Formal Modeling and Analysis of Flexible Automated Production Workflows

Awatef Hicheur and Kamel Barkaoui
CEDRIC-CNAM 292, Rue Saint-Martin Paris 75003 France
A_hicheur@cnam.fr, barkaoui@cnam.fr

Abstract- **The aim of this paper is to show the relevance of our model, namely Recursive ECATNets (RECATNets), in the modeling and analysis of flexible workflows in manufacturing systems. Recursive ECATNets are defined on the basis of a sound combination of ECATNets (Extended Concurrent Algebraic Term Nets) and Recursive Petri nets. Moreover, since RECATNets semantics is defined in the conditional rewriting logic framework, the use of formal verification techniques to prove liveness and safety properties becomes possible.**

I. Introduction

In order to stay competitive in a rapidly changing environment, manufacturing enterprises are more and more adopting workflow management systems (WfMSs for short), to define, execute and control their manufacturing and logistics processes [1], [11], [12], [13]. It is, also, well known that the process of specifying a practical workflow schema (i.e. a workflow definition) is complex and prone to errors. Among the limitations of WfMSs currently available in the marketplace, one can note that the workflow schemas used by many WfMSs are often not formally specified which makes it hard to test the correctness of these workflow definitions before putting them into production (e.g. we need to be able to check that the workflow eventually terminates or that there are no potential deadlocks)[19]. The use of formal methods for the specification of workflows of production systems reduces ambiguities and open possibilities for improving production processes through analysis and verification. Another limitations of currents WfMSs, is that only few of them provide efficient ways of managing flexible workflows which require dynamic adaptation of their structure at the occurrence of exceptional situations and failures [9], [14]. It is prevalent that faults are events that cannot be ignored in the automation of manufacturing processes [15]. The development of workflow systems that consider not only normal processes but also detection and treatment of faults is essential for improving the flexibility in the planning of shop floor operations [15]. In this paper, we propose a modeling approach for flexible automated production workflows, based on our model, namely recursive ECATNet (RECATNet) [8]. This model offers mechanisms allowing to manage dynamic structural changes of systems (i.e. direct and intuitive support of dynamic creation and suppression of processes). This ability is particularly relevant for a faithful and a correct flexibility description in workflow planning and execution (e.g. alternate planning, modification of planning and execution of workflow processes; dynamic creation and duplication of processes). Moreover, these dynamic mechanisms are adequate to model, in a concise way, the most complex routing constructs (or flow patterns) [2] such the multiple instantiation of (sub)processes and the cancellation of activities and cases (i.e. cancellation of running sub-processes or global processes). The RECATNets model extend the classical ECATNets formalism [3], [4] with the recursion concept firstly introduced in the recursive Petri nets (RPNs) [6]. ECATNets (Extended Concurrent Algebraic Term Nets) are a kind of algebraic nets which combine the expressive power of high-level Petri nets and algebraic abstract types. Their semantics is expressed in terms of rewriting logic [5]. The RPNs, for their part, are introduced as a strict extension of the ordinary Petri net (PNs) [6]. We define recursive ECATNets semantics in the conditional rewriting logic framework. The structure of the this paper is as follows: In section 2, we recall our RECATNet model. Section 3 shows how RECATNets can be used in the modeling of flexible workflows in manufacturing systems. In section 4, we express the semantics of RECATNets in the conditional rewriting logic framework and we show how the formal verification of the given example can be done using the LTL Model-Checker of Maude. Finally, section 5 concludes the paper.

II. Recursive ECATNets

A. *ECATNets Review*

An ECATNet is a high level net ε = (*Spec*, *P*, *T*, *sort*, *IC*, *DT*, *CT*, *Cap*, *TC*) where: *Spec* = (Σ, *E*) is an algebraic specification of an abstract data type given by the user (with *E* its set of equations and Σ the set of operations and sorts) and in which places marking are multisets of Σ-terms. The graphical representation of a generic ECATNet is given in Fig.1. Note that $T_{\Sigma,E}(X)$ is the Σ-algebra of the equivalence classes of the Σ-terms with variables in *X*, modulo the equations *E*. $MT_{\Sigma,E}(X)$ represents the free commutative monoid of the terms $T_{\Sigma,E}(X)$ endowed with the internal operator $\oplus$ and having $\varnothing$ as the identity element. CATdas(*E*, *X*) is the structure of equivalence classes formed from the multisets of $MT_{\Sigma,E}(X)$ modulo the associative, commutative and identity axioms for the operator $\oplus$ [3]. -- *P* is a finite set of places;
-- *T* is a finite set of transitions (with $P \cap T = \varnothing$);

-- *sort* : $P \rightarrow S$ (with S the set of sorts of *Spec*);
-- *Cap*: $P \rightarrow$ CATdas(E, $\emptyset$), (Places Capacity);
-- IC: $P \times T \rightarrow$ CATdas(E, X)*, (Input Condition), where CATdas(E, X)* = $\{\alpha^+ \,/\, \alpha \in$ CATdas(E, X) $\} \cup \{\alpha^- \,/\, \alpha \in$ CATdas(E, X) $\} \cup \{\alpha^0 \,/\, \alpha \in$ CATdas(E, X) $\} \cup \{\alpha_1 \wedge \alpha_2 \,/\, \forall\, i$ $\alpha_i \in$ CATdas(E, X)$^*\} \cup \{\alpha_1 \vee \alpha_2 \,/\, \forall\, i\ \alpha_i \in$ CATdas(E, X)$^*\}$. For a given transition t (Fig.1.), the expression $IC(p,\ t)$ specifies conditions on the marking of the input place p for the enabling of t. It takes one of the form given in the following table:

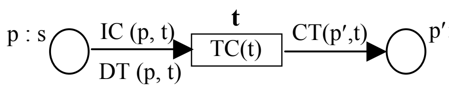


Fig. 1. A generic representation of an ECATNet

-- *DT*: $P \times T \rightarrow$ CATdas(E, X), (Destroyed Tokens); the expression $DT(p,\ t)$ specifies the multiset of tokens to be removed from the marking of the input place p when t is fired. Note that the multiset $DT(p,\ t)$ must be included in the marking of p (i.e. $DT(p,\ t) \subseteq IC(p,\ t)$).
-- *CT*: $P \times T \rightarrow$ CATdas(E, X), (Created Tokens); the expression $CT(p',\ t)$ specifies the multiset of tokens to be created in the output place p', when t is fired.
-- *TC*: $T \rightarrow$ CATdas(E, X)$_{\text{bool}}$, (Transition Condition); the expression $TC(t)$ is a boolean term which specifies an additional enabling condition for the transition t. It specifies some conditions on the values taken by local variables of t. So, $TC(t)$ may contain variables occurring in the expressions $IC(p,\ t)$ and $DT(p,\ t)$ related to the all input places of t. Note that when $TC(t)$ is omitted, the default value is the term *True*. A transition t is fireable when several conditions are satisfied simultaneously: (1) every $IC(p,\ t)$ is satisfied for each input place p of t; (2) the transition condition $TC(t)$ is true and (3) the addition of the tokens $CT(p',\ t)$ to the output place p' of t must not result in p' exceeding its capacity when this capacity is finite. When t is fired, the multiset $DT(p,\ t)$ is removed from the input place p and simultaneously $CT(p',\ t)$ is added to the output place p'.

IC(p, t)	Enabling condition
α^0	The marking of p must be equal to α. ($IC(p,\ t) = \emptyset^0$ means that the marking of p has to be empty)
α^+	The marking of the place p must include α. ($IC(p,\ t) = \emptyset^+$ means condition always satisfied).
α^-	The marking of the place p must not include α (with $\alpha \neq \emptyset$)
$\alpha_1 \wedge \alpha_2$	conditions α_1 and α_2 are both true
$\alpha_1 \vee \alpha_2$	α_1 or α_2 is true

B. Recursive ECATNets

A recursive ECATNet has the same structure as an ordinary ECATNet except that the transitions are partitioned into two categories: *abstract* transitions (represented by a double border rectangle, see Fig. 2) and *elementary* transitions (see Fig. 3).

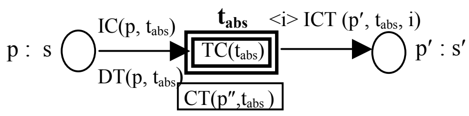


Fig. 2. A generic abstract transition

The enabling rule of a transition t (elementary or abstract) of a recursive ECATNet is specified by the expressions *IC(p, t)* and *TC (t)*. A transition t is fireable when several conditions are satisfied simultaneously: (1) every $IC(p$, t) is satisfied for each input place p of the transition t; (2) the condition $TC(t)$ is true and (3) the addition of tokens in the output places of this transition must not result in exceeding the capacity of these places when this capacity is finite. The execution of a recursive ECATNet generates a dynamical tree of threads (denoting the fatherhood relation) where each of these threads has its own activity. One can note that all the threads of this tree can be executed simultaneously.
-- When a thread fires an abstract transition t_{abs}, it consumes the multiset of tokens $DT(p,\ t_{abs})$ from the input place p and simultaneously it creates a new child (i.e. a new thread) which starts its execution with the initial marking (indicated in a frame) associated to this abstract transition. The starting marking of a created thread may depend on the state reached in the thread which gave birth to it.
-- A family of boolean terms Υ is defined and associated to a RECATNet in order to describe the termination conditions (i.e. final markings) of the created threads. This family is indexed by a finite set I whose items are called termination indexes. This set is simply deduced from the enumeration of all the defined final markings. So, if a thread reaches a final marking Υi (with $i \in I$), it terminates aborting its whole descent of threads. Then, it produces (in the token game of its father) and for the abstract transition t_{abs} which gave birth to it, the multiset of tokens $ICT(p',\ t_{abs},\ i)$ in the output place p' of t_{abs}. Such a firing is called a *cut step* and denoted τi (with $i \in I$). An arc from an abstract transition t_{abs} to its output place p' is labeled by the following algebraic expression: <*i*> $ICT(p',\ t_{abs},$ $i)$. Such an arc can be omitted if the term $ICT(p',\ t_{abs},\ i)$ is null. Note that if a cut step occurs in the root of the tree of threads, one obtains the *empty tree* denoted $\perp$.

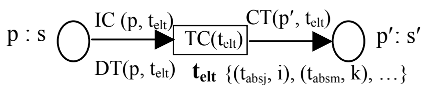


Fig. 3. A generic elementary transition

-- The behavior 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. Basically, if a thread fires an elementary transition t_{elt}, it updates its internal marking as a transition of ordinary ECATNets. Moreover, if the set specified by K is non empty, the firing of this elementary transition performs the appropriate cut step to each subtree generated by the abstract transitions specified by K (depending on the termination index associated to it).

Definition (Recursive ECATNets). A Recursive ECATNet is a tuple *RECATNet = (ECATNet, I, Υ, K, ICT)* where :

-- $T = T_{abs} \cup T_{elt}$ ($\cup$ denotes the disjoint union) is the set of RECATNets transitions, partitioned into abstract and elementary ones;

-- I is a finite set of indexes; Υ is a family, indexed by I, of boolean terms defined in order to describe the termination conditions of threads. These conditions can be specified by a system of linear inequalities or equalities on the places marking (We require that $Spec = (\Sigma, E)$ is a many sorted algebra with finite number of sorts).

-- $K : T_{elt} \rightarrow T_{abs} \times I$, is a partial function which associates to an elementary transition the set of interrupted abstract transitions and their associated termination index.

-- ICT: $P \times T_{abs} \times I \rightarrow$ CATdas(E, X), (Indexed Created Tokens). The created tokens in the output places of an abstract transition depend on the termination index of the final marking reached in the generated thread child.

