On Formalizing UML2 Activities Using TPNets: Case Studies

Sabine Boufenara
Department of Computer Engineering,
École polytechnique de Montréal,
*P.O Box 6079, Station Centre-ville, Montréal, Québec
sabine.boufenara@polymtl.ca

Kamel Barkaoui
CEDRIC-CNAM,
Rue Saint-Martin,
Paris, France
barkaoui@cnam.fr

Faiza Belala
Department of Computer Science,
Université Mentouri
BP 325, Route Ain El Bey 25017 Constantine,
Algérie
belalafaiza@hotmail.com

Hanifa Boucheneb
Department of Computer Engineering,
École polytechnique de Montréal,
*P.O Box 6079, Station Centre-ville, Montréal, Québec
hanifa.boucheneb@polymtl.ca

ABSTRACT. Transactional Petri Nets (TPNets) are a new class of high-level Zero-Safe Nets (ZSNs), defined as a more suitable semantic framework for UML2 activity diagrams. Indeed, they ensure reactivity and synchronization of concurrent flows triggering with their junction. Reactivity is guaranteed due to the real time massive cancellation semantics based on the definition of new dynamic enabling rules and the imposed priority among executions. Global synchronization in turn is assured thanks to non-locality principle, an outcome of exploiting atomic stable transactions. Rewriting logic is defined as the operational semantics framework of TPNets.

KEYWORDS: Zero-safe nets, UML2 activity diagrams, reactivity, non-local behaviors, synchronization.

1. INTRODUCTION

UML2 represents a milestone in the evolution of software development technologies. This is essentially due to the object management group launching of the UML modeling in the context of Model Driven Architecture 'MDA' (OMG, 2003). In particular, activity diagrams syntax has been written from scratch in UML2[1] (OMG, 2005), new constructions have been defined to meet complex systems modeling requirements; defining thus non-local behaviors that can also be reactive and transactional. Indeed, to rigorously represent the behavior of such complex systems with UML activities, it is essential to capture non-local actions triggering related behaviors as well as cancellation patterns and the advanced resulting synchronization mechanisms. Indeed, elementary Petri nets based semantics for these new concepts still currently ambiguous and too basic. The lack of formalization in UML2 can lead to confusion and different interpretations in the analysis model, which reduces the ability to develop required tools to assist formal specification process. For their part, designers require more adequate formal models to verify, validate and refine UML models.

Moreover, the standard does not provide adequate examples to illustrate the intent behind these concepts and clarify their meaning. A series of Bock publications (Bock, 2003a), (Bock, 2003b), (Bock, 2003c) (Bock, 2004) (Bock, 2005), a founder of UML2 activities, clarified some concepts and dynamics of inherent execution, yet, in a natural language.

The UML2 standard hinted that there is a quasi isomorphism between activities and Petri nets; this has launched many researchers' attempts to investigate in the formalization of activities via Petri nets. But they quickly noticed that place/transition Petri nets are not enough expressive to handle such complex semantic (streams, expansions, exceptions, etc). Two solutions are suggested to overcome the problem: 1) Expand some activity parts and then formalize them via place/transition Petri nets concepts (Barros, et al., 2003) and 2) Resort directly to high level class Petri nets. Yet, giving rise to a multitude of models, each one handling a particular activity concept. Our works are situated in this mind set. We have shown in our previous publications (Boufenara, et al., 2009a) (Boufenara, et al., 2009b) (Boufenara, et al., 2010) that the place/transition Petri nets do not preserve

[1] The current published version of UML is the UML2.3 (OMG, 2010). With regard to Activity Diagrams, UML versions 2.1.1 to 2.3 are minor revision to UML2.0 specification.

the semantics of the original addressed activity concepts and we have proposed its formalization via another Petri nets high level class inspired from Zero Safe Nets (ZSNs) of Bruni (Bruni, et al., 1997). In this paper, our purpose is to give a unified formal model (TPNets) that encompasses the whole activity model and resolves problematic situations, whilst being intuitive; we offer solutions to many concepts that have not been treated in the literature: reactivity, non-local behavior, synchronization of fork/join nodes, special synchronization at the final node, streaming parameters, etc.

The paper is structured in 6 sections: in Section 2, we give the most pertinent related work and discuss their weaknesses. Section 3 is devoted to presenting zero-safe nets, UML2 activity diagrams (by means of our proposed formalism AD2s "Activity Diagrams in UML2") and rewriting logic. In Section 4, we informally (for lack of space) present our approach which encompasses the proposed model TPNets along with syntactic and behavioral aspects and the transformation of AD2s to TPNets. Three case studies are presented in Section 5 to illustrate our formalization approach of UML2 activities. Finally, we conclude by giving the most relevant contribution of the paper.

2. RELATED WORK

The feasibility of a particular semantic approach depends largely on how it addresses the characteristics of the diagrams it covers. The most interesting activities formalization are those that define a literally formal semantics contribution. In the state of the art, few works are relevant to mention; Barros and Gomes proposed in (Barros, et al., 2003) a translation of a subset of activities concepts, including streaming and expansions, to Petri nets. They focus on some streaming aspects and try to overcome the failure of place/transition Petri nets to support this concept by defining a rich semantics for actions. To do so, they expand actions to activities. In fact, this is an improper manipulation of activity diagrams, since the UML standard states that an action is elementary and represents a single step and may be complex in its effect (refer to (OMG, 2005) p.321) so it can not in any way be interrupted. Effectively, when translating actions, considered now as activities, to Petri nets, additional places are created, leading to intermediate states and hence to the UML underlying semantics violations. Moreover, only local behaviors are considered in place/transition Petri nets, while the non-locality semantics is an important and innovative feature of UML2.

A series of Störrle culminating publications focuses on a thorough study of activities semantics and proposes different formalizations of some concepts. In (Störrle, 2004a), the author defines a denotational and compositional semantics of activities by means of transformation rules to the procedural Petri nets. Emphasis is set on control flows; including procedural calls, nevertheless without any indication about data. The first work is extended in (Störrle, 2004b) to cover exceptions. These ones involve non-local control flow to capture the overall state distribution which is obviously complex to model with place/transition and procedural Petri nets. To address the problem, the author defines an exception colored Petri nets based approach. The author only focuses on directing the control flow to the exception handler. Exception semantics is related to the internal behavior of action raising the exception, creating thus a non-deterministic internal choice. We do not need adding artifacts because deterministic choice naturally directs control to the exception handler. In (Boufenara, et al., 2009b), we give an intuitive solution to exception problems. In (Störrle, 2004b), the author formalizes structured nodes by non-formal transformations (examples) to procedural colored Petri nets. We consider that Störrle work is quite relevant, it is rather criticized the multitude semantic domains while a UML model typically includes multiple concepts at once, which cannot be formalized according to the different Störrle work to obtain a single Petri net.

In our previous papers (Boufenara, et al., 2009a) (Boufenara, et al., 2009b) (Boufenara, et al., 2010), we solved many related problems to activities formalization and adapted zero-safe nets to best meet activities requirements. However, during our research, we noticed that ZSNs are not completely adapted to preserve the whole UML2 activities features. That was a revelation for a real change in our study. In this paper, we define a unified and precise operational semantics of UML2 activities. We develop a double sided integrated approach: On one hand, we define a new dedicated model to UML2 activities called Transactional Petri Nets (TPNets in short). TPNets are based on a stable transaction semantics inherited from Zero-Safe Nets: ZSNs (Bruni, et al., 1997), a class of high level Petri nets. On the other hand we define a more structured way to describe an abstraction of activity diagrams called AD2s (these ones cover all previously addressed concepts, and new ones: synchronization at join nodes, initial and final nodes and the overall activities static and operational semantics). AD2s resume UML2 activities syntactic concepts, while limited to most but not all expressiveness levels. They serve to define TPNets model rigorously assigned to UML2 activities. We also define, in our work, algebraic equations for formal mapping AD2s to TPNets. This is a critical key step in our development. In fact, it allows transparency of formalization process; we build on earlier analyses rather than discard them, so developers have no longer to worry about rewriting their informal activities models in formal ones (TPNets over here) thereby engendering transformation omissions and mistakes. We moreover define TPNets operational semantics under the rewriting logic, so TPNets firing rules are described by rewriting rules and firing conditions are expressed via membership equations.

