed by flow dependencies. Especially the merging behavior is of interest, as already outlined in the motivation of our work (cf. Section 3). For example, BPMN defines different kinds of gateways, like parallel, exclusive and inclusive gateways. In the proposed approach, process flows are modeled without any explicitly defined gateway, but splitting and merging behavior can be modeled even without introducing gateways. Avoiding gateways is a design decision of our approach to keep it as lightweight as possible. In particular, the number of introduced new modeling elements is kept minimal (i.e., only the flow dependency connector is introduced).

Figure 6 contains a process flow created by flow dependencies which is semantically equal to the BPMN process in Figure 5 (a). The execution of `A` is mandatory (indicated by the exclamation mark). With the start of `A`, the flow-driven process starts. At the completion of `A`, all outgoing flow dependencies become satisfied. Depending on the state of the data attributes `a` and `b` at the completion of `A`, either all or subsets of the task set {`B`, `C`, `D`} become activated because of their incoming flow dependencies. If none of the entry criteria of `B`, `C` and `D` has been evaluated to true, then the process flow ends. Otherwise, Task `E` will be activated eventually. The behavior of the three incoming flow dependencies to `E` is as if there was an inclusive gateway (i.e., a synchronizing merge) in between them and the task. That is, `E` becomes activated once at least one incoming flow dependency is satisfied and the later satisfaction of at the same time unsatisfied incoming flow dependencies is not possible (due to the lack of upstream activity that would be required to satisfy the dependency eventually).

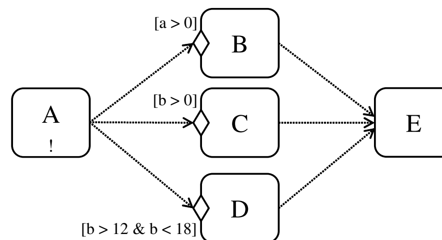

Figure 6: Branching flow scenario

Figure 7 shows different scenarios for merging incoming flows. Figure 7 (a) shows three incoming flows to the entry criterion of `Task 1`. This entry criterion is evaluated once at least one flow dependency is satisfied and there is no upstream activity that might lead to the satisfaction of the unsatisfied ones in the future. At that time, the entry criterion is evaluated either to true or false, and `Task 1` can only be started if the entry criterion has turned out to be true. Figure 7 (b) is an extended version of (a) that has three additional flow dependencies directly incoming to the task. In this case, `Task 2` is activated once the same as in (a) holds and at least one of the three incoming flow dependencies directly to `Task 2` is satisfied and there is no upstream activity that might lead to the satisfaction of the unsatisfied ones in the future. In the presence of multiple entry criteria, like in Figure 7 (c), the positive evaluation of one of the entry criteria activates `Task 3`. In Figure 7 (d), one of the entry criteria can activate the task, once the flow dependencies that are directly incoming to `Task 4` are satisfied sufficiently, as already discussed for Figure 7 (b).

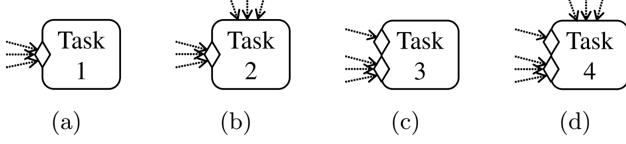

Figure 7: Merging scenarios

4.3 Completion of Stages

Additionally to the existing requirements for stage completion (no active elements, all required elements in terminal state - cf. [10]), we define an additional requirement to prevent the premature termination of process flows: If there is any contained task or stage of a stage s enabled or active that is part of a flow dependency structure (i.e., having an incoming or outgoing flow dependency, or having an entry criterion with an incoming flow dependency, or having an exit criterion with an outgoing flow dependency) or a flow dependency of a stage s is satisfied, then the stage s may not complete.

