Infinite Horizon Decision Support For Rule-based Process Models

Michaela Baumann*, Michael Heinrich Baumann†, Dominik Franz-Xaver Gruber‡, and Stefan Jablonski*
*Institute for Computer Science, University of Bayreuth, Germany
†Institute for Mathematics, University of Bayreuth, Germany
‡Faculty of Computer Science, Hochschule Kempten – University of Applied Sciences, Germany
email: {michaela.baumann, michael.baumann, stefan.jablonski}@uni-bayreuth.de, dominik.f.gruber@stud.fh-kempten.de

***Abstract*—In recent years, process models tend to turn away from common procedural models to more flexible, rule-based models. These models are characterized by the fact that in each execution step users usually have to decide between several rule-consistent tasks to perform next. Precise execution paths are not given, which is why adequate execution support needs to be provided. Simulation is one means to facilitate the users' decisions. In this context, we suggest an execution simulation tool with an infinite horizon, i.e., in each (simulated) step, users are informed about the tasks that in any case still need to be done to properly finish one process instance, and about tasks that may no longer be executed. The forecasts consider an actual or a simulated history of the process instance and the rules given by the model. In contrast to systems that show consequences of decisions for the next 1, 2, 3, ... steps, our system deals with impacts until the end of the process instance, independent of the length of the instance, i.e., the number of steps, which is what we call "infinite horizon."**

Keywords–Process execution; Rule-based process models; Process decision support;

I. Introduction

This paper is based on the ICCGI 2015 contribution [1].

In many fields of economy, industry, and research, process models are used for supporting the execution of operating processes, for designing work steps, for documentation purposes, and so on [1]. Usually, these process models are a sort of procedural process models, where the execution order of the process steps is prescribed through the control flow. Notations for this kind of process models are, for example, Event-driven Process Chains (EPCs) [2], the Business Process Model and Notation (BPMN) [3], and petri nets [4]. See the left side of Figure 1 for an imperative process model example in BPMN. Other execution orders than the prescribed ones are not provided. This is why computational offloading ("the extent to which differential external representations reduce the amount of cognitive effort required to solve informationally equivalent problems" [5]) is quite well achieved in procedural process models. For rule-based process models, this is not the case [6], as they take a different modeling and representation approach. They are typically used when procedural process models are too restrictive or get too complicated when complex facts shall be displayed. The approach of rule-based process models is to provide a set of tasks, firstly without stating any execution order, and then to restrict all possible execution orders by adding rules or constraints that should be met during the execution. An example for such a rule could be: "If task A has been executed, afterwards task C needs to be eventually executed, too." See Figure 2 or the right side of Figure 1 for an example of a graphically represented rule-based process model in the ConDec language [7] whose elements will be explained in Section III. In some related work ConDec is also called DECLARE [8], but the term DECLARE is also used for a constraint-based system that supports LTL-based models like ConDec.

The two notations of Figure 1 express the same circumstance, namely that task A has to be executed at least two and at most five times in one process instance. As [9] states, declarative process models may be transformed into imperative ones, but the resulting model will probably look like a so-called "spaghetti-model" as the number of execution paths is incredibly large [10]. An example of such a transformation is given in Figure 1. Especially for rule-based process models, guidance for the user through the process is necessary, as the execution sequences leading to a proper process completion are not easy to see [11]. The paper at hand provides one mechanism for such a guidance through rule-based process models.

The work proceeds as follows: Section II briefly describes the research question in contrast to related work. Section III presents the declarative modelling language ConDec, which is based on linear temporal logic and considered in the further course of this paper. Section IV introduces the basic idea of the infinite horizon decision support whereas Section V then presents the concrete mechanism through deriving so-called status values out of the declarative process rules. Section VI briefly introduces a prototypical implementation. Section VII

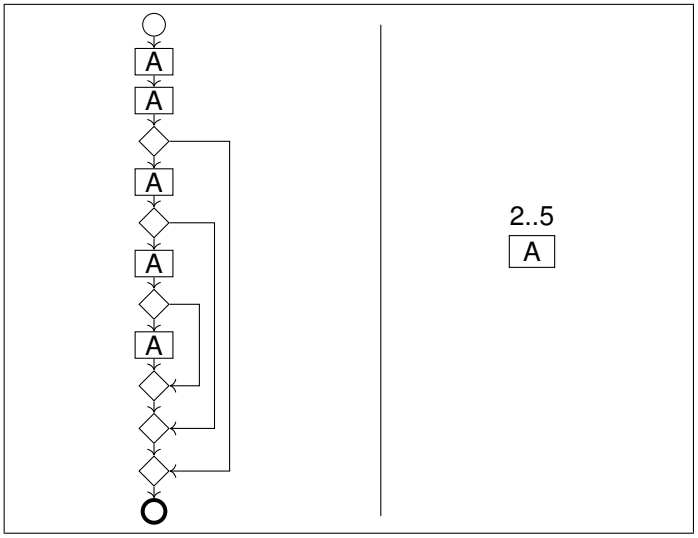

Figure 1: The same process model: on the left hand side an imperative representation, on the right hand side a declarative one.

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

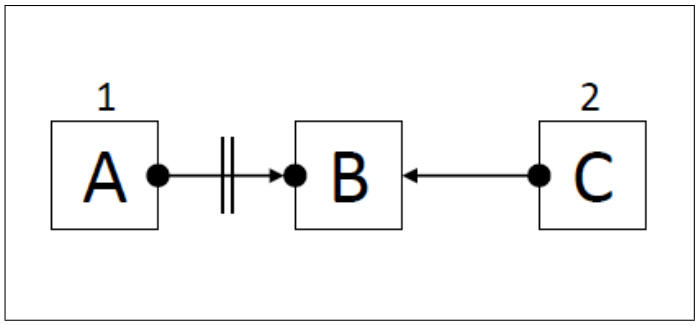

Figure 2: An example process model where ν has to look forward intelligently.

provides an exemplary process execution with infinite horizon status indicators and Section IX concludes the paper with topics for ongoing research.

II. Related Work and Research Question

For imperative process models a variety of execution support tools are available [12], like Workflow Management Systems (WfMSs) and electronical or paper-based checklists [13]. These support tools help the user to proceed the process adequately and can be based on the underlying process models but also on log data gained from previous process instances [14]. This is typically done by telling the user which tasks he has to perform or is allowed to perform next, or which tasks still have the possibility to be performed and which are no longer allowed due to previous decisions. Also, advice and frequently performed task orders together with some kind of evaluation may be provided to the user. For declarative process models, the basic functions of execution support tools are the same. However, due to the differences in the modelling approaches, there may be differences in the tools as well, for example the handling of the high level of flexibility when executing declarative process models [15]. In the work at hand, a mechanism for answering the following list of questions when processing a declaratively modelled process shall be presented:

(a) Which tasks still need to be executed during the process instance? How many times?
(b) Which tasks still can be executed during the process instance (but do not have to)? Is there a maximum number of executions?
(c) Which tasks are no longer allowed to be executed during the process instance?
(d) What changes take place for (a), (b), and (c) if a certain task would be performed next?

We do not want to answer the question, which tasks may be executed in the next step (finite horizon with step size 1). This has been done in other work, e.g., in [16] for ConDec models via automata. Instead, we want to look at the infinite horizon, i.e., a possibly infinite number of future steps, to give hints about process activities *valid for an indefinite future time frame*, exactly as specified in (a)-(d). To the best of the authors' knowledge, such a feature is not yet available. However, we assume that there is a mechanism ν available, like the one of [16], that chooses tasks for the next step in a way that every resulting process history is model conform and that dead ends are avoided. Furthermore, we need process models that do not contain conflicting constraints, i.e., that there is at least one possibility to finish the process successfully (satisfiability of the model) and that do not contain dead activities [16].

Our approach is in general independent of the underlying modelling language, i.e., also independent of the underlying logic the rules are based on, e.g., linear temporal logic or predicate logic. One reason, why we cannot use state automata for answering the above questions (a)-(d) is that it is not clear whether all declarative process models can be expressed as finite state automata at all [17]. If it is possible to express a model via an automaton, then for realistic models the automata usually suffer from a state explosion [18] and their computation gets very costly [19]. Furthermore, we want to be able to express exact numbers of activity recurrences (see (a) and (b)).

For run-time support, recommendations for effective execution [15] can be given. However, these recommendations are usually based on past experiences and need a specific goal, i.e., a rating of experiences in terms of desirability [8], as input. Due to this preselection, parts of the executable tasks are hidden from the executing agent. The decision support we head for is somehow different as we do not intend to give recommendations based on a specific goal (as input into the system) but to provide an overview over *the impact of each of his decisions* to the process participant. He can then decide, according to the overview and a goal (which is only in his mind), which step to execute next. The system and the model do not need to be changed, which may cause history-based violations when done at run-time [8]. As in declarative process models there is not necessarily a limit of steps till the end of an instance for answering questions (a)-(d), we talk of *infinite horizon* in this context. A use case for this approach could be the following example situation: in one section of a company problems have occurred (e.g., power failure, machine failure). Now, all processes that are executed until the problem is fixed shall be proceeded without consulting this section. A change of the workflow system is not needed as the problem can be fixed any time.