III. RECATNETS-BASED MODELING OF FLEXIBLE AUTOMATED PRODUCTION WORKFLOWS

In production systems, manufacturing processes of each part consist in sequences of operations, each of which must be processed during an uninterrupted time period on a given machine, in order to transform raw materials into finished (or semi-finished) goods. During the production run, disturbances in manufacturing processes have to be handled in order to minimize the unavailability of the system [17]. These disturbances are caused by the occurrence of exceptions from internal or external environments such as machine breakdown, unavailability of operators or materials, violation of delays, low quality productions, modification or cancellation of a manufacturing order. Therefore, the specified workflows for manufacturing process automation should be flexible enough to react rapidly to the occurrence of failures that degrades the original production plan and also to create plans that anticipate the occurrence of disturbances in order to minimise their effects [13]. For this purpose, a method is proposed in this paper based on our model, namely, recursive ECATNets for the modeling of flexible automated production workflows. Recursive ECATNets offer a powerful means to deal with dynamic structural aspects of systems. This is handled via mechanisms for a direct modeling of dynamic creation and suppression of processes. In a RECATNet based workflow model of a production system, transitions represent tasks and events occurring in a manufacturing process (leading to changes in the states of objects or the status of operations). RECATNets offer a natural way to define and manage advanced data structures (specifying information on the status of operations and internal states of manipulated objects) of production systems via the algebraic specifications. Moreover, transitions in a RECATNet have the ability to check for context conditions: positive contextual conditions (known as read arc) and negative contextual conditions (also called inhibitor arcs). This ability allows to model, faithfully, the different kinds of complex causal dependencies arising among events encountered in manufacturing processes dealing with e.g. shared resources [18] and priority. Recursive ECATNets may be used to specify flexible automated production workflows whose structures can be modified, extended or reduced dynamically during their execution. Two types of tasks are introduced in workflow processes at this end: *elementary tasks* (modelled by elementary transitions) and a*bstract tasks* (modelled by abstract transitions, whose execution generates dynamically, in a lower level thread, a new operation plan from the previous one). The exceptional situations which may occur, during the execution of a plan, can be reflected (in this plan) by the firings of abstract transitions (i.e. dynamic creation of threads) or by the firing of cut steps executed when the associated final markings are reached or when extended elementary transitions are fired. So, a production process may handle exceptions by generating a new operation plan or by terminating the current process. A RECATNet-based workflow model of a production system is depicted in Fig. 4. In this example, a company, distributed over many sites, manufactures a number of products ($Prod_1$,…, $Prod_n$), at request. The workflow covers the process from the reception of the manufacturing order at the shop floor level to the production completion of the ordered parts. Note that, for notation convenience, the multiset $IC(p, t)$ (or $DT(p, t)$) is omitted in the graphical representation of RECATNets, when $IC(p, t) = DT(p, t)$. Also, the term t is noted instead of the equivalence class of the term $[t]$. The initial state of this net is a tree containing a single thread (i.e. the root thread) with a token (N#, ListParts) in the place "OrderReceived". This token represents the waiting manufacturing order which contains the order ID number and the list of ordered parts. Each element of this list consists in a couple (PrId, qty) representing, respectively, the ordered part ID number and the requested quantity for this part. The workflow process starts by the firing of the transition "HandelOrder" (i.e. when a new manufacturing order is received at the shop floor). Then, each ordered part is considered individually, following their order of priority (i.e. the transition "HandelParts" returns the head of the list ListParts). For each couple (PrId, qty) produced by the transition "HandelParts", the abstract transition "StartFab" is enabled. Each firing of this last one, for a token (PrId, qty), initialises an instance of the specific manufacturing process sequence associated to the part PrId (the sequence of operations to be performed on lots in order to manufacture this product). In fact, a new thread son is created dynamically in the tree of threads with the initial marking associated to the abstract transition "StartFab". This initial marking depends on the consumed token (PrId, qty) such that the initialised manufacturing process at each firing of "StarFab" corresponds to the appropriate part (PrId). Note that an instance of a manufacturing process sequence of a part PrId is recognized as a job to manufacture a batch of identical discrete part items of PrId (qty being the size of this batch). Due to the space limitation, only the manufacturing process sequence associated to the part (Prod1) is represented in Fig. 4. In this example, the production system is distributed over a number of areas; consequently, different production plans or instances of the same production plan (to manufacture a part) may run in parallel in the shop floor. In order to manufacture parts

