

Activity Diagrams Patterns for Modeling Business Processes (Report version)*

Étienne André, Christine Choppy, and Gianna Reggio

Abstract Designing and analyzing business processes is the starting point of the development of enterprise applications, especially when following the SOA (Service Oriented Architecture) paradigm. UML activity diagrams are often used to model business processes. Unfortunately, their rich syntax favors mistakes by designers; furthermore, their informal semantics prevents the use of automated verification techniques. In this paper, (i) we propose activity diagram patterns for modeling business processes, (ii) we devise a modular mechanism to compose diagram fragments into a UML activity diagram, and (iii) we propose a semantics for the produced activity diagrams, formalized by colored Petri nets. Our approach guides the modeler task (helping to avoid common mistakes), and allows for automated verification.

1 Introduction

Business processes are collections of related and structured activities or tasks, producing a specific service or product. Being able to model and to analyze business processes is of paramount importance, not only for the design of such processes, but also in the field of the software development whenever the SOA (Service Oriented

Étienne André
Université Paris 13, Sorbonne Paris Cité, LIPN, F-93430, Villetaneuse, France
e-mail: Etienne.Andre@lipn.univ-paris13.fr

Christine Choppy
Université Paris 13, Sorbonne Paris Cité, LIPN, F-93430, Villetaneuse, France
e-mail: Christine.Choppy@lipn.univ-paris13.fr

Gianna Reggio
DIBRIS, Genova, Italy
e-mail: gianna.reggio@unige.it

* This work is partially supported by project #12 “Méthode de modélisation des systèmes dynamiques” (CREI, Université Paris 13, Sorbonne Paris Cité).

Architecture) paradigm [Erl07] is followed. The most common modeling notations for business processes are the BPMN² and the UML [UML] activity diagrams. We consider in this paper the UML since it offers also many other diagrams (classes, state machine, etc.), providing an integrated way to model all the aspects of a business as the used data and the participant entities; also it may be used in all the other phases of the software development. Furthermore, there is no relevant difference between the readability of the UML and of the BPMN (see, e.g., [PBA⁺08, BKO10]).

Although UML diagrams are widely used, they suffer from some drawbacks. Indeed, since UML specification is documented in natural language, inconsistencies and ambiguities may arise. First, their rich syntax is quite permissive, and hence favors common mistakes by designers. Second, their informal semantics in natural language prevents the use of automated verification techniques, that could help detecting errors as early as the modeling phase. We take as a basis for our work the latest version (2.4.1) of the UML specification.

Our contribution is twofold. First, we define precise activity diagrams for modeling business processes. These precise activity diagrams are based on patterns, that can be inductively composed so as to build complex activity diagrams. Our approach also takes classes into account. We have selected a minimal subset of the useful UML activity diagram constructs (viz., sequence, fork, join, choice, merge, loops). This paper does not consider accept and timed event, which is the subject of ongoing work. Second, we give a semantics to these patterns, by translating them into Colored Petri Nets (CPNs) [JK09] in a modular way. Petri net is a natural formalism as result of the translation: the UML specification explicitly mentions them, and the informal semantics of activity diagrams is given in terms of token flows.

Related Works. The first issue we address is that of an adequate notation and approach for business process modeling. [RRS⁺11, CDR⁺11] compare different styles of activity diagrams (precise, “ultra-light”) in experiments. The workflow pattern initiative [Wor] issued a collection of workflow patterns for business modeling. These patterns address the modeling of control, data, etc., and are expressed in Petri nets.

Another issue is to propose a formal associated semantics to UML diagrams using a formal notation, which is important to allow for automated verification [FELR98]. This has been addressed in quite a variety of works using automata, different kinds of Petri nets, etc., so we mention only a few. Instantiable Petri nets are the target of transformation of activity diagrams in [KT10], and this is supported by tool BCC (Behavioral Consistency Checker); however they do not consider data, whereas we do. In [DSP11, BM07], the issue is performance evaluation, from activity diagrams and others (use case, state diagrams, etc.) to stochastic Petri nets. In [ZL10] and [ACK12], various syntactic features of UML state machines are translated into CSP# and colored Petri nets, respectively. Also note that [GRR10] proposes an operational semantics of the activity diagrams (for UML 2.2). Börger [Bör07] and Cook *et al.* [CPM06] present other formalizations of the workflow patterns of [Wor] using formalisms different from Petri nets, viz., Abstract State Machines and Orc, respec-

² <http://www.bpmn.org/>

tively. In [MGT09], patterns for specifying the system correctness are defined using UML statecharts, and then translated into timed automata. The main differences with our approach are that the authors mainly focus on real-time properties, and the patterns of [MGT09] do not seem to be hierarchical: the “composition” of patterns in [MGT09] refers to the simultaneous verification of different properties in parallel. In [KH10], a reactive semantics is defined for a subset of UML activities, which makes it a precise design language for reactive systems. The same authors also define in [KH09] an automated compositional mechanism for UML activities together with an interface (a so-called External State Machine), seen as building blocks.