4.4 Ordinary Dependencies

Ordinary dependencies (i.e., dependencies that are not flow dependencies) can be combined with flow dependencies. Figure 8 shows two scenarios in which dependencies and flow dependencies are used at the same time. In Figure 8 (a), the dependency is directly attached to an entry criterion with three incoming flow dependencies. The only possible way to activate `Task 1` is given by satisfying the entry criterion. The entry criterion is evaluated once (1) at least one incoming flow dependency is satisfied and there is no upstream activity that might lead to the satisfaction of the unsatisfied ones in the future, and (2) all incoming dependencies to the entry criterion are satisfied. Figure 8 (b) shows `Task 2` with three incoming flow dependencies, and an entry criterion with three incoming flow dependencies and two ordinary dependencies. This is an extension of the example in Figure 7 (b). In addition to that scenario, at least one incoming flow dependency to the task must be satisfied and there must not be any upstream activity on yet unsatisfied incoming flow dependencies.

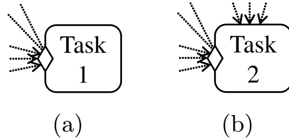

Figure 8: Scenarios with dependencies

4.5 Ordinary Entry and Exit Criteria

Entry and exit criteria do not necessarily need to be associated with a flow dependency. An entry criterion without any incoming flow dependency functions as an entry point to the flow. For example, it might make sense to use entry criteria to limit the access to tasks and stages at the begin of the process flow. An exit criterion without any outgoing flow dependency may be used to notify a task or stage outside of the flow through an ordinary dependency.

4.6 Cycles

Some situations might require the introduction of cycles. Dependent elements (i.e., elements that have an ordinary dependency defined on an element of the process flow) are not automatically instantiated multiple times by this looping behavior. Figure 9 shows a four tasks. Tasks `A`, `B`, `C` are part of a process flow, and `D` is just dependent on `C`. There is a cycle that leads to the creation of a new instance of `B` after the completion of `C` if the entry criterion attached at the bottom of `B` is satisfied. If the entry criterion of `D` is met at the first completion of `C`, then `D` becomes activated. Further loop iterations will not affect `D` or its entry criterion anymore. However, if the entry criterion of `D` is not satisfied in the first iteration of the loop, then it will be continuously reevaluated. An exception of this behavior is given if `D` is decorated as repeatable (by the '#' symbol). In that case, the ordinary repetition behavior of CMMN is effective, and `D` may be instantiated multiple times.

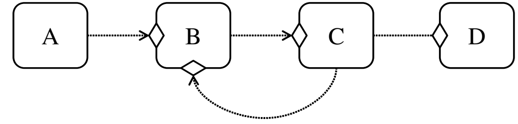

Figure 9: Process with a loop formed by flow dependencies (`B`, `C`) and an ordinary dependency (entry criterion of `D` is dependent on `C`)

5. FORMALIZATION

A case model with flow dependencies $\mathcal{M}_\mathcal{F}$ is a tuple $\mathcal{T}$, $\mathcal{G}$, $\mathcal{S}$, $\mathcal{E}$, $\mathcal{X}$, $\mathcal{C}$, $\mathcal{R}$, $\mathcal{V}$, $\mathcal{D}$, $\mathcal{D}_\mathcal{F}$, $\zeta_\mathcal{E}$, $\zeta_\mathcal{X}$, δ, κ_{req}, κ_{rep}, $\mathcal{H}_{req}$, $\mathcal{H}_{rep}$, $\kappa_\mathcal{C}$, $\mathcal{AM}$, $\sigma_{\mathcal{AM}}$, $\mathcal{CM}$, $\sigma_{\mathcal{CM}}$, where