III. The Declarative Process Model

At first we have to specify the process model that will serve as input for the execution support tool. As already mentioned in Section II, the model shall be declarative, i.e., based on certain rules. One declarative modelling language is ConDec [7], a language based on LTL. It is usable in the DECLARE system [8], but it only allows for rules considering the tasks, i.e., the functional perspective of a process model in the five perspective approach of [20]. Rules for the other process perspectives like the data perspective, the operational perspective, or the organizational perspective are not covered. The control-flow perspective regarding the tasks is induced by the rules themselves. The limitation to the functional and the control-flow perspective through the use of ConDec simplifies the presentations but still makes clear the method of our apporach. As our approach is not dependent on ConDec but ConDec is just an example, we can extend the infinite horizon decision support to other rules involving other process perspectives as well. This extensibility comes from the fact that we do not directly use the particular modelling language's elements but the meaning behind the specific constructs, which can be called dependency patterns or *generic activity relationships*, a

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE I: LTL symbols and their meanings, taken from [24].

$\wedge$, $\vee$, $\neg$	Common logical operators (AND, OR, NOT)
$\Rightarrow$, $\Leftrightarrow$	Abbreviations for expressions formed with above operators (IMPLICATION, EQUIVALENCE)
$\bigcirc\phi$	ϕ has to hold next in the process history (NEXT)
$\Box\phi$	ϕ has to hold always in the subsequent process history (ALWAYS)
$\Diamond\phi$	ϕ has to hold eventually in the subsequent process history (EVENTUALLY)
$\phi \cup \psi$	ϕ has to hold in the process history at least until ψ holds where ψ holds eventually in the process history (UNTIL)

term used in [21]. Examples for other declarative modelling languages are the Case Management Model and Notation (CMMN) [22] specification of the Object Management Group (OMG) or the textual Declarative Process Intermediate Language (DPIL) [17]. The model elements of CMMN are to some extent very particular and exceed the scope of our approach, which is why we did not consider CMMN. DPIL is at that time not yet fully established, but it contains rules also for the other perspectives as well as cross-perspective rules. As ConDec is very common, we illustrate our approach with this graphical notation.

Independently of the underlying modelling language we have tasks $A, B, C, \ldots \in \mathcal{A}$ that may be executed in one process instance. The rules applied on the tasks of $\mathcal{A}$ are summarized in set $\mathcal{R}$. In the ConDec language there exist 19 basic rule types, also representable in linear temporal logic (LTL) [23], which will be considered further on. The basic LTL elements are listed in Table I.

The 19 basic rules of ConDec are the following ones. Every rule has a textual macro, an LTL representation indicated in square brackets, and a graphical representation, see [24]. The graphical representations are in Figures 3 and 4.

i) $existence(A, m)$: Task A has to be executed at least m times, $0 < m < \infty$
$[\Diamond(A \wedge (\bigcirc existence(A, m-1)))$ where $existence(A, 1) = \Diamond A]$
ii) $absence(A)$: Task A may not be performed
$[\neg existence(A, 1)]$
iii) $absence(A, n+1)$: Task A may only be executed up to n times, $0 < n < \infty$
$[\neg existence(A, n+1)]$
iv) $exactly(A, n)$: Task A has to be performed exactly n times, $0 < n < \infty$
$[existence(A, n) \wedge absence(A, n+1)]$
v) $init(A)$: Each process instance has to start with the execution of task A
$[A]$
vi) $respondedExistence(A, B)$: If task A appears in the process instance, then task B has to appear at some point, too
$[\Diamond A \Rightarrow \Diamond B]$
vii) $coExistence(A, B)$: If task A appears in the process instance, then task B has to appear at some point, too, and vice versa
$[\Diamond A \Leftrightarrow \Diamond B]$
viii) $response(A, B)$: If task A appears in the process instance, then task B has to appear after A, too
$[\Box(A \Rightarrow \Diamond B)]$
ix) $precedence(A, B)$: Task B can only be executed if task A has already been executed, i.e., already appears in the process history
$[(\neg B \cup A) \vee \Box(\neg B)]$
x) $succession(A, B)$: If task A shall be executed, then task B needs to be executed at some point afterwards and if B shall be executed, A needs to be performed in advance
$[response(A, B) \wedge precedence(A, B)]$
xi) $alternateResponse(A, B)$: If task A appears in the process instance, then task B has to appear after A, too, and A may not appear a second time before B
$[\Box(A \Rightarrow \bigcirc(\neg A \cup B))]$
xii) $alternatePrecedence(A, B)$: Task B can only be executed if task A has already been executed and between two executions of B at least one execution of A has to appear
$[precedence(A, B) \wedge \Box(B \Rightarrow \bigcirc(precedence(A, B)))]$
xiii) $alternateSuccession(A, B)$: Task A has to be followed by B and B has to be preceded by A and two executions of A need to have an execution of B before the second A and two executions of B need to have an execution of A after the first B
$[alternateResponse(A, B)$
$\wedge\ alternatePrecedence(A, B)]$
xiv) $chainResponse(A, B)$: Every execution of task A has to be directly followed by B
$[\Box(A \Rightarrow \bigcirc B)]$
xv) $chainPrecedence(A, B)$: Every execution of task B has to be directly preceded by A
$[\Box(\bigcirc B \Rightarrow A)]$
xvi) $chainSuccession(A, B)$: Every execution of task A has to be directly followed by B and every execution of task B has to be directly preceded by A
$[\Box(A \Leftrightarrow \bigcirc B)]$
xvii) $notCoExistence(A, B)$: Once task A is executed the first time, task B may not be executed any more and vice versa
$[\neg(\Diamond A \wedge \Diamond B)]$
xviii) $notSuccession(A, B)$: Once task A is executed, task B may not be executed any more after A
$[\Box(A \Rightarrow \neg(\Diamond B))]$
xix) $notChainSuccession(A, B)$: Task B may not be executed directly after A
$[\Box(A \Rightarrow \bigcirc(\neg B))]$

Graphical representations of the constraints are presented in Figure 3 and 4. These representations correspond to that ones shown in [16]. However, rules i)-iv) or i)-v) may be combined into the following two rules that are also shown in Figure 3 with $0 \leq m \leq n \leq \infty$ and $m < \infty$. We consider rules i') and ii') instead of the original ones in the further course of the paper.

i') $existence(A, m, n)$: Task A has to be performed at least m times but not more than n times
ii') $initExistence(A, m, n)$: Task A has to be performed at least m times but not more than n times and every process instance has to start with the execution of A

The LTL representation of each of these templates implies that formal model checking and verification can be applied. This is done via a translation to automata [16]. We require that set $\mathcal{R}$ is executable, i.e., that the underlying set of LTL formulas is satisfiable. Rules i)-v) or i') and ii'), resp., are also known as existence templates, rules vi)-xvi) as relation

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

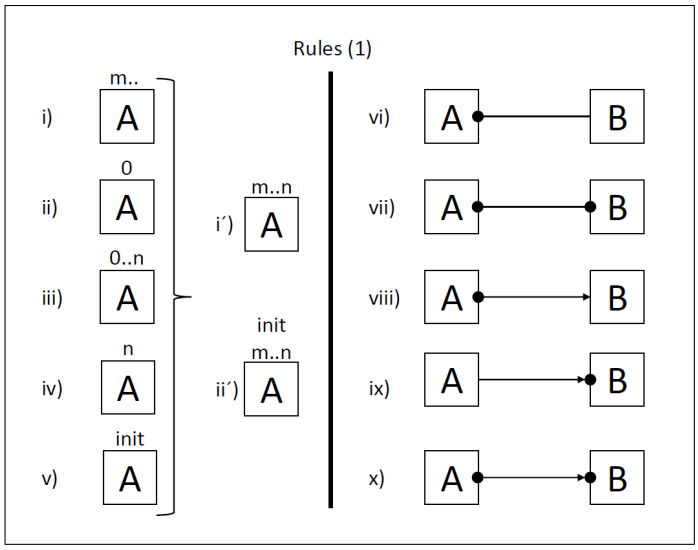

Figure 3: ConDec rules i)-x) and rules i') and ii').

templates and rules xvii)-xix) as negative relation templates. A fourth group of rules, choice templates, has been skipped like in [24] where the first three groups are referred to as basic rule types. In [16], LTL formulas for the choice templates are listed. However, as the rules base on LTL expressions, arbitrary rules can be added or existing rules modified or removed. The templates are always evaluated regarding their LTL expression, not their shortcut or graphical representation that are only used for better readability.

