

# The Resultmaker Online Consultant: From Declarative Workflow Management in Practice to LTL \*

Mukkamala Raghava Rao  
Industrial PhD student  
IT University of Copenhagen &  
Resultmaker A/S  
rao@resultmaker.com

Thomas Hildebrandt  
IT University of Copenhagen  
Rued Langgaardsvej 7  
2300 Copenhagen, Denmark  
hilde@itu.dk

Janus Boris Tøth  
Resultmaker A/S,  
Vester Farimagsgade 3,  
1606 Copenhagen, Denmark  
jbt@resultmaker.com

## Abstract

We present the process model employed in the Resultmaker Online Consultant (ROC) workflow management system as an example of a declarative workflow language used in practice. We describe and formalize the key primitives of the ROC process model as Linear time Temporal Logic (LTL) formulas, in line with a recent proposal of van der Aalst and Pesic to use LTL as the foundation for flexible declarative process languages. The work is one of the first steps in a recently initiated research project (TrustCare) aiming at contributing to the foundations for workflow management for trustworthy pervasive healthcare services by combining research in formal process models, pervasive user interfaces, and development of research based prototype extensions to the ROC workflow management system.

## 1 Introduction

Research in the flexibility of workflow management systems deals with the problems of how to maintain freedom to select as many different flows as possible in process specifications, how to accommodate dynamic changes of workflow processes, and the ability to easily reuse process descriptions in different contexts [1, 2].

As pointed out in [3] most work on flexibility of workflow has so far focussed on *imperative* process languages, in particular languages based on a notion of flow graphs as employed in the majority of the currently used business process and workflow management systems. However, the authors argue that the use of imperative process languages often leads to *over specification*, which imposes too many con-

straints on the flows and consequently amplifies the need for changes to the specified process. Based on this observation, the authors propose a paradigm shift replacing the imperative process languages with *declarative* process languages, in which one specifies the constraints between work activities rather than exactly how these constraints are resolved. Concretely, they propose the open languages ConDec [3] and DecSerFlow [4], which are based on the idea of using pattern templates for Linear time Temporal Logic (LTL) formulas [5, 6] accompanied by a graphical notation as the primitives of a flexible and extensible declarative process language.

In the present paper we report on our initial work on formalizing the *Process Matrix*, which is the patented<sup>1</sup> declarative process model employed in the Resultmaker Online Consultant (ROC) workflow management system. In line with the approach proposed by van der Aalst and Pesic we describe and formalize the key primitives of the Process Matrix as LTL formulas. The work is one of the very first steps of a recently initiated research project on Trustworthy Pervasive Healthcare Services (TrustCare) [7]. The aim of the project is to contribute to the foundations of workflow management systems for trustworthy pervasive healthcare services by combining research in formal process models, logic and domain specific languages, research in pervasive user interfaces, and prototype development of workflow management systems at Resultmaker. The research will in particular focus on uses of formal process models as a foundation for workflow management systems supporting trustworthy dynamic composition and changes of processes, and the prototype development will involve a re-implementation of the ROC using the knowledge gained from the formalization.

The ROC workflow management system has evolved from Resultmaker's industrial experiences obtained during the process of authoring solutions for the Danish public sec-

\*This work was funded in part by the Danish Research Agency (grant no.: 2106-07-0019, no.: 274-06-0415), Resultmaker A/S, and IT University of Copenhagen (the TrustCare and CosmoBiz projects).

<sup>1</sup>US Patent # 6,895,573

tor, and has been used with success in practice for several years. It is based on a shared data architecture and electronic forms (updating the shared data) as the key basic activity. Hereto comes activities for connecting to external systems, inviting participants and digitally signing data, that we will ignore in the present paper.

The key primitives of the ROC Process Matrix is that of *sequential* and *logical* predecessor relations between activities A and B, along with *activity conditions* and *dependency expressions* for each activity. That A is a sequential predecessor of B informally means that in each instance of the process, activity A (for that instance) must be executed before B (for that instance) can be executed. Note that an action by default can be executed any number of times within an instance as long as it is executed at least once. If A is declared as a *logical* predecessor of B, it informally means that it is a sequential predecessor with the additional constraint that B must be re-executed at some point after any re-execution of A. A prototypical example of logical predecessor is when A is the activity of filling out a loan or grant application and B is the activity of evaluating or signing it. Activity conditions and dependency expressions refer to values of variables in the shared data store and are dynamically evaluated after each step of the workflow. An activity condition determines if an activity is currently included in the workflow instance (i.e. it is active) and a change in a dependency expression determines that an activity must be re-executed. Activity conditions make it very easy to reuse a process description for a different purpose in a different variant: One just adds a new boolean variable to the shared data store and use it to toggle the inclusion or exclusion of activities. Dependency expressions allow for a description of logical dependency similar to the logical predecessor constraint, but are based on changes in data rather than re-executions of activities and thus allow declaring a more fine-grained dependency. In the example of filling out a grant application, one may e.g. use a dependency expression to declare that the signature activity has to be re-executed if the data in the budget is changed, but not if the name of the project is changed, even though both values are entered in the grant application form.

