Modelling and Analysis of Flexible Healthcare Processes Based on Algebraic and Recursive Petri Nets

Awatef Hicheur[1], Amel Ben Dhieb[2], and Kamel Barkaoui[1]

[1] Cedric-Cnam Paris, France
[2] LSITI Enit, Tunis, Tunisia
{awatef.hicheur,kamel.barkaoui}@cnam.fr

Abstract. Healthcare involves distributed and interacting processes which have to be handled in a flexible way due to the variety of individual patient state of health and different kinds of exceptions and deviations that may occur. First, we show how recursive and algebraic workflow Nets (RecWF-Nets) are a promising formalism for modelling and analysis of flexible medical treatment processes where data management and control flow aspects are closely related. Secondly, owing to their semantics defined in terms of generalized rewriting logic, we show that we can check efficiently generic and medical properties of healthcare processes using the Maude LTL model checker.

Keywords: Workflow technology, Distributed and flexible healthcare processes, Recursive and algebraic workflow Nets, LTL Model Checking.

1 Introduction

The recent push for healthcare reform has lead healthcare organizations to reengineer their processes in order to deliver high quality care while at the same time reducing costs and improving their financial assets [6], [10]. For these reasons, the clinical staff tries to optimize patient treatment time in any possible way while keeping the same quality of service by modelling and automating their healthcare processes. Workflow Management Systems (WfMSs) are used in a minority of clinical processes. This is due, in particular, to the fact that healthcare processes (or careflows) are highly unpredictable and extremely dynamic [4], [10], [15], [20] and therefore this poses flexibility requirements on their modelling. Many kinds of exceptions and deviations always occur in clinical processes. Moreover, complex, distributed and interacting processes, with different level of granularity, are often involved in healthcare. Consequently, specifying a real-life healthcare workflow is prone to errors. Incorrectly specified healthcare workflows result in erroneous situations, which may cause disastrous problems in the clinical organisation where they are deployed or more dangerously, on a patient health. Therefore, it is crucial to be able to verify the correctness of a workflow definition before it becomes operational by means of rigorous analysis techniques. For instance, we need to be able to check that a healthcare process always terminates correctly or that contraindications are never administrated. In this paper, we show how we can use Recursive Workflows Nets (abbreviated RecWF-Nets) [4] to model flexible and distributed healthcare processes,

J. Weber and I. Perseil (Eds.): FHIES 2012, LNCS 7789, pp. 1–18, 2013.

allowing the users to deviate from the pre-modelled process plan during run-time by offering other alternatives (i.e. creating, deleting or reordering some sub-processes). RecWF-Nets are a sub-class of the Recursive ECATNets [1] which are a special kind of high-level algebraic nets offering a practical *recursive* mechanism for a direct and intuitive support of the dynamic creation, suppression and synchronization of concurrent processes. Furthermore, we show how to check if defined healthcare processes behave correctly before putting them into production. We distinguish two types of properties which must be met by healthcare processes: generic properties related to their control-flow correctness and domain sensitive (medical) properties related to their medical quality and safety requirements [3]. For this purpose, we use the MAUDE system [7] (an implementation system of the rewriting logic [5]) as a simulation environment for the RecWF-nets specifications. In this framework, the LTL (Linear Temporal Logic) Model-Checker tool of MAUDE [24] is used on the RecWF-nets prototypes to check their general and medical sensitive properties. The rest of the paper is organized as follows: In section 2, we discuss flexibility requirements of healthcare workflows. In section 3, we recall the semantics of the Recursive Workflow Nets. In section 4, we illustrate the appropriateness of Recursive Workflow Nets in the healthcare domain through a simple but significant oncology treatment example. In section 5, we show how some properties of healthcare processes can be formally verified using the LTL model-checker of the MAUDE system. In Section 6, we discuss links to related works. Section 7 concludes this paper and provides directions for future studies.

2 Flexibility Requirements of Healthcare Workflows

Workflow models such as those conceived by classic WfMSs are a description of a process of ideal work generally represented in a rigid way [15], [16], [21]. Such representations are not well suited to the reality of organizations where processes are often led to deviate from their initial plans like in healthcare organizations. In fact, healthcare workflows involve coordination of a heterogeneous set of professionals, patients, organizations and sectors and must be able to adapt to inevitable changes of treatment processes, organizational rules [13], environmental conditions and patients requirements. This can be done by opening alternate execution paths, which may not have been foreseen at design-time and not explicitly catered for by the process modelling [20], [12]. This fact challenges traditional WfMSs using an imperative process modelling language such as Business Process Modelling Notation (BPMN) in which the control flow is modelled explicitly. Declarative process languages, allowing any flow that fulfils the specified constraints, have been suggested by a number of researchers as being more appropriate for representing workflow processes requiring a high degree of flexibility [13], [16]. The need for flexible workflow systems and the deployment of standard processes (so called “clinical pathways”) [10] seem to be crucial to deliver high-quality services and to reduce staff idle time.

Recently, diverse approaches for enhancing flexibility in workflow processes are proposed where this flexibility requirement is interpreted in different manner, following its application domain [18], [23]. In [18], a set of five distinct approaches are recognized and resumed in the following flexibility patterns:

(1) *Flexibility by design* involves the introduction of advanced modelling constructs into a process model, at design-time, such as cancellation or multiple instantiation of sub-processes.

(2) *Flexibility by underspecification* involves the partial definition of a process model at design-time where the whole structure of some sub-processes will become known only during the execution time by allowing, for example, the late selection or the late modelling of process fragments. (3) *Flexibility by deviation* involves allowing process instances to temporarily deviate from their process definition at runtime, for example, by bypassing or to undoing some tasks. (4) *Flexibility by momentary change* involves changing the structure of a process instance at runtime (for example, by adding or deleting some tasks). (5) *Flexibility by permanent change* involves changing the structure of a process definition at runtime, taking into account the impact of these modifications on all its running process instances (migration instances problem). Based on this referential, an important question to ask is which kind of flexibility is the more adequate for healthcare processes [12], [21], [20]. In this paper, we propose a modelling approach for healthcare processes, based on recursive workflow nets (RecWF-nets), where we focus on design-time flexibility (by design and underspecification). In a first step, these two types of flexibility are sufficient [20] to handle with the required exception mechanisms (e.g. interruption of processes) and the advanced behaviours (e.g. recursion and multiple instantiation of processes) encountered in almost healthcare processes.

3 Recursive Workflow Nets

3.1 Recursive ECATNets Review

Recursive ECATNets (abbreviated RECATNets) [1], [4] are a kind of high level algebraic Petri nets combining the expressive power of abstract data types and Recursive Petri nets [11]. Each RECATNet is associated to an algebraic specification. Each place in such a net is associated to a sort (i.e. a data type of the underlying algebraic specification). A place can contains tokens which are multisets of closed algebraic terms of the same sort of this place. Moreover, transitions are partitioned into two types: *abstract* transitions and *elementary* transitions. Each abstract transition is associated to a *starting marking* represented graphically in a frame.