IV. Concept of the Infinite Horizon Decision Support

We already stated in Section I that a function or machine called ν exists that correctly returns all tasks that are executable in the next step. This means that according to every given and rule-consistent history $h \in \mathcal{H}$, where $\mathcal{H}$ is the space of all admissible histories, function $\nu : \mathcal{H} \rightarrow \mathcal{P}(\mathcal{A})$

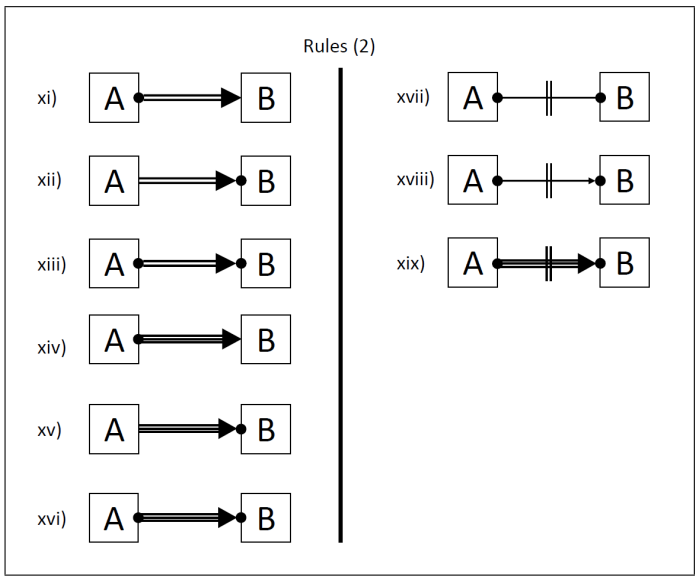

Figure 4: ConDec rules xi) - xix).

returns those tasks that will not inevitably lead to an erroneous instance state. From every task supposed by ν at least one correct path, i.e., a rule-consistent task sequence, to a successful process completion has to exist. Regard Figure 2 for an example. Three tasks, namely A, B, and C may be executed during an instance of this example process. More precisely, task A has to be executed exactly once and task C exactly twice. However, after the execution of A, task B may no longer be executed. But after any execution of C task B must eventually be executed. Let $h = \odot$ be the empty history at the beginning of a process instance. Then $\nu(\odot) = \{B, C\}$ and not $\{A, B, C\}$ because if A would be executed at the very beginning, task B may no longer be executed. But the execution of task C, and C has to be executed even twice, requires at least one execution of B afterwards. This is a contradiction and would lead to a non-correct process completion. This is why function ν may not suggest task A as first task in the process instance.

The example in Figure 2 also illustrates the possibly infinite length of a process instance. A valid process execution would be to do task C twice, then B, then A: $h = C.C.B.A$. This is indeed the shortest possible execution. But it would also be valid to do B ten times: $h = C.C.B.B.B.B.B.B.B.B.B.A$, or even a hundred times, as the number of executions of task B is not restricted. If the user of a process support system has insight only into the tasks executable in the next step, he will not be able to recognize the impact of his decisions more than one step and possibly infinite steps ahead. For answering the questions (a)-(d) from Section I we introduce two counters for each task. One counter for the mandatory executions of a task and one for the optional executions. If the mandatory counter of a task has value $a \geq 0$, this means that no matter which decisions are made from now on, this task has to be executed at least a times. If the optional counter of a task has value $b \geq 0$, this means that from now on there is no path through the process where this task may be executed more than b times. It always holds that the mandatory counter is less than or equal to the optional counter. Furthermore, the optional counter cannot increase during the execution, otherwise the previous counter value would have been wrong. If the optional counter value is zero, the corresponding task may no longer be executed during this process instance. Mandatory counters can only decrease through the execution of a task. A process can only be successfully finished when the mandatory counters of all tasks are zero.

According to the declarative nature of the process model, the initial values of the counters are zero for the mandatory counters and infinite for the optional counters. Thus, the initial status of every task is $status(A) = (man(A), opt(A)) = (0, \infty)\ \forall A \in \mathcal{A}$. The initial status of the process of Figure 2 when subsuming the statuses of all tasks into one matrix would be the following:

TABLE II: Initial status table for the example process of Figure 2; $h = \odot$.

Task	$mandatory$	$optional$
A	0	∞
B	0	∞
C	0	∞

In the next step, these status values must be updated

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

according to the process model rules given in $\mathcal{R}$. Every process model rule implies several status update rules. The specific update rules are derived in Section V, but we can state several general conditions for the change of a status value. For all model rules, we check these general conditions whether they apply for a model rule and determine the consequences. The possible conditions are the occurrences of the following events:

- The process instance has just been started ($start$)
- A certain task has just been executed ($exec(task)$ with $task \in \mathcal{A}$)
- The mandatory value of a certain task is greater than zero ($man(task) > 0$ with $task \in \mathcal{A}$)
- The optional value of a certain task is equal to zero ($opt(task) == 0$ with $task \in \mathcal{A}$)
- The optional value of a certain task is finite ($opt(task) \neq \infty$ with $task \in \mathcal{A}$)
- A certain task appears somewhere in the instance history ($task \in h$ with $task \in \mathcal{A} \wedge h \in \mathcal{H}$)
- A certain task appears on the last place in the instance history ($task = \ell(h)$ with $task \in \mathcal{A} \wedge h \in \mathcal{H}$ and ℓ a one-input function returning the last element of the history)
- A certain task appears more recently in the instance history than another task ($\ell(h, task, anothertask) = task$ with $task, anothertask \in \mathcal{A} \wedge h \in \mathcal{H}$ and ℓ a three-input function returning the more recent element of two of the history)
- A certain task does not appear somewhere in the instance history ($task \notin h$ with $task \in \mathcal{A} \wedge h \in \mathcal{H}$)
- A certain task does not appear on the last place in the instance history ($task \neq \ell(h)$ with $task \in \mathcal{A} \wedge h \in \mathcal{H}$ and ℓ a function returning the last element of the history)

The statuses $man(task) == 0$ and $opt(task) == \infty$ for $task \in \mathcal{A}$ do not have any consequences as theses statuses are the default ones. Of course, a condition can also be the conjunction of several of the above events. The possible consequences are the following events:

- The mandatory value of a certain task is reduced by one if it is greater than zero ($man(task) \leftarrow \max\{0, man(task) - 1\}$ with $task \in \mathcal{A}$)
- The mandatory value of a certain task is set to a fixed value ($man(task) \leftarrow n$ with $task \in \mathcal{A} \wedge n \in \mathbb{N}$)
- The mandatory value of a certain task is set to at least one or the value it was before ($man(task) \leftarrow \max\{1, man(task)\}$ with $task \in \mathcal{A}$)
- The mandatory value of a certain task is set to at least the mandatory value of another task (maybe increased or decreased by one) and the value it was before ($man(task) \leftarrow \max\{man(task), man(anothertask[\pm 1]\}$ with $task, anothertask \in \mathcal{A}$)
- The optional value of a certain task is reduced by one if it is greater than zero ($opt(task) \leftarrow \max\{0, opt(task) - 1\}$ with $task \in \mathcal{A}$)
- The optional value of a certain task is set to a fixed value ($opt(task) \leftarrow n$ with $task \in \mathcal{A} \wedge n \in \mathbb{N}$)
- The optional value of a certain task is set to zero ($opt(task) \leftarrow 0$ with $task \in \mathcal{A}$)
- The optional value of a certain task is set to at most the optional value of another task (maybe increased or decreased by one) and the value it was before ($opt(task) \leftarrow \min\{opt(task), opt(anothertask[\pm 1])\}$ with $task, anothertask \in \mathcal{A}$)

The reduction of the mandatory and the optional value of a certain task by one is only done when this task is executed.

If $exec(A)$: $\max\{0, man(A) \leftarrow man(A) - 1\}$;
If $exec(A)$: $\max\{0, opt(A) \leftarrow opt(A) - 1\}$;

Setting the mandatory and optional values to a fixed value is only done once at the beginning when existence rules are evaluated. For the remaining consequences it holds that the optional value of a task never increases and the mandatory value of a task never reaches infinity during the execution of the process.

For every process rule we now have to check which conditions imply which consequences for the involved tasks, i.e., we have to find the specific status update rules for every process rule.

V. Task Status Update Rules

For deriving the update rules we begin with the existence rules as these are the first ones to be evaluated after starting a process. The relation rules come next but without the chain rules, as these require a more thorough investigation. As third we examine the negation rules again without the negated chain rule. Finally, we analyze the relation and negation chain rules and observe particular behaviour for these rules because they do not directly influence the infinite horizon by definition. The derivation of the update rules is accompanied by the example process of Figure 2.

A. Existence Rules

The ConDec existence rules are equivalent to rules i') and ii'). These rules only change the status values of the tasks at the beginning of the process. After $start$ they are evaluated once and for the rest of the execution they are no longer relevant. What the update rules do is that they set the mandatory and optional values to the fixed values of the existence template that state that a task has to be executed at least m times and at most n times. If a value is not specified it remains the inital mandatory or optional value of 0 and ∞, respectively.