We believe that the study in this paper forms the starting point for a valuable cross-fertilization between development of workflow management systems in practice and research in theoretical computer science. The predecessor primitives of the Process Matrix are similar to the primitives considered by van der Aalst and Pesic, and can thus be quite naturally formalized as LTL formulas. However, the use of activity conditions suggests in our opinion interesting variants of the constraint templates given for the ConDec and DecSerFlow languages. On the other hand, the formalization also suggests extensions to the Process Matrix. In particular one may follow the approach in DecSerFlow and

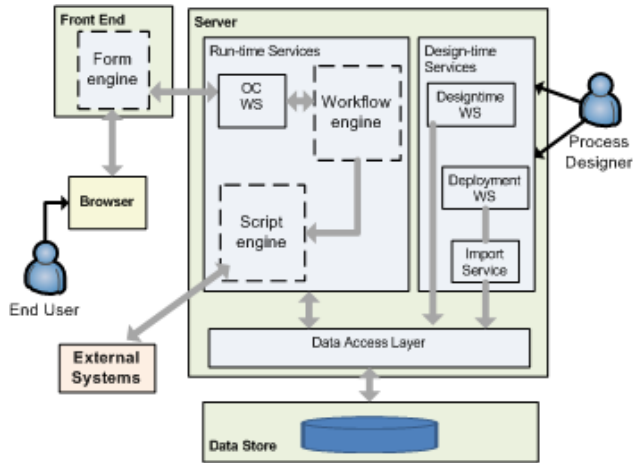


Figure 1. The Online Consultant Architecture.

consider allowing designers to specify new process primitives as LTL formulas based on an open set of templates.

The structure of the paper is as follows. In Sec. 2 we informally describe the ROC Architecture and the Process Matrix using a fictive loan application process as example. In Sec. 3 we recall the approach of the DecSerFlow language and formalize the key primitives of the ROC *Process Matrix* as LTL formulas. We end in Sec. 4 by a conclusion and outlining future work.

## 2 The Online Consultant

In this section we introduce the ROC workflow architecture and key components, in particular the declarative primitives of the ROC process model, referred to as the *Process Matrix*.

### 2.1 The Online Consultant Architecture

ROC is a user-centric workflow management system based on a shared data store and so-called *eForms* as its principal activities. An eForm is a web based questionnaire presented to the users of the system by the front end Form engine. The fields in the eForms are mapped to variables in the shared data store.

Fig. 1 shows the overall architecture of ROC. The Run-time services constitute components that execute a ROC process instance, while the Design-time services constitute e.g. tools for process description and design of eForms. ROC has its own eForm designer tool, but also supports forms developed in Microsoft InfoPath.

### 2.2 Process Modeling Primitives

Below we describe the key process modeling primitives used in ROC.

**Activities:** The ROC has 4 pre-defined activity types.

*eForm Activity:* As described above, it is the principal activity of ROC. The data filled in by the users will be available to all activities of the workflow instance through the shared data store. *eForms* are appended to ROC activities in process definitions and each activity can contain only one eForm. At run-time when an eForm activity is executed, the corresponding eForm will be displayed to the user for human interaction. If any of the variables or activities an eForm activity *A* depends on is changed by another activity while the form is being displayed (and edited) by the user, the activity *A* will be skipped when the form is attempt to submit by the user and the user will be notified. In this way eForm activities are guaranteed to run atomically and in isolation.

*Invitation Activity:* This type of activity attaches a role to an external user (identified by an email address) and sends him an invitation link to the process instance via email notification.

*Signing Activity:* In order to provide authentication for the data filled in by the users, the ROC uses Signing Activity. The user data on eForms will be digitally signed by using XML digital signatures syntax [8] and users digital identity certificates. A single signing activity supports signing of data from multiple eForms.

*External Activity:* Via a general script engine it is possible to connect to any external system, e.g. for automated tasks. In the remainder of the paper we will only consider eForm activities.