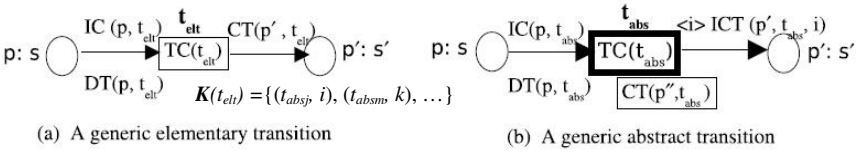

Fig. 1. Transition Types

In a RECATNet, an arc from an input place p to a transition t (elementary or abstract) is labelled by two algebraic expressions $IC(p, t)$ (*Input Condition*) and $DT(p, t)$ (*Destroyed Tokens*). The expression $IC(p, t)$ specifies the partial condition on the marking of the place p for the enabling of t (see table 1). The expression $DT(p, t)$

specifies the multiset of tokens to be removed from the marking of the place p when t is fired. Also, each transition t may be labelled by a Boolean expression $TC(t)$ which specifies an additional enabling condition on the values taken by *contextual* variables of t (i.e. local variables of the expressions labelling all the input arcs of t). When the expression $TC(t)$ is omitted, the default value is the term *True*.

Table 1. The different forms of the expression $IC(p, t)$ for a given transition t

$IC(p, t)$	Enabling condition
α^0	The marking of the place p must be equal to α (e.g. $IC(p, t) = \varnothing^0$ means the marking of p must be empty).
$\alpha+$	The marking of the place p must include α (e.g. $IC(p, t) = \varnothing^+$ means condition is always satisfied).
$\alpha-$	The marking of the place p must not include α, with $\alpha \neq \varnothing$.
$\alpha 1 \wedge \alpha 2$	Conditions $\alpha 1$ and $\alpha 2$ are both true.
$\alpha 1 \vee \alpha 2$	$\alpha 1$ or $\alpha 2$ is true.

For an elementary transition t, an output arc (t, p') connecting this transition t to a place p' is labelled by the expression $CT(p', t)$ (*Created Tokens*). However, for an abstract transition t, an output arc (t, p') is labelled by the expression $< i > ICT(p,' t, i)$ (*Indexed Created Tokens*). These two algebraic expressions specify the multiset of terms created in the output place p' when the transition t is fired.

Note that in a RECATNet, a capacity associated to a place p specifies the number of algebraic terms which can be contained in this place for each element of the sort associated to p. In the graphical representation of a RECATNet, if $IC(p, t) =_{def} DT(p, t)$ on an input arc (p, t) (i.e. $IC(p, t) = \alpha^+$ and $DT(p, t) = \alpha$), the expression $DT(p, t)$ is omitted on this arc. Fig2 illustrates an example of a RECATNet associated to an underling algebraic specification *SpecRecatnet* (described in Fig. 2) and its unbounded places are of the sort *Data* and *CoupleData*.

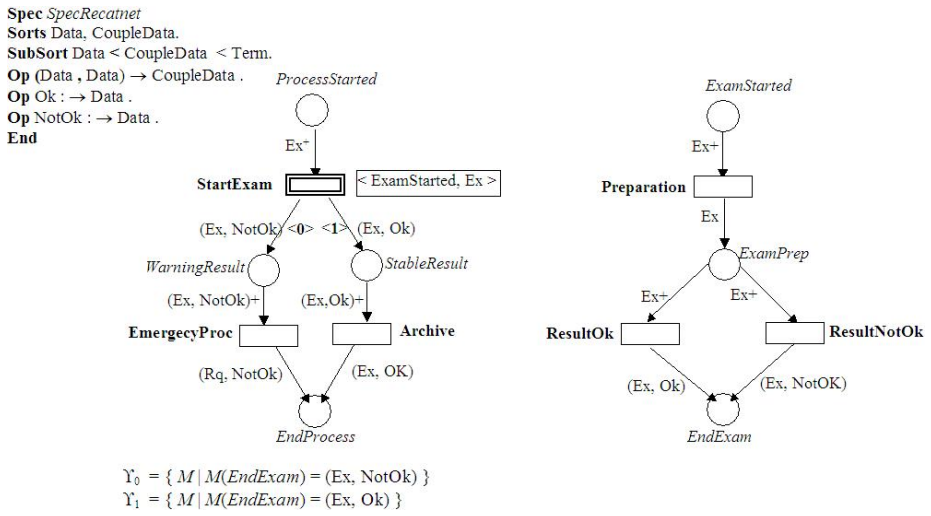

Fig. 2. A RECATNet example

This net has an abstract transition ***StartExam*** (associated to the starting marking <*ExamStarted*,*Ex*>), five elementary transitions ***Preparation***, ***ResultOk, Archive, ResultNotOk*** and ***EmergencyProc*** and two termination sets Υ_0 and Υ_1.

On the arc (*ProcessStarted*, *StartExam*), *IC*(*ProcessStarted*, *StartExam*) = Ex^+ and *DT*(*ProcessStarted*, *StartExam*) = *Ex*, consequently only the expression Ex^+ is represented in Fig.2.

Informally, the behaviour of a RECATNet can be explained as follows: First, let us note that a particular feature of such net is that there is a clear distinction between the *firing condition* of a given transition *t* (i.e. condition on the marking of its input place *p*) and the tokens which may be destroyed from this place *p* during *the firing action* of *t* (respectively specified via the expression *IC*(*p, t*) and *DT*(*p, t*)).

Secondly, a RECATNet generates during its execution a dynamical tree of *marked threads* (i.e. sub-processes with an internal marking describing the tokens distribution over their places) called *an extended marking*, which reflects the global state of such net. This latter denotes the fatherhood relation between the generated threads (describing the inter-thread calls). Each of these threads has its own execution context. All threads of such tree can be executed simultaneously independently from each other i.e. a thread can't access to other threads' internal states. A *step* (an *event occurrence*) of a RecWF-net is thus a step of one of its threads. There are three types of events in a RECATNet: the firing of an *abstract transition*, the firing of an *elementary transition* or a *cut step execution*.

The evaluation of the firing conditions of a transition *t* (elementary or abstract) is always done under a *firing mode* (noted *sub*). *A firing mode is* derived from a consistent *substitution* of the contextual variables of this transition. A transition *t* is then enabled in a mode *sub* (we say that *t* is *sub-enabled*) if under this particular variable's substitution, (1) the transition condition *TC*(*t*) is evaluated to true and (2) the evaluation of the input condition *IC*(*p, t*) of each input arc (*p, t*) of this transition *t* is satisfied in the current marking of its input place *p* and finally (3), the addition of the created tokens to each output place of *t* must not result in exceeding the capacity of this place when this capacity is finite. So, if a transition *t* is *sub-enabled* in a thread, it *may fire* in this same mode *sub*. In Fig. 3, we give a firing sequence of the RECATNet of Fig.2, where a black node in the depicted tree of threads denotes the thread in which the following step is fired. For the sake of clarity, each thread is associated to its internal marking, noted in a grey frame.