Rule i)':
$\forall\ existence(A, m, n) \in \mathcal{R}$:
If $start$: $man(A) \leftarrow m$;
If $start$: $opt(A) \leftarrow n$;

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Whether a task has the $init$ property or not does not affect the infinite horizon, thus rule ii') is exactly the same regarding the status update values. Of course, if a task is marked as $init$, its mandatory value has to be at least one and there may only exist at most one $initExistence$ rule.

Rule ii)':
$\forall\ initExistence(A, m, n)$: $existence(A, \max\{1, m\}, n)$

The example in Figure 2 has two existence rules, namely $existence(A, 1, 1)$ and $existence(C, 2, 2)$. According to the update rules above the status matrix in Table II changes to the matrix shown in Table III.

TABLE III: Status table for the example process of Figure 2 after evaluating the update rules resulting from existence rules; $h = \odot$.

Task	$mandatory$	$optional$
A	1	1
B	0	∞
C	2	2

B. Relation Rules without Chain Rules

The ConDec relation rules are rules vi)-xiii) plus $chainResponse$, $chainPrecedence$, and $chainSuccession$, but we consider the $chain$ rules not until Section V-D. We start with the $response$ rule. As $response(A, B)$ states that after A has been executed, B has to be eventually executed, too, the execution of A sets the mandatory value of B to at least one. But we also know that if A still has to be executed (not matter how often), B also has to be executed at least once. And we know that if B can no longer be executed, A must be prevented from execution, too, by setting $opt(A) == 0$. The $response$ rule is not history dependent.

Rule viii):
$\forall\ response(A, B) \in \mathcal{R}$:
If $exec(A)$: $man(B) \leftarrow \max\{man(B), 1\}$;
If $man(A) > 0$: $man(B) \leftarrow \max\{man(B), 1\}$;
If $opt(B) == 0$: $opt(A) \leftarrow 0$;

In contrast to the $response$ rule, $precedence$ is history dependent. We do not want to state that for rule $precedence(A, B)$, if A has already been executed, B may be executed in the next step, but we want to give long-range information. That means, if A has not yet been executed (is not in h) and can no longer been executed, the optional value of B has to be set to zero, because the prerequisites for executing B can never be fulfilled. The other way round, if B still needs to be executed and A is not yet executed, its mandatory value has to be at least one.

Rule ix):
$\forall\ precedence(A, B) \in \mathcal{R}$:
If $opt(A) == 0 \wedge A \notin h$: $opt(B) \leftarrow 0$;
If $man(B) > 0 \wedge A \notin h$: $man(A) \leftarrow \max\{man(A), 1\}$;

For $succession$ both the update rules of $response$ and $precedence$ hold.

Rule x):
$\forall\ succession(A, B) \in \mathcal{R}$:
$response(A, B) \wedge precedence(A, B)$

$respondedExistence$ is a variant of $response$ and causes the same update rules as $response$ in case that the second input task not already appears in the history. If this task already appears in the history, then the $respondedExistence$ process rule has no implications on the status values.

Rule vi):
$\forall\ respondedExistence(A, B) \in \mathcal{R}$:
If $exec(A) \wedge B \notin h$: $man(B) \leftarrow \max\{man(B), 1\}$;
If $man(A) > 0 \wedge B \notin h$: $man(B) \leftarrow \max\{man(B), 1\}$;
If $opt(B) == 0 \wedge B \notin h$: $opt(A) \leftarrow 0$;

For $coExistence$, the update rules of $respondedExistence$ with the tasks in original order and transposed order hold.

Rule vii):
$\forall\ coExistence(A, B) \in \mathcal{R}$:
$respondedExistence(A, B)\ \wedge$
$respondedExistence(B, A)$

The status update rules implied by the ConDec $alternate$ rules make use of the function $\ell : \mathcal{H} \times \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ that returns the more recent task out of two tasks under the current history h. If only one of the two tasks is in History h then ℓ returns this task. If none of the tasks is in h, it returns $NULL$. The update rules implied by the $alternate$ rules are similar to that of $response$, $precedence$, and $succession$, except that for two of the update rules, a case differentiation based on the current history has to be made. The $alternateResponse$ is, in contrast to the $response$ rule, history dependent:

Rule xi):
$\forall\ alternateResponse(A, B) \in \mathcal{R}$:
If $exec(A)$: $man(B) \leftarrow \max\{man(B), 1\}$;
If $opt(B) \neq \infty \wedge \ell(h, A, B) == A$:
$opt(A) \leftarrow \min\{opt(B) - 1, opt(A)\}$;
If $opt(B) \neq \infty \wedge \ell(h, A, B) == B$:
$opt(A) \leftarrow \min\{opt(B), opt(A)\}$;
If $\ell(h, A, B) == A$:
$man(B) \leftarrow \max\{man(B), man(A) + 1\}$;
If $man(A) > 0 \wedge \ell(h, A, B) == B$:
$man(B) \leftarrow \max\{man(B), man(A)\}$

The second and third as well as the fourth and fifth update rule can be combined via an indicator function. This is in line with the possible $NULL$ return of ℓ if neither A nor B appears in h:

Rule xi):
$\forall\ alternateResponse(A, B) \in \mathcal{R}$:
If $exec(A)$: $man(B) \leftarrow \max\{man(B), 1\}$;
If $opt(B) \neq \infty$:
$opt(A) \leftarrow \min\{opt(B) - \mathbb{I}_{\ell(h,A,B)==A}, opt(A)\}$;
If $man(A) + \mathbb{I}_{\ell(h,A,B)==A} > 0$:
$man(B) \leftarrow \max\{man(B), man(A) + \mathbb{I}_{\ell(h,A,B)==A}\}$;

Also for $alternatePrecedence$, the usual $precedence$ can be applied modified by another history dependency:

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Rule xii):
$\forall\ alternatePrecedence(A, B) \in \mathcal{R}$:
If $opt(A) \neq \infty$:
$opt(B) \leftarrow \min\{opt(B), opt(A) + \mathbb{I}_{\ell(h,A,B)==A}\}$;
If $man(B) - \mathbb{I}_{\ell(h,A,B)==A} > 0$:
$man(A) \leftarrow \max\{man(A), man(B) - \mathbb{I}_{\ell(h,A,B)==A}\}$;

The update rules implied by $alternateSuccession$ are the conjuction of the update rules of $alternateResponse$ and $alternatePrecedence$:

Rule xiii):
$\forall\ alternateSuccession(A, B) \in \mathcal{R}$:
$alternateResponse(A, B) \wedge$
$alternatePrecedence(A, B)$

C. Negation Rules without Chain Rule

Like the relation rules, the negation rules xvii) and xviii) are at first considered without the $chain$ rule, in this case without the $notChainSuccession$ rule xix. For $notSuccession(A, B)$ it holds that as soon as task A is executed, task B may no longer be executed, i.e., the optional status value of B becomes zero. The fact that as long as B has to be executed ($man(B) > 0$) prohibits A from execution cannot be considered in the infinite horizon but this restraint has to be realized in ν but this is not focussed in this paper.

Rule xviii):
$\forall\ notSuccession(A, B) \in \mathcal{R}$: If $exec(A)$: $opt(B) \leftarrow 0$;

For the $notCoExistence$ rule the fact that a mandatory value of one of the tasks is greater than zero influences the infinite horizon status update rules as there is no time dependency for this rule.

Rule xvii):
$\forall\ notCoExistence(A, B) \in \mathcal{R}$:
If $exec(A)$: $opt(B) \leftarrow 0$;
If $exec(B)$: $opt(A) \leftarrow 0$;
If $man(A) > 0$: $opt(B) \leftarrow 0$;
If $man(B) > 0$: $opt(A) \leftarrow 0$;

We can now futher analyze the model of Figure 2 and its implications on the status update table. Table III shows the statuses after $start$ of the process. The update rules implied by the relation and negation process rules have to be checked next, still before having executed the first task. We have to consider the following update rules, induced by $response(C, B)$ and $notSuccession(A, B)$:

- If $exec(C) : man(B) \leftarrow \max\{man(B), 1\}$;
- If $man(C) > 0 : man(B) \leftarrow \max\{man(B), 1\}$;
- If $opt(B) == 0 : opt(C) \leftarrow 0$;
- If $exec(A) : opt(B) \leftarrow 0$;

Clearly, the update rules activated by $exec$ have not effect at the moment. Only one update rule ($man(C) > 0$) is activated. The status changes are in Table IV. The mandatory value of B has increased by one. Note that the status update rules excluding that ones with $exec$ and $start$ can in general influence each other. The order of processing them does not matter but they have to be processed as many times until nothing more changes.