3. BACKGROUND

The objective of this section is to present elementary concepts of formalisms used in this paper: Zero-Safe Nets model, UML2 activity diagrams and rewriting logic.

3.1 Zero safe nets

Zero-safe nets have been introduced by Bruni in (Bruni, et al., 1997) to define synchronization mechanism among transitions, without introducing any new interaction mechanism besides the ordinary token-pushing rules of nets. Their role is to ensure the atomic execution of complex transitions collections, which can be considered as synchronized. Formally, a ZSN is a 6-tuple B = (S_B, T_B, F_B, W_B, u_B, Z_B) where N_B = (S_B, T_B, F_B, W_B, u_B) is the underlying place/transition net S_B is a non-empty set of places T_B is a non-empty set of transitions, $F_B \subseteq (S_B \times T_B) \cup (T_B \times S_B)$ is a set of directed arcs W_B is the weight function that associates a positive integer to each arc u_B is the places marking associating positive tokens number to each place $Z_B \subseteq S_B$ is the set of zero places (also called synchronization places and pictured by small circles). The places in $S_B \backslash Z_B$ are called stable places. A stable marking is a multiset of stable places. The presence of one or more zero places in a given marking makes it unobservable, while stable markings describe observable states of the system.

A firing sequence *s* = $u_0[t_1>u_1 \ldots u_{n-1}[t_n>u_n$ is a stable step of a ZSN if it guarantees the two following properties: 1) The concurrent enabling property which insures the initial simultaneous not conflicting enabling of all sequence transitions by stable places and not only those transitions allowing the initial triggering of the first execution[2] and 2) The stable fairness property which assumes that u_0 and u_n are stable markings.

A stable step *s* is a stable transaction[3] of B if in addition: 3) markings $u_1, \ldots, u_{n-1}$ are not stable and 4) The perfect enabling property that ensures the consummation of all initial stable tokens before the transaction ends, is satisfied.

3.2 UML2 Activities

Activities are a kind of graphs having their nodes defined in the meta-model of UML standard (OMG, 2005). We formally define an underlying structure of UML2 activities called AD2s. It takes back the first four activities expressivity levels and some aspects of the fifth one. We incrementally define the AD2s by first designing the core composed of the lowest level including basic fundamental concepts of activities, which is ADs. Its structure is a tuple AD = (EN, BN, CN, IN, fN, CF, Guard) where elements stand respectively In the underlying activity diagram for executable nodes, branch nodes (decisions and merges), concurrent nodes (forks and joins), initial nodes (will represent the initial marking in the TPNet), final node, control flows and guards on decisions output edges. On the other hand, an AD2 is more expressive, it is defined by a tuple AD2 = (AD, ON, OF, CR, SA, EA, IAR, EVT, Cancel, Weight) where elements respectively stand for AD defined above, object nodes (pins), object flows (data flows), concurrent region (a sub-activity diagram delimited by a global fork and a global join), streaming actions (actions with streaming parameters), exception actions (actions with exception outputs), interruptible activity region (a sub activity diagram containing interruptible actions due to reception of an external interrupting event)[4], interrupting event, cancel (action handling the exception) and finally edges weight.

3.3 Rewriting logic

In rewriting logic, a dynamic system is represented by a rewriting theory $\mathfrak{R} = (\mathbf{\Sigma}, \mathbf{E}, \mathbf{R}, \mathbf{L})$ describing the complex structure of its states and the various possible transitions between them. In rewriting theory definition, $(\mathbf{\Sigma}, \mathbf{E})$ represents an equational membership theory, $\mathbf{L}$ is a set of labels and $\mathbf{R}$ is a set of labelled conditional rewriting rules. These rewriting rules can be of the following form:

$$(\forall \mathbf{X})\mathbf{r}: \mathbf{t} \rightarrow \mathbf{t}' \, \mathbf{if} \bigwedge_{\mathbf{j} \in \mathbf{J}} \mathbf{W_j}: \mathbf{S_j} \wedge \bigwedge_{\mathbf{l} \in \mathbf{L}} \mathbf{t_l} \rightarrow \mathbf{t'_l}$$

Where $\mathbf{r}$ is a labeled rule, all the terms (p_i ,q_i ,w_j ,s_j ,t_l ,t_l') are $\mathbf{\Sigma}$-terms and the conditions can be

[2] This property prohibits the consummation of stable tokens produced in the step.

[3] In a stable transaction, each transition represents a micro-step carrying out the atomic evolution through invisible states. Stable tokens produced during the transaction are frozen. They become active in the system, only at the end of the transaction.

[4] Only nodes that might generate places via transformation rules in the TPNet are considered in the IAR. Since our purpose is not modeling but defining static and operational semantic, so it is of no interest to reconsider all syntactic IAR elements. Only those important for interrupting the activity in the region are to be considered. In a perspective of TPNets, interrupting the execution of some transitions means to disable them by destroying tokens of their preconditions.

rewritings, membership equations in (Σ, E), or any combination of both. Given a rewriting theory, we say that $\Re$ implies a formula $[t] \rightarrow [t']$ if and only if, it is obtained by a finite application of the rewriting logic deduction rules, Reflexivity, Congruence, Replacement and Transitivity (Meseguer, 1992). The theoretical concepts of the rewriting logic are implemented through the Maude language (clavel, et al., 2006). Its objective is to extend the use of the declarative programming and the formal methods to specify and verify critical and concurrent systems. A Maude program represents a rewriting theory, i.e., a signature and a set of rewriting rules. The computation in this language corresponds to the deduction in rewriting logic. Furthermore, it is implemented through a running environment, allowing prototyping and formal analysis of concurrent and complex systems.

In (Meseguer, 1992), the author has shown that rewriting logic naturally describes place/transition Petri nets and correctly captures atomic behavior and concurrency in these nets according to semantic choices that may be interleaving or true concurrency. In our work, we do not directly execute TPNets, we rather define executions by means of rewriting logic. This will make us avoid the combinatory explosion of the Petri net states space.

4. TRANSACTIONAL PETRI NETS FOR UML2 ACTIVITIES

Petri nets are not reactive, well known by their locality feature, while most of systems are reactive and might admit a non-local activation of computational steps. Looking ahead to faithfully describe the operational semantics of activities reactivity and global synchronization patterns, we define a new variant of ZSNs called TPNets, with special enabling and firing rules well adapted to model complex systems, including a priority among transitions execution and many other features that make them eligible to constitute a formal semantic framework of UML2 activities.

TPNets include static net structures to unfold activities actions. Actions appear atomic, but really they enclose a complex internal behavior, such as actions with exception outputs or streaming parameters. In UML2 activities, no synchronization is needed at the final node, whereas Petri nets just define ‘and’ synchronization at transitions entries. In TPNets, we define special transitions synchronization at final nodes.

For lack of space, we do not give detailed definitions in this paper; we refer interested reader to (Boufenara, 2010).