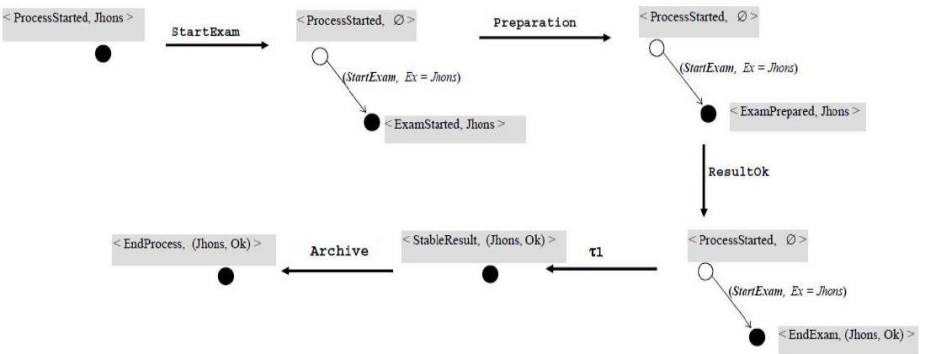

Fig. 3. Firing Sequence of the RECATNet of Fig. 2

Firing of an abstract transition: When a thread of an extended marking fires an abstract transition t_{abs} under a *firing mode sub*, it destroys from each input place p of this transition, the multiset of algebraic terms resulting from the evaluation of the expression $DT(p, t_{abs})$ in this mode *sub*. Simultaneously, it *creates* a new thread (called its child thread and representing a new sub-process) which starts its execution with, as an initial marking, the multiset of terms resulting from the evaluation under the mode *sub* of the starting marking associated to t_{abs}. We note that the fatherhood relationship between these two threads is memorised in the extended marking (we keep track of the name of the abstract transition and the mode of its firing). For instance, in Fig.3, the firing of the abstract transition ***StartExam*** in the root node of the tree, under the mode (*Ex* = *Jhons*), generates a new thread with, as a starting marking, a term *Jhons* in the place *ExamStarted.*

Execution of a cut step: The family Υ of final markings associated to a RECATNet is indexed by a finite set of items called *termination indices* (*i.e.* $\Upsilon = (\Upsilon_i)_{i \in I}$). For instance, in Fig. 2, the termination indices belong to the set $I = \{0, 1\}$. When a thread reaches a final marking belonging to a termination set Υi (with $i \in I$), the two following actions occur *simultaneously*: (1) this thread *terminates and aborts* its whole descent of threads. (2) Then, it *produces* (in its father thread) the multiset of algebraic terms in the output places of the abstract transition t_{abs} which gave birth to it. Such a firing is called a *cut step* and is noted τ_i. In this case, the produced terms in each output place p' of this abstract transition t_{abs} result from the evaluation of the expression $ICT(p', t_{abs}, i)$ (where <*i*> is the termination index of the final marking reached) under the firing mode in which the transition t_{abs} had fired, giving birth to the terminating thread. Therefore, when an abstract transition is fired, the production of tokens in the output places of this transition is delayed until its child thread (i.e. the thread generated by the firing of this transition) terminates and a cut step is executed at this level. Particularly, if a cut step occurs in the root of the tree of threads, it leads to the *empty tree*, noted by $\perp$, from which neither transition nor cut step can occur. In *Fig. 3*, the execution of a cut step τ_i aborts the generated thread (i.e. process) where a final marking is reached (belonging to the termination set Υ_i), reducing the tree of threads to its root process and produces a term (*Jhons, Ok*) in the place *StableResult*.

The produced terms in this place is done using the variable substitution (*Ex* = *Jhons*), which is the firing mode of the abstract transition ***StartExam*** memorised in the tree of threads.

Firing of an elementary transition: The behaviour of an elementary transition t_{elt} is twofold and depends on a partial function K which associates to it a set of abstract transitions to interrupt and for each of these transitions a termination index. In the graphical representation of a RECATNet, the name of an elementary transition t_{elt} is followed by the set $K(t_{elt})$ when this set in non empty (in Fig. 1, $K(t_{elt}) = \{(t_{absj}, i), (t_{absm}, k), \ldots\}$). Basically, when a thread of an extended marking fires an elementary transition t_{elt} under a *firing mode sub*, the two following actions occur at the same time*:*

(1) *Update of the internal marking of the thread where* t_{elt} *is fired*: for each input place p of t_{elt}, it removes the multiset of algebraic terms resulting from the evaluation of the expression $DT(p, t_{elt})$ in this mode *sub* and it creates the multiset of algebraic terms $CT(p', t_{elt})$ (evaluated under that same mode *sub*) in each output place p' of t_{elt}.

(2) *Interruption of the threads generated by the abstract transitions associated to* t_{elt}: if the function K is defined for this elementary transition t_{elt}, the firing of this transition performs then the appropriate cut step to each sub-tree generated by the abstract transitions specified by K. So, all threads which are generated by one of the abstract transitions specified by K are aborted and, depending on the termination index associated to it, the output tokens of these abstract transitions are produced in the thread where the firing takes place. In the RECATNet of *Fig 2*, the set K associated to each elementary transition is empty. Consequently, these two elementary transitions update only the internal marking of the thread where they are fired. For instance, in the firing sequence of *Fig. 3*, the firing of the elementary transition (under the mode *Ex* = *Jhon*) removes the term *Jhon* for its input place *ExamStarted* and produce a term *Jhon* in its output place *ExamPrep*.

3.2 Recursive Workflow Nets

The Recursive Workflow Nets (noted *RecWF-Net)* are a sub-class of the RECATNets model, dedicated to the modelling of flexible and distributed workflows. Consequently, *RecWF-Nets* have structural restrictions which reflect the particular concepts of typical workflows where there is a well-defined starting point and a well-defined ending point. In a *RecWF-Net,* each connected component (i.e, subnet) is called *a workflow component* which specifies the behaviour of a workflow sub-process. A *workflow component* has one source place (i.e. a place without input transitions) and one sink place (i.e. a place without output transitions). Moreover, every place or transition of this subnet is on a directed path from its source to its sink place. In practice, a RecWF-net describes the composition of workflow sub-processes initialised by a principal (i.e. root) process. Let us note that in a RecWF-net, we associate a *finite capacity* to each place connected to an inhibitor arc (i.e. if the input condition on this arc is of the form $IC(p,t) = \overline{\alpha}$). Consequently, interesting properties such as accessibility, boundedness and finiteness remain decidable for RecWF-nets [14]. For instance, the RECATNet depicted in *Fig.*2 is an example of a RecWF-net. It describes a simplified process for managing a set of a patient's exams. In this net, we distinguish two workflow components: (1) the component delimited by the source place ***ProcessStarted*** and the sink place ***EndProcess*** (representing the *root process*) and (2) the component having the source place ***ExamStarted*** and the sink place ***EndExam.***

In our compositional modelling approach of flexible healthcare processes we propose to introduce two types of tasks in RecWF-nets: *Elementary tasks* (represented by *elementary* transitions) and *abstract tasks* (represented by *abstract* transitions). The execution of an abstract task dynamically generates a new (lower-level) plan of actions in a workflow process. This plan terminates when it reaches a predicated termination state (a final marking) or when it is interrupted by an exception occurrence in a higher level plan of actions (i.e. by the firing of an elementary transition). In these two cases, the whole descent of action plans, generated by it, are aborted (i.e. a cut step is executed) and the results are returned to the caller abstract task. In fact, a dynamic tree of action plans (with an independent context) describes the structure of a workflow process where all plans can be executed simultaneously. The root plan of such a tree represents the principal process by which the whole specified workflow starts and terminates.