- $\mathcal{T}$ is a set of tasks, $\mathcal{G}$ is a set of goals (i.e., milestones), $\mathcal{S}$ is a set of stages, $\mathcal{E}$ is a set of entry criteria, $\mathcal{X}$ is a set of exit criteria, $\mathcal{C} = \mathcal{E} \cup \mathcal{X}$ is a set of criteria, $\mathcal{R}$ is a set of rules, $\mathcal{V}$ is a set of event listeners,
- $\mathcal{D}$ is a set of dependencies,
- $\mathcal{D}_\mathcal{F}$ is a set of flow dependencies,
- $\zeta_\mathcal{E} : \mathcal{E} \mapsto \mathcal{T} \cup \mathcal{G} \cup \mathcal{S}$ is a total non-injective function which maps an entry criterion to a task, goal, or stage,
- $\zeta_\mathcal{X} : \mathcal{X} \mapsto \mathcal{T} \cup \mathcal{S}$ is a total non-injective function which maps an exit criterion to a task or stage,
- $\delta : \mathcal{T} \cup \mathcal{G} \cup \mathcal{S} \mapsto \mathcal{S}$ is a partial non-injective function which maps a task, goal, or stage to a parent stage,
- $\kappa_{req} : \mathcal{T} \cup \mathcal{S} \mapsto \mathcal{R}$ is a total function which maps a task, goal, or stage to a required rule,
- $\kappa_{rep} : \mathcal{T} \cup \mathcal{S} \mapsto \mathcal{R}$ is a total function which maps a task, goal, or stage to a repetition rule,
- $\mathcal{H}_{req} : \mathcal{R} \mapsto \{true, false\}$ is a total non-injective surjective function which preserves the evaluation of a required rule,

- $\mathcal{H}_{rep} : \mathcal{R} \mapsto \{true, false\}$ is a total non-injective surjective function which preserves the evaluation of a repetition rule,
- $\kappa_{\mathcal{C}} : \mathcal{C} \mapsto \mathcal{R}$ is a total function which maps a criterion to a rule,
- $\mathcal{AM} = \{automatic, manual\}$ is a set of activation modes for tasks and stages,
- $\sigma_{\mathcal{AM}} : \mathcal{T} \cup \mathcal{S} \mapsto \mathcal{AM}$ is a total non-injective surjective function which maps a task or stage to an activation mode.
- $\mathcal{CM} = \{automatic, manual\}$ is a set of completion modes for stages,
- $\sigma_{\mathcal{CM}} : \mathcal{T} \cup \mathcal{S} \mapsto \mathcal{CM}$ is a total non-injective surjective function which maps a stage to a completion mode.

The function $EvaluateRule(r \in \mathcal{R})$ evaluates a rule r to either $true$ or $false$, depending on the runtime states of data at the time of invocation.

The function $EvaluateIncomingFlowDependencies(e \in \mathcal{E} \cup \mathcal{T} \cup \mathcal{S})$ (cf. Algorithm 1) returns true if there is at least one incoming flow dependency satisfied and there is no upstream activity for yet unsatisfied flow dependencies.

Algorithm 1 Evaluation of incoming flow dependencies

```
1: function EvaluateIncomingFlowDependencies(e ∈ ℰ ∪ 𝒯 ∪ 𝒮)
2:     atLeastOneSatisfied := ∃d | d = (d_s, d_t) ∈ 𝒟_ℱ ∧ d_t = e  (d.state = Satisfied)
3:     noUpstreamActivity := true
4:     for all d | d = (d_s, d_t) ∈ (𝒟_ℱ \ d^⊤ | d^⊤ = (d_s^⊤, d_t^⊤) ∈ 𝒟_ℱ ∧ d_t^⊤ = e  (d^⊤.state = Satisfied)) ∧ d_t = e do
5:         if EvaluateUpstreamActivity(d) = true then      ▷ cf. Algorithm 2
6:             noUpstreamActivity := false
7:             break
8:     return atLeastOneSatisfied ∧ noUpstreamActivity
```

Algorithm 2 Evaluation of upstream activity

```
1: function EvaluateUpstreamActivity(d = (d_s, d_t) ∈ 𝒟_ℱ)
2:     if d.state = Satisfied) then
3:         return true
4:     if d_s ∈ 𝒯 ∪ 𝒮 ∧ (d_s.state = Enabled ∨ d_s.state = Active) then
5:         return true
6:     if ζ_𝒳(d_s) ∈ 𝒯 ∪ 𝒮 ∧ (d_s.state = Enabled ∨ d_s.state = Active) then
7:         return true
8:     if d_s ∈ 𝒢 ∧ (d_s.state = Completed) then
9:         return true
10:    for all d^c | d^c = (d_s^c, d_t^c) ∈ 𝒟_ℱ  (d_t^c = d_s ∨ ζ_ℰ(d_t^c) = d_s) do
11:        if EvaluateUpstreamActivity(d^c) = true then
12:            return true
13:    return false
```