Outline. Section 2 presents the ingredients of our UML-based modeling for business processes (static view, activity diagram, etc.), details the activity diagram features we consider, and describes how to compose them in a modular way. Then, we provide a translation of the considered activity diagrams into colored Petri nets in Section 3 (activity diagram) and Section 4 (static view). We use as a running example an electronic commerce system EC. Section 5 concludes, gives some hints on our implementation, and sketches future directions of research.

2 Business Process Modeling

2.1 Precise Business Process Models

Business processes are collections of related and structured activities or tasks, producing a specific service or product. In this section, we consider *precise* models of business processes. The word “precise” means here that we define such models in a sharper way than usual; the word is used in several related works on models (see, e.g., [RRS⁺11]). A precise model of a business process consists of (1) the *static view*, i.e., a class diagram defining the types of all the entities in the process; (2) the list of the *process participants* and of the used data typed using the classes and the datatypes in the static view; and (3) an *activity diagram* representing the process behavior.

The process participants are entities taking part in a process, and can be classified as: (i) business worker, if they correspond to human beings acting in the process, (ii) system, if they are software or hardware systems with a role in the process, and (iii) business object, when they are passive entities used in the activities of the workers and of the systems. The classes in the static view may be stereotyped by <<worker>>, <<system>> and <<object>> to explicit which kind of entities they model. A class with these stereotypes is called an *entity class*.

The operations of the classes stereotyped by either <<worker>> or <<system>> represent the atomic activities that they are able to perform in the business process. These classes may have also some auxiliary operations stereotyped by <<aux>> not modeling any activity (indeed they are UML queries, i.e., they have no side effects, and always have a return type).

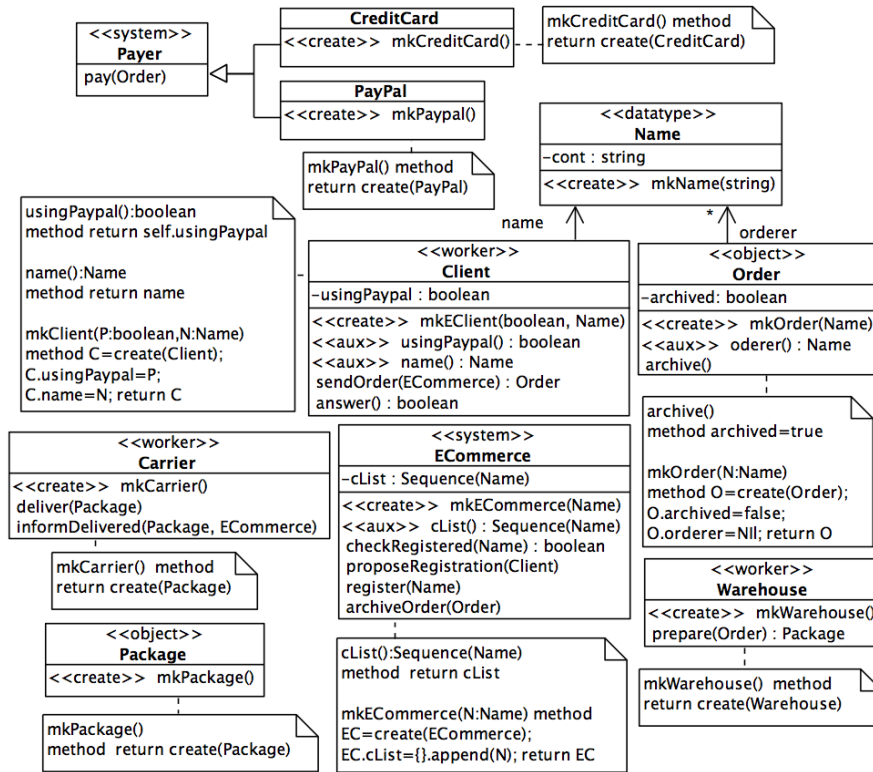


Fig. 1 EC example: static view

The operations of the classes stereotyped by <<object>> represent the atomic activities that may be performed over them. The constructor operations³ for any class have the stereotype <<create>>; an operation stereotyped by <<create>> has the class as return type (and for readability it will be not visualized in the class diagram), and it is static.