This ability offers the following advantages:

1) Flexibility in workflow planning and execution at design time is introduced naturally. Indeed, RecWF-nets capture the flexibility by design (choice, parallelism, recursion, cancellation and multiple instantiation of processes) and the flexibility by underspecification (*late selection* of process component).

2) Distributed execution of the interacting sub-processes related to healthcare process life-cycle (administrative process, lab testing process, radiology process, care process…etc.) is faithfully reflected.

3) Data and knowledge management of healthcare processes can be easily integrated in RecWF-nets due to state algebraic description.

4 Modelling Chemotherapy Treatment Process Using RecWF-Nets

Based on the characteristics of clinical procedures and medical tasks mentioned in [15], we illustrate the suitability of RecWF-nets in the modelling of a chemotherapy treatment following a breast cancer surgery trough the RecWF-net depicted in *Fig.4* and *Fig.5*.

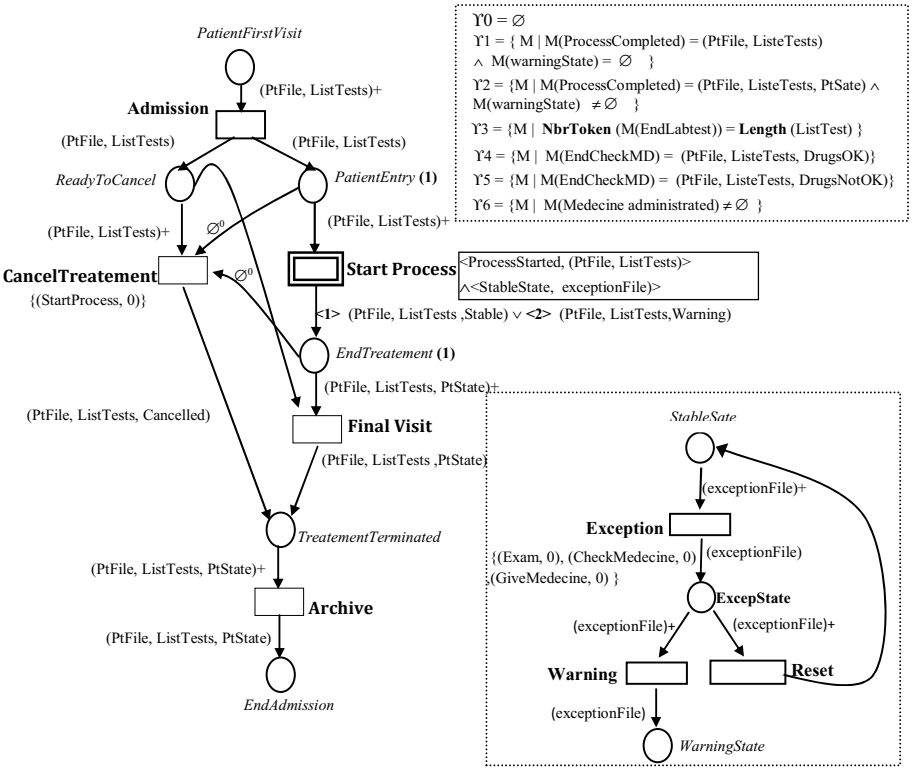

Fig. 4. An example of chemotherapy treatment workflow (part 1)

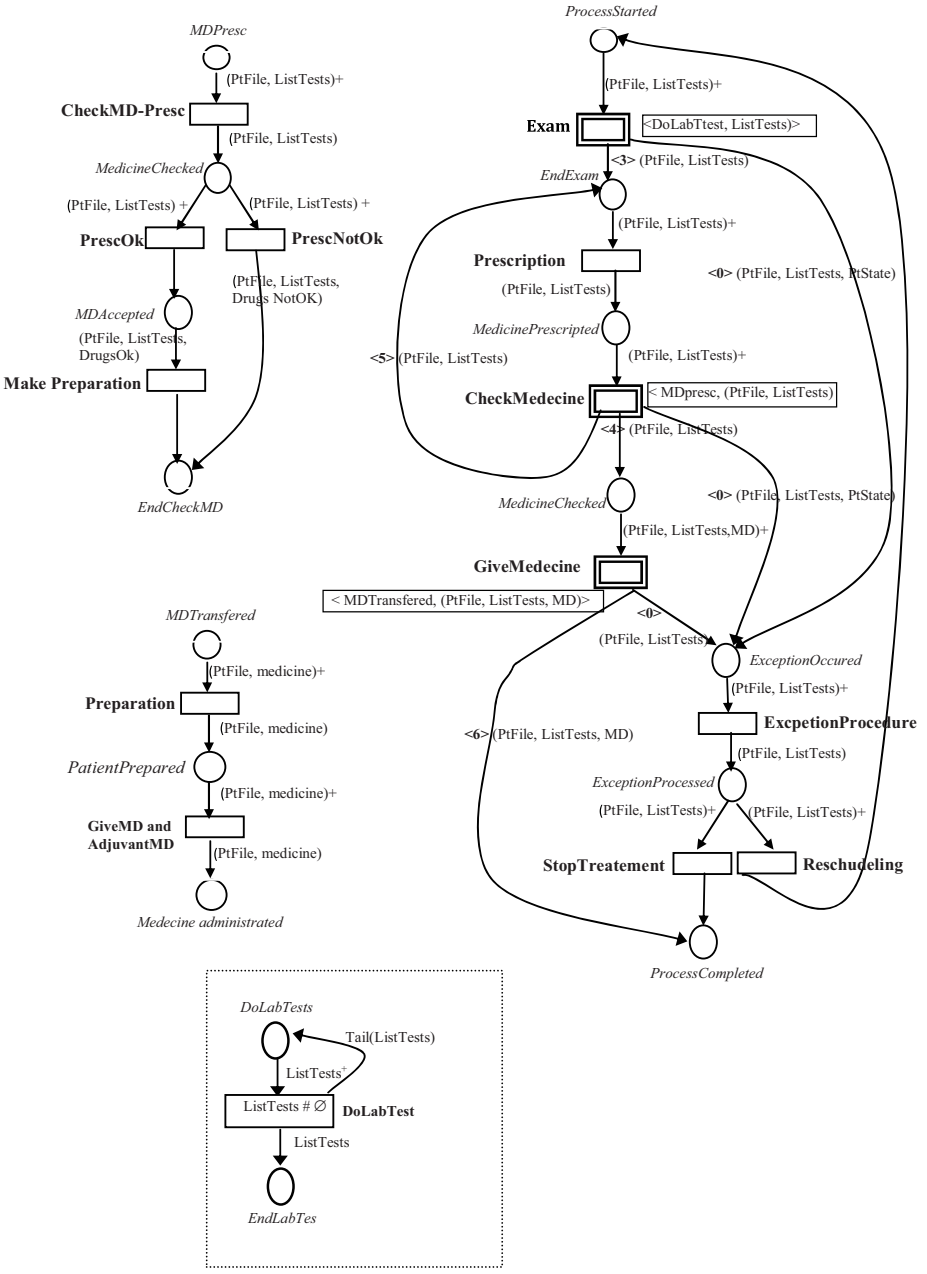

Fig. 5. An example of chemotherapy treatment workflow (part 2)

All the chemotherapy treatment process is based on a flowchart in which basic information about the patient is registered in his file (e.g. weight, height, lab results). All the places of this RecWF-Net are associated to a sort *PatientData*. Let us note that the initial state of this net is a tree containing a single thread with a token (*PtFile,*

ListTests) in the place *PatientFirstVisit* (i.e. the starting place of the principal admission process). This token represents the file of the patient and the list of tests required. A firing sequence of this example is presented in *Fig.6* where we note by (t_1 Seq. t_2) the sequential firing of transitions t_1 and t_2.