4.1 Definitions

A TPNet is the tuple TPN = (P, T, F, W, Z, RF, S_{IAR}, Cancel, Z_{cancel}, NTR_{except}, NS_{stream}, IP, sp, t_{fin}). It is also a bipartite graph composed of two types of nodes: places and transitions. We define six types of places: stable places, zero places *Z*, source (initial) places *IP*, final place *sp*, cancellation places Z_{cancel} (interface places) and decision zero places. Each stable initial place has only one output transition, cancellation places are zero places used as the system interface with external events[5], and decision places are zero colored places used to make atomic non-deterministic choices. A TPNet contains only one stable final place.

Transitions are of three types: simple (elementary) transitions, cancellation transitions *Cancel* and final transition t_{fin}. Elementary transitions are ordinary transitions, cancellation transitions have only S_{IAR}[6] as input places, defined to exclusively handle massive cancellation behavior (emptying S_{IAR} places, the set S_{IAR} is constructed by the AD2/TPNet transformation process) and directing control flow to exception handler. They are pictured by a highlighted rectangle in Figure 1. Final transition has only one output stable place sp, it is pictured via two parallel lines.

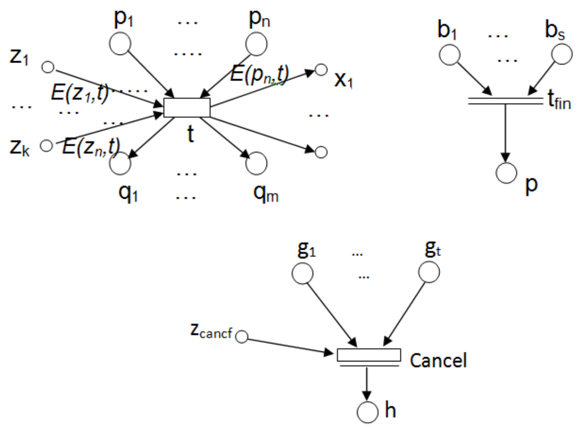

Figure 1: *A generic TPNet*

Places are connected to transitions via three types of directed arcs: non-labeled weighted/non-weighted arcs, labeled arcs (edges with guards) but with a weight equal to ‘1’ and reset arcs which can be interpreted as functions (places×transitions) that allow to a transition to delete all tokens of its input place connected via the reset arc. TPNets may also include sub-nets for exceptions NTR_{except}[7], or for

[5] Interface places are necessary to express, once marked, the reception of an interrupting event
[6] S_{IAR} is a set of places that have to be emptied to disable transitions of the interruptible region.
[7] NTRexcept is a sub-TPNet with a predefined static structure, composed of a decision zero place along with output transitions that are enabled according to the color of the decision place token. It is introduced to allow posting tokens exclusively at some output places of the same transition. We introduced some mechanisms that make the sub-net non-interruptible.

streams NS_{stream}[8]. Concurrent regions, meanwhile, do not need special notations or constructions and that is why they do not appear in this section. The passage from AD2s to TPNets will make them stand out. Figure 1 shows a generic TPNet.

4.2 TPNets behaviors

A TPNet can evolve in two different states: observable (stable) states and non-observable (atomic) states. This is possible thanks to stable places and ZSNs zero-places along with the resulting atomic execution semantics. This feature has been exploited in TPNets to establish coordination between fork nodes (triggers of parallel threads) and join (parallel threads synchronization node). Although fork and join are simply mapped into transitions in TPNets, the related semantics is preserved via a global synchronization mechanism. Fork and Join nodes have special semantics under the principle of traverse-to-completion[9], which is a global behavior principle of tokens offer and acceptance.

In TPNets, behaviors are related to tokens initial distribution in the graph, called initial marking, and transitions firing rules among the net. The initial marking is not user-defined; it rather assigns a positive number of tokens to each source place *ip* $\in$ *IP*. $M_0 = \{ip_1, ip_2, \ldots, ip_n\}$ such that *n* is the initial places number in the net.

Graphically, a token is a dark dot inside the place. There are two kinds of tokens; stable ones, associated to stable places and zero tokens associated to zero places. A marking M is said to be stable iff $\forall p \in M$, $p \in P \backslash Z$. In other words, a marking is stable if it does contain any zero token. It is non-observable otherwise. Initial marking is stable.

A transaction is a firing sequence which includes a set of transitions t_i with place/transition Petri nets elementary enabling and firing rules. Enabling transaction on the other hand, requires a whole concurrent enabling of internal transitions. Firing atomic transitions of a transaction *tr* does not require any new mechanism besides place/transition Petri nets one. But it is constrained by static rules inherent to ZSNs and which guarantee atomicity, stable fairness and perfect firing.

Cancel transitions are necessary to abort special regions in the net called S_{IAR}. They need to satisfy two enabling conditions: Initial Enabling condition: $z_{canc} \in M$ and Effective Enabling condition: The effective marking M that enables a transition cancel $\in$ Cancel is calculated dynamically (at runtime) and is equal to the instant marking of incoming places (each place p $\in$ S_{IAR}) of the transition Cancel calculated when z_{canc} is marked. Firing an enabled transition cancel $\in$ Cancel, destroys a token of interface place z_{canc} and all tokens of places p such that p $\in$ S_{IAR} and creates tokens in the outgoing places of cancel, enabling hence the transition corresponding to exception handler.

A transition t having incoming edge labeled by an expression from the set Expr should satisfy in addition to enabling rules the following rule: The tokens type (color) of each place p required to enable t is equal to the arc (p, t) label. Firing such transitions destroys tokens of the same color as the (place,transition) edge label.

Enabling final transition t_{fin} assumes that no tokens synchronization is required. Its firing is the same as Petri nets elementary transitions firing. We give some case studies in section 5 to better clarify TPNets principles.

In our work, we assign an operational semantics to TPNets. To achieve this, we first define a revised rewriting logic based semantic framework for ZSNs. Then we adapted this mathematical model to give a sufficient and precise semantics to TPNets. Thus, we do not only define an operational semantics to the TPNet behavior, but we also be able to produce executable specifications through Maude system that may be formally analyzed. Maude (Clavel, et al., 2006) is a practical environment implementing rewrite theories.

4.3 From AD2 to TPNets

In this step, a formal mapping is established between AD2s and TPNets. It defines a formal semantic to the syntactic structure of AD2s using algebraic equations. In what follows, we give transformation equations. These ones are illustrated via case studies in section 5.

Let AD2 be an activity diagram *AD2 = (AD, ON, OF, CR, SA, EA, IAR, EVT, Cancel)* where *AD = (EN, BN, CN, IN, fN, CF, Garde)*. The semantic model φ(AD2) of AD2 is a marked transactional Petri net $(TPNet, M_0) = (P, T, F, W, Z, RF, S_{IAR}, Cancel, Z_{cancel}, NTR_{except}, NS_{stream}, IP, sp, t_{fin}, M_0, Couleur, Expr)$ such as $NS_{stream} = (synch_1, synch_2, connect, p_{st}, z_{st}, t_{st1}, t_{st2})$ and $NTR_{except} = (z_{dec}, EF, t_{ac}, t_N, T_E)$. The corresponding TPNet is obtained by the following rules:

[8] NS_{stream} is a sub-TPNet introduced to bypass the basic Petri nets synchronization at transitions inputs/outputs and to allow global synchronization where streams can move along without any interne synchronization. No interruption of the execution of the NS_{stream} is permitted.
[9] ttc