TABLE IV: Status table for the example process of Figure 2 after evaluating all update rules without having executed a task; $h = \odot$.

Task	$mandatory$	$optional$
A	1	1
B	1	∞
C	2	2

Tasks executable next (the output of function ν with respect to history $h = \odot$) are B and C. The execution of B implies the following changes: B is executed and thus $man(B) \leftarrow 0$ and $opt(B) == \infty$ (reducing ∞ by one is still ∞). Then the update rule $man(C) > 0$ activates and the mandatory value of B is again set to one. The situtation is as before. When executing C, $h = C$, the statuses change according to Table V.

TABLE V: Status table for the example process of Figure 2 after execution of C: $h = C$.

Task	$mandatory$	$optional$
A	1	1
B	1	∞
C	1	1

Now, we have $\nu(C) = \{B, C\}$. Executing B in the second step would again lead to no changes, but executing C will result in a status table as shown in Table VI.

TABLE VI: Status table for the example process of Figure 2 after execution of C again; $h = C.C$.

Task	$mandatory$	$optional$
A	1	1
B	1	∞
C	0	0

The output of ν under $h = C.C$ is now only B, thus, task B needs to be executed next. This results in statuses as seen in Table VII. The output of ν under $h = C.C.B$ is now A and B.

TABLE VII: Status table for the example process of Figure 2 after execution of B; $h = C.C.B$.

Task	$mandatory$	$optional$
A	1	1
B	0	∞
C	0	0

As soon as A is executed, the process has successfully finished as all mandatory values have reached zero. Furthermore, all optional values are zero as well, which means that no task may be executed again as well, see Table VIII. Before executing A, B can be executed arbitraritly often without changing the status values.

D. Relation and Negation Chain Rules

The $chainResponse$ behaves similar to the $alternateResponse$ with the difference that in the sequence of several tasks A and B, B has to directly follow A. Therefore, the update rules of $alternateResponse$ can be

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE VIII: Status table for the example process of Figure 2 after execution of A; $h = C.C.B.A$.

Task	$mandatory$	$optional$
A	0	0
B	0	0
C	0	0

modified through considering $\ell(h)$ instead of $\ell(h, A, B)$ with $\ell : \mathcal{H} \rightarrow \mathcal{A}$ returning the most recent task in instance history h.

Rule xiv):
$\forall\ chainResponse(A, B) \in \mathcal{R}$:
If $exec(A)$: $man(B) \leftarrow \max\{man(B), 1\}$;
If $opt(B) \neq \infty$:
$opt(A) \leftarrow \min\{opt(A), opt(B) - \mathbb{1}_{\ell(h)==A}\}$;
If $man(A) + \mathbb{1}_{\ell(h)==A} > 0$:
$man(B) \leftarrow \max\{man(B), man(A) + \mathbb{1}_{\ell(h)==A}\}$;

The same modifications hold for $chainPrecedence$.

Rule xv):
$\forall\ chainPrecedence(A, B) \in \mathcal{R}$:
If $opt(A) \neq \infty$:
$opt(B) \leftarrow \min\{opt(B), opt(A) + \mathbb{1}_{\ell(h)==A}\}$;
If $man(B) - \mathbb{1}_{\ell(h)==A} > 0$:
$man(A) \leftarrow \max\{man(A), man(B) - \mathbb{1}_{\ell(h)==A}\}$;

For $chainSuccession$, the update rules of $chainResponse$ and $chainPrecedence$ are conjuncted.

Rule xvi):
$\forall\ chainSuccession(A, B) \in \mathcal{R}$:
$chainResponse(A, B)\ \wedge\ chainPrecedence(A, B)$

Rule xix): $notChainSuccession$ has no direct implications on the status update rules but affects them indirectly. We address these indirect consequences in Section V-F.

E. Additional notSuccession and notChainSuccession caused by chainResponse and chainSuccession

For $chainResponse$, as well as for $chainSuccession$ as it includes the $chainResponse$, we can make another observation. As soon as A is executed, B has to be executed next, and so all status changes B causes are actually already caused through the execution of A. For the status update rules, this transfer caused by a $chainResponse$ resp. $chainSuccession$ is already done indirectly except for the $notSuccession$ and $notChainSuccession$ rules. The

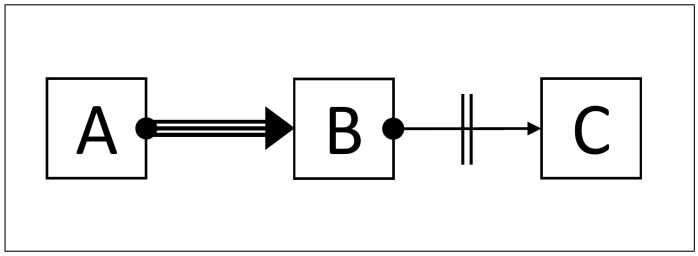

Figure 5: ConDec model where $notSuccession(B, C)$ causes the addition of $notSuccession(A, C)$ because of $chainResponse(A, B)$.

$notSuccession$ only has one update rule, which is triggered by the execution of a certain task, the $notChainSuccession$ has no update rules at all. Figure 5 shows a situation where a $notSuccession$ causes the addition of another rule. Task A has to be directly followed by B and then, after the execution of B, C may no longer be executed because of the rule $notSuccession(B, C)$. The update rule implied by the $notSuccession$ is, however, If $exec(B)$: $opt(C) == 0$ although it is already clear when executing A, C can no longer be executed. We want the update rule If $exec(A)$: $opt(C) == 0$. This can be achieved by including a $notSuccession(A, C)$ in addition to the original $notSuccession(B, C)$. Similar to this, it holds for $notChainSuccession$: $chainResponse(A, B)\ \wedge\ notChainSuccession(B, C)$: add $notChainSuccession(A, C)$ to the model. The addition of a $notChainSuccession$ may look like a redudant thing to do because the $chainResponse$ prohibits all other tasks except the responding task in the next step, but the problem gets clear with another observation described in Section V-F.

The $chainPrecedence$ is only history dependent and does not imply additions of process rules.

F. Infinite horizon implications of notChainResponse

At a first glance, the $notChainSuccession$ rule does not affect the infinite horizon as it prevents the execution of a certain task only in the next step (which would be a matter for ν). But indeed, it can have a considerable impact on the infinite horizon. See Figure 6 for an example of the infinite horizon effect of a $notChainSuccession$. The model consists only of three tasks, A, B, and C, where A needs to be executed at least once and C may not be executed directly after A. As soon as B is executed, C may no longer be executed. When starting with A, there is no direct infinite horizon status change for C. The $notSuccession$ causes no status changes at this point as well, as it only triggers through the execution of B. But in fact, C may no longer be executed as the next task after A is again A or B (C is prohibited through the $notChainSuccession$). Executing A again implies no changes, but the execution of B sets the optional value of C to zero. That means, through $exec(A)$ is should have followed immediately $opt(C) == 0$.

The detection of such a connection can be very complicated as it can involve any number of steps. A helpful approach is to conduct a review of the $notChainSuccession(A, B)$ rules under the current history if A is executed. In our example in Figure 6, the $notChainSuccession$ implies the same process behavior when executing A as a $notSuccession$ under $h = \odot$, i.e., it actually affects the infinite horizon statuses. So, for the situation $h = \odot$ the $notChainSuccession$ rule is translated into a $notSuccession$ implying all status update rules of the $notSuccession$. We call these translated $notChainSuccession$ rules $deFactoNotSuccession$ rules. They turn back to $notChainSuccession$ after one process step as their transformation is history dependent. That is, the status update rules implied by a $deFactoNotSuccession$ only hold for one turn. Figure 7 shows a situation where the $notChainSuccession$ not necessarily turns into a $notSuccession$. Before execution, the rule $notChainSuccession(A, D)$ has to be added to the original model because of $chainResponse(A, B)$ and

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

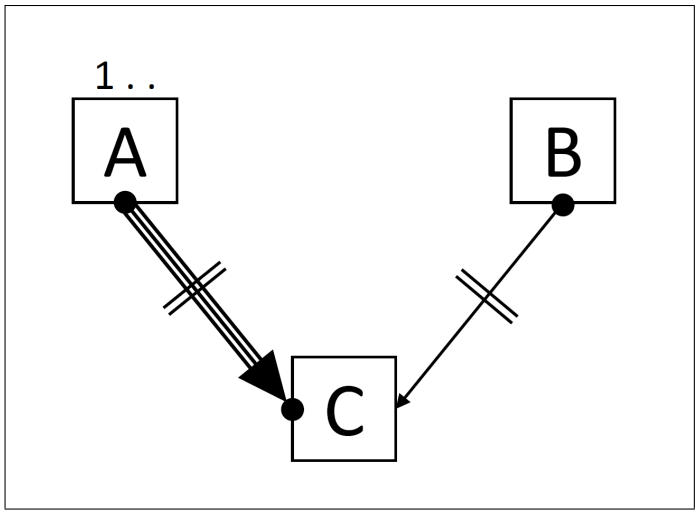

Figure 6: ConDec model where $notChainSuccession(A, C)$ is a $deFactoNotSuccesion(A, C)$.