**Transactions:** A ROC Transaction holds a group of activities to be executed in transaction mode. The transactions differ from standard transactional semantics in that they are long running and cannot be rolled back. Instead, as also found in web-service orchestration languages such as WS-BPEL, they can have a compensating logic to be executed in case a transaction has to be aborted. Transactions can be either signed or unsigned. Signed transactions involves signing the data using digital certificates by single/multiple parties containing many eForms. We will leave the further investigation and formalization of transactions for future work.

**Resources/Roles:** ROC has a simple resource model that uses Roles to define allowed behaviour of different users within the system. Each Role is assigned an access right for each activity of a workflow. The possible access rights are Read (R), Write (W) and Denied (D). The Read access is the default access right that allows a user with the particular role to see the data of an activity. Write access right allows the user to execute an activity and also to input and submit data for that activity. A Denied access right has the effect of making the activity invisible to the user. As for transactions, we will leave the formalization of Roles for future work.

Activities are executed by default at least once, but pos-

sibly many times in a process instance. The state of ROC records whether an activity has been executed or not in an instance. If an activity has state *executed*, its state can be reset to *not executed* under certain circumstances described below.

**Control Flow Primitives:** ROC contains the following control flow primitives which controls the execution of process instances.

*Activity Condition:* Every activity in ROC has an attached activity condition. An activity condition is a boolean expression that reference variables from the shared data store. When this condition evaluates to true, the activity is included in the workflow instance for execution. If the condition evaluates to false the activity will be skipped from the list of activities stacked for execution. The boolean variables used in activity conditions are referred to as *purposes*. The reason for this terminology is that activity conditions makes it easy to reuse a process description for a different purpose in a different variant: One just adds a new purpose variable and use it in activity conditions to toggle the inclusion of relevant and exclusion of irrelevant activities. Activity Conditions are re-evaluated whenever necessary during the execution of an instance, so the inclusion of an activity in the workflow can be changed within in the lifetime of the workflow instance. As described below, changing an activity from non-active to active may influence the state of other activities that logically depend on the activity.

*Sequential Predecessors:* If *A* is declared to be a sequential predecessor of *B* then in any process instance, activity *A* must be executed before *B* can be executed. However, the sequential predecessor has only effect if the predecessor activity *A* is included in the workflow instance as per its activity condition. That is, if the activity condition for *A* is false at a certain point of time, then activity *B* can be executed even if *A* is a sequential predecessor of *B* and has state non-executed. If activity *A* becomes part of the workflow instance after *B* has been executed (e.g. because the activity condition for activity *A* changes from false to true), it will not affect the execution status of activity *B*.

*Logical Predecessors:* If an activity *A* is declared to be a *logical* predecessor of activity *B*, then *A* is a sequential predecessor of *B*, but in addition, if activity *A* is re-executed, reset, or becomes part of the workflow after activity *B* has been executed and activity *B* is active at the time, then activity *B* is also reset, and thus must be re-executed at a later time (unless it stays in-active for the rest of the instance lifetime, i.e its activity condition continuously evaluates to false). Note that activity resets in this way can propagate through a chain of (currently active) logical predecessors.

As also mentioned in the introduction the *Process Matrix* model includes an additional advanced feature called *dependency expressions*. A dependency expression is a set of expressions attached to an activity. Like activity condi-

tions, dependency expressions can also contain references to variables in the shared store. However, where an activity condition evaluates to a boolean value, a dependency expression can evaluate to any value, and any change in the value of a dependency expression associated to an activity will reset the activity state to non-executed.

### 2.3 The Process Matrix

There is yet no formal graphical notation for ROC workflow processes. However, there is a guideline for how to identify and specify activities, roles/actors and constraints in a tabular format. This table is referred to as the *Process Matrix*, which is also used as name for the process model. Practical experience has shown that the guideline and the Process Matrix have been useful to extract process descriptions from domain experts.

Below we describe a small fictive example of a loan application process represented by the Process Matrix shown in Fig. 2.<sup>2</sup> Each row of the matrix represents an activity of the process: Filling in the application (Application), Registering customer information (Register Customer Info), Approval of the application (Approval 1 and 2), Payment, Express Payment, Rejection and Archive. The columns are separated in 3 parts: The first set of columns describes the access rights for the different roles (*Applicant* (App), the *Case Worker* (CW) and *Manager* (Mgr) in the figure). The Roles columns indicate that the applicant can fill out applications, but the case worker and manager can only read the content of the application. Everyone can register customer information, but only the case worker can perform approval 1 and only the manager can perform approval 2, and both approval steps are invisible to the applicant. The remaining actions can only be performed by the case worker - they can be read by the manager and applicant, except for the archiving which is invisible to the applicant. The next row describes the predecessor constraints, where we indicate by a \* that the predecessor is a logical predecessor. That is, activity Approval 1 has activity Register Customer Info as sequential predecessor and activity application as logical predecessor. Thus, the customer may at any time re-submit the basic info (e.g. address and phone number) without causing a re-execution of the approval activity. However, if the application is changed the approval must also be carried out again. (If only changes in the amount given in the Application activity should cause Approval to be re-executed, one could make the application a sequential predecessor of the Approval activities, but add the amount of the loan as a dependency expression to the Approval activities). Finally, the last row describes the activity condition. For instance, the condition  $Hurry \wedge Accept$  of activity Express Payment