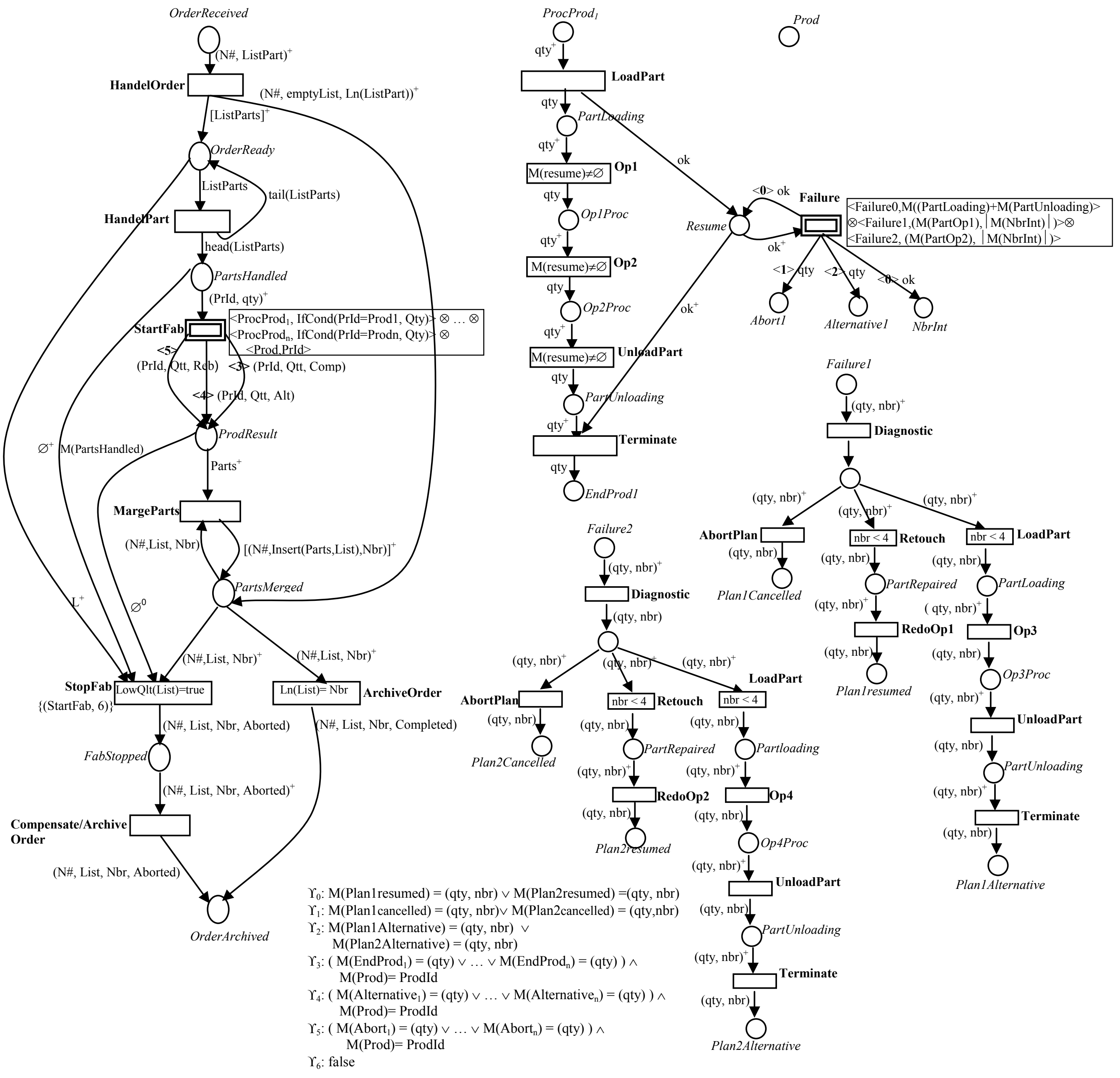


Fig. 4. A RECATNet based workflow model of a flexible production process.

satisfying quality and due time, the shop floor controls all the steps of the production plans. This control is modelled in the manufacturing process of the part (prod1) by the abstract transition "Failure" which is trigged by the external environment when a failure occurs during the production of a part (prod1). The initial marking associated to this abstract transition is parameterised. So when the transition "Failure" is fired, the running manufacturing process is frozen and a recovery process is initialised (created as a new thread in the tree of threads) which depends on the current state of the manufacturing process (i.e. the running operation) when the breakdown is detected. Only the recovery processes which correspond to the handling of exceptions occurring during the processing of the operations "op1" and "op2" are represented in Fig. 4 (due to the space constraints). These last ones suggest recovery treatments, depending on the state of the produced parts (if a physical damage is identified on the manufactured parts or not). For instance, the recovery process associated to the operation "op1" suggests, either, (1) to cancel the running process, (2) to redo the operation "op1" after reparation of the part and to resume, afterward, the interrupted manufacturing process or (3) to initialise an alternative manufacturing process to produce an alternative version of the part, after reparation. Many faults may be detected during the production of a part.

The number of tolerated interruptions resulting from the detection of a fault is limited to 3 (three) after what the part is considered as a rubbish (waste) and has to be removed. The initial state associated to the transition "Failure" is also parameterised by the number of interruptions occurring during the manufacturing process of the part via the term $|$M(NbrInt)$|$ which returns the number of token in the place "NbrInt". The completion of this recovery process is indicated by a token (qty, nbr) in one of the places "Plan$_i$Cancelled", "Plan$_i$Resumed" or "Plan$_i$Alternative". In this case, a cut step is executed at this level of recursion (the corresponding thread is aborted from the tree) and depending on the final marking reached in the recovery process (i.e. Υ_0, Υ_1 or Υ_2) the places "Resume" and "NbrInt", the place "Abort" or the place "Alternative", respectively, are marked in the previous recursion level. Consequently, the interrupted manufacturing process may resume its execution or may stop it. The manufacturing process sequence terminates if one of the following final markings, Υ_3, Υ_4 or Υ_5, is reached. Then, another cut step is executed (i.e. the thread is aborted) at this level of the tree and, depending on the index of this termination (i.e. <3>, <4> or <5>), the outputs of the abstract transition "StartFab" are created (i.e. a token (PrId, qty, Comp), (PrId, qty, Alt) or (PrId, qty, Reb), respectively, is produced in the place "ProdResult" at the root level of the three of threads). The processing of the manufacturing order terminates when all the ordered parts are completed (i.e. ln(List) = Nbr, the length of the list of the completed parts corresponds to the length of the ordered parts). Furthermore, during the processing of the order, the shop floor has the possibility to stop the fabrication of the ordered parts (by executing the task "StopFab") as long as the corresponding order is non completed, if the quality of the completed parts of this order is exceptionally evaluated below the tolerated threshold (i.e. the quantity of waste products is superior to the third of the quantity to be produced. This condition is evaluated by the function *LowQlt*). Note that "StopFab" is an extended elementary transition (the associated list of interrupted abstract transitions is not empty). It interrupts the abstract transition "StartFab" with the termination index <6>. Consequently, when the transition "StopFab" is fired, the threads generated by the transition "StartFab" are aborted and a token (N, list, Nbr, Aborted) is produced in the place "FabStopped". In fact, all the running manufacturing processes (corresponding to the parts currently manufactured) are stopped and all the tokens of the place "PartsHandled" (corresponding to the parts waiting to be produced) are removed (note that IC(PartsHandled, StopFab)=$\varnothing^+$ and DT(PartsHandled, StopFab)=M(PartsHandled,)). After that, the shop floor executes compensation operations for the stopped manufacturing processes. No token is produced in the output place of "StartFab", when it is interrupted by the elementary transition "StopFab". An equivalent modeling of the flexible structure of this workflow with other high level Petri nets (e.g. colored Petri nets) would be very complicated. The resulting net, for the modeling of the cancellation of the running manufacturing processes, for instance, would contain spaghetti-like arcs to remove tokens from all combinations of all places [2]. Note that with rewriting semantics given to RECATNets, the concurrent execution of different running manufacturing or recovery processes (independently in different production areas) is well described.