We consider an e-commerce EC as a running example of a precise business process. Fig. 1 presents its static view, while Fig. 2 presents its activity diagram and the list of the participants with the used data.

The EC business process has seven participants, and two of them, ORDER and PACK, are created during the process execution. Two boolean values, ANS and RES, are set during the process execution. It is important to note that the listed participants and data are not specific individuals, but roles that can be instantiated in many different ways. If a participant/data is marked by <<out>>, then it means that it is created/defined during the process execution.

³ The UML does not provide any native constructors.

The static view should be complemented with methods defining the meaning of the operations of the datatypes, of the classes stereotyped by <<object>>, and of any operation stereotyped by <<aux>> or <<create>>. In Fig. 1, the various methods are reported in notes attached to the corresponding classes. The behavior of the classes stereotyped by <<worker>> or <<system>> will be defined by state machines, where all events are calls of their operations not stereotyped by <<aux>>. In the case of the EC process, these state machines are not shown here. They have a simple “daisy form”, with a unique state and with a transition leaving and entering this state for any operation. This corresponds to say that the instances of these classes may perform anytime any atomic activity represented by an operation.

The following subsection describes how the business process behavior is modeled by a precise activity diagram.

2.2 *Precise Activity Diagrams*

2.2.1 UML Activity Diagrams

We first briefly recall UML activity diagrams [UML]. They feature in particular an *initial node* (e.g., the top node in Fig. 2), and two kinds of final nodes: *activity final*, that terminate the activity globally (“final1” and “final2” in Fig. 2), and *flow final*, that terminate the local flow [UML, Section 12.3.6, p.340]. More precisely:

“A token reaching an activity final node terminates the activity. [...] In particular, it stops all executing actions in the activity, and destroys all tokens in object nodes, except in the output activity parameter nodes. [...] If it is not desired to abort all flows in the activity, use flow final instead. Using a flow final will simply consume the tokens reaching it without aborting other flows.”

[UML, Section 12.3.6, p.340]

They also feature *choice* (e.g., “dec1”), i.e., the ability to follow one path among different possibilities, depending on guards, and *merge* (e.g., “Merge1”), i.e., the converse operation. They also feature *fork*, i.e., the ability to split the flow into different subactivities executed in parallel (e.g., the large line below “Merge1”), and *join*, i.e., the converse operation (e.g., the large line below “Merge3”).

2.2.2 Activity Diagrams Patterns

General Scheme for Patterns. We now introduce precise activity diagrams. Whereas UML activity diagrams provide a lot of freedom in the syntax, we give here precise rules for building activity diagrams in an iterative and modular way. First, from years of experience in the area of modeling, we believe that some of the syntactic