A transition of a task or stage instance $ts \in \mathcal{T} \cup \mathcal{S}$ from `Available` to `Enabled` (if $\sigma_{\mathcal{AM}}(ts) = manual$) or from `Available` to `Active` (if $\sigma_{\mathcal{AM}}(ts) = automatic$), or of a goal instance $g \in \mathcal{G}$ from `Available` to `Completed` is performed iff $(\exists d \mid d = (d_s, d_t) \in \mathcal{D}_{\mathcal{F}} \wedge d_t = ts) \rightarrow EvaluateIncoming\!-\!FlowDependencies(ts) = true$, and either

- $\exists e \mid e \in \mathcal{E} \wedge \zeta_{\mathcal{E}}(ts) = e \wedge EvaluateRule(\kappa_{\mathcal{C}}(e)) = true \wedge (\forall d^o \mid d^o = (d^o_s, d^o_t) \in D \wedge d^o_t = e \;\; (d.state = Satisfied)) \wedge ((\exists d^f \mid d^f = (d^f_s, d^f_t) \in \mathcal{D}_{\mathcal{F}} \wedge d^f_t = e) \rightarrow EvaluateIncomingFlowDependencies(e) = true)$, or
- $\nexists e \mid e \in \mathcal{E} \wedge \zeta_{\mathcal{E}}(ts) = e$.

Simply put, a task or stage becomes enabled (manual activation) or active (automatic activation) if and only if at least one incoming flow dependency is satisfied and there is no upstream activity for the unsatisfied incoming flow dependencies (if there exist flow dependencies directly to it), and at least one entry criterion is satisfied (if it has any). A goal completes if and only if all incoming flow dependencies are satisfied and at least one entry criterion (also called completion criterion) is satisfied (if it has any).

A stage instance $s \in \mathcal{S}$ may undergo a manual completion when $\sigma_{\mathcal{CM}}(s) = manual$ or undergoes an automated completion when $\sigma_{\mathcal{CM}}(s) = automatic$ (transition from `Active` to `Completed`) iff all of the following holds:

- $\nexists ts \mid ts \in \mathcal{T} \cup \mathcal{S} \wedge \delta(tg) = s \wedge ts.state = Active$,
- $\forall tgs \mid tgs \in \mathcal{T} \cup \mathcal{G} \cup \mathcal{S} \wedge \delta(tgs) = s \wedge \mathcal{H}_{req}(\kappa_{req}(tgs)) = true \;\; (tgs.state \in \{Disabled, Completed, Terminated, Failed\})$,
- $\nexists ts \mid ts \in \mathcal{T} \cup \mathcal{S} \wedge \delta(tgs) = s \wedge (\exists d \mid d = (d_s, d_t) \in \mathcal{D}_{\mathcal{F}} \wedge (d_t = ts \vee \zeta_{\mathcal{E}}(d_t) = ts \vee d_s = ts \vee \zeta_{\mathcal{X}}(d_s) = ts)) \wedge tgs.state = Enabled$, and
- $\nexists d \mid d \in \mathcal{D}_{\mathcal{F}} \wedge d.state = Satisfied.$

That is, there must not be any active elements, all required tasks/stages/goals must be in a terminal state, there must not be any enabled tasks/stages that are part of a flow, and all flow dependencies must have been consumed (there must not be any satisfied flow dependencies).

A satisfied flow dependency instance $d \mid d = (d_s, d_t) \in \mathcal{D}_{\mathcal{F}} \wedge d.state = Satisfied$ is said to be *consumed* ($d.state = Consumed$) iff $d_t \in \mathcal{T} \cup \mathcal{S}$ and d_t undergoes a transition from `Available` to `Enabled` or `Active`, or $d_t \in \mathcal{G}$ and d_t undergoes a transition from `Available` to `Completed`, or $d_t \in \mathcal{E} \wedge EvaluateRule(\kappa_{\mathcal{C}}(d_t)) = true \wedge (\forall d^o \mid d^o = (d^o_s, d^o_t) \in D \wedge d^o_t = d^t \;\; (d.state = Satisfied)) \wedge EvaluateIncomingFlow\!-\!Dependencies(d_t) = true$ and one of the following holds:

- $\zeta_{\mathcal{E}}(d_t) \in \mathcal{T} \cup \mathcal{S}$ and d_t undergoes a transition from `Available` to `Enabled` or `Active`, or
- $\zeta_{\mathcal{E}}(d_t) \in \mathcal{G}$ and d_t undergoes a transition from `Available` to `Completed`,

or $d_t \in \mathcal{E} \wedge EvaluateRule(\kappa_{\mathcal{C}}(d_t)) = false \wedge (\forall d^o \mid d^o = (d^o_s, d^o_t) \in D \wedge d^o_t = d^t \;\; (d.state = Satisfied)) \wedge EvaluateIn\!-\!comingFlowDependencies(d_t) = true.$

In other words, a flow dependency instance is consumed if and only if it takes an active part in the enabling/activation of a task/stage or the completion of a goal, or the rule of the associated criterion is negatively evaluated at the point in time where all dependencies and at least one flow dependency have arrived and there is no upstream activity.

For repetitions introduced by loops, the notion of CMMN's repetition rule (cf. [10]) is used, which may result in the

creation of additional instances. This happens implicitly for $\forall e \mid e \in \mathcal{T} \cup \mathcal{G} \cup \mathcal{S}$ $(\exists d \mid d = (d_s, d_t) \in \mathcal{D}_{\mathcal{F}} \wedge (d_t = e \vee \zeta_{\mathcal{E}}(d_t) = e \vee d_s = e \vee \zeta_{\mathcal{X}}(d_s) = e))$ (i.e., for all tasks, goals, and stages that are in a flow), so that $\kappa_{rep}(e) = true$. It is not required to specify the repetition decorator explicitly.

6. EVALUATION OF EXPRESSIVENESS

This section evaluates the expressiveness of the approach on the basis of workflow patterns [14], which are a comprehensive collection of flow-driven business process functionality.

Table 1 lists all workflow patterns, and whether the pattern can be realized by the proposed approach. Moreover, it contains the degree of support that is currently provided by the CMMN standard. The improvements achieved by our approach are highlighted in boldface. Pattern 01 (Sequence) is supported by the flow dependency connector that can be used to activate (or enable) a case element (e.g., task) after the completion of another in our approach. A sequence can also be realized in CMMN, but in contrast to our approach it requires the use of two modeling elements, namely an ordinary dependency in combination with an entry criterion. Pattern 02 (Parallel Split) can be realized by multiple outgoing flow dependencies, and Pattern 03 (Synchronization) by multiple incoming flow dependencies. Similarly to Pattern 01 (Sequence), in CMMN, the realization of these patterns requires the combination of entry criteria and ordinary dependencies. It is possible to model Pattern 04 (Exclusive Choice) if the conditions of involved criteria can only be satisfied one at a time. It is impossible to model this behavior in CMMN since the rules in criteria may become true at a later time. That is, it is impossible in CMMN to guarantee an XOR decision. As a result, Pattern 05 (Simple Merge) cannot be realized in CMMN because the pattern's assumption that none of the alternative branches is ever executed in parallel does not hold. Whether a merge in our approach is a Pattern 05 (Simple Merge) depends on the applied splitting behavior, so if the splitting behavior is an exclusive choice, then the merging behavior will be a simple merge. In contrast to Pattern 04 (Exclusive Choice), Pattern 06 (Multi-Choice) enables the satisfaction of conditions of several involved criteria at once. If this kind of splitting behavior is applied, then the merging behavior will be Pattern 07 (Synchronizing Merge). A multi-choice (OR-split) can be realized in CMMN as well, but, as already discussed for Pattern 04 (Exclusive Choice), rules of involved criteria can become true at a later time. This seems to contradict the pattern, which assumes the immediate evaluation. As already discussed in the motivating example, a synchronizing merge cannot be realized in CMMN. Pattern 08 (Multi-Merge) has no synchronization, so every incoming flow dependency may lead to the activation or enabling of a case element. This behavior can be realized by multiple entry criteria, each having just a single incoming flow dependency. In CMMN, the pattern can be realized by multiple entry criteria, each having just a single ordinary incoming dependency, and the case element must have a positively evaluated repetition rule (i.e., it must be repeatable). Pattern 09 (Discriminator) is not supported by the proposed approach. A discriminator is activated once an incoming edge is triggered, thereafter it blocks further activations until it resets when everything expected has arrived. Supporting the modeling of such a behavior would require the introduction of an additional modeling element. Pattern 10 (Arbitrary Cycles) can be modeled using both CMMN and the proposed approach. Pattern 10 (Implicit Termination) is both supported by CMMN and our approach by the automated completion of stages.