IV. Recursive ECATNets Semantics in the Conditional Rewriting Logic Framework

In this section, we show how RECATNets semantics is naturally expressed in terms of conditional rewriting logic. Consequently, we can benefit from the use of the Maude system [7] (a high-performance interpreter and language based on rewriting logic) as a simulation environment for RECATNets where the models can also be analysed and model checked. For a more detailed description on rewrite logic definitions, interested readers may refer to [5]. In what follows we adopt Maude syntax to present specific equational or rewrite theories.

A. *RECATNets Semantics in terms of rewriting logic*

The structure of the state space: The global distributed state of a RECATNet is described as a *dynamical tree Tr* of threads marking. Each thread (node) *Th* of the tree *Tr* is described by a term [M_{Th}, t_{abs}, *ThreadChilds*] of sort `Thread`, where: M_{Th} represents the *marking* of the thread *Th* which is expressed as multi-sets of pairs of the form <p,$[m]_\oplus$>, where p is a place of this thread and $[m]_\oplus$ a multi-set of algebraic terms. The operator ⊗ denotes the multi-set union on the pairs <p, $[m]_\oplus$>. The sub-term t_{abs} represents the name of the abstract transition whose firing (in the thread father) gave birth to the thread *Th*. Note that the root thread is not generated by any abstract transition (the abstract transition which gave birth to the root thread is, then, represented by the constant `nullTrans`). The sub-term *ThreadChilds* represents the threads generated by this thread *Th* in the current tree *Tr*. They are described as a finite multiset of terms of sort *Thread*. The constant *nullThread* represents the empty thread and the operator _ _ (the underline indicates the position of the parameter) is the corresponding multiset union operator which is associative, commutative and has the constant *nullThread* as the identity element.

```
fmod THREAD  is protecting  Marking .
sorts  Thread Trans TransAbs . subsort TransAbs < Trans .
op  nullTrans :-> Trans .
op  nullThread :-> Thread .*** The empty thread and the empty Tree (⊥)
op [_,_,_] Marking TransAbs Thread -> Thread .
op _ _: Thread Thread -> Thread [assoc comm id : nullThread] .
op Initial : Making -> Thread .
var  T : TransAbs .  var  L : List .  vars  mth mthf : Thread .
var M : Marking . vars Minit Moutput mts : multiset .
Eq Initial (M) = [M, nullTrans, nullThread] .
**************************************************
op DeleteThread : Thread List -> Thread .
eq DeleteThread(nullThread, L) = nullThread .
eq DeleteThread(Thf [M, T, Th], L) = if    find(T, L)  then
            DeleteThread(Thf, L) else [M, T, Th] DeleteThread(Thf, L) fi  .
**************************************************
op CreateTokens : Multiset Thread Trans Multiset -> Multiset .
```

eq CreateTokens(Minit, nullThread, T, Moutput)= Moutput.
eq CreateTokens(Minit, Thf [M, T, Th], T, Moutput) = createTokens(Minit, Thf, T, Moutput ⊕ Minit).
eq CreateTokens(Minit, Thf [M, Tf, Th], T, Moutput) = createTokens(Minit, Thf, T, Moutput) [owise] . **endfm**

RECATNets rewrite rules: Each transition (abstract or elementary) and each cut step in a RECATNet *RN* is formally expressed as a rewrite rule. These rewrite rules are partitioned into four distinct types which are the *abstract* rules, the *elementary* rules, the *extended elementary* rules and the *pruning* rules. The general form of these rules is given as follows:

*** Elementary rules: For each elementary transition t_{elt} with $K(t_{elt})=\varnothing$
crl [t_{elt}]: <p, mp ⊕ DT(p, t) > ⊗ <p', mp'> → < p, mp > ⊗ <p', mp' ⊕ CT(p', t)> **if** (Nbr(mp′ ⊕ CT(p′, t)) ≤ Cap(p′)) and (InputCond) and (TC(t)) .
…
**** Extended elementary rules: For each elementary transition with
**** $K(t_{elt})$ defined,(i.e. $K(t_{elt})=\{(t_{absj},i),(t_{absm},k), …\}$) ***************
crl [t_{elt}]: [M ⊗ <p, mp ⊕ DT(p, t) > ⊗ <p′, mp′ > ⊗ <p'_j, mp'_j > ⊗ <p'_m, mp'_m > …, T, mTh] → [M ⊗ <p, mp > ⊗ <p′,mp′ ⊕ CT(p′, t)> ⊗ <p'_j, mp'_j ⊕ CreateTokens(ICT(p'_j, t_{absj}, i), mTh, t_{absj}, Ems)> ⊗ <p'_m, mp'_m ⊕ CreateTokens(ICT(p'_m, t_{absm}, k), mTh, t_{absm}, Ems)> … , T, DeleteThread(mTh, t_{absj} ;; t_{absm} ;; …)] **if** (InputCond) **and** (Nbr(mp′ ⊕ CT(p′,t))≤Cap(p′)) **and** (TC(t)) .
…
**** Abstract rules **
crl [t_{absi}]:[M ⊗<p, mp ⊕ DT(p, t) >, T, mTh]→[M ⊗ <p, mp>, T, mTh [<p″,CT(p″,t_{absi})> ⊗ <p_1, Ems> ⊗…⊗ <p_n, Ems>, t_{absi}, nullThread]] **if** [(InputCond) **and** (TC(t)) .
…
**** Pruning rules **
*** A pruning rule associated to a cut step likely to occur in a thread (different
*** from the root),generated by an abstract transition t_{absj}:
crl [τi]: [Mf ⊗ <p′, mp′>, Tf,[M ⊗ <p_{final}, mp_{final}>,t_{absj}, mTh] mThf]→[Mf ⊗ <p′, mp′ ⊕ ICT(p′,t_{absj},i)>, Tf, mThf] **if** (Υ_i) **and** (Nbr (mp′ ⊕ ICT(p′, t_{absj}, i)) ≤ Cap(p′)).
…
*** A pruning rule associated to a cut step likely to occur in the root thread,
*** has the form
crl [τi]:[M ⊗<p_{final}, mp_{final}>, nullTrans, mTh] → nullThread **if** (ϒi) .