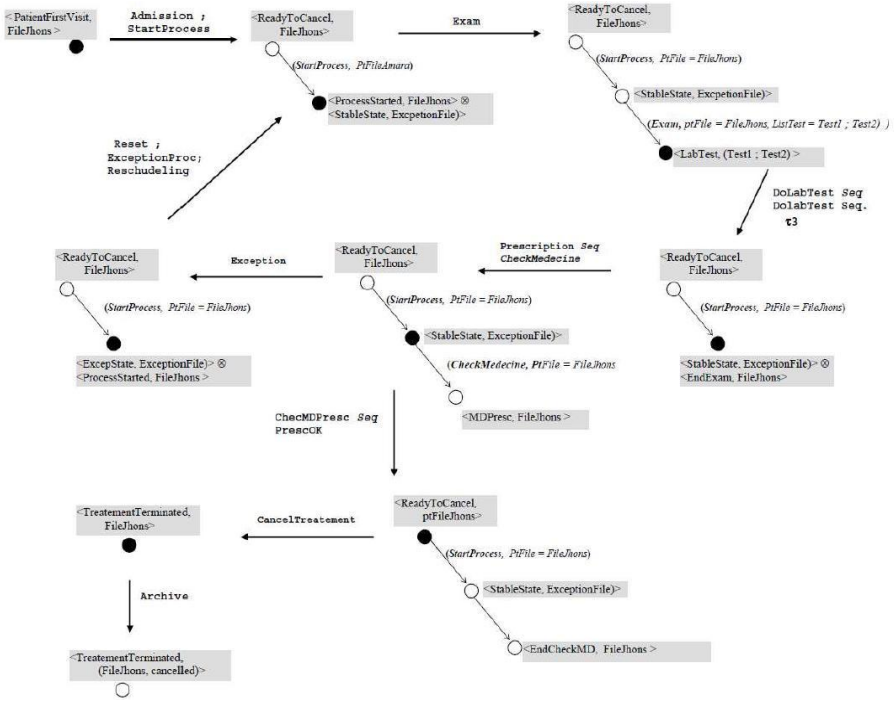

Fig. 6. Firing sequence of the healthcare RecWF-net of Fig. 4 and Fig. 5

This healthcare workflow process starts by the firing of the elementary task "*Admission*" (i.e. a new patient is entered to the chemotherapy department). Then the abstract task "*StartProcess*" dynamically creates a new thread in the tree of threads of the RecWF-net by calling in parallel two sub-processes: one which gives the steps of the treatment (i.e. the subnet with the initial place *ProcessStarted*) and another one which describes an exception detection procedure (i.e. the subnet with the initial place *StableState).*This latter controls tasks execution in the treatment sub-process. The first abstract task "*Exam*" generates a new thread in the tree of threads where some laboratory tests are done and printed in a file. This thread completes when all the lab tests are done (See the termination state γ_3). After this step, the elementary task *"Prescription"* of the adequate medicine is executed. Then the abstract task *"CheckMedicine"* is executed, dynamically creating a new thread.

Through this new thread, a pharmacy controller checks that drugs dose calculated by the doctor is matched or not with the patient data file (laboratory tests results, measures and adverse effects). The completion of this thread is indicated by a token in the place *EndCheckMD* (See termination states γ_4 and γ_5). If the medicine prescription is not trusted (γ_5 is reached), a token is created in the place *EndExam*

leading to the re-firing of the task "*Prescription"* allowing the assigned doctor to recalculate drug dosages and correct the medicine prescription in the flowchart. If the medicine prescription is trusted (Υ_4 is reached), the abstract task *"GiveMedicne"* is fired. In this case, another thread is created in the tree of threads, with the starting marking <*MDTransfered*, (*ptFile*, *ListTests, medicine)*>. This new created thread completes when the place *MedecineAdminstrated* is marked (See the final marking Υ_6) after what the treatment sub-process completes (the place ProcessCompleted is marked). During the processing of the treatment sub-process, if the elementary transition "*Exception"* is fired (i.e. a medical alert is raised and an exception file is produced), an *interruption* is raised and all the sub-processes produced by the abstract tasks "*Exam*", "*CheckMedicine*" and "*GiveMedcine*" are stopped and aborted. Note that the elementary transition "*Exception"* interrupts all the threads generated by either the abstract transition "*Exam"*, "*CheckMedicine*" or "*GiveMedcine*" with the termination index <0> (i.e. the list of interrupted abstract transitions associated to this elementary transition is not empty). In this case, a term is produced in the place *ExcepState* and an exception procedure is lunched. After that, depending on the doctor's decision, the treatment sub-process is stopped or rescheduled. Let us note that if the treatment is rescheduled, the exception detection procedure can either be reset to a stable state or ends in a warning state. Depending on the exception file produced, the treatment subprocess terminates in a stable state or in a warning state which has to be watched by future medical procedures (See the termination states Υ_1 and Υ_2). Finally, at the level of the principal admission process, the doctor supervising the patient treatment has the possibility to stop the whole treatment sub-process along with the exception detection procedure (i.e. the thread is aborted by the firing of the elementary transition "*CancelTreatment")* as long as the corresponding treatment is not completed. This can happen if the patient decides to leave before the start of the treatment or for another very exceptional reason (e.g. the patient dies). The elementary transition *"CancelTreatment"* interrupts the abstract transition *"StartProcess"* with the termination index <0>. When this transition is fired a term *(ptFile,ListTests, Cancelled)* is produced in the place *TreatementTerminated* but no token is produced in the output place of *StartProcess* (i.e. *ICT*(*StartProcess*, *EndTreatement*, 0) =∅). In *Fig.6*, each firing of an abstract transition leads to the creation of a new node in the tree of threads. Also, when a final marking Υ_3 is reached in a thread, a cut step τ_3 is executed. The firing of the elementary transition "*Exception"* aborts the thread generated by the abstract transition "*CheckMedecine"*. Then, the exception detection procedure is reset to its stable state and the treatment sub-process is rescheduled. Moreover, the firing of the elementary transition "*CancelTreatement"* aborts the whole treatment sub-process and the exception detection procedure, reducing the tree of threads to its root process with a term *FileJhons* in the place *TreatmentTerminated*.