- $P = BN \cup ON \setminus \{p_{oi} / \exists of = (p_{oi}, p_{oi+1})$ and $\neg \exists tb$ on $of\} \cup \{IN, fN\} \cup \{p_c / c \in CF\} \cup \{p \in (NTR_{except} \cup NS_{stream})\}$.
- $T = EN \setminus \{SA \cup EA\} \cup CN \setminus \{j_1, \ldots, j_{m'}\}$ for $(j_i$ OF $on_f) \vee (j_i$ CF $A_{f'}) \cup \{t_{oi} / \exists of \in OF \wedge of = o_i \times o_{i'}$ and $\exists tb$ on $of\} \cup \{t_{didi'} / \exists (d_i, d_{i'}) \in (OF \cup CF)\} \cup \{t_{mimi'} / \exists (m_i, m_{i'}) \in (OF \cup CF)\} \cup \{cancel_i / cancel_i \in spec_i\} \cup \{t_{of} / of \in OF \wedge (of.source \in IN \wedge of.target \in ON) \vee (of.source \in ON \wedge of.target = fN)\} \cup \{t \in (NTR_{except} \cup NS_{stream})\} \cup t_{fin}$.
- $F = (OF \cup CF) /$ if $\{x', y'\} \subset (EN \cup CN) \Rightarrow ((x, q), (q, y)) / ((x \in T) \wedge (y \in T) \wedge (q \in P)) \cup \{(x, y) / (x, y) \in (NTR_{except} \cup NS_{stream})\}$.
- $W =$ *Weight.*
- $M_0 = IN$.
- $Z = \{p / (p = n) \wedge (n \in CR)\} \setminus \{p / \neg\exists A_i / (f, A_i) \in (OF^* \cup CF^*) \wedge (A_i, j) \in (OF^* \cup CF^*)\} \cup EVT \cup \{z / -z \in (NS_{stream} \cup STR_{except}\}$.
- $S_{IAR} = \{p / (p \in P) \wedge (p = n / n \in IAR\}$.
- *Cancel = Cancel.*
- $Z_{cancel} = EVT$.
- $NTR_{except} = EA$.
- $NS_{stream} = SA$.
- $IP = IN$.
- $sp = fN$.
- $t_{fin} = A_i / (A_i, fN) \in CF$.
- *Expr = Garde.*
- *Couleur* $= \{o \mapsto o.type / o \in ON\}$

Decisions and merges, initial nodes and final node are transformed into TPNet places. Also, a place is created for each control flow arc. When two object nodes are connected through an edge not carrying a behavior, then they are mapped into one place for both. If the edge does carry any transformation behavior, then two places are created for each object node. Executable nodes are transformed into elementary transitions. Concurrent nodes fork are transformed into elementary transitions and join nodes followed by actions (and thus pins) are not transformed, i.e. they do not create any transition in the corresponding TPNet. Join nodes which outputs are different from actions with input pins are transformed into transitions. Edges carrying a transformation behavior *tb* also generate transitions. Other transitions are generated for sequential decision nodes and sequential merge nodes. For each action cancel, a special transition cancel of type Cancel is created. A transition is created when a source node is directly attached to an object node or when an object node is directly linked to a final node. Actions with input/output stream are converted into TPNets net structures NS_{stream} and actions with exception output parameters are converted to net structures NTR_{except}. All generated places of concurrent regions are zero ones. Some exceptions to that rule ensure the preservation of the traverse-to-completion principle[10]. A final transition corresponds to the last action in the underlying AD2.

5. CASE STUDIES

In our definition of TPNets, we do not associate to them any application domain, preventing hence a restricted use of such a powerful model. They are merely associated to UML2 activities as shown in figure 2.

To better illustrate our contribution, we present three case studies selected from literature and adapted to the modeling of some of the relevant aspects of our approach. The first one, business activity of the order process, illustrates the application of TPNets to capture the non-interruptible concurrent region activity. A variant of the basic diagram of this case study illustrates how TPNets handle interruptible regions. Finally, through the third example (Automatic cash dispenser), we illustrate how powerful are TPNets to detect deadlocks due to non-observation of the traverse-to-completion property in activities.

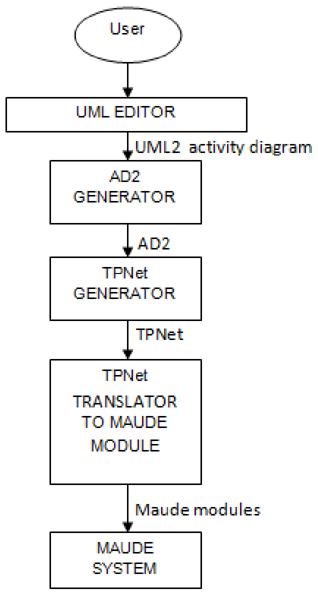

Figure 2: *Our Transformation process*

5.1 Process Order Commercial Activity

Figure 3 presents a classical example of the commercial activity of the ordering process taken and adapted from the UML2 specification (OMG, 2005). Partitions are illustrated in the example to

[10] The traverse-to-completion principle assumes that the edge traversal is constrained by the token offer acceptation by both source and destinations. The principle requires in some special cases a global synchronization between the fork and join nodes.

highlight responsible actors of different actions in the workflow.

The control begins when a customer sends an order "Send order", this one triggers sequential actions "Check availability" and "Calculate amount". A data "Price estimation is generated and sent to customer to make a choice (decision node, pictured here by a diamond) whether to cancel order and pass control to close customer record action "Close record" via the guard [Cancel] or change his order or validate order and hence route control to two concurrent threads via the fork node. Each concurrent thread is composed of sequential actions that are synchronized once all involved threads are finished via the join node. Hence, if a thread finishes before the other one, it cannot rout the control to join node because of the traverse-to-completion principle waiting until all threads finish and can offer their tokens to join node for synchronization. The final action terminates control in the activity diagram.

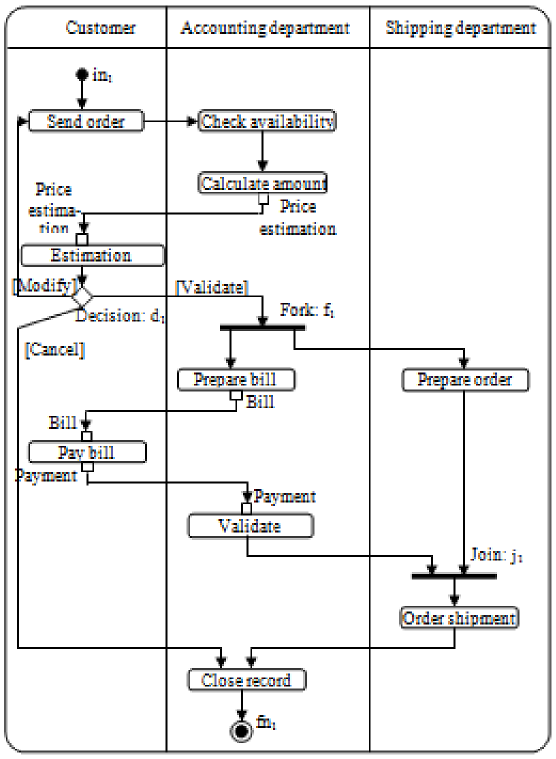

Figure 3. *Activity diagram of the process order commercial activity*

5.1.1. Identifying AD2 elements

Figure 3 presents AD2, the abstraction of the activity diagram corresponding to this case study.

AD2 = (EN_1, BN_1, CN_1, IN_1, fN_1, CF_1, ON_1, OF_1, CR_1, $Garde_1$) where:

EN_1 = {Send-order, Check-availability, Calculate-amount, Estimation-acceptance, Prepare-bill, Pay-bill, Validate-payment, Prepare-order, Order-shipment, Close-record}.