The component *InputCond* is determined from the expression *IC*(*p*, *t*) as follows (where *mp* is the marking of the input place *p* of *t*):

$$InputCond = \left\{\begin{array}{ll} mp \equiv \varnothing & \text{if } IC(p,t) = empty \\ mp \equiv \alpha & \text{if } IC(p,t) = [\alpha]^0 \\ \alpha \textbf{ Inclu } mp & \text{if } IC(p,t) \equiv [\alpha]^+ \\ \text{not } (\beta \textbf{ Inclu } mp) & \text{if } IC(p,t) \equiv [\beta]^- \end{array}\right\}$$

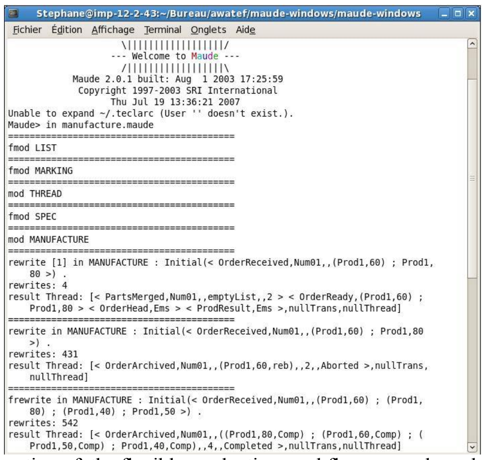
Example: In what follows, we give one rule from the rewrite theory MANUFACTURE which describes the behavior of the RECATNet given in Fig. 4. Since, we have the case $[IC(p, t)]_\oplus = [DT(p, t)]_\oplus$, for the majority of the transitions (elementary or abstract) *t* of this RECATNet, we choose to simplify the form of the associated rewrite rules as follows:

mod MANUFACTURE **is protecting** THREAD SPEC .
subsort Expression < Tokens. **subsort** ExpressionList < Tokens .
vars M Mf : Marking . **vars** T Tf : Trans . **vars** mTh mThf : Thread .
vars mts : Multiset . **vars** N ProdId : Data. **vars** Nbr Qty: Nat .
vars List L : ExpressionList. **ops** StartFab Failure :-> TransAbs .
ops OrderReceived … Plan2Alternative :-> Place . ….
…
************* abstract rules **********************************
rl [StartFab] : [M ⊗ <ProdParts , mts ⊕ (ProdId , Qty) >, T, Th] → [M ⊗ <ProdParts , mts>, T, Th [<ProcProd1, IfCond(ProdId == Prod1, Qty)>) ⊗ …⊗ <ProcProdn , IfCond(ProdId == Prodn , Qty)> ⊗ <Prod , ProdId>, StartFab, nullThread]] . …
…
endm

B. RECATNets Analysis using the Maude System

Since we express each RECATNet semantics in terms of rewriting logic [5], a RECATNet theory will constitute an executable specification of the concurrent system that it represents, directly used for formal analysis and verification [10]. In the framework of this work, we use as platform, the version 2.1.1 of the Maude system under Linux. The Maude system has a collection of formal tools supporting simulation, accessibility analysis and various forms of logical reasoning to check program properties like the LTL (linear temporal logic) Model-Checker [16] which allows to prove liveness properties and safety properties related to (finite) workflow schemas [19].

Simulation: The simulation is performed as an interactive (i.e. user-controlled) execution of the workflow [19]. In Fig. 5, an execution for the previous example is given in Maude environment using in the first case the command 'rewrite' and in the second case the command 'frewrite'. We remind that the 'rewrite' and 'frewrite' commands of Maude rewrite a given expression, each of them using its own strategy, until no more rules can be applied. Maude allows the user to specify the maximum number (enclosed in brackets) of rule applications when using these two commands. This ability is useful in debugging a model or in the case where such a computation may not terminate.