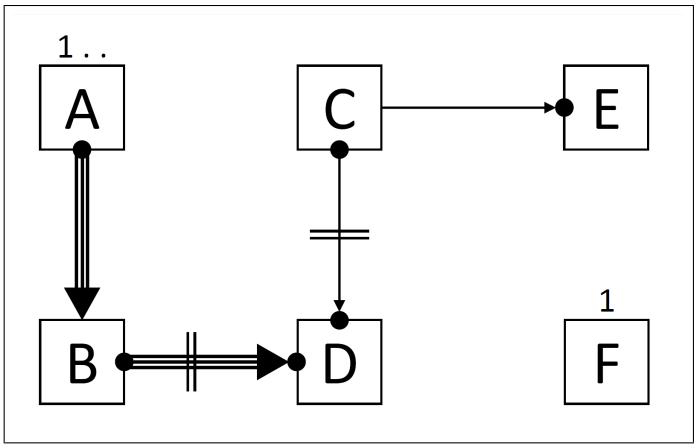

Figure 7: ConDec model where $notChainSuccession(A, D)$ is a $deFactoNotSuccesion(A, D)$ with underlying history $h = F$.

$notChainSuccession(B, D)$. It is possible, to start the process with execution of A: $h = A$. Now it has to be checked, if task D is only locked for the next step because of $notChainSuccession(A, D)$ or if it is a $deFactoNotSuccession(A, D)$ setting $opt(D) == 0$. With history $h = A$ we find the following path to execute D afterwards: task B has to follow immediately: $h = A.B$. But also here, D is not allowed at least in the next step and we also have to check for $deFactoNotSuccession(B, D)$. Task E requires C, but C immediately prohibits any further execution of D. Instead, task F can be executed: $h = A.B.F$. After execution of F, execution of D does no longer conflict with $notChainSuccession(A, D)$ and $notChainSuccession(B, D)$ and thus, D can be executed next. However, if we consider the situation where F is executed first and then A, we are no longer able to find a way to executed D afterwards. The only possible way to executed D after A was to do F, but it has exact execution number of one, i.e., its mandatory and optional values have turned to zero after the first process step. So, with history $h = F.A$ the execution of A causes the rule $notChainSuccession(A, D)$ to transform to a $notSuccession(A, D)$ and the optional value of D is immediately reduced to zero. When executing B afterwards, rule $notChainSuccession(B, D)$ does not need to be checked for being a $deFactoNotSuccesion(B, D)$ as already $opt(D) == 0$.

Rule xix):
$notChainSuccession(A, B)$: As this rules points exactly one task execution into the future, it has no direct implications for the infinite horizon. When A is executed, it must be checked whether it is an actual $notChainSuccession$ or a $deFactoNotSuccession$, which applies the status update rules of a $notSuccession$, depending on the current history.

This possible translation of $notChainSuccession$ rules into $deFactoNotSuccession$ rules is the reason why $notChainSuccession$ rules have to be added due to $chainResponse$ rules as described in Section V-E. The conduction whether a $notChainSuccession$ is a $deFactoNotSuccession$ or an actual $notChainSuccession$ can be done via model verification: For $notChainSuccession(A, B)$ simulate the execution of A based on the current history h (simulated history: $h' = h.A$) and check the model for freeness of conflicts when adding the rule $response(A, B)$. If the original model was conflict-free and now a conflict occurs, it means that B cannot be executed after A and thus, the $notChainSuccession(A, B)$ is a $deFactoNotSuccession(A, B)$.

VI. Implementation

In this section, we demonstrate the functioning of the derived infinite horizon status update rules with a prototypical proof-of-concept implementation of the infinite horizon system in Java. This implementation is a stand-alone program reflecting the infinite horizon mechanism and can be integrated into existing process execution systems. Figure 8 presents a reduced UML class diagram that shows the structure of the infinite horizon update rule system. The rules not shown in Figure 8 indicated through the "incomplete" annotation are included in the same way as the respresented ones. The update rules derived in Section V are implemented as methods in the respective rule classes.

Because we did not implement interfaces yet in the prototype, the process models we checked were entered by hand. If necessary, we added transferred $notSuccesion$ and $notChainSuccession$ rules. The derivation of those tasks that are executable next, i.e., the functionality of ν, can be achieved with an automaton based approach as described in [8]. The same automaton based approach can be used when checking for $deFactoNotSuccessions$ as described in Section V-F: for every $notChainSuccession(A, B)$, history $h = history.A$ with $history$ being the status of the process instance so far and the model has to be entered into the verification system. Additionally, the rule $response(A, B)$ is needed to check whether there are any conflicts or not, i.e., whether B is eventually executable or not. If a conflict occurs, the $notChainSuccession(A, B)$ is a $deFactoNotSuccession$ and the respective update rule has to be added into our system.

Basically, the procedure of the update rules trigger mechanism is the following where start-update rules (s-update rules)

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

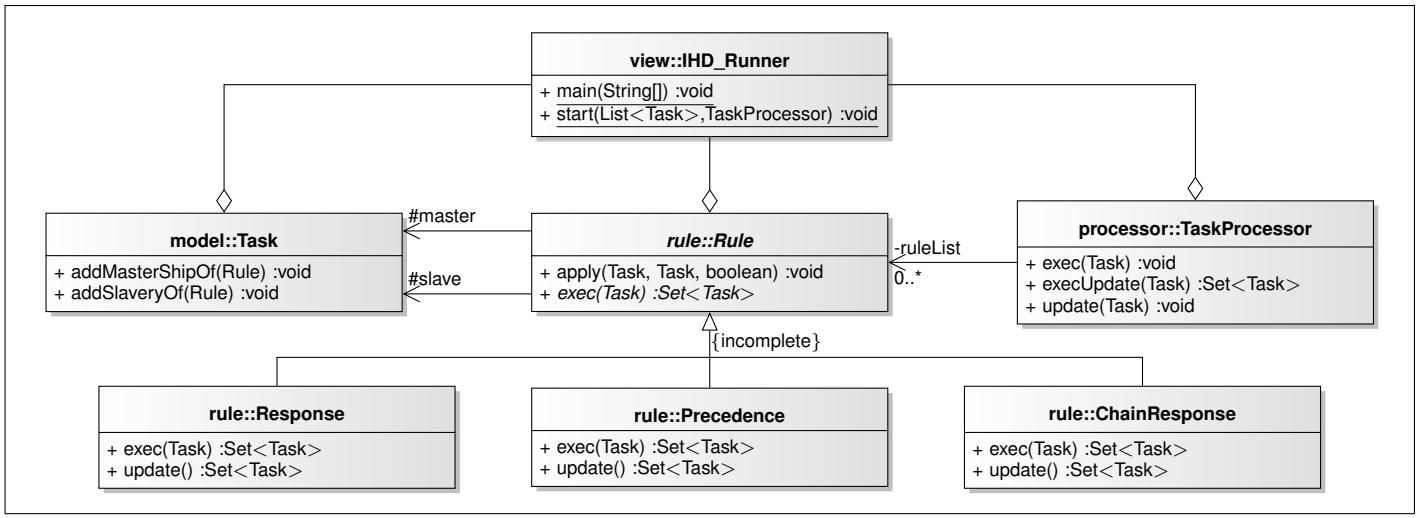

Figure 8: UML class diagramm of the infinite horizon status update rule system.

are those beginning with "If $start$" and execution-update rules (e-update rules) are those beginning with "If $exec(\cdot)$":

1) $h \leftarrow \odot$;
   initialization of status table (mandatory values zero, optional values ∞);
2) start process:
   go through s-update rules once;
   go through other update rules (except e/s-update rules) as many times until nothing more changes;
3) execution of $task \in \nu(h)$:
   $h \leftarrow task$;
   go through e-update rules once;
   go through other update rules (except e/s-update rules) as many times until nothing more changes;
4) execution of $task \in \nu(h)$:
   $h \leftarrow h.task$;
   go through execution-update rules once;
   go through other update rules (except e/s-update rules) as many times until nothing more changes;
5) etc.

The order of the rules when going through s-update rules, e-update rules, or all other update rules is not important, as long as the other update rules are processed until in one run no changes apply to the status values. We tested the behavior of the infinite horizon status update mechanism with several distinct process models and always reached the desired, correct behavior of the status updates. One such execution example is provided in Section VII to illustrate the functionality of the implemented prototype.