$BN_1 = \{d_1\}$

$CN_1 = \{f_1, j_1\}$

$IN_1 = \{in_1\}$

$fN_1 = \{fn_1\}$

$CF_1 \subset ((EN_1, BN_1, CN_1, IN_1) \times (EN_1, BN_1, CN_1, fN_1))$

$ON_1 = \{dt_1, dt_2, dt_3, dt_4, dt_5, dt_6\}$ where *$dt_1 = dt_2$ = Price-estimation, $dt_3 = dt_4$ = Bill* and *$dt_5 = dt_6$ = payment.*

$OF_1 = \{(dt_1, dt_2), (dt_3, dt_4), (dt_5, dt_6)\}$

CR_{11} = {{Prepare-bill, Pay-bill, Validate-payment, Prepare-order}$\cup\{dt_3, dt_4, dt_5, dt_6\}\cup$\{($f_1$, Send-order), ($f_1$, Prepare-order), (dt_3, dt_4), (dt_5, dt_6), (Validate-payment, j_1), (Prepare-order, j_1)\}}

$Garde_1$ = {modify, validate, cancel}

5.1.2. Design of the corresponding TPNet

Figure 4 models the generated TPNet. This one is generated using equations defined in the transformation model.

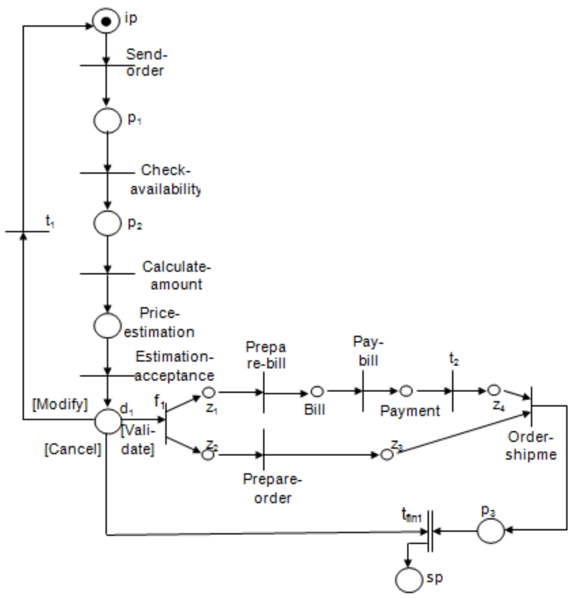

Figure 4. *TPNet model of the process order commercial activity*

TPNet = (P_1, T_1, F_1, Z_1, IP_1, sp_1, t_{fin1}, M_{01}, $Couleur_1$, $Expr_1$) such as:

P_1 = {ip_1, p_1, p_2, p_3, sp_1, Price-estimation, d_1, Bill, Payment, z_1, z_2, z_3, z_4}

T_1 = {Send-order, Check-availability, Calculate-amount, Estimation-acceptance, Prepare-bill, Pay-bill, Prepare-order, Order-shipment, f_1, t_{fin1}, t_1, t_2}

$F_1 \subseteq (P_1 \times T_1) \cup (T_1 \times P_1)$
$Z_1 = \{z_1, z_2, z_3, z_4, Bill, Payment\}$
$IP_1 = \{ip_1\}$
$M_{01} = \{ip_1\}$
$Couleur_1$: $d_1 \mapsto \{Modify, Validate, Cancel\}$
$Expr_1 = \{Modify, Validate, Cancel\}$

Note. Place d_1 corresponds to decision node. Firing transition *Estimation*-*acceptance* generates the d_1 internal token which value is Price-estimation. d_1 type is composite, at a logical level it is a record containing at least two fields: Price-estimation and acceptance-state that is calculated by *Estimation-acceptance.* This later takes one of three values *modify, validate or cancel*.

5.1.3. Model Execution and comments
The TPNet of figure 4 may be associated to the following rewrite theory $T_{B1} = (\Sigma_{B1}, E_{B1}, L_{B1}, R_{B1})$, defined by:

Σ_{B1} = {*Marking, ip_1 p_1 p_2 p_3 sp_1 Price-estimation d_1:→ splace, ∅ :→ Marking, Bill Payment z_1 z_2 z_3 z_4 :→ zplace, init-marking, fin-marking, _⊗_ :Marking Marking→ Marking, _∨_: Marking Marking→ Marking*}
$E_{B1} = \{p \otimes \varnothing = p,\ p \otimes p' = p' \otimes p,\ (p \otimes p') \otimes p'' = p \otimes (p' \otimes p''),\ p \vee \varnothing = p,\ p \vee p' = p' \vee p,\ (p \vee p') \vee p'' = p \vee (p' \vee p'')\}$
L_{B1} = {*Send-order, Check-availability, Calculate-amount, Estimation-acceptance, Prepare-bill, Pay-bill, Prepare-order, Order-shipment, f_1, t_{fin1}, t_1, t_2*}
R_{B1} = {
Send-order: $<ip_1,id> \rightarrow <p_1,id>$,
Check-availability: $<p_1,id> \rightarrow <p_2,id>$,
Calculate-amount: $<p_2,id> \rightarrow$ *<Estimation-price,id>*,
Estimation-acceptance: *<Estimation-price,id>*$\rightarrow <d_1,c_1>$,
f_1: $<d_1,c_1>E(d_1,f_1> \rightarrow <z_1,id> \otimes <z_2,id>\ E(d_1,f_1)$ *if* c_1*=validate,*
Prepare-bill: $<z_1,id> \rightarrow$ *<Bill,id>*,
Pay-bill: *<Bill,id>*$\rightarrow$ *<payment,id>*,
Prepare-order: $<z_2,id> \rightarrow <z_3,id>$,
Order-shipment: $<z_4,id> \otimes <z_3,id> \rightarrow p_3$,
t_{fin1}: $<p_3,id> \vee <d_1,c_1>E(d_1,t_{fin1}) \rightarrow <sp_1,id>E(d_1,t_{fin1})$ *if* c_1*=cancel,*
t_1: $<d_1,c_1>E(d_1,t_1> \rightarrow <ip_1,id>\ E(d_1,t_1)$ *if* c_1*=modify,*
t_2: *<payment,id>*$\rightarrow <z_4,id>$
}

Notice that no synchronization is defined on final transition. This is expressed by the operator ∨ (logic OR) on rule t_{fin1}. If we transformed our AD2 to a place/transition Petri net, this would have synchronized flows at the final transition, synchronizing hence *order-shipment* and the cancel process. What is semantically wrong and would lead to a deadlock because P_3 and *cancel* cannot be both marked. The traverse-to-completion principle assumes that in our case study, no interruption of the concurrent region is allowed. Transforming our AD2 to a place/transition Petri net will create intermediate stable places, thus intermediate states that can be interrupted. If we assume that once the billing process is started, no interruption is allowed, then the TPNet meets this requirement. Creating zero-places in the concurrent region will make it impossible to abort, because no zero token can be left at a stable state and a correct firing sequence has to end at a stable one.

In what follows, we give a simple execution example of our proposed model through some rewriting rules of T_{B1}. The initial marking is {ip_1}.

Send-order: $<ip_1,id> \rightarrow <p_1,id>$;
Check-availability: $<p_1,id> \rightarrow <p_2,id>$;
Calculate-amount: $<p_2,id> \rightarrow$ *<Estimation-price,id>;*
Estimation-acceptance:<Estimation-price,id>$\rightarrow <d_1,c_1>$; *--init-marking*
f_1: $<d_1,valider> \rightarrow <z_1,id> \otimes <z_2,id>$;
Prepare-bill: $<z_1,id> \rightarrow$ *<Bill,id>;*
Pay-Bill: *<Bill,id>*$\rightarrow$ *<Payment,id>;*
t_2: *<Payment,id>*$\otimes <z_2,id> \rightarrow <z_4,id>$)//
(*Prepare-order*: *<Payment,id>*$\otimes <z_2,id> \rightarrow <z_3,id>$);
Order-shipment: $<z_4,id> \otimes <z_3,id> \rightarrow p_3$; *--fin-marking;*
t_{fin1}: $<p_3,id> \vee <d_1,Cancel> \rightarrow <sp_1,id>$.