Such a construction adequately describes the flexible and distributed structure of healthcare workflows where sub-processes may be created or cancelled dynamically (when an exception is raised), leading to rescheduling of some sub-processes. In comparison, modelling cancellation of workflow cases with Coloured Petri Nets would result in net containing spaghetti-like arcs to remove tokens from all combinations of all places [14]. To improve such a modelling, cancellation regions in workflows are often implemented by means of *Reset Nets* [9].

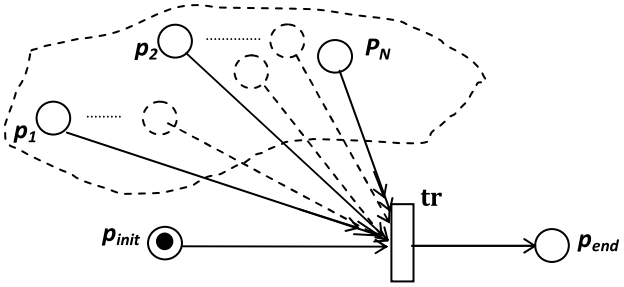

Fig. 7. Modelling a cancellation region with Reset Nets

A Reset Net is a Petri Net with special arcs (called *reset arcs,* represented by a *double headed arrow*), that allow a transition to remove *all* tokens (independently of their number) from its input places when this transition fires. However, the number of reset arcs used in such a modelling depends on the number of places in the cancellation regions of a workflow process (See *Fig. 7*). For instance, in *Fig. 7,* transition *tr* is enabled if and only if there is a token in place p_{init}. After the firing of *tr* all tokens are removed from the places p_1 to p_n and a token is produced in p_{end}. Modelling the cancellation of sub-processes via cut step executions in RecWF-Nets is much more concise because it is independent of the number of places in the aborted processes.

5 Analysis of Healthcare Recursive Workflow Nets

5.1 Correctness Properties of Healthcare RecWF-Nets

With regard to the sensitive nature of a clinical workflow, where human lives are at stake, it is primordial to determine if such a process, based on its definition, behave correctly before its deployment. Testing techniques are not sufficient to prove with confidence the absence of errors in careflow definitions. Testing can find errors but it cannot prove the absence of errors. Therefore, the verification of critical properties of healthcare processes must be done by means of rigorous analysis techniques [3] such as model checking. Model checking is an automatic method which determines if a specified property (formulated in a suitable temporal logic like the Linear Temporal Logic) is satisfied by a description model of a system and its initial state. The model checker will either terminate with the answer true, indicating that the model satisfies the property, or give a counterexample that shows an execution path in which the formula is not satisfied. We distinguish two types of properties in healthcare processes: *generic* properties and *medical* (domain specific) properties [3].

Generic properties specify the control-flow correctness requirements which must be satisfied by every workflow process, regardless of its application domain. For instance, one wants to check (1) if a clinical process can eventually terminate without leaving scheduled or uncompleted tasks, (2) if there is a deadlock or (3) if there is a task which can never be executed. These questions can be resumed into the *soundness*

property of a workflow which requires that this latter is always able to terminate properly by reaching its final predicted state and every task of such a process can potentially happen. The soundness of a RECATNet is based on two criteria interpreted on the level of its root process:

1. *Proper completion (termination):* Starting from an initial extended marking reduced to its root node where only the source place of the principal workflow component is marked, it is always possible to reach a final extended marking reduced to its root node where only the sink place of the principal workflow component is marked.

2. *No dead task:* In each initially marked workflow component, every transition can fire, at least, once.

Medical properties specify medical constraints and recommendations on healthcare processes, such as relevant clinical parameters or general safety requirements concerning actions of medical staff members [3]. Examples of typical medical properties are: "A patient case must be evaluated by a doctor before beginning treatment", "Contraindications are never administrated" or "A nurse administrates only the medicines given by a doctor".

5.2 RecWF-Net Analysis in the MAUDE System

Since the RecWF-nets semantics is expressed in terms of the generalised rewriting logic [5], [19] each RecWF-net *RN* is defined as a *rewrite theory* $\Re_{RN}= (\Sigma_{RN}, E_{RN}, L_{RN}, R_{RN})$ where the underlying equational theory (Σ_{RN}, E_{RN}) describes the tree structure of its extended marking. Moreover, transitions firing or cut step executions of this net are formally expressed by labelled rewrite rules of the set R (with L the set of their labels). A RecWF-net rewrite rule is of the general form "*Th => Th´ if C*" which means that a fragment of the RecWF-net state fitting pattern *Th* can change to a new local state fitting pattern *Th´*, *concurrently with any other state change*, if the condition *C* holds. Consequently, a *firing sequence* in a RecWF-net is described by a sequence of *concurrent rewritings* in its associated rewrite theory.

Maude is a high-level language [7] and an efficient system based on rewriting logic. The Maude linear temporal logic (LTL) model checker supports on-the-fly explicit-state model checking of concurrent systems expressed as rewrite theories with performance comparable to that of current tools of that kind, such as SPIN [24]. We apply, below, the Maude LTL model checker on recWF-Nets with respect to generic and medical properties.

Generic Properties

1. *Proper termination* ($Prop_1$): This criterion is expressed in LTL by the following formula *Prop*1: ***F FinalState*** where the proposition *FinalState* is valid in extended marking *Tr* if this latter is reduced to its root process with only one token in its sink place. The temporal operator ***F*** (*Eventually*) is noted by ⟨ ⟩, in MAUDE notation.

2. *No dead task* ($Prop_2$): We define the parameterised proposition *Excu*(*t*) which is valid in an extended marking *Tr*, if this transition *t* is enabled in its root node. Thus, to check that there is no dead transitions (transitions which can't fire), we express the

negation of this criterion as the following LTL formula $Prop_2$: $\bigvee_{t \in Tc}$ (***G*** ¬***Execu*(*t*)**). In this case, such a formula is valid if there is at least one transition is non fireable in the root component of a RecWF-net. The temporal operators ***G*** (*Globally or always*) and ¬(*not*) are noted, respectively, by **[ ]** and **~** in MAUDE notation. In the following (see *Fig 8*), we apply the LTL Model-Checker of MAUDE to check these two properties on the RecWF-net of *Fig.4&5*, taking, as an initial state, an extended marking with only a root process and one token (*FileJhons*) in the source place *PatientFirstVisit.* There, we can see that the first property is valid which means the process can always terminate properly. The second property is not valid (a counter-example is returned) which means that there is no transition which is not fireable in the root process of the healthcare RecWFnet. We deduce from these two results, that the healthcare RecWF-net of *Fig.4&5* is *sound*.