Patterns 12-15 are concerned with multiple instances. The support for these pattern requires meta-data, more specifically the number of activations and completions of case elements, which must be provided by the specific implementation. Consequently, the support is implementation dependent. Figure 10 contains realization examples for those patterns. In Figure 10 (a), Pattern 12 (Multiple Instances Without Synchronization) is realized. `Task 3` is started multiple times (depending on the data attribute `v`), but a synchronization of the created instances is not necessary. Figure 10 (b) models multiple instances with a priori design time knowledge, so it is known already at design time, how often a task will be instantiated at runtime (in this example it is four times). Since the number of times is static, the stage can complete automatically (indicated by the filled rectangle icon). Figure 10 (c) shows the modeling of multiple instances with a priori runtime knowledge that the task must be instantiated `v` times. In contrast to Figure 10 (b), the stage is not completed automatically because it is a possibility that the data attribute `v` is increased at any time, and an automated termination might not result in the desired behavior. In Figure 10 (d), multiple instances can be created, but the number of instances is not known, so the user can create instances as desired (Pattern 15: Multiple Instances Without a Priori Runtime Knowledge). There is however still a synchronization of the create instances necessary before the flow can continue.

Figure 11 shows a realization attempt for Pattern 16 (Deferred Choice). There is a deferred choice between `B` and `C`, triggered by events. However, this choice is not exclusive (XOR) as demanded by the description of the pattern in [14]. Consequently, this pattern cannot be represented using our approach. Pattern 17 (Interleaved Parallel Routing) requires the execution of a set of case elements in any order and sequentially (one at a time). This kind of behavior cannot be modeled using our approach at this time. However, it is simple to extend the current approach by introducing an additional decorator for stages that indicates that contain elements must be executed one at a time. Pattern 18 (Milestone) can be realized using a deferred choice, which cannot be represented using our approach [14]. Canceling a case and canceling activities (Patterns 19 & 20) are supported by the stage instance state top-down propagation and by state changes in the lifecycle of case elements (cf. [10]).

7. DISCUSSION

This paper proposes a novel approach for the modeling of process flows in case management models. The approach introduces a single new element, the so-called *Flow Dependency*, to form process flows as first class citizens in case models. An evaluation of the approach shows that it increases the support of the fundamental workflow patterns discussed in [14] from 60% to 80%. Most importantly, essen-

Table 1: Evaluation of workflow pattern support

Workflow pattern	Supported by CMMN	Supported by our approach
01: Sequence	fully	fully
02: Parallel Split	fully	fully
03: Synchronization	fully	fully
04: Exclusive Choice	no	**fully**
05: Simple Merge	no	**fully**
06: Multi-Choice	limited	**fully**
07: Synchronizing Merge	no	**fully**
08: Multi-Merge	fully	fully
09: Discriminator	no	no
10: Arbitrary Cycles	fully	fully
11: Implicit Termination	fully	fully
12: Multiple Instances Without Synchronization	implementation dependent	implementation dependent
13: Multiple Instances With a Priori Design Time Knowledge	implementation dependent	implementation dependent
14: Multiple Instances With a Priori Runtime Knowledge	implementation dependent	implementation dependent
15: Multiple Instances Without a Priori Runtime Knowledge	implementation dependent	implementation dependent
16: Deferred Choice	no	no
17: Interleaved Parallel Routing	no	no
18: Milestone	no	no
19: Cancel Activity	fully	fully
20: Cancel Case	fully	fully