The operator «;» expresses sequential rewriting steps whereas the operator « // » expresses parallel rewriting steps.

States between *init-marking* and *fin-marking* are of type *zmarking,* so they are not interruptible. In our definition of rewriting rules, we impose that once a rewriting rule generates a *z-marking,* at runtime the left hand side of the rule is considered as an *init-marking* and that we must obtain a *fin-marking* so that proof is correct if not this one is forbidden. Hence, once billing process and order processing have begun, cancellation is no longer allowed.

5.2 Variant of the Process Order Commercial Activity

In the previous section, we presented a classical example of a commercial order treatment process where the customer belonging to the system could intervene and stop the order processing procedure through the action *Estimation-Acceptance*. Note that there exists only one point where customer could interrupt the process.

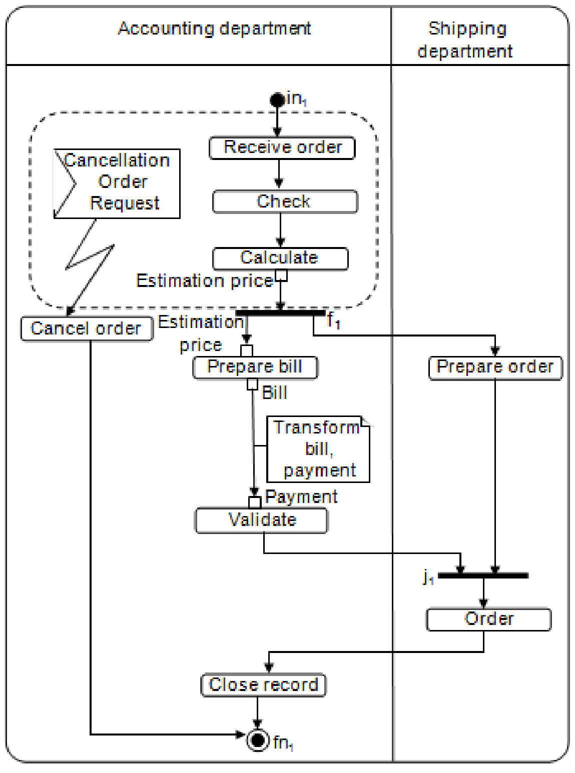

Figure 5. *Activity diagram of the process order commercial activity, including Interruptible region*

In this section, we discuss the same process but from another view (see figure 5): customer becomes an external entity to the system and therefore, he interacts with it through external events. Interrupting event *«cancellation»*: *Cancellation-Order-Request* corresponds to the guard *cancel* of figure 3, however not restricted to some exact and predefined interruption points. In the current view, the system becomes more flexible, accepting external real time interrupting events at multiple points of the interruptible region delimited by dashed rectangle. In the example, we do not accept interrupting order process once billing process is started. Hence, billing process does not belong to the interruptible region.

5.2.1. Identifying AD2 elements

Abstracting case study 2 activity diagram (figure 5) gives raise to the following AD2.

AD2 = (EN_2, CN_2, IN_2, fN_2, CF_2, ON_2, OF_2, CR_{12}, IAR_2, EVT_2, $Cancel_2$) where:

EN_2 = {Receive-order, Check-availability, Calculate-amount, Prepare-bill, Validate-payment, Prepare-order, Order-shipment, Close-record, Cancel-order}. Transform.bill.payment is a transformation behavior noted tb_1:

$CN_2 = \{f_1, j_1\}$

$IN_2 = \{in_1\}$

$fN_2 = \{fn_1\}$

$CF_2 \subset ((EN_2, CN_2, IN_2) \times (EN_2, CN_2, fN_2))$

$ON_2 = \{dt_1, dt_2, dt_3, dt_4, dt_5, dt_6\}$ where dt_1= dt_2 = Estimation-price, dt_3= Bill and dt_4= Payment.

$OF_2 = \{(dt_3, dt_4)\}$

CR_{12} = {{Prepare-bill, Validate-payment, Prepare-order}$\cup$\{dt_2,dt_3,dt_4\}$\cup$\{(f_1,dt_2), (f_1,Prepare-order), (dt_3,dt_4), (validate-payment, j_1), (Prepare-order, j_1)}}

IAR_2 = {Receive-order, Check-availability, Calculate-amount }

EVT_2 = {Cancellation-order-request}

$Cancel_2$ = {Cancel-order}

5.2.2. Generating the corresponding TPNet

Figure 6 models the corresponding TPNet of the given AD2, its formal definition is omitted and replaced by explanation.

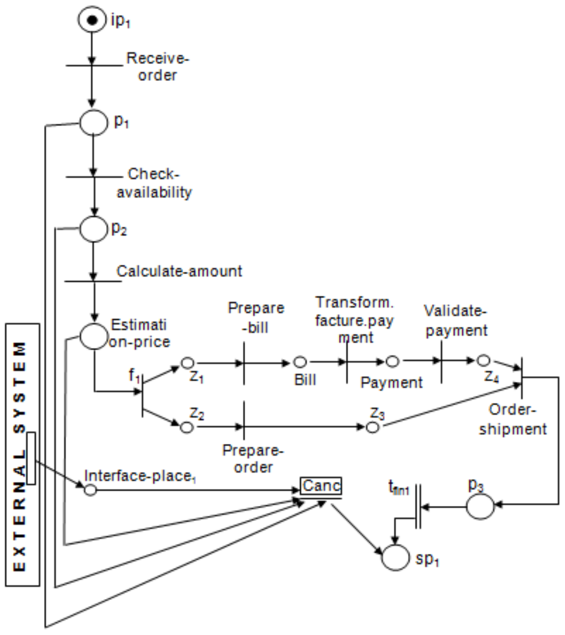

Figure 6. *A TPNet handling cancellation event*

Once receiving a cancellation event i.e. once *interface-place* is marked, we do need to satisfy two conditions: 1) Raising and handling the exception: all places of the S_{IAR2}= *{p_1,p_2,Estimation-price}* have to be emptied at once and the control is moved to the handler (transition *cancel*) and 2) Priority and isolation of the abort execution: receiving an external event triggers the immediate blocking and abort activity in the interruptible region. In terms of Petri nets: firing two simultaneously enabled transitions is non-deterministic. The TPNet of figure 6 satisfies both conditions.

5.2.3. Discussion

Behaviors of the process order commercial activity including interruptible region are discussed via executions of the proposed TPNet model. In a similar way, we obtain $T_{B2} = (\Sigma_{B2}, E_{B2}, L_{B2}, R_{B2})$, the rewrite theory associated to *TPNet,* for lack of space, it is not given here.

In what follows, we give four model execution examples (proofs). Only ip_1 is initially marked.

Proof 1.

Receive-order: $<ip_1,id> \rightarrow <p_1,id>$;
Check-availability: $<p_1,id> \rightarrow <p_2,id>$;
Calculate-amount: $<p_2,id> \rightarrow <Estimation\text{-}price,id>$; *--init-marking*
f_1 : $<d_1,Validate> \rightarrow <z_1,id> \otimes <z_2,id>$;
(Prepare-bill: $<z_1,id> \rightarrow <Bill,id>$;
Transform-bill-payment : $<Bill,id> \rightarrow <Payment,id>$;
Valider-payment : $<z_2,id> \otimes <Payment,id> \rightarrow <z_4,id>$) //
(Prepare-order: $<z_2,id> \otimes <Payment,id> \rightarrow <z_3,id>$);
Order-shipment: $<z_4,id> \otimes <z_3,id> \rightarrow p_3$; *--fin-marking;*
t_{fin1} : $<p_3,id> \rightarrow <sp_1,id>$.