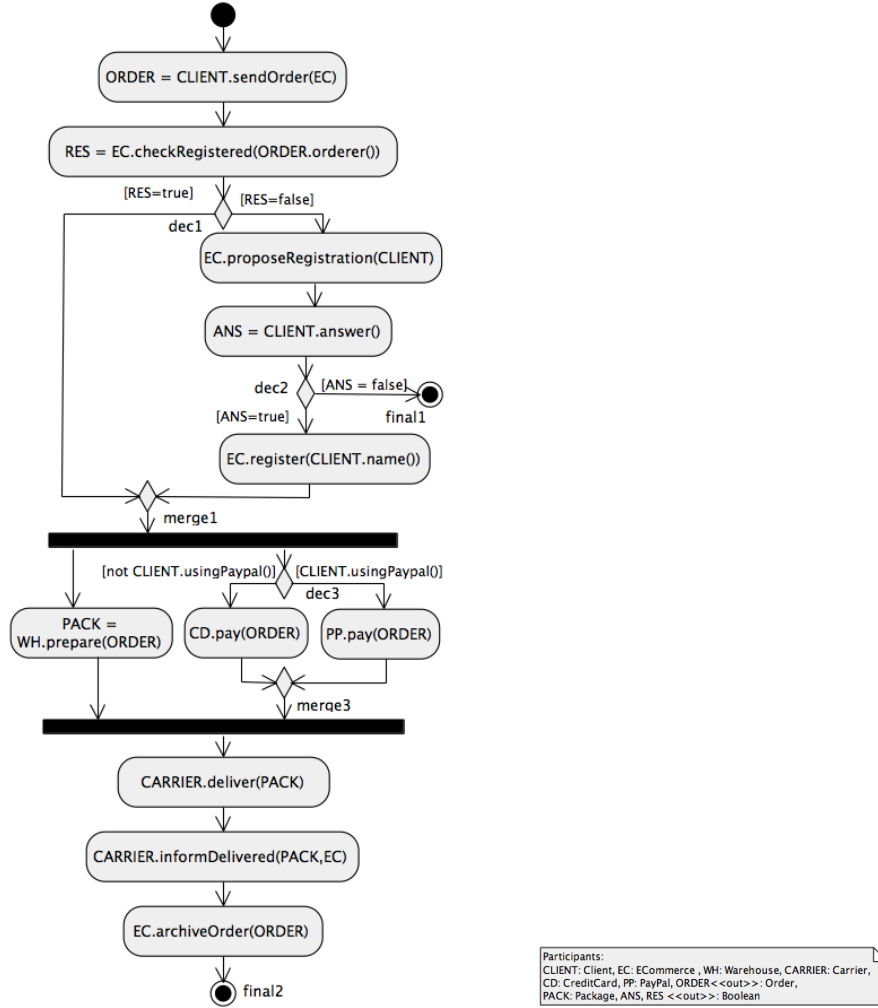


Fig. 2 EC example: activity diagram

features of UML activity diagrams are not often used in practice, or are ambiguous, and are then discarded here. Second, some constructions can reflect ill-formed diagrams. For example, we make here compulsory that a fork must always be eventually followed by a join, except in very particular cases. Hence, following these patterns can help the designer to avoid common mistakes (see, e.g., [RLR11]).

Providing these precise activity diagrams with a semantics will be the subject of Section 3. Note that, different from software engineering design patterns [GHJV95], that can be inserted into freely written code, precise activity diagrams are exclusively made of activity diagram patterns composed with each other.

Inductive Rules. We assume the static view and the list of the participants of the business process are already defined. Now, the set **PACT** of the precise activity diagrams is inductively defined below using a set of rules. Each rule defines an activity diagram pattern. For each activity diagram pattern in **PACT**, we define a *begin node* and an *end node*. Either the begin or the end node may be undefined, but not both. When composing the activity diagram fragments, we denote by \perp the fact that a fragment has no end node.

In the following **EXP** denotes the set of the OCL (Object Constraint Language) expressions built on the participant names, the operations of the datatypes defined in the static view, and the operations of the entity classes appearing in the static view stereotyped by $\ll\text{aux}\gg$. Such expressions are without side-effects on the process since the stereotype $\ll\text{aux}\gg$ requires an operation to be a query.

Rules 1–4 define simple patterns, whereas rules 5–8 define complex patterns by composing fragments built using the patterns. We also compare our patterns with those of [Wor], when applicable.



Rule 1: Initial. The initial node \bullet belongs to **PACT**, and its begin node is undefined, while its end node is itself.

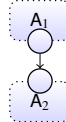
Rule 2: Activity final. The activity final node \odot belongs to **PACT**, and its begin node is itself, while its end node is undefined.

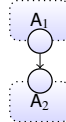
Rule 3: Flow final. \otimes belongs to **PACT**, and its begin node is itself, while its end node is undefined.

Rule 4: Action. If X is a participant of the process, Exp , $\text{Exp}_1, \dots, \text{Exp}_n$ belong to **EXP**, and op is an operation of a class stereotyped either by $\ll\text{worker}\gg$, $\ll\text{system}\gg$, $\ll\text{object}\gg$ in turn not stereotyped by $\ll\text{aux}\gg$ or $\ll\text{create}\gg$, then $\boxed{X := \text{Exp}}$ (4a), $\boxed{X := \text{Exp.op}(\text{Exp}_1, \dots, \text{Exp}_n)}$ (4b), and $\boxed{\text{Exp.op}(\text{Exp}_1, \dots, \text{Exp}_n)}$ (4c) belong to **PACT**, and their begin and end nodes coincide with themselves.





Rule 5: Sequence. This pattern corresponds to pattern 5 (“sequence”) in [Wor]. If

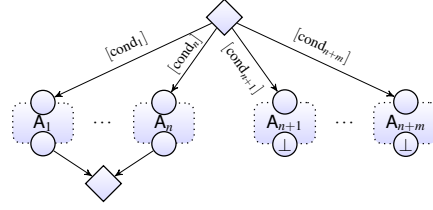
 (with a defined end node) and  (with a defined begin node) belong to





PACT, then  belongs to **PACT**, and has the begin node of A_1 and the end node A_2 , if they exist. Note that A_1 and A_2 represent here activity diagrams fragments inductively defined using our set of rules. The begin node of A_1 (resp. end node of A_2) is not depicted: this means it can either be defined or not. These conventions will be used throughout this section.