tial patterns that have a high degree of practical relevance (i.e., choice and merge) now become supported by our approach. To reach such a high percentage in workflow pattern support is a surprisingly good result, when we consider that merely a single new element is added to the already existing modeling elements of CMMN. To keep the approach as lightweight as possible, no further modeling elements are introduced. Naturally, additional modeling elements would make a support for the yet unsupported workflow patterns possible as well, so there is a trade-off between simplicity and expressiveness. The current degree of expressiveness of our approach might in many cases be sufficient, because usually merely a small subset of process modeling elements are used in practice (cf. [8]). Consequently, the approach might be suitable to supersede existing solutions that depend on both case modeling (such as CMMN or similar proprietary solution) and classical flow-driven business process modeling solution (such as BPMN, UML activity diagrams, EPCs, ...) in order to alleviate the overall modeling complexity (cf. [6]). Please note that there also exists an extended pattern catalog [12]. In the evaluation we focus on the original catalog of workflow patterns since the elements with the most practical

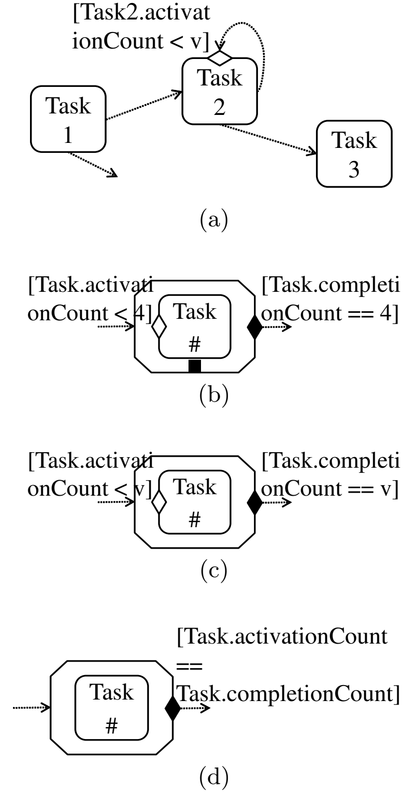

Figure 10: Multiple instance pattern realizations: (a) Without Synchronization, (b) With a Priori Design Time Knowledge, (c) With a Priori Runtime Knowledge, (d) Without a Priori Runtime Knowledge

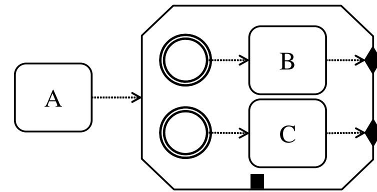

Figure 11: Attempt to model a deferred choice

relevance are covered (cf. [8]).

Whether the dependency flow element and the formed process flows are understood well by users must be further evaluated. It must be further investigated whether the introduced dotted arrow shape can be distinguished sufficiently from an ordinary dependency. The physics of notations framework by Moody [7] may function as a starting point for this investigation and for improvements. For example, it might be useful to use color encodings to distinguish elements that are part of a process flow from those that are not. Such kinds of refinements of the approach are opportunities for future research. Moreover, experimental studies with users may be used to evaluate the overall understandability of the approach and to compare it against existing flow-driven modeling approaches such as UML activity dia-

grams and BPMN.

An evaluation of potential modeling errors and resulting inconsistencies is unfortunately out of scope of this paper. For example, a user might arrange flow dependencies in a way that creates an endless looping behavior. This inconsistency would hinder the completion of a stage. We consider it as important to enable the automated detection of such errors in future work.

8. RELATED WORK

We are not aware of any existing approach that integrates process flows in case management models in a similar way to our approach. However, there exist approaches that combine different business process modeling approaches. Westergaard and Slaats [16] propose the combination of the declarative workflow approaches *Declare* and *DCR Graphs* to increase the overall expressiveness. Maggi et al. [5] propose the automated discovery of hybrid process models that consist of nested flow-driven and *Declare* models. Van der Aalst et al. [13] propose the combination of different modeling approaches in a service-oriented architecture. De Giacomo et al. [3] propose an extension of BPMN, called BPMN-D, for the combination of declarative and procedural process models.