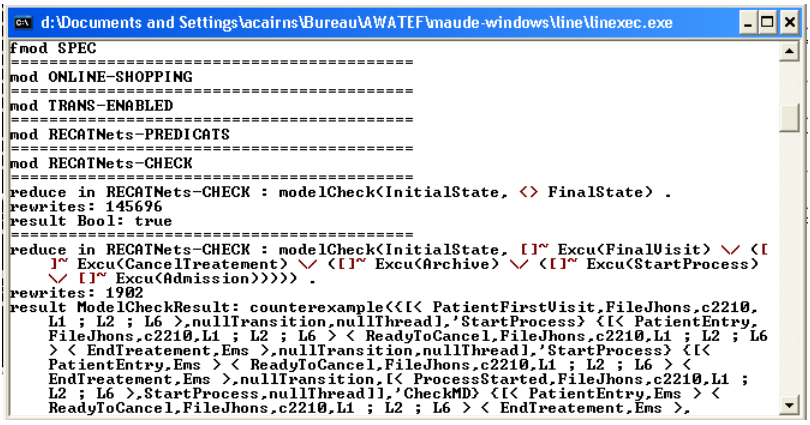

Fig. 8. Verification of the soundness of the RecWF-net (Fig 4&5) using Maude LTL model checker

Medical properties. For instance, for the RecWF-Net of *Fig.4&5*, we use the LTL Maude model checker to check the two following domain specific properties (Fig. 9).

1. $Prop_3$: When an exception is raised, the examination, the treatment and the lab testing processes are stopped which means that all running subprocesses launched by the tasks *Exam*, *CheckMedecine* and *GiveMedecine* are immediately aborted. Such a property is expressed by the following LTL formula where the proposition *RunningSubProcesses* is valid in an extended marking if the thread generated by the abstract task *StarProcess* has at least one subprocess.

$$\boldsymbol{G(Enabled(Exception) \wedge Next(Fired(Exception)) \rightarrow Next(\neg RuningSubProcesses)).}$$

2. $Prop_4$: A medicine is administrated to a patient only if it is prescribed by her/his doctor. This property is expressed by the following LTL formula $\boldsymbol{G(\neg(Enabled(Prescription) \wedge Next(Excu(Prescription))) \mapsto \neg(Excu(GiveMD)))}$

The temporal operators ***Next*** and ***Leads-to*** are noted, respectively, by **O** and **|->** in MAUDE notation. In *Fig. 9*, the returned result shows that the properties $Prop_3$ and $Prop_4$ are true. Let us not that the property $Prop_3$ shows the particular feature of

recWF-Nets where elementary transitions have the ability to interrupt all the threads generated by several abstract transitions, independently of the number of these threads, in *one step*.

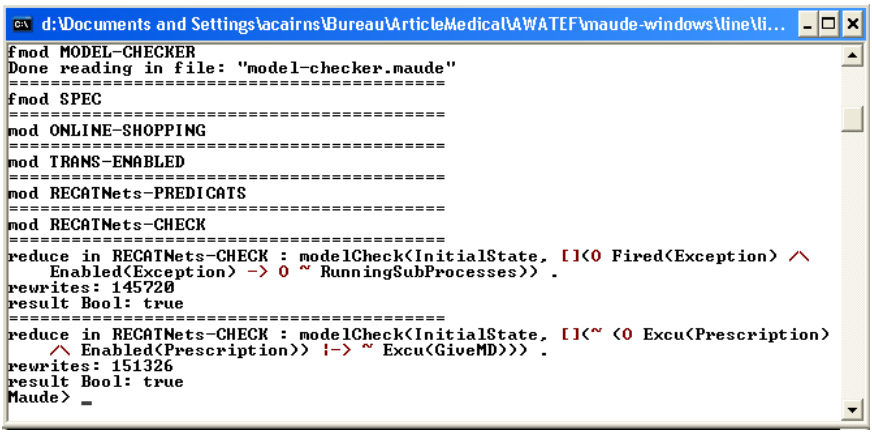

Fig. 9. Verification of two medical properties ($Prop_3$ and $Prop_4$) of the RecWF-net (Fig 4&5) using Maude LTL model checker

6 Related Works

Adaptive workflow nets [12] are an instance of the "nets in nets" paradigm, where tokens in a (higher-level) net can be nets themselves. Adaptive workflow nets [12], like RecWF-nets, can change their execution plans by allowing the modelling of the dynamic creation and suppression of processes. However, the advantage of the RecWF-nets is that the distributed execution of workflows and their verification by model checking are intrinsic via the given rewriting semantics. Also, RecWF-nets are more descriptive than Adaptive workflow nets. For instance, in *Fig.5*, when the elementary transition *ExceptionProcedure* is fired, all the running threads, generated by the abstract transitions *Exam*, *CheckMedicine* and *GiveMedicine* are cancelled, independently of their number. Then, the tokens produced in the output places of these abstract transitions depend on the number of the aborted threads. Let us note that the cancellation of these threads and the production of the tokens in the output places of these abstract transitions *happen in one step*. Modelling such a construction is not that direct and simple using Adaptive Nets. In [17], YAWL4Healthcare are introduced to model flexible healthcare processes.

YAWL allows a direct modelling of most complex control-flow structures involving cancellation, multiple instantiation and advanced synchronization, via its predefined constructors. However, the soundness property is not decidable for YAWL specifications where cancellation constructors are used, due to the semantics of the underlying Reset Nets [9]. Consequently, the decision procedures which are developed for the analysis of these YAWL specifications are only partial. In contrast, RecWF-nets allow the modelling of cancellation of sub-processes via cut steps execution while the soundness property remains decidable if their state space is finite.

Finally, BPMN (Business Process Modelling Notation) is widely used to describe the behaviour of clinical workflows [22]. BPMN allows to model cancellation region and compensation actions in such processes. vBPMN framework is defined in [8] as a combination of a BPMN adaptation patterns catalogue and a set of business rules which permit to dynamically activate, on the fly, process fragments creating new workflow variants when exceptions are encountered. In comparison, our model can naturally integrate such a mechanism since its semantics is described in rewriting logic. Indeed, organisational rules and medical rules (for instance, bad interactions between two types of drugs) can be formally expressed as rewrite rules. Consequently, the rewriting logic framework offers a formal unifying framework for the RecWF-Nets specifications and the clinical rules (describing general medical constraints and recommendations) of the healthcare environment where they are deployed.

7 Conclusion

In this paper, we show the ability of Recursive Workflow Nets (recWF-nets) for the modelling and the formal verification of flexible healthcare workflow processes [1], [4]. In future work, we intend to use the temporal extension of recWF-nets, namely, the Temporal Recursive Workflow nets (abbreviated T-RecWF-net) [2] for the modelling and the analysis of real-life healthcare workflows where temporal constraints are preponderant. For instance, it is primordial to specify medical task duration, minimal and maximal time between medicine administration, duration of blood samples or time-out on medical sub-processes. Consequently, time-constrained medical properties in healthcare workflows (e.g. A patient must have been evaluated by a doctor within three weeks before beginning chemotherapy) can be evaluated using the timed LTL model-Checker of MAUDE [7]. Actually, we are working on the implementation of a graphical tool based on rewriting logic (taking MAUDE system as an underlying engine) for creating and analysing recWF-nets. This tool allows edition (via a graphical editor), simulation and verification of recWF-nets using the reachability analysis and the LTL Model checking tools of Maude [24]. Such a tool will propose control flow and flexibility patterns to facilitate workflow modelling.