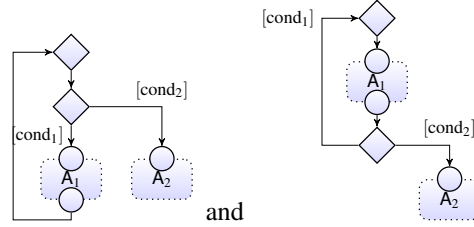
Rule 6: Decision/merge. Let $n \geq 1$, $m \geq 0$, $n + m \geq 2$.

If , ..., , , ...,  belong to **PACT**, if A_{n+1}, \dots, A_{n+m} have no defined end node, and if $\text{cond}_1, \dots, \text{cond}_n, \text{cond}_{n+1}, \dots, \text{cond}_{n+m}$ belong to **EXP**







such that $\bigvee_{i=1, \dots, n+m} \text{cond}_i = \text{true}$, then belongs to **PACT**. Its begin node is the decision node, and its end node is the merge node. This pattern can be seen as a combination and a generalization of patterns 4 (“exclusive choice”) and 5 (“simple merge”) in [Wor]. However, there are several differences: (1) we make the merge compulsory after a choice; (2) we allow some activities ($n+1$ to $n+m$) not to merge, providing they terminate (which is encoded by the fact that they have no end node); and (3) our choice is not exclusive (several guards may be true simultaneously, in which case the choice is nondeterministic).

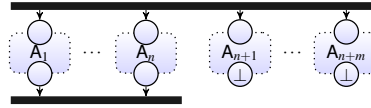
Rule 7: Loop. If  and  belong to **PACT**, and $\text{cond}_1, \text{cond}_2$ belong to

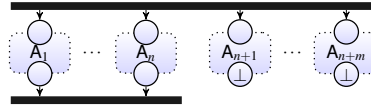


EXP with $\text{cond}_1 \vee \text{cond}_2 = \text{true}$, then belong to **PACT**; their begin node is the merge node, and their end node is the end node of A_2 . We name these two rules 7a (“while”) and 7b (“repeat until”) respectively. Rule 7a (resp. 7b) is similar to the while variant (resp. repeat variant) of pattern 21 (“structured loop”) in [Wor].

Rule 8: Fork/join. Let $n \geq 0, m \geq 0, n+m \geq 2$.

If , ..., , , ...,  belong to **PACT**, if A_{n+1}, \dots, A_{n+m} have



no defined end node, then  belongs to **PACT**. Its begin node is the fork node, and its end node is the join node if $n > 0$, otherwise it is undefined. This pattern can be seen as a combination and a generalization of patterns 2 (“parallel split”) and 3 (“synchronization”) in [Wor]. However, we make the join compulsory after a fork; and we allow some activities ($n+1$ to $n+m$) not to join, providing they terminate.

3 Translation of the Activity Diagram

In the remaining of the paper, we consider the translation into a CPN of the business process models introduced in Section 2. On the one hand, the translation of the static view and of the lists of the participants of a business process will result in a set of declarations of types and of functions over them defining a special type *State*, whose values represent the current situation of the process participants and of the process data during the process execution. On the other hand, the translation of the activity diagram will result in a CPN. This CPN will use the type declarations and functions in its inscriptions.

We first recall the formalism of CPNs (Section 3.1), and then introduce the translation of the activity diagram (Section 3.2). The translation of the static view will be the subject of Section 4.

3.1 Colored Petri Nets with Global Variables

We briefly recall here colored Petri net (CPNs) [JK09]. CPNs are an extension of Petri nets with color sets, or types. In CPNs, places, tokens and arcs have a *type*. In Fig. 3(a), place p_1 has type \mathbb{N} , whereas p_2 has type $\mathbb{N} \times \mathbb{B}$. Arcs can be labeled with *arc expressions* modifying the (colored) token (e.g., $(i, true)$ in Fig. 3(a)). Transitions can have a *guard*, hence enabling the transition only if the guard is true (e.g., $[i \neq 2]$). We use for arc inscriptions and guards the syntax of CPN ML, an extension of the functional programming language Standard ML, and used by CPN Tools [JK09].