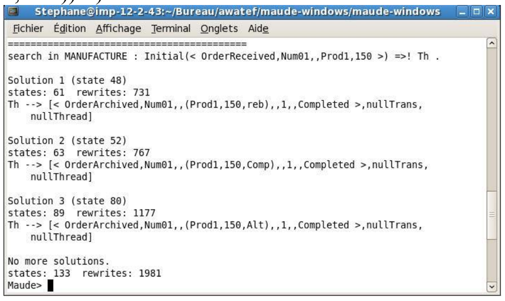

Stephane@imp-12-2-43:~/Bureau/awatef/maude-windows/maude-windows
Fichier Édition Affichage Terminal Onglets Aide

```
                  \||||||||||||||||||/
                --- Welcome to Maude ---
                  /||||||||||||||||||\
           Maude 2.0.1 built: Aug  1 2003 17:25:59
            Copyright 1997-2003 SRI International
                 Thu Jul 19 13:36:21 2007
Unable to expand ~/.teclarc (User '' doesn't exist.).
Maude> in manufacture.maude
==========================================
fmod LIST
==========================================
fmod MARKING
==========================================
mod THREAD
==========================================
fmod SPEC
==========================================
mod MANUFACTURE
==========================================
rewrite [1] in MANUFACTURE : Initial(< OrderReceived,Num01,,(Prod1,60) ; Prod1,
    80 >) .
rewrites: 4
result Thread: [< PartsMerged,Num01,,emptyList,,2 > < OrderReady,(Prod1,60) ;
    Prod1,80 > < OrderHead,Ems > < ProdResult,Ems >,nullTrans,nullThread]
==========================================
rewrite in MANUFACTURE : Initial(< OrderReceived,Num01,,(Prod1,60) ; Prod1,80
    >) .
rewrites: 431
result Thread: [< OrderArchived,Num01,,(Prod1,60,reb),,2,,Aborted >,nullTrans,
    nullThread]
==========================================
frewrite in MANUFACTURE : Initial(< OrderReceived,Num01,,(Prod1,60) ; (Prod1,
    80) ; (Prod1,40) ; Prod1,50 >) .
rewrites: 542
result Thread: [< OrderArchived,Num01,,((Prod1,80,Comp) ; (Prod1,60,Comp) ; (
    Prod1,50,Comp) ; Prod1,40,Comp),,4,,Completed >,nullTrans,nullThread]
```

Fig. 5. Execution of the flexible production workflow example under Maude environment.

Proper Termination Checking Using Accessibility Analysis: A typical requirement for any workflow schema is that: (1) the process eventually terminates and (2) there must be no deadlock (a state different from the expected final state where the workflow execution come to a halt). In the workflow depicted in Fig. 4, this requirement corresponds to the presence of only one token in the place *OrderArchived* at the root thread (a root thread which has no thread child). By using the command 'search' of Maude system, we can know if a certain

state is reachable or not from a given initial state. One can impose a bound to the command `search` and so specifying a limit to the number of solutions searched for. In our example, we want to check if starting from the initial marking (one token in the place *OrderReceived*), it is possible to reach the workflow final marking. The command given in what follows allows to obtain all the terminal states (states which cannot be rewritten further) reachable from the initial state (e.g. one token (Num01 ,, (Prod1 , 150)) in the place *OrderReceived*). In this command we precise a general final state which, in the case of recursive ECATNet, is of the form Th:Thread:
search in MANUFACTURE : Initial(< OrderReceived , (Num01 ,, (Prod1 , 150)) >) =>! Th:Thread .


Stephane@imp-12-2-43:~/Bureau/awatef/maude-windows/maude-windows
Fichier Édition Affichage Terminal Onglets Aide

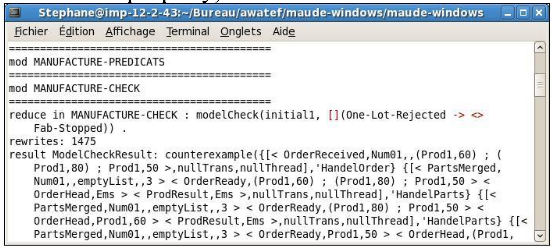
```
==========================================
search in MANUFACTURE : Initial(< OrderReceived,Num01,,Prod1,150 >) =>! Th .

Solution 1 (state 48)
states: 61  rewrites: 731
Th --> [< OrderArchived,Num01,,(Prod1,150,reb),,1,,Completed >,nullTrans,
    nullThread]

Solution 2 (state 52)
states: 63  rewrites: 767
Th --> [< OrderArchived,Num01,,(Prod1,150,Comp),,1,,Completed >,nullTrans,
    nullThread]

Solution 3 (state 80)
states: 89  rewrites: 1177
Th --> [< OrderArchived,Num01,,(Prod1,150,Alt),,1,,Completed >,nullTrans,
    nullThread]

No more solutions.
states: 133  rewrites: 1981
Maude> █
```

Fig. 6. Termination checking of the flexible production workflow example under Maude environment.

From the obtained results (see Fig. 6), we deduce that the workflow process terminates properly i.e. the expected final state can be reached and moreover there is no deadlock i.e. terminal state not expected. If one want to see the sequence of rewrites that allows us to reach one of these states (for instance the state 48), it's enough to write after this request, the following command: 'show path 48'. We can also obtain all the accessibility graph of this workflow (from the given initial state) using the command 'show search graph' or using the following command:
search in MANUFACTURE : Initial(< OrderReceived , (Num01 ,, (Prod1 , 150))) >) =>* Th:Thread .

Checking Workflow Properties Using Maude Model-Checker: We can use the Maude Model-Checker to prove general or domain specific properties (expressed as LTL formula) related to a workflow system (when the set of states reachable from an initial state is finite). Note that, thanks to the rewriting logic reflective nature, well supported by the Maude engine (i.e. the capability to represent rewrite specifications as objects and control their execution at the meta level), different rewrite strategies can be implemented for RECATNets verification (in the case of infinite state system). This particular feature is not the subject of this paper. In what follows we apply the Model-Checker of Maude system to prove some properties related to our flexible production workflow example (which is finite).

Property 1. If one produced lot is rejected (considered as a waste), all the production workflow is stopped and the process is compensated. This property is formulated as the LTL formula: [] (One-Lot-Rejected -> <> Fab-Stopped) (see Fig. 7). The proposition One-Lot-Rejected is valid, if one token of the form (ProdId, Qty, reb) is in the place *ProdResult* (at the root thread). The proposition Fab-Stopped is valid, if the transition Stop-Fab is executed. By applying Maude `Model-Checker` (with the initial state <OrderReceived , (Num01 ,, ((Prod1 , 60) ; (Prod1, 80) ; (Prod1, 50)) >) we obtained the expected result. This last one denotes that the property is false. In this case, Maude `Model-Checker` returns a counterexample (an execution trace that violates the property).