Limitations. Although, recWFnets are suitable to model and to verify both clinical processes and medical diagnostic protocols, there remains much more to investigate:

1). Healthcare processes entail substantial amounts of concurrency, data and exception handling which lead to very large state spaces. The next step in our work, is to elaborate a more effective analyze procedure for the recWF-nets, to manage the state space explosion problem induced by the reachability graph of huge systems.

In this case, abstraction techniques or hierarchical verification procedures may be adopted. Thanks to the reflective capabilities of the rewriting logic, well supported by the MAUDE system [7] (i.e. the capability to represent rewrite specifications as objects and control their structure and their execution at the meta-level), one can define different rewrite strategies to control the rewriting process in recWF-nets. Such strategies allow, for instance, to partially explore the reachability graph of a recWF-net, in an *hierarchical manner*, for a partial verification of its properties. One can also

specify and implement abstraction strategies to reduce the state space of recWF-nets while preserving their interesting properties. In this case these preserved properties are verified by exploring the produced abstraction graphs.
2). reWF-Nets focus on the workflow flexibility requirements which are expressed during build-time (flexibility by *design* and by *underspecification*) [18]. To extend our approach (*flexibility by change*), we can use rewriting strategies to define different structural modification operations on the recWF-Nets specifications to add/change/ remove, on the fly, process fragments, places or transitions.
3). We intend also to extend recWF-nets with shared resource concept allowing us to study the efficiency of healthcare workflows taking into account a limited number of available resources (medical staff member and materials).

References

1. Barkaoui, K., Hicheur, A.: Towards Analysis of Flexible and Collaborative Workflow Using Recursive ECATNets. In: ter Hofstede, A.H.M., Benatallah, B., Paik, H.-Y. (eds.) BPM 2007 Workshops. LNCS, vol. 4928, pp. 232–244. Springer, Heidelberg (2008)
2. Barkaoui, K., Boucheneb, H., Hicheur, A.: Modelling and Analysis of Time-Constrained Flexible Workflows with Time Recursive ECATNets. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 19–36. Springer, Heidelberg (2009)
3. Bäumler, S., Balser, M., Dunets, A., Reif, W., Schmitt, J.: Verification of medical guidelines by model checking – a case study. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 219–233. Springer, Heidelberg (2006)
4. Ben Dhieb, A., Barkaoui, K.: On the Modeling of Healthcare Workflows Using Recursive ECATNets. In: Daniel, F., Barkaoui, K., Dustdar, S. (eds.) BPM 2011 Workshops, Part II. LNBIP, vol. 100, pp. 99–107. Springer, Heidelberg (2012)
5. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. J. Theor. Comput. Sci. 360(1-3), 386–414 (2006)
6. Cardoen, B., Demeulemeester, E., Beliën, J.: Operating room planning and scheduling: A literature review. European Journal of Operational Research 201(3), 921–932 (2010)
7. Clavel, M., Duran, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., Talcott, J.: Maude Manual (Version 2.3). SRI International and University of Illinois at Urbana-Champaign (2007), `http://maude.cs.uiuc.edu/maude2-manual/`
8. Döhring, M., Zimmermann, B.: vBPMN: Event-Aware Workflow Variants by Weaving BPMN2 and Business Rules. In: Halpin, T., Nurcan, S., Krogstie, J., Soffer, P., Proper, E., Schmidt, R., Bider, I. (eds.) BPMDS 2011 and EMMSAD 2011. LNBIP, vol. 81, pp. 332–341. Springer, Heidelberg (2011)
9. Dufourd, C., Finkel, A., Schnoebelen, P.: Reset nets between decidability and undecidability. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 103–115. Springer, Heidelberg (1998)
10. Dadam, P., Reichert, M., Kuhn, K.: Clinical Workflows - The Killer Application for Process-oriented Information Systems? In: Proc. BIS 2000, pp. 36–59 (2000)
11. Haddad, S., Poitrenaud, D.: Recursive Petri nets: Theory and Application to Discrete Event Systems. Acta Informatica 40(7-8), 463–508 (2007)
12. van Hee, K.M., Schonenberg, H., Serebrenik, A., Sidorova, N., van der Werf, J.M.: Adaptive Workflows for Healthcare Information Systems. In: ter Hofstede, A., Benatallah, B., Paik, H.-Y. (eds.) BPM 2007 Workshops. LNCS, vol. 4928, pp. 359–370. Springer, Heidelberg (2008)

13. Hildebrandt, T., Rao Mukkamala, R., Slaats, T.: Declarative Modelling and Safe Distribution of Healthcare Workflows. In: Liu, Z., Wassyng, A. (eds.) FHIES 2011. LNCS, vol. 7151, pp. 39–56. Springer, Heidelberg (2012)
14. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Monographs in Theoretical Computer Science. Springer (1997)
15. Lenz, R., Reichert, M.U.: IT Support for Healthcare Processes - Premises, Challenges, Perspectives. Data & Knowledge Engineering 61(1), 39–58 (2007)
16. Lyng, K.M., Hildebrandt, T., Mukkamala, R.R.: From paper based clinical practice guidelines to declarative workflow management. In: Ardagna, D., Mecella, M., Yang, J. (eds.) BPM 2008 Workshops. LNBIP, vol. 17, pp. 336–347. Springer, Heidelberg (2009)
17. Mans, R.S., et al.: Supporting healthcare processes with YAWL4Healthcare. In: Ludwig, H., Reijers, H.A. (eds.) Pro: Demo Track of the Nineth Conf. on BPM, pp. 1–6 (2012)
18. Mulyar, N., Russell, N., Van der Aalst, W.M.P.: Process flexibility patterns. Working paper WP 251, Beta Research School (2008)
19. Meseguer, J.: Rewriting Logic as a Semantic Framework for Concurrency. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 331–372. Springer, Heidelberg (1996)
20. Reijers, H.A., Russell, N., van der Geer, S., Krekels, G.A.M.: Workflow for Healthcare: A Methodology for Realizing Flexible Medical Treatment Processes. In: Rinderle-Ma, S., Sadiq, S., Leymann, F. (eds.) BPM 2009 Workshops. LNBIP, vol. 43, pp. 593–604. Springer, Heidelberg (2010)
21. Reuter, C., Dadam, P., Rudolph, S., Deiters, W., Trillsch, S.: Guarded Process Spaces (GPS): A Navigation System towards Creation and Dynamic Change of Healthcare Processes from the End-User's Perspective. In: Daniel, F., Barkaoui, K., Dustdar, S. (eds.) BPM 2011 Workshops, Part II. LNBIP, vol. 100, pp. 237–248. Springer, Heidelberg (2012)
22. Richard, M., Rogge-Solti, A.: BPMN for Healthcare Processes. In: Eichhorn, D., Koschmider, A., Zhang, H. (eds.) 3rd Central-European Workshop on Services and their Composition. CEUR Workshop Proceedings, vol. 705, pp. 65–72 (2011)
23. Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features–enhancing flexibility in process-aware information systems. Data & Knowledge Engineering 66(3), 438–466 (2008)
24. Ekera, S., Meseguer, J., Sridharanarayananb, A.: The Maude LTL Model Checker. In: Proc. of Rewriting Logic and Its Applications (WRLA 2002). Electronic Notes in Theoretical Computer Science, vol. 71, pp. 162–187 (2002)