No interrupt event raises in this execution of T_{B2} so it ends in a normal final state sp_1.

Proof 2.

Receive-order: $<ip_1,id> \rightarrow <p_1,id>$;
$canc_1$: $<Interface\text{-}place_1,id> \otimes <\_p_1,id> \otimes <\_p_2,id> \otimes <\_Estimation\text{-}price,id> \rightarrow <sp_1,id>$.

Proof 2 ends in a normal final state sp_1 upon receipt of a cancellation event (marking *interface-place*$_1$) while processing *Receive-order*. Indeed, the initial enabling condition is satisfied when *interface-place*$_1$ is marked, on the other hand effective enabling condition is calculated dynamically and contains $\{p_1\}$ in this case. Isolation is ensured because running *Check-availability* prior to $canc_1$ will disable this latter and keep the system in a non-observable state. One can think that p_1 can be provided another token to continue executing the TPNet, but this is not allowed since other tokens generated tokens will be frozen until the transaction ends.

Similarly, Rewriting rules 3 and 4 end after receiving cancellation event respectively while processing *Check-availability* and *Calculate-amount.*

Proof 3.

Receive-order: $<ip_1,id> \rightarrow <p_1,id>$;
Check-availability: $<p_1,id> \rightarrow <p_2,id>$;
$canc_1$: $<interface\text{-}place_1,id> \otimes <\_p_1,id> \otimes <\_p_2,id> \otimes <\_Estimation\text{-}price,id> \rightarrow <sp_1,id>$.

Proof 4.

Receive-order: $<ip_1,id> \rightarrow <p_1,id>$;
Check-availability: $<p_1,id> \rightarrow <p_2,id>$;
Calculate-amount: $<p_2,id> \rightarrow <Estimation\text{-}price,id>$;
$canc_1$: $<interface\text{-}place_1,id> \otimes <\_p_1,id> \otimes <\_p_2,id> \otimes <\_Estimation\text{-}price,id> \rightarrow <sp_1,id>$.

5.3 Automatic Cash Dispenser

In this case study, we illustrate how powerfull TPNets are to reveal conception errors in the UML activities development process. This is done through a hypothetical example of an automatic cash dispenser (see figure 7).

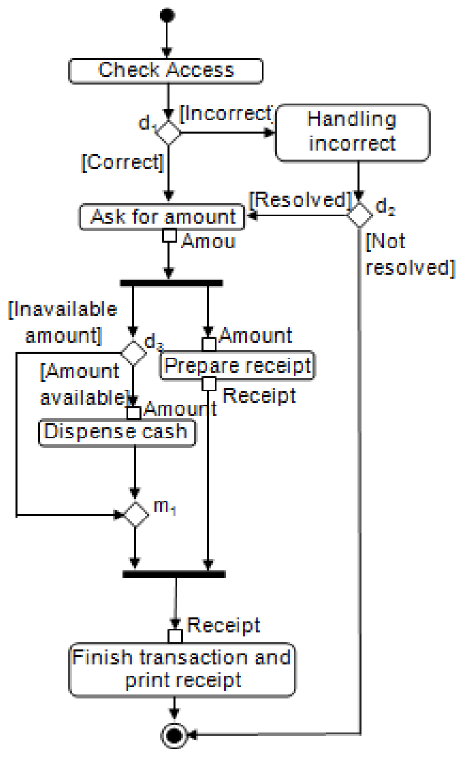

Figure 7. *Activity diagram of the automatic cash dispenser process*

Fork node triggers two parallel branches; the left one may contain, in some executions, no executable node. According to the traverse-to-completion principle, the activity diagram contains a deadlock due to circular dependency between fork and join nodes. A fork/join global synchronization is necessary to capture such global behavior; nevertheless this is not possible within place/transition Petri nets. TPNets can observe such deadlock and forbid executions leading to it.

5.3.1. Identifying AD2 elements

This AD2 is the abstraction of case study 3 activity diagram. *AD2 = (EN_3, CN_3, BN_3, IN3, fN_3, CF_3, ON_3, CR_{13})* such as:

- *EN_3 = {Check-access-code,* Handling-incorrect-access-code, *Ask-for-Amount, Prepare-receipt, Dispense-cash,* Finish-transaction}.
- *$CN_3 = \{f_1, j_1\}$*
- *$BN_3 = \{d_1, d_2, d_3, m_1\}$*
- *$IN_3 = \{in_1\}$*
- *$fN_3 = \{fn_1\}$*
- *$CF_3 \subset ((EN_3, CN_3, BN_3, IN_3) \times (EN_3, CN_3, BN_3, fN_3))$*
- *$ON_3 = \{dt_1, dt_2, dt_3, dt_4\}$, dt_1, dt_2, dt_3, dt_4* respectively stand for *Access-code, Result, Amount and Receipt.*
- *CR_{13} = {{Prepare-receipt, Dispense-cash}$\cup$ $\{dt_3, dt_4\} \cup \{(f_1, dt_3), (f_1, d_3), (d_3, dt_3), (dt_1, d_1), (dt_2, d_2), (dt_4, j_1)$, (Dispense-cash, m_1)}}*

$Garde_3$ = {correct, incorrect, Resolved, Not-resolved, Amount-available, Unavailable-amount}

5.3.2. Generating the corresponding TPNet

In the same way, we obtain the corresponding *TPNet = (P_3, T_3, F_3, Z_3, IP_3, sp_1, t_{fin1}, M_{03}, $Couleur_3$, $Expr_3$).*

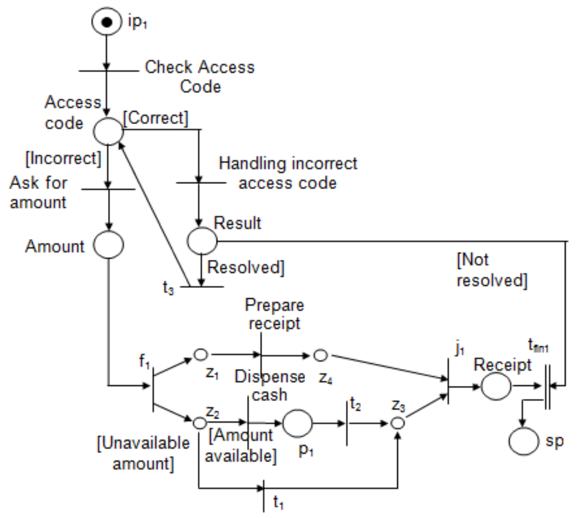

***Figure 8.** Automatic dispense cash TPNet*

Notice that p_1 is stable, hence once marked, the token is frozen until the transaction ends (no zero token is left), this is impossible because of the circular dependency in the region. The TPNet execution semantic ensures to forbid such runnings.