Stephane@imp-12-2-43:~/Bureau/awatef/maude-windows/maude-windows
Fichier Édition Affichage Terminal Onglets Aide

```
==========================================
mod MANUFACTURE-PREDICATS
==========================================
mod MANUFACTURE-CHECK
==========================================
reduce in MANUFACTURE-CHECK : modelCheck(initial1, [](One-Lot-Rejected -> <>
    Fab-Stopped)) .
rewrites: 1475
result ModelCheckResult: counterexample({[< OrderReceived,Num01,,(Prod1,60) ; (
    Prod1,80) ; Prod1,50 >,nullTrans,nullThread],'HandelOrder} {[< PartsMerged,
    Num01,,emptyList,,3 > < OrderReady,(Prod1,60) ; (Prod1,80) ; Prod1,50 > <
    OrderHead,Ems > < ProdResult,Ems >,nullTrans,nullThread],'HandelParts} {[<
    PartsMerged,Num01,,emptyList,,3 > < OrderReady,(Prod1,80) ; Prod1,50 > <
    OrderHead,Prod1,60 > < ProdResult,Ems >,nullTrans,nullThread],'HandelParts} {[<
    PartsMerged,Num01,,emptyList,,3 > < OrderReady,Prod1,50 > < OrderHead,(Prod1,
```

Fig. 7. Verification of workflow properties using Maude LTL Model-Checker

V. CONCLUSION

In this paper we have proposed a modeling approach based on our model, namely, recursive ECATNets for the specification of flexible workflows in manufacturing systems. Indeed, the high level description of RECATNets is well-suited for modeling workflows where the dynamic reconfiguration of their structure is required. The proposed model is also particularly adequate for handling the most advanced routing constructs (flow patterns) [2]. The RECATNets semantics is defined in the conditional rewriting logic framework [5]. Consequently, using a rewriting logic language implementation such as the system Maude [7], it is possible to create rapid prototype on which one can apply formal verification methods such model checking technique.

REFERENCES

[1] W.M.P. van der Aalst and K.M. van Hee, Workflow Management: Models, Methods, and Systems. MIT press, Cambridge, MA, 2002.
[2] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, "Workflow patterns," Distributed and Parallel Databases, vol. 14, pp. 5-51, 2003.
[3] M. Bettaz and M. Maouche, "How to specify non determinism and true concurrency with algebraic terms nets," in LNCS, No 655, Springer-Verlag, pp. 164-180, 1992.
[4] N. Zeghib, K. Barkaoui and M. Bettaz., "Contextual ECATNets semantics in terms of conditional rewriting logic," in Proc. 4th ACS/IEEE Int. Conf. on Computer Systems and Application, UAE, March 2006, pp. 936- 943.
[5] R. Bruni, J. Meseguer: Semantic foundations for generalized rewrite theories. Theor. Comput. Sci. 360(1-3): 386-414 (2006).
[6] S. Haddad and D. Poitrenaud, "Modeling and analyzing systems with recursive Petri nets," in Proc. 5th Workshop on Discrete Event Systems , Belgium, Kluwer Academic Publishers, August 2000, pp. 449-458.
[7] M. Clavel and al. "Maude manuel" (Version2.1), 2004, http://maude.cs.uiuc.edu.
[8] A. Hicheur, K. Barkaoui, N. Boudiaf, Modeling Workflows with Recursive ECATNets. In Proc. of the 8th SYNASC'06, IEEE Computer Society , pp. 389-398, 2006.
[9] J.J Halliday, S.K. Shrivastava, S.M. Wheater, Flexible workflow management in the OPENflow system, in fifth IEEE International Enterprise Distributed Object Computing Conference, 2001.

[10] K. Barkaoui, A. Hicheur, a natural semantics for RECATNets in terms of conditional rewriting logic. Internal Technical Report, Cedric Lab. February, 2007

[11] M. Dong, and F. Chen, Petri Net-Based Workflow Modeling and Analysis of the Integrated Manufacturing Business Process, Int. J. Advanced Manufacturing Technology, 2005, 26 (9-10), 1163-1172.

[12] R.Y.K. Fung, Y.M. Au, and A.W.H. IP, Petri-net Based Workflow Management Systems for In-process Control in a Plastic Processing Plant, 2003, J. of Materials Processing Technology, 139, 302-309.

[13] Y. He, H. Yang, W. He, W. Zhang, X. He, Flexible Workflow Driven Job Shop Manufacturing Execution and Automation Based on Multi Agent System, Proc. of the IEEE/WIC/ACM (IAT'06), 2006, USA, 695-699.

[14] C. Hagen, G. Alonso, Exception Handling in Workflow Management Systems, 2000, IEEE Transactions on Software Engineering (TSE) 26, 943–958.

[15] Riascos, L.A.M., Moscato, L. A. and Miyagi, P.E., 2004, Detection and treatment of faults in manufacturing systems based on Petri Nets, J. Braz. Soc. Mech. Sci. & Eng., 26(3), 280-289.

[16] S. Eker, J. Meseguer and A. Sridharanarayana, The Maude LTL Model Checker and its Implementation . in Proc. of the 10th SPIN Workshop LNCS 2648. May,2003. Pages 230–234.

[17] Bruccoleri M., La Diega S.N. and Perrone G., 2003, An object oriented approach for flexible manufacturing control systems analysis and design using the unified modeling language, Int J. Flexible Manuf. Systems, 15, 195-216.

[18] K Barkaoui, L. Petrucci, Structural Analysis of Workflow Nets with Shared Resources, Proc. Workflow Management Net-based Concepts, Models, Techniques and Tools (WFM'98), Eindhoven, 82-95.

[19] C. Karamanolis, D. Giannakopoulou, J. Magee, S.M. Wheater, "Model Checking of Workflow Schemas," edoc, p. 170, Fourth International Enterprise Distributed Object Computing Conference (EDOC'00), 2000.