	Activities	Roles			Predecessors	Activity Condition
		App	CW	Mgr		
1	Application	W	R	R		
2	Register Customer Info	W	W	W		
3	Approval 1	D	W	R	* 1,2	
4	Approval 2	D	R	W	* 1,2	$\neg Rich$
5	Payment	R	W	R	* * 3,4	$\neg Hurry \wedge Accept$
6	Express Payment	R	W	R	* * 3,4	$Hurry \wedge Accept$
7	Rejection	R	W	R	* * 3,4	$\neg Accept$
8	Archive	D	W	R	* * * 5,6,7	

Figure 2. Loan application Process Matrix

indicates that the boolean values *Hurry* and *Accept* in the shared data store must both be set to true for this activity to be included in the flow. To fit the table within one column of the paper we have left out a column stating which eForm is attached to an activity, and which values in the shared data store are accessed and changed by the eForm: The Application form changes the variables *Rich* and *Hurry*, and the Approval forms toggle the *Accept* variable. Concretely, in the online example of the loan application process the purposes *Rich* and *Hurry* are set by radio buttons in the eForm attached to the Application activity in step 1, and the purpose *Accept* is toggled in the eForms attached to Approval 1 and Approval 2.

The Activity Conditions in the last column depend on the purposes *Rich*, *Hurry* and *Accept*. A rich applicant only needs an approval from the case worker, while a poor applicant also needs an approval from the manager in the bank. If the purpose *Hurry* is set to true, the application is treated as an express payment. The result is that the Express payment activity (step 6) is included and not the Payment activity (step 5). Conversely, if the purpose *Hurry* is set to false, the (normal) Payment activity in step 5 is included and not the express payment activity. Both payment activities require the purpose *Accept* to be true.

### 2.4 Process Execution

In Fig. 3 we show a possible state of the system during an instance of the workflow where a poor applicant applies for a non-express loan. The purpose *Hurry* is set to false thus the activity Express Payment is excluded. The activity condition for all other activities except activity Express Payment is set to true and they are included for execution, i.e.

<sup>2</sup>The example is available for online demonstration at ([www.resultmaker.com](http://www.resultmaker.com)).

	Activities	Activity Condition	Activity Status
1	Application	true	executed
2	Register Customer Info	true	executed
3	Approval 1	true	can start
4	Approval 2	true ( $\neg Rich$ )	executed
5	Payment	true	can not start (wait for {3})
6	Express Payment	false( $\neg Hurry$ )	inactive ( $\neg Hurry$ )
7	Rejection	true	can not start (wait for {3} $\wedge$ $\neg Accept$ )
8	Archive	true	can not start (wait for $(\{3\} \wedge \{4\}) \vee (\{3\} \wedge \{7\})$ )