5.2.3. Discussion

We associate the rewrite theory T_{B3} to this *TPNet. $T_{B3} = (\Sigma_{B3}, E_{B3}, L_{B3}, R_{B3})$:*

Σ_{B3} = {Marking, ip_1 p_1 sp_1 Access-code Amount Result $Receipt_1$:$\rightarrow$ splace, $\varnothing$:$\rightarrow$ Marking, z_1 z_2 z_3 z_4:$\rightarrow$ zplace, init-marking, fin-marking, _$\otimes$_: Marking Marking$\rightarrow$ Marking, _$\vee$_: Marking Marking$\rightarrow$ Marking}

$E_{B3} = \{p \otimes \varnothing = p,\ p \otimes p' = p' \otimes p,\ (p \otimes p') \otimes p'' = p \otimes (p' \otimes p''),\ p \vee \varnothing = p,\ p \vee p' = p' \vee p,\ (p \vee p') \vee p'' = p \vee (p' \vee p'')\}$

L_{B3} = {Check-access-code, Handling-incorrect-access-code, Ask-for-amount, Prepare-receipt, Dispense-cash, f_1, j_1, t_{fin1}, t_1, t_2, t_3}

R_{B3} = {

Check-access-code: $<ip_1,id> \rightarrow <$Access-code,$c_1>$,

Handling-incorrect-access-code: < Access-code,c_1> E(Access-code, Handling-incorrect-access-code)$\rightarrow$ <Result,id>E(Access-code, Handling-incorrect-access-code -incorrect) *if c_1=correct,*

Ask-for-amount: <Access-code,c_1>E(Access-code,Ask-for-amount)$\rightarrow$ <Amount,id>E(Access-code,Ask-for-amount) if c_1=incorrect,

t_3: <Result,c_2>E(Result,t_3)$\rightarrow$ <Access-code,c_1>E(Result,t_3) if c_2=Resolved,

f_1: <Amount,id>$\rightarrow$ <z_1,id>$\otimes$ <z_2,c_3>,

Prepare-receipt: <z_1,id>$\rightarrow$ <z_4,id>,

Dispense-cash: <z_2,c_3>E(z_2, Dispense-cash)$\rightarrow$ <p_1,id>E(z_2, Dispense-cash) if c_3=Amount-available,

t_1: <z_2,c_3>E(z_2,t_1)$\rightarrow$ <z_3,id>E(z_2,t_1) if c_3=Unavailable-amount,

t_2: <p_1,id>$\rightarrow$ <z_3,id>--p_1 token is frozen untill reaching a smarking ,

j_1: <z_3,id>$\otimes$ <z_4,id>$\rightarrow$ <$Receipt_1$,id>

t_{fin1}: <$Receipt_1$,id> $\vee$ <Result,c_2>E(Result,t_{fin1}) $\rightarrow$ <sp_1,id>E(Result,t_{fin1}) if c_2=Not-resolved

}

The following executions show how the defined TPNet preserves activity initial operational semantics.

Check-access-code: <ip_1,id>$\rightarrow$ <Access-code,c_1> ;

Ask-for-amount: <access-code,c_1>$\rightarrow$ <Amount,id>;--init-marking

f_1 : <Amount,id>$\rightarrow$ <z_1,id>$\otimes$ <z_2,c_3> ;

(Prepare-receipt: <z_1,id>$\otimes$ <z_2,c_3> $\rightarrow$ <z_4,id>)//

(Dispense-cash: <z_1,id>$\otimes$ <z_2,c_3> $\rightarrow$ <p_1,id>)

Running T_{B3} gives a *zmarking <z_4,id>$\otimes$<p_1,id>* from which no other rule is applied. We cannot apply rule t_2 because p_1 token is frozen since we still are in a *zmarking*, Hence the underlying TPNet can be stuck in a non-final state. To avoid this, TPNet will forbid this execution.

6. CONCLUSION

This paper has addressed the issue of defining a unified semantic framework for UML2 activities by means of TPNets model. It is a new class of ZSNs that offers a non-local behavior related to transitions enabling allowing handling cancellation and advanced synchronization patterns. The TPNets are reactive since they are receptive to external events via zero-places also called interface places. The handling of external events is immediate in TPNets, suggesting execution priority semantic that defies Petri nets semantics. In fact, a Petri net enabled transition may be fired or not, also with no execution priority. Taking into account external events is done in transactional mode thereby ensuring isolation property, i.e. transaction is the unique to see the data it manipulates and that other system actions (transitions in TPNets) see only statements prior to the transaction (statements which triggered the transaction). Durability is an important feature of TPNets because it allows them to maintain the system stability lasting after a transaction is finished. This property is necessary to prevent calculations that may lead to unstable states (eg. not completed transactions). None of Petri nets variants can express reactivity and priority of transitions firings as we do by isolation and with no conflict in executions. Extended Petri nets (Petri nets with priority) explicitly express priority among transitions by associating a non negative integer to each transition, this leads to indirect conflict situations among concurrent transitions having the same priority.

REFERENCES

Barros, J. P. and Gomes, L. 2003. Actions as Activities and Activities as Petri nets. Bernhard Rumpe, Robert France, and Eduardo B. Fernandey Jan J¨urjens.Proc.Ws.Critical SystemsDevelopment with UML. 2003, pp. 129-135.
Bock, C. 2003a. Activity and action models part 2: Actions. 2003a, Vols. Journal of Object Technology, 2(5): pp. 41-56.
Bock, C. 2003b. Uml 2 Activity and action models. Journal of Object Technology. 2003b, Vol. 2(4), pp. 43-53, July-August 2003.
Bock, C. 2003c. Uml 2 activity and action models part 3: Control nodes. Journal of Object Technology. 2003c, Vol. 2(6), pp. 7-23, 2003.
Bock, C. 2004. Uml 2 activity and action models part 4: Object nodes. Journal of Object Technology. pp. 27-41, January-February 2004, 2004, Vol. 3, 1.
Bock, C. 2005. UML 2 Activity and action models Part 6: Structured Activities. Journal of Object Technology. May-June 2005, 2005, Vol. 4, 4.
Boufenara, S. 2010. Les Réseaux de Petri Transactionnels (TPNs), Un Cadre Sémantique des Activités dans UML2. [Thèse d'état]. Constantine, Département d'Informatique : Université Mentouri, Octobre 2010.
Boufenara, S., Belala, F. and Barkaoui, K. 2010. Mapping UML2.0 Activities to Zero-Safe Nets. J. Software Engineering & Applications JSEA. 2010, Vol. doi: 10.4236/jsea.2010.35048, 3.
Boufenara, S., Belala, F. and C.Bouanaka. 2009a. Les Zero-Safe Nets pour la préservation de la TTC dans les diagrammes d'activité d'UML. Revue des Nouvelles Technologies de l'Information RNTI-L-3, 15ème Conférence Internationnale sur les Langages et Modèles à Objets : LMO 2009. Cépaduès éditions, 2009a, pp. 91-106.
Boufenara, S., F. Belala and Debnath, N. 2009b. On Formalizing UML 2.0 Activities: Stream and Exception Parameters. 22nd International Conference on Computers and Their Applications in Industry and Engineering CAINE. 2009b.
Bruni, R. and Montanari, U. 1997. Zero-safe Nets, or transition synchronization made simple. Proceedings EXPRESS'97, ENTCS 7, Elseive. 1997.
Clavel, M., et al. 2006. All About Maude. 1997-2006 SRI International, 2006.
E.Guerra and J.D.Lara. 2003. A framework for the verification of UML models. Examples using Petri nets. .JISBD. 2003, pp. 325-334.
Meseguer, J. 1992. Conditioanl rewriting logic as a unified model of concurrency. Theoretical computer science. 1992, Vol. 96(1), pp. 73-155.
OMG. 2003. MDA Guide 1.0.1. Object management group. 2003, Document 03-06-01.
OMG. 2005. Unified Modelling Language: Superstructure. Version 2.0. s.l.: Object Management Group. Available at www.omg.org, 2005.
OMG. 2010. Unified Modelling Language: Superstructure. Version 2.3. s.l.: Object Management Group. Available at www.omg.org, 2010.
Störrle, H. 2004a. Semantics of Control-Flow in UML 2.0 Activities. VLFM. 2004a.
Störrle, H. 2004b. Semantics of Exceptions in UML 2.0 Activities. Journal of Software and Systems Modeling. 2004b.
Trickovic, I. 2000. Formalizing activity diagrams of UML by Petri nets. Novi Sad J. Math. 2000, Vol. 30, 3, pp. 161-171.