Definition 1 (Colored Petri Net). A *colored Petri net* (CPN) [JK09] is a tuple $CPN = (P, T, A, Labels, B, V, C, G, E, I, L)$ such that:

1. P is a finite set of *places*,
2. T is a finite set of *transitions* such that $P \cap T = \emptyset$,
3. $A \subseteq P \times T \cup T \times P$ is a set of directed *arcs*,
4. B is a finite set of non empty *color sets* (types),
5. V is a finite set of *typed variables* such that $\forall v \in V, Type[v] \in B$,
6. $C : P \rightarrow B$ is a *color set function* assigning a color set to each place,
7. $G : T \rightarrow Expr(V)$ is a *guard function* assigning a guard to each transition such that $Type(G(t)) = \mathbb{B}$, and $Var[G(t)] \subseteq V$,
8. $E : A \rightarrow Expr(V)$ is an *arc expression function* assigning an arc expression to each arc such that $Type(E(a)) = C(p)_{MS}$, where p is the place connected to the arc a , and MS denotes the multiset, and
9. $I : A \rightarrow Expr(V)$ is an *initialization function* assigning an initial marking to each place such that $Type(I(p)) = C(p)_{MS}$.

A *marking* M of a CPN is a function providing for each place $p \in P$ a content that is of type $C(p)_{MS}$. The *initial marking* is denoted M_0 . Note that a CPN marking expresses the current state of the CPN.



Fig. 3 Example of a use of global variables

We use here the concept of *global variables*, a notation that does not add expressive power to CPNs, but renders them more compact. Global variables can be read in guards and updated in transitions. Some tools (such as CPN Tools) support these global variables. Otherwise, one can simulate a global variable using a “global” place, in which a single token (the type of which is the variable type) encodes the current value of the variable. An example of use is given in Fig. 3(a). The variable v (of type \mathbb{N}) is a global variable updated to the expression $v + i$. This CPN construction is equivalent to the one in Fig. 3(b). The case where a global variable is read in a guard is similar, with the difference that v is not modified.

3.2 Translation

The translation of the precise activity diagrams belonging to **PACT** (defined in Section 2.2) will be given compositionally following the rules defined there.

3.2.1 Assumptions

We make the following choice: each translated activity diagram fragment must start and finish with a place, so that the composition of the translations of the subparts is straightforward: it suffices to connect the places the same way as for the nodes we defined for the activity diagram patterns.

We define two global variables go : BOOL and s (see Section 4). In particular, variable go records whether the CPN should still execute, or should be completely stopped. This go variable is used to encode the activity final pattern (rule 2); if such a state is entered, then the whole process must immediately stop. Here, we assume that, for each transition of the CPN, the guard includes a check $[go = \text{true}]$ (for sake of conciseness, this variable will not be depicted in our graphics). This go variable is initialized with true , and will be set to false when entering the CPN transition encoding the activity final state (see Fig. 4(c)).

Note that all edges and places have type “UNIT”, i.e., the same type as in place/transition nets (we omit that type in Fig. 4 for sake of conciseness). Never-

theless, our CPN is still colored because of the use of global variables, guards in transitions, and functions updating the variables in transitions.

3.2.2 Translation of the Rules

We now give in Fig. 4 the translation of the rules from Section 2.2.2. The translation of each activity diagram pattern will result in a CPN fragment having the shape of Fig. 4(a). Modular composition is performed using the begin and end nodes, using the same way as for activity diagram patterns in Section 2.2.2.

Rule 1: Initial. The initial state is encoded into an initial place, containing the only initial token of the resulting CPN, followed by a transition assigning `InitState` to the global variable `s` (`InitState` will be detailed in Section 4). Finally, an outgoing place allows connection with the next component. The scheme is given in Fig. 4(b).

Rule 2: Activity final. An activity final pattern is translated (see Fig. 4(c)) into a transition updating the global variable `go` to `false`. Hence, since each transition has an implicit guard checking that `go=true`, the execution of the CPN is immediately stopped.

Rule 3: Flow final. A flow final pattern is translated (see Fig. 4(d)) into a simple place; hence local execution is terminated, without any consequence on the rest of the system.

Rule 4: Action. Recall from Section 2.2 that this rule translates the actions using three different schemes (i.e., Rules 4a, 4b, and 4c). We give the translation of the 3 rules in Fig. 4(f)–4(h).

Rule 5: Sequence. We give the translation of the sequence in Fig. 4(e). We translate A_1 and A_2 inductively, and we directly merge the end node of A_1 with the begin node of A_2 .

Rule 6: Decision/merge. Here, (only) one of the transitions will fire (depending on the guards⁴). If the corresponding activity has an end node (activities 1 to n), then the process continues afterwards from the outgoing place below; otherwise (activities $n + 1$ to $n + m$), it is stopped when the activity stops. The translation is given in Fig. 4(j).

Rule 7: Loop. The translation of the while loop (resp. repeat until loop) is given in Fig. 4(k) (resp. Fig. 4(l)).