**Figure 3. The Process Matrix at Run Time.**

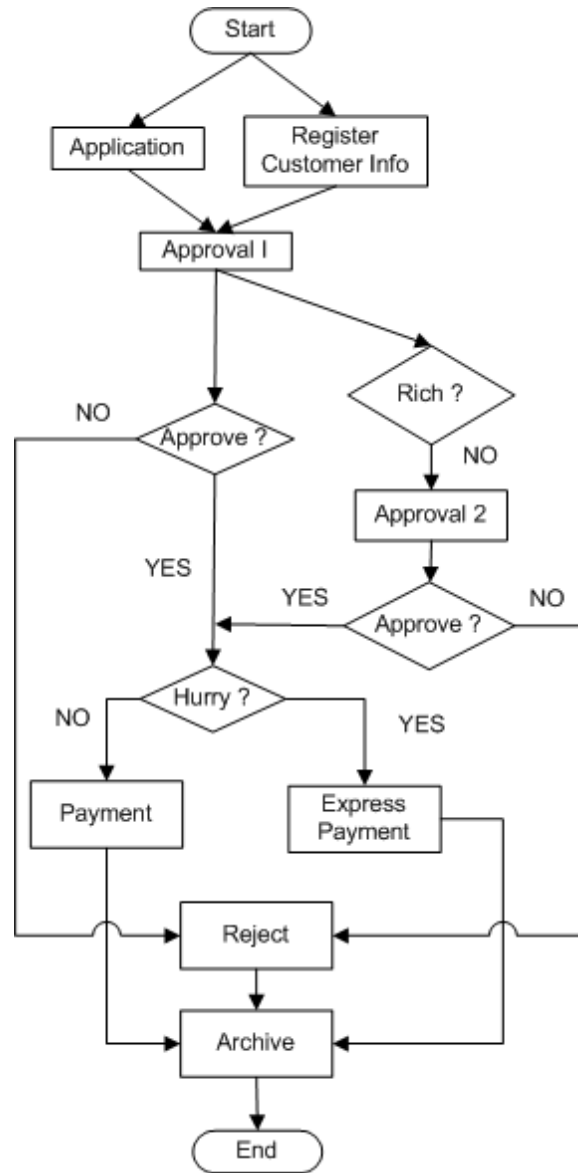
the activity Approval 2 is included because the purpose *Rich* evaluates false. The activities Application, Register Customer Info, Approval 2 have already been executed and their activity status is thus executed. The activity Approval 1 is ready for execution, but it has not started executing. Note that the activities Payment and Rejection can not be started because of their predecessors, but only one of them will be executed in future as the value of purpose *Accept* makes the other activity to be excluded. The activity Archive will be executed eventually after all its predecessors, as it does not have any purposes attached to it. As mentioned above, activity conditions will be re-evaluated after execution of each activity which makes the dynamic inclusion or exclusion of activities possible at runtime.

Note that the registration of customer information can be done either before or after the application, and can be redone arbitrarily often without affecting any of the other steps.

### 2.5 Process Matrix vs Flow Charts?

We conclude the presentation of the *Process Matrix* by a very brief comparison of the Process Matrix for the loan application to a typical flow chart description as shown in Fig. 4. Note that the diagram only shows a possible description of the loan process that one may come up with during design - it is not equivalent to the process described by the Process Matrix. In particular, it assumes that every activity is carried out once. To model the flexibility provided in the Process Matrix, that every step can be re-executed, one

would need loops back from every activity to any preceding activity. Clearly, adding these arcs to the flow chart diagram would make it much more complex.



**Figure 4. Example in Flow chart.**

## 3 Formalizing the Online Consultant in LTL

In this section we provide formalizations of the key primitives of the Online Consultant process matrix described in Sec. 2 in terms of Linear time Temporal Logic (LTL) [5, 6] formulas. First we briefly recall LTL and the approach in [4, 3].

### 3.1 Executable LTL for Workflow

LTL is a temporal logic extending propositional logic to infinite sequences of states. This is done using the temporal modal operators  $\mathbf{O}P$  (in the next state of the sequence formula  $P$  holds),  $\mathbf{\square}P$  (in the current and all of the following states of the sequence formula  $P$  holds),  $\mathbf{\diamond}P$  (in the current or at least one of the following states of the sequence formula  $P$  holds), and  $Q \mathbf{U} P$  (in the current or at least one of the following states of the sequence formula  $P$  holds and formula  $Q$  holds in all states *until* that state is reached).

LTL has been extensively used as property language for automatic verification of reactive systems, also referred to as *model checking* [9]. The basic principle of model checking is to use an automatic tool to check if a system, usually described by an automaton, satisfies a property specified in a property language, which is often a temporal logic. In this case one says that the system is a *model* of the property.

The key idea of the paradigm shift proposed in [4] is to turn this around and use the declarative, temporal logic language to provide the system (workflow) definition. The system is then defined as a formula that characterizes the valid completed sequences of activities, e.g. that in a completed instance execution a certain activity must always occur before some other activity.<sup>3</sup> In acknowledgement to the fact that LTL formulas may be too difficult to understand for process designers, the authors in [4] propose to use so-called *constraint template formulas*, also referred to as policies or business rules. These templates are further equipped with a graphical notation.

It is worth noting, that a similar paradigm shift was in fact also proposed by Gabbay in [10] where he suggests to use LTL formulas as execution language for interactive systems. Moreover, Gabbay showed that one could ease the description of systems by using LTL extended with past time modalities by proving that any LTL formula with past time modalities can be rewritten to an equivalent (but in the worst case exponentially longer [11]) LTL formula with only future time modalities. We exploit the use of past time modalities below to give more succinct formalizations of the activity resets in ROC.