Another closely related approach is case handling [11, 15]. It introduces three primitives, called execute (i.e., a task must be performed), skip (i.e., a task can be performed, though it is not mandatory), redo (i.e., a task can be repeated) and precedence (i.e., some task must be executed before another is allowed to happen), to allow for more flexibility during the execution of a process. CMMN offers similar constructs (i.e., repetition rule, required rule, dependencies). Like CMMN, case handling seems to have only limited support for merging behavior. The proposed approach alleviates this kind of problem.

In a recent study, de Carvalho et al. [2] evaluate the expressiveness of CMMN with regards to workflow patterns. However, the proposed representation of some workflow patterns seems to be inaccurate. For example, the exclusive choice between *Task B* and *Task C* is modeled as follows: If *Task B* is started, then the exit criterion of *Task C* is satisfied, which is ought to terminate *Task C* if it is active at that time. Obviously, this is not modeling the exclusive choice pattern accurately because both tasks enter a state of execution. Clearly, this underpins the need for a better support for the modeling of flow-driven aspects in case models, which our approach is able to provide.

9. CONCLUSION & FUTURE WORK

This paper proposes a lightweight approach for enabling the seamless modeling of process flows in case management models. Just a single new modeling element, the so-called *Flow Dependency* enables a high degree of expressiveness for the modeling of procedural workflows. In particular, 80% of all workflow patterns that are discussed in [14] are supported by the proposed approach. The further evaluation of the approach, including potential improvements, as outline in Section 7, as well as the identification of potential inconsistencies, and finding a solution for the automated detection of modeling errors are opportunities for future work.

Acknowledgments The research leading to these results has received funding from the FFG project CACAO, no. 843461 and the WWTF, Grant No. ICT12-001.

10. REFERENCES

[1] C. Czepa, H. Tran, U. Zdun, S. Rinderle-Ma, T. Tran, E. Weiss, and C. Ruhsam. Supporting structural consistency checking in adaptive case management. In *CoopIS'15*, pages 311–319, October 2015.

[2] R. M. de Carvalho, H. Mili, A. Boubaker, J. Gonzalez-Huerta, and S. Ringuette. On the analysis of cmmn expressiveness: revisiting workflow patterns. In *AdaptiveCM'16*, September 2016.

[3] G. De Giacomo, M. Dumas, F. M. Maggi, and M. Montali. *Declarative Process Modeling in BPMN*, pages 84–100. Springer, Cham, 2015.

[4] Forrester Research. The Forrester Wave^{TM}: Dynamic Case Management, Q1 2016.

[5] F. M. Maggi, T. Slaats, and H. A. Reijers. *The Automated Discovery of Hybrid Processes*, pages 392–399. Springer, Cham, 2014.

[6] M. A. Marin, H. Lotriet, and J. A. Van Der Poll. Measuring method complexity of the case management modeling and notation (cmmn). In *SAICSIT'14*, pages 209:209–209:216. ACM, 2014.

[7] D. Moody. The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Trans. Softw. Eng.*, 35(6):756–779, Nov. 2009.

[8] M. z. Muehlen and J. Recker. *How Much Language Is Enough? Theoretical and Practical Use of the Business Process Modeling Notation*, pages 465–479. Springer, 2008.

[9] OMG. BPMN 2.0. http://www.omg.org/spec/BPMN/2.0/PDF. Last accessed: December 1, 2016.

[10] OMG. CMMN 1.0. http://www.omg.org/spec/CMMN/1.0/PDF. Last accessed: December 1, 2016.

[11] H. A. Reijers, J. H. M. Rigter, and W. M. P. van der Aalst. The case handling case. *Journal of Cooperative Information Systems*, 12(03):365–391, 2003.

[12] N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical report, 2006.

[13] W. M. P. van der Aalst, M. Adams, A. H. M. ter Hofstede, M. Pesic, and H. Schonenberg. *Flexibility as a Service*, pages 319–333. Springer, 2009.

[14] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[15] W. M. P. van der Aalst and M. Weske. Case handling: A new paradigm for business process support. *Data Knowl. Eng.*, 53(2):129–162, May 2005.

[16] M. Westergaard and T. Slaats. *Mixing Paradigms for More Comprehensible Models*, pages 283–290. Springer, 2013.