VII. Example

As an illustrational example of the infinite horizon decision support we take the process given in Figure 9 representing a (fictional) clinical diagnosis process. The example contains three $precedence$ rules, one $response$ rule, one $respondedExistence$ rule und one $notChainSuccession$ rule. In this example, the $notChainSuccession$ rule never turns into a $deFactoNotSuccession$ rule as task "magnetic resonance tomography" can always be executed. Thus, the $notChainSuccession$ rule has no influence on the infinite horizon. With the abbreviations MRT (magnetic resonance tomography), $assistant$ (assistant physician's round), $blood$ (blood test), $head$ (head physician's round), and $Xray$ (X-ray examination) for the tasks and the status rules applied after initializing the states the table for history $h = \odot$ looks like Table IX.

TABLE IX: Diagnosis process: status table with history $h = \odot$.

Task	$mandatory$	$optional$
MRT	1	∞
$blood$	0	1
$assistant$	1	∞
$Xray$	0	∞
$head$	0	1

Regarding the console output of our prototype in Figure 10, we get the same result.

Table IX says that there are two tasks that need to be done in any case, even in every instance of this process as the history is empty at the moment, namely tasks MRT and $assistant$. All other tasks may be done at some point during an instance, however, not necessarily all tasks in one instance. Now imagine that there is a patient for whom a diagnosis has to be made without performing the X-ray examination in this particular case. Is there a possibility to perform the diagnosis process without this specific task? Table IX states "yes" as the $mandatory$ value of $Xray$ is 0. This means, there is at least one successful execution without having to perform the X-ray examination.

Possible tasks in the first step give by $\nu(\odot)$ are MRT, $blood$ and $XRay$. When executing $blood$, the output of the prototype is shown in Figure 11.

Summarized in a status table, the status values for $h = blood$ are represented in Table X. The status values reveal that if the blood test is done then the X-ray examination eventually needs to be done, too. This is not desired for our patient, so the blood test should not be the choice in the first step.

Instead of the blood test the MRT is executed in the first step (this task has to be executed anyway during the process), so we have history $h = MRT$. The prototype output of Figure

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE X: Diagnosis process: status table with history $h = blood$.

Task	$mandatory$	$optional$
MRT	1	∞
$blood$	0	0
$assistant$	1	∞
$Xray$	1	∞
$head$	1	1

TABLE XI: Diagnosis process: status table with history $h = MRT$.

Task	$mandatory$	$optional$
MRT	0	∞
$blood$	0	1
$assistant$	1	∞
$Xray$	0	∞
$head$	0	1

12 and the summarized status table in Table XI show that this choice is fine since the X-ray examination is still optional, but not mandatory.

The only remaining mandatory task is $assistant$. If this task is performed next, then the diagnosis process can be finished successfully as all $mandatory$ values are 0. This is apparent from both Figure 13 and summarized status Table XII.

So, the answer to the question if the process can be performed without having to do the X-ray examination can be given directly after initializing the tasks' states, but the concrete steps that have to be taken to reach this goal can only be detected through simulation. However, the execution $h = MRT.assistant.end$ is one possibility to finish the process under the given constraints. There may exist others as well.

VIII. Inclusion of the Next Function

The function that returns the tasks executable in the next step is not in the focus of this paper. However, to obtain a satisfactory decision support tool, the functionality of such a mapping ν has to be integrated into the system. At the moment, the output of function ν is generated externally. We

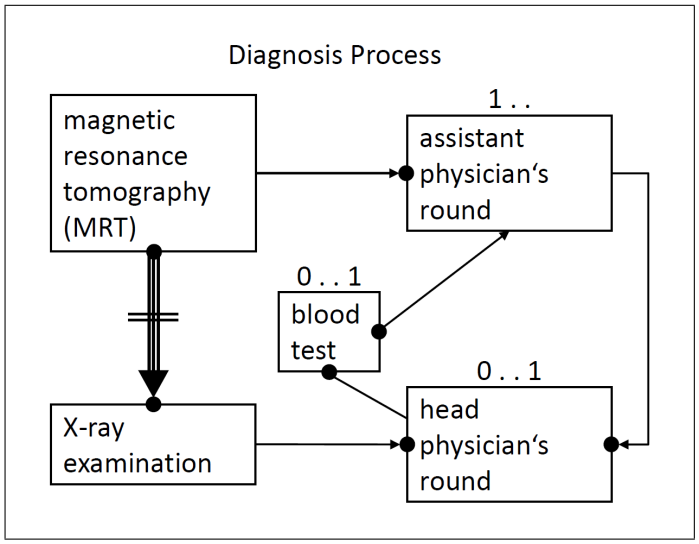

Figure 9: A fictional clinical process model representing a diagnosis process.

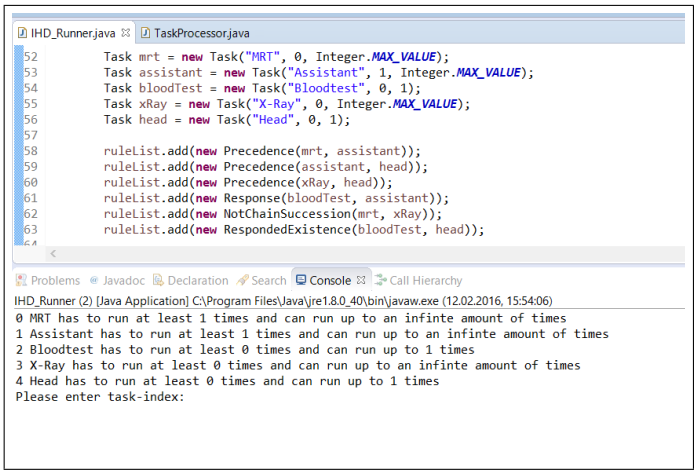

Figure 10: Prototype output of the Diagnosis process for $h = \odot$.

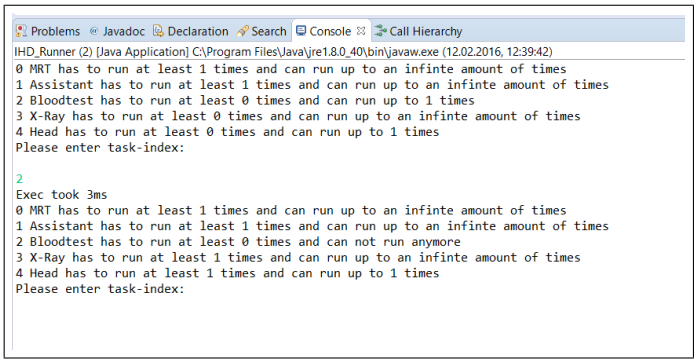

Figure 11: Prototype output of the Diagnosis process for $h = blood$.

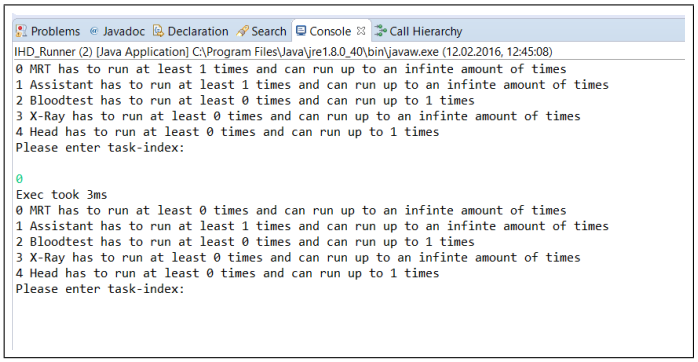

Figure 12: Prototype output of the Diagnosis process for $h = MRT$.

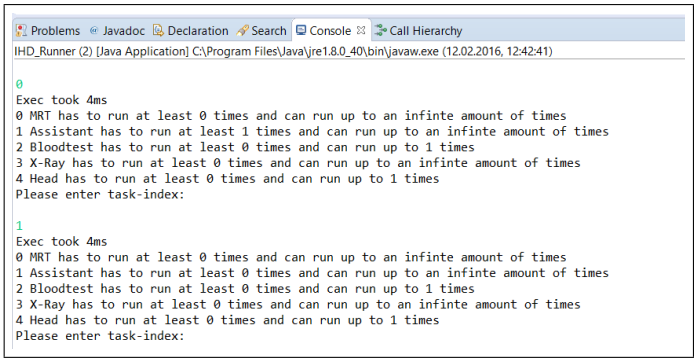

Figure 13: Prototype output of the Diagnosis process for $h = MRT.assistant$.

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE XII: Diagnosis process: status table with history $h = MRT.assistant$.

Task	$mandatory$	$optional$
MRT	0	∞
$blood$	0	1
$assistant$	0	∞
$Xray$	0	∞
$head$	0	1

TABLE XIII: Diagnosis process: user decision support with next executable tasks and their implications with $h = \odot$.

	$mandatory$	$impossible$
Current	MRT, $assistant$	
MRT	$assistant$	
$blood$	MRT, $assistant$, $Xray$, $head$	$blood$
$Xray$	MRT, $assistant$	

are currently working on the inclusion of ν in the same manner as the mandatory and optional values are generated. We add a third type of status value to each task indicating whether a task is locked by certain process rules or not. The update rules have to be extended by locking rules.