It is important to recall, that the difference between using a declarative language as opposed to an imperative language is on the *ease* and *flexibility* of expression and not on expressiveness: Any LTL formula can be automatically translated to an equivalent finite automaton over infinite sequences and vice versa [6]. The point made in [4, 10] is that one may use this correspondence to let the workflow engine construct an automaton from the declarative LTL description that can be used for execution of the process.

<sup>3</sup>Note that a partial execution sequence need not satisfy the formula, as long as it is possible to complete the sequence in a way that makes the formula satisfied.

As described in the previous section the Process Matrix employed in ROC is in fact an example of a declarative workflow language used in practice. Our aim in the following is to give a translation from the Process Matrix model to LTL, which translates any Process Matrix process  $M$  into an LTL formula  $\llbracket M \rrbracket$  such that the sequences of states for which  $\llbracket M \rrbracket$  is true is exactly the sequences of states that constitute valid executions of the process  $M$ . Concretely, our formalization is defined as extensions to the LTL template formulas given in [4, 3]. As in [4, 3] we assume a discrete time model where any step between two consecutive states in the sequence corresponds to the execution of one activity in the workflow, and we deal with the fact that workflow executions are finite and LTL is interpreted over infinite sequences by using the standard stutter extension, assuming that the finite workflow executions are terminated by an infinite sequence of steps with no change in the state. The basic propositional formulas we employ will be boolean formulas over propositions on the state space and the current activity. In particular, the proposition ( $\mathbf{act} == A$ ) is true in a state if the last executed activity is  $A$ .

A basic example of an LTL template in the DecSerFlow language is the constraint template *existence*( $A : activity$ ) formalized as  $\mathbf{\diamond}(\mathbf{act} == A)$  in LTL. It simply states that there exists a step in which activity  $A$  is carried out.

An example of a so-called *relation formula* [4] is the constraint *precedence*( $A : activity, B : activity$ ) which states that an activity  $B$  is preceded by an activity  $A$ , i.e. the activity  $B$  can not be executed before activity  $A$  has been executed. This template formula uses the existence template as a sub formula and is expressed in LTL as

$$existence(B) \implies (!(\mathbf{act} == B) \mathbf{U} (\mathbf{act} == A))$$

where  $!$  denote the the boolean negation. Reading the formula, it expresses that if there exists a state in the sequence in which  $B$  is carried out then there exists a state in the sequence in which  $A$  is carried out for which  $B$  is not carried out in any of the preceding states. This is equivalent to the intended property that the activity  $B$  can not be executed before activity  $A$  has been executed.

Another example of a relation formula is the constraint *response*( $A : activity, B : activity$ ) which expresses that whenever the activity  $A$  is executed then  $B$  must also be executed after it. This formula is expressed in LTL as

$$\mathbf{\square}((\mathbf{act} == A) \implies existence(B))$$

From the response and precedence templates one may build composite relation templates, such as the template *succession*( $A : activity, B : activity$ ) expressed in LTL simply as a conjunction of the two templates:

$$response(A, B) \wedge precedence(A, B)$$

The formula expresses that every execution of activity  $A$  must be followed by an execution of  $B$  and any execution of  $B$  must be preceded by an execution of  $A$ .

The reader may already have noticed similarities with the primitives in the Process Matrix. In the following section we will see that the Process Matrix primitives can indeed be formalized similarly to the templates given above, but with some interesting variations due to the use of activity and dependency conditions. We do not consider the roles nor dependency expressions.

### 3.2 From the Process Matrix to LTL

To define the translation from the Process Matrix model to LTL we describe how the individual primitives can be expressed as templates in LTL. The formalization of a Process Matrix workflow  $M$  will then be an LTL formula  $\llbracket M \rrbracket$  which is a set of formulas in conjunction obtained by instantiating the templates according to the entries in the Process Matrix. Our aim is that  $\llbracket M \rrbracket$  is true exactly for the sequences of states that constitute valid executions of the process  $M$ . However, we leave for future work to evaluate the correctness of the formalization.

In the following we assume a Process Matrix workflow  $M$ . We let  $A$  and  $B$  range over activities in  $M$  and write  $actcon(A)$  for the activity condition specified in the Process Matrix  $M$  for an activity  $A$ .