Rule 8: Fork/join. The translation is quite straightforward and is given in Fig. 4(i). The $n + m$ activities are subject to a fork; then, only the n first activities are merged later.

A full translation of the activity diagram in Fig. 2 is available in [ACR13].

⁴ If several guards are true simultaneously, the choice is nondeterministic, according to the CPN semantics.

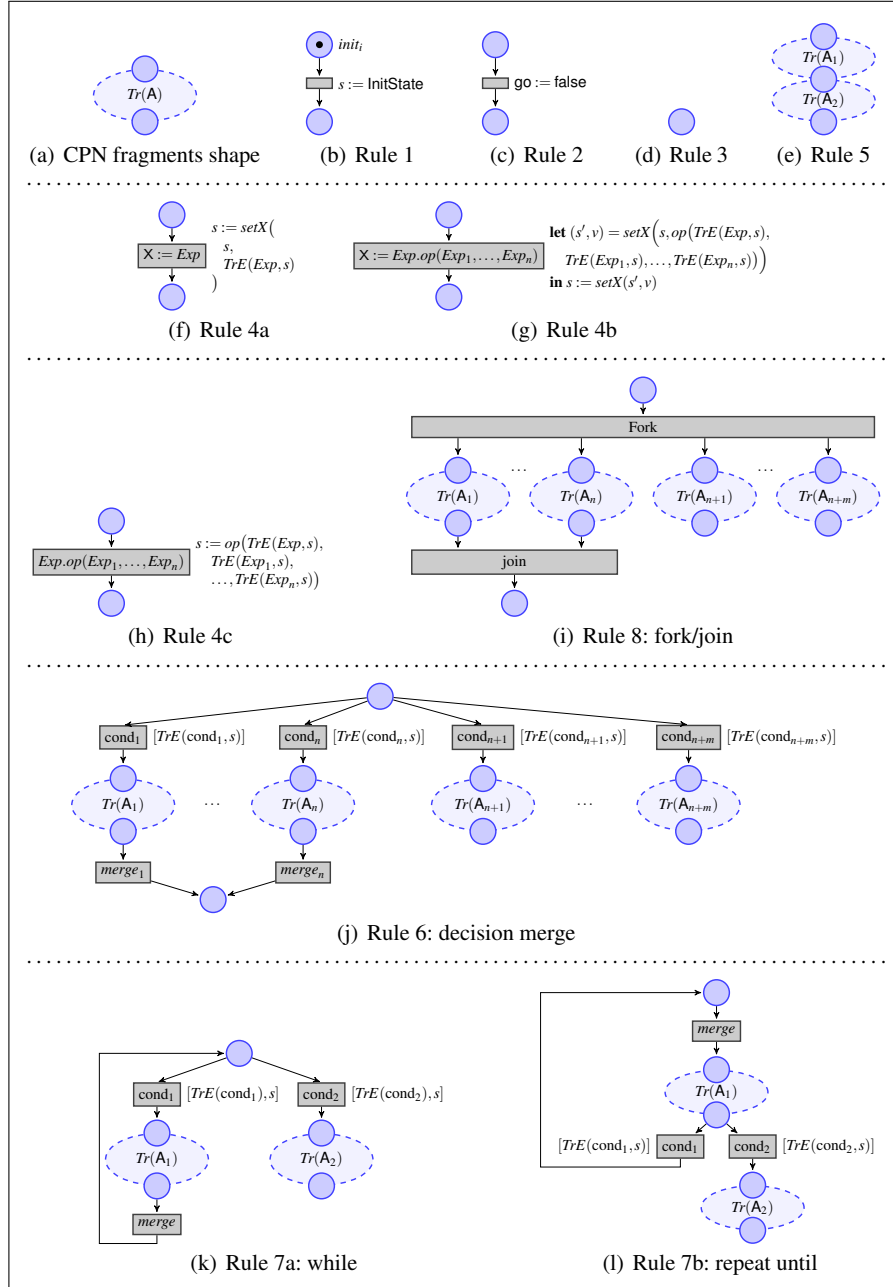


Fig. 4 Translating precise activity diagrams patterns into colored Petri nets fragments

4 Translation of the Static View and of the Participant List

In this section, we translate the static view and the participant list into a set of CPN ML declarations. In particular, we translate the type (color set) *State* together with a set of declarations of auxiliary types and of functions needed to handle them, used by the CPN defined in Section 3. Recall that the values of *State* represent the current situation of the process participants and of the process data during the execution of the process itself.