The $response$ rule does not have any influence on the locking of tasks, but for the $precendence$ rule, we have to add the following status update rule: $\forall\ precedence(A, B) \in \mathcal{R}$: If $A \notin h$: $lock(B)$. This functionality could be realised with a map in Java to ensure that a lock is released only once as one task may have several locks from different process rules. This locking mechanism wourld detach the dependency on process models representable as finite state automata (see, e.g., [16]) but extent the set of possible process models.

A schematical representation of the user decision support with next executable tasks is depictured in Table XIII. The process is that one of Figure 9 with an empty history. Table XIII shows the current mandatory/optional values not with their concrete values but only distiguishes between a value of 0 or > 0. This current state is shown in the row above the double horizontal line: MRT and $assistant$ have to be executed somewhen to finish the process under the current situation. There are not any non-executable tasks at the beginning. The tasks executable next, i.e., in the first step, are shown in the left column below the double horizontal line as well as their implications on the infinite horizon in the respective rows: the tasks that are mandatory after the execution of one of the next executable tasks and the tasks that are impossible to execute in the further process instance. For example, if $blood$ is executed in the next, i.e., first, step, then it cannot be executed any more but $Xray$ and $head$ get mandatory, too. MRT and $assistant$ remain mandatory as they have not been executed yet. The optional values are indirectly given. If a task has an optional value of zero, it is listed in column $impossible$. For example, $blood$ gets impossible for all future steps after execution of $blood$ in the first step.

IX. Conclusion and Ongoing Research

The work at hand presents a possibility for decision support for declarative process models. The presented decision support applies to an infinitely long forecast horizon. It makes use of the process rules and their implications to the states of the tasks they refer to. The rules considered in this paper only concern the sequence flow of a process so an extension to other rule types, e.g., concerning roles and agents, is worthwhile. Furthermore, the tasks are only points in time, i.e., only their final execution is registered. However, due to concurrency issues it would be beneficial to split one task into different events, at least into $taskstarted$ and $taskfinished$. According to these events, the update rules need to be adjusted. Furthermore, tasks can also serve as containers implying subprocesses, like activities marked as subprocesses in BPMN do. For such nested process models the update rules can also be nested, meaning, the execution of a container task initializes a separate set of update rules for the subprocess. For larger process models, the status value tables can be stored via a hash key once they are computed to reduce runtime.

The mechanism presented in this paper can be seen as an extension for existing decision support systems or process navigation tools where it can be embedded into. Interfaces for the respective target system have to be built then, but the mechanism, i.e., the logic of the task status update rules, can be taken one to one.

Acknowledgement

The work of Michael Heinrich Baumann is supported by a scholarship of "Hanns-Seidel-Stiftung (HSS)", which is funded by "Bundesministerium für Bildung und Forschung (BMBF)." The authors wish to thank Josef Noll (Basic Internet Foundation and University of Oslo, University Graduate Center (UNIK), Norway), session chair at The Tenth International Multi-Conference on Computing in the Global Information Technology (ICCGI 2015) October 11 - 16, 2015 - St. Julians, Malta.

References

[1] M. Baumann, M. H. Baumann, and S. Jablonski, "An idea on infinite horizon decision support for rule-based process models," in *ICCGI 2015, The Tenth International Multi-Conference on Computing in the Global Information Technology*, H. Kaindl, K. György, and D. Tamir, Eds. Think Mind, 2015, vol. 5, pp. 73–75.

[2] A.-W. Scheer, O. Thomas, and O. Adam, "Process modeling using event-driven process chains," *Process-Aware Information Systems*, pp. 119–146, 2005.

[3] P. Wohed, W. van der Aalst, M. Dumas, A. ter Hofstede, and N. Russell, "On the suitability of bpmn for business process modelling," in *Business Process Management*, ser. LNCS, S. Dustdar, J. Fiadeiro, and A. Sheth, Eds. Springer Berlin Heidelberg, 2006, vol. 4102, pp. 161–176.

[4] J. Desel, "Process modeling using petri nets," *Process-Aware Information Systems: Bridging People and Software through Process Technology*, pp. 147–177, 2005.

[5] M. Scaife and Y. Rogers, "External cognition: how do graphical representations work?" *International Journal of Human-Computer Studies*, vol. 45, no. 2, pp. 185–213, 1996.

[6] S. Zugal, J. Pinggera, and B. Weber, "Creating declarative process models using test driven modeling suite," in *IS Olympics: Information Systems in a Diverse World*, ser. LNBIP, S. Nurcan, Ed. Springer Berlin Heidelberg, 2012, vol. 107, pp. 16–32.

[7] M. Pesic and W. van der Aalst, "A declarative approach for flexible business processes management," in *Business Process Management Workshops*, ser. LNCS, J. Eder and S. Dustdar, Eds. Springer Berlin Heidelberg, 2006, vol. 4103, pp. 169–180.

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[8] M. Pesic, H. Schonenberg, and W. van der Aalst, "DECLARE: Full support for loosely-structured processes," in *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*, 2007, pp. 287–287.

[9] D. Fahland, D. Lübke, J. Mendling, H. Reijers, B. Weber, M. Weidlich, and S. Zugal, "Declarative versus imperative process modeling languages: The issue of understandability," in *Enterprise, Business-Process and Information Systems Modeling*, ser. LNBIP, T. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, and R. Ukor, Eds. Springer Berlin Heidelberg, 2009, vol. 29, pp. 353–366.

[10] C. Günther, S. Schönig, and S. Jablonski, "Dynamic guidance enhancement in workflow management systems," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC '12. New York, NY, USA: ACM, 2012, pp. 1717–1719.

[11] W. M. van der Aalst, M. Weske, and D. Grnbauer, "Case handling: a new paradigm for business process support," *Data & Knowledge Engineering*, vol. 53, no. 2, pp. 129 – 162, 2005.

[12] S. Jablonski, "Do we really know how to support processes? considerations and reconstruction," in *Graph Transformations and Model-Driven Engineering*, ser. LNCS, G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, Eds. Springer Berlin Heidelberg, 2010, vol. 5765, pp. 393–410.

[13] M. Baumann, M. Baumann, S. Schönig, and S. Jablonski, "Enhancing feasibility of human-driven processes by transforming process models to process checklists," in *Enterprise, Business-Process and Information Systems Modeling*, ser. LNBIP, I. Bider, K. Gaaloul, J. Krogstie, S. Nurcan, H. Proper, R. Schmidt, and P. Soffer, Eds. Springer Berlin Heidelberg, 2014, vol. 175, pp. 124–138.

[14] W. van der Aalst, T. Weijters, and L. Maruster, "Workflow mining: discovering process models from event logs," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 16, no. 9, pp. 1128–1142, Sept 2004.

[15] W. van der Aalst, M. Pesic, and H. Schonenberg, "Declarative workflows: Balancing between flexibility and support," *Computer Science - Research and Development*, vol. 23, no. 2, pp. 99–113, 2009.

[16] M. Pesic, "Contraint-based workflow management systems: Shifting control to users," Ph.D. dissertation, Technische Universiteit Eindhoven, 2008.

[17] M. Zeising, S. Schönig, and S. Jablonski, "Towards a common platform for the support of routine and agile business processes," in *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2014 International Conference on*, Oct 2014, pp. 94–103.

[18] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, *Informatics: 10 Years Back, 10 Years Ahead*. Springer Berlin Heidelberg, 2001, ch. Progress on the State Explosion Problem in Model Checking, pp. 176–194.

[19] M. Montali, "Specification and verification of declarative open interaction models – a logic-based framework," Ph.D. dissertation, Alma Mater Studiorum Universitá di Bologna, 2009.

[20] S. Jablonski and C. Bussler, *Workflow management: modeling concepts, architecture and implementation*. International Thomson Computer Press, 1996.

[21] P. Dourish, J. Holmes, A. MacLean, P. Marqvardsen, and A. Zbyslaw, "Freeflow: Mediating between representation and action in workflow systems," in *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, ser. CSCW '96. ACM, 1996, pp. 190–198.

[22] Object Management Group, "Case management model and notation version 1.0," 2014. [Online]. Available: http://www.omg.org/spec/CMMN/1.0/PDF/ [accessed: 2015-07-19]

[23] F. Maggi, M. Montali, M. Westergaard, and W. van der Aalst, "Monitoring business constraints with linear temporal logic: An approach based on colored automata," in *Business Process Management*, ser. LNCS, S. Rinderle-Ma, F. Toumani, and K. Wolf, Eds. Springer Berlin Heidelberg, 2011, vol. 6896, pp. 132–147.

[24] F. Maggi, A. Mooij, and W. van der Aalst, "User-guided discovery of declarative process models," in *Computational Intelligence and Data Mining (CIDM), 2011 IEEE Symposium on*, April 2011, pp. 192–199.

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org