The first formula used for the formalization is then the LTL formula  $act\_include(A : activity)$  given by

$$\Box(\mathbf{O}(\mathbf{act} == A) \implies actcon(A))$$

It expresses that an activity  $A$  can only be executed in the next step if it is included in the present, i.e. its activity condition is true. The formula  $act\_include(A)$  is then included in the conjunction in  $\llbracket M \rrbracket$  for every activity  $A$  in  $M$ .<sup>4</sup>

To formalize the remaining ingredients we define a few templates used as sub formulas. The first such template is  $act\_including(A, B) = (\mathbf{act} == A) \wedge actcon(B)$  which expresses that activity  $A$  is executed and at the same time the activity  $B$  is included in the process (because the activity condition for  $B$  is true).

The second template is  $existence\_act\_including(A, B) = \Diamond act\_including(A, B)$  which extends the existence template for DecSerFlow to express that an activity  $A$  is eventually executed and at the same time the activity  $B$  is included in the process.

We now go on to formalize the control flow primitives of the Process Matrix.

**Sequential Predecessor:** The sequential predecessor constraint is similar to the precedence formula in DecSerFlow

<sup>4</sup>We also include the formula  $\bigwedge_{A \in M} \Box(\mathbf{act} == A)$  in the conjunction stating that no activities are carried out before the initial state.

described above, except for the use of the activity condition in the Process Matrix. We define the constraint template  $sequential\_predecessor(A : activity, B : activity)$  stating that  $A$  is a sequential predecessor of  $B$  by the LTL formula  $existence\_act\_including(B, A) \implies (\neg act\_including(B, A) \mathbf{U} (\mathbf{act} == A))$ . Let  $A <_M B$  denote that  $A$  is a sequential predecessor of  $B$  in  $M$ . We then include the formula  $sequential\_predecessor(A, B)$  in the conjunction  $\llbracket M \rrbracket$  for any pair  $A <_M B$ .

**Activity Reset:** To formalize the logical predecessor constraint, we need to formalize the somewhat complex handling of *activity resets* in ROC. We want to define a template  $reset(A)$  which expresses that the activity  $A$  is being reset in the current state. Here we exploit the past time modality *Since* written as  $Q \mathbf{S} P$  and the past time modality *YP*. The *Since* modality is the dual of the until modality and is true if in the current or at least one of the *preceeding* states the formula  $P$  holds and formula  $Q$  holds in all states *since* that state. The past time modality *YP* is true if  $P$  holds "Yesterday", i.e. in the previous state. As described in [10] we can translate the formalization including past time modalities into a pure present and future time formula.

Let  $A \overset{*}{<}_M B$  denote that  $A$  is a logical predecessor of  $B$  in  $M$ . If there is a chain of logical predecessors  $A_0 \overset{*}{<}_M A_1 \overset{*}{<}_M \dots \overset{*}{<}_M A_k$ , for which  $actcon(A_i)$  is true for  $i \in \{0, \dots, k\}$ , i.e. the activities  $A_i$  are all included in this state, and the first activity  $A_0$  is executed or changes from not-included in the previous state to included in this state, then the activity  $A_k$  will be reset in the Process Matrix. To formalize this, first define the template  $included(A : activity) = \mathbf{Y} \neg actcon(A) \wedge actcon(A)$  and define  $chain(A_0, A) = \{[A_0, A_1, \dots, A_k] \mid A_0 \overset{*}{<}_M A_1 \overset{*}{<}_M \dots \overset{*}{<}_M A_k = A\}$ , i.e. the set of all chains of logical predecessors with  $A_0$  as first and  $A$  as the last activity. Then we define the template  $resetchain(A) = \bigvee_{B \in M, c \in chain(B, A)} (\bigwedge_{A' \in c} actcon(A') \wedge (included(B) \vee (\mathbf{act} == B)))$ .

Finally, we define the template  $reset(A) = \neg(\mathbf{act} == A) \mathbf{S} resetchain(A)$ , which we will use below.

**Logical Predecessor:** Logical Predecessor is a strengthening of the Sequential Predecessor constraint. The template  $reset(A)$  allows us to formalize the template  $logical\_predecessor(A : activity, B : activity)$  in LTL as  $sequential\_predecessor(A, B) \wedge \Box(reset(A) \implies sequential\_predecessor(A : activity, B : activity))$  We then include the formula  $logical\_predecessor(A, B)$  in the conjunction  $\llbracket M \rrbracket$  for any pair  $A \overset{*}{<}_M B$ .

**Activity Execution:** The final part of the formalization, is to express when an activity should be executed. We use the template  $executed(A : activity) = \neg reset(A) \mathbf{S} (\mathbf{act} == A)$ , i.e. using the template  $reset(A)$  and the since modality to describe that an activity has status executed if there exist a

state in the past where it is executed and it has not been reset since. The execution formula can then finally be formalized as

$$(\diamond\Box\text{executed}(A)) \vee (\diamond\Box!\text{actcon}(A))$$

which is included in the conjunction  $\llbracket M \rrbracket$  for every activity  $A$  in  $M$ . The formula expresses that either the activity  $A$  has status executed continuously in some future state, or it is excluded from the process. (Recall that we interpret LTL over infinite sequences and assume the execution sequences of ROC to be terminated by an infinite sequence of states with no change)

This concludes our formalization of the Process Matrix.

## 4 Conclusion and Future Work

We have presented the process modeling primitives employed in the Resultmaker Online Consultant workflow management system as an example of a flexible declarative workflow language used in practice. We formalized the core primitives (sequential predecessor, logical predecessor, and activity conditions) as LTL formulas along the lines of the recently proposed approach in [4, 3]. The somewhat complex propagation of activity resets were the most challenging part. The use of activity conditions suggests, in our opinion, interesting variants of the constraint templates given for the ConDec and DecSerFlow languages. We want to remark that the formalization has not yet been validated experimentally, which we plan to do as the next step to support the claim that the formalization is really describing the valid execution traces. One should be aware that other temporal logics for specifying computations exists, notably the *Computational Tree Logics* CTL\* and CTL [9], which are so-called branching time logics that differ from LTL with respect to the ability to specify *when* decisions are made in a process. We leave for future work the possible uses of branching time logics and primitives derived from such logics for specifying work flow processes.

We believe that the study in this paper forms the starting point for a valuable cross-fertilization between development of workflow management systems in practice and research in theoretical computer science. The formalization suggests extensions to ROC and the Process Matrix, e.g. allowing designers to specify new process primitives as LTL formulas based on an open set of templates as in ConDec and possibly using similar graphical notation. As future work we will research how the formalization can be used to support prototype implementations of ROC, and extensions to allow safe dynamic changes and compositions of processes.

Future work will also include studies of the use of the ROC Process Matrix for healthcare services as initiated in [12]. In practice, clinical guidelines are right now being implemented using the Process Matrix as part of a Polish Electronic Health Record project (EHR-PL).

## References

- [1] Heintz, P., Horn, S., Jablonski, S., Neeb, J., Stein, K., Teschke, M.: A comprehensive approach to flexibility in workflow management systems. In: In Proceedings of WACC '99, ACM Press (1999) 79–88
- [2] van der Aalst, W.M.P., Jablonski, S.: Dealing with workflow change: identification of issues and solutions. *International Journal of Computer Systems Science & Engineering* **15**(5) (2000) 267–276
- [3] van der Aalst, W., Pesic, M.: A declarative approach for flexible business processes management. In: In Proceedings of Workshop on Dynamic Process Management (DPM 2006). Volume 4103 of LNCS., Springer Verlag (2006) 169–180
- [4] van der Aalst, W., Pesic, M.: DecSerFlow: Towards a truly declarative service flow language. In Bravetti, M., Nunez, M., Zavattaro, G., eds.: In Proceedings of Web Services and Formal Methods (WS-FM 2006). Volume 4184 of LNCS., Springer Verlag (2006) 1–23
- [5] Pnueli, A.: The temporal logic of programs. In: In Proceedings of 18th IEEE FOCS. (1977) 46–57
- [6] Sistla, A., Vardi, M., Wolper, P.: Reasoning about infinite computation paths. In: In Proceedings of 24th IEEE FOCS. (1983) 185–194
- [7] Hildebrandt, T.: Trustworthy pervasive healthcare processes (TrustCare) research project. Webpage (2008) (<http://www.trustcare.dk/>).
- [8] D. Eastlake, J. Reagle, D.S.: Rfc 3275: Xml-signature syntax and processing (2002) (<http://www.ietf.org/rfc/rfc3275.txt>).
- [9] M. Clarke, E., Grumberg, O., A. Peled, D.: Model Checking. MIT Press (1999)
- [10] Gabbay, D.M.: The declarative past and imperative future: Executable temporal logic for interactive systems. In: Temporal Logic in Specification, London, UK, Springer-Verlag (1987) 409–448
- [11] Laroussinie, F., Markey, N., Schnoebelen, P.: Temporal logic with forgettable past. In: In Proc. 17th IEEE Symp. Logic in Computer Science (LICS'2002), Copenhagen, Denmark, IEEE Computer Society Press (2002) 383–392
- [12] Lyng, K.M., Hildebrandt, T., Mukkamala, R.R.: From paper based clinical practice guidelines to declarative workflow management. In: In Proc. of 2nd International Workshop on Process-oriented information systems in healthcare (ProHealth 08), Milan, Italy (2008)