We first present the part of the translation generating the definition of *State* (Section 4.1). Then we give the translation of the expressions (Section 4.2). We terminate with the part concerning the definition of the initial state (Section 4.3), the particular value of *State* representing the situation at the beginning of the process execution. We use the EC example to illustrate our approach throughout the section. The complete model can be found in Appendix 6.2.

In the following $E_1: T_1, \dots, E_n: T_n$ are the participants of the business process, $Class_1, \dots, Class_m$ are all the entity classes introduced by the static view (i.e., those stereotyped by `<<object>>`, `<<worker>>` or `<<system>>`), and $Datatype_1, \dots, Datatype_h$ are all datatypes included in the static view.

4.1 State Definition

As mentioned earlier, the values of type *State* represent all possible states of the process participants during the process execution. *State* is defined by the list of type and function declarations shown in Fig. 5(a). The first n components of *State* are used to record the associations between the names of the participants (E_1, \dots, E_n) and the CPN ML value identifying them; whereas, given *Class* a class, then *classes*: *CompType*(*Class*) is the component of *State* recording all existing instances (objects) of the class *Class* with their current states. Function *CompType* returns the proper types for the various components of *State*. *Comp* generates all the functions and type declarations needed to handle the *State* component corresponding either to a process participant or to all the instances of a class, whereas *Decls* generates the data structures and the relative functions needed to represent a class/datatype.

We give below the definition of *State* in the case of the EC example.

```
colset State = record
  CLIENT : ClientID *      EC : ECommerceID *      WH : WarehouseID *
  CARRIER : CarrierID *   CC : CreditCardID *      PP : PaypalID *
  ORDER : OrderID *       PACK : PackageID *       ANS : BOOL *
  RES : BOOL *            clients : Clients *       eCommerces : ECommerces *
  warehouses : Warehouses * carriers : Carriers *   creditCards : CreditCards *
  paypals : Paypals *      orders : Orders *       packages : Packages;
```

Function *Decls* (defined in Fig. 5(b) and 5(c)) transforms a class/datatype present in the static view into the set of CPN ML type and function declarations needed to

```

Decls(Datatype1) ... Decls(Datatypeh); Decls(Class1) ... Decls(Classm)
Comp(E1: T1) ... Comp(En: Tn); Comp(Class1) ... Comp(Classm)
colset State = record
  E1: CompType(T1) * ... * En: CompType(Tn)
  class1s: CompType(Class1) * ... * classms: CompType(Classm); ;

```

(a) State translation

```

let att1: T1, ..., attk: Tk be the attributes of Class
colset ClassID = int;
if Class is stereotyped by <<object>> then
colset ClassState = record att1: TrType(T1) * ... * attk: TrType(Tk);
otherwise
colset ClassControl = with s1 | ... | sh;
colset ClassState = record att1: TrType(T1) * ... * attk: TrType(Tk) * control: ClassControl;
where s1, ..., sh are the states of the state machine associated with Class

```

(b) Definition of *Decls*(Class)

```

let att1: T1, ..., attk: Tk be the attributes of Datatype
colset DatatypeVal = record att1: TrType(T1) * ... * attk: TrType(Tk);
for any op(T1, ..., Tn): T operation of Datatype
op: TrType(T1) * ... * TrType(Tn) → TrType(T)
these operations must be defined by looking at the associated methods in the static view

```

(c) Definition of *Decls*(Datatype)

```

fun setE: State × TrType(T) → State

```

(d) Definition of *Comp*(E: T)

```

colset Classes = list product ClassID * ClassState;
upClass: Classes * ClassID * ClassState → Classes
getClass: Classes * ClassID → ClassState
for any op(T1, ..., Tn) operation of Class
op: State * ClassID * TrType(T1) * ... * TrType(Tn) → State
for any op(T1, ..., Tn): T operation of Class not marked by <<aux>>
op: State * ClassID * TrType(T1) * ... * TrType(Tn) → (State * TrType(T))
for any op(T1, ..., Tn): T operation of Class marked by <<aux>>
op: State * ClassID * TrType(T1) * ... * TrType(Tn) → TrType(T)

```

(e) Definition of *Comp*(Class)

Fig. 5 Translation of the static view

represent its values and to handle them. The values of a datatype *Datatype* are represented by the type *DatatypeVal*, i.e., a record having a component for each attribute of *Datatype*. A class *Class* determines a set of objects having an identity, typed by *ClassID*, and a local state typed by *ClassState*. The local state is a record having a component for each attribute of *Class* and, in the case of active objects and extra

component corresponding to the control state, typed by `ClassControl`, and defined by the state machine associated with `Class`.

In the EC example, the `WarehouseState` is defined as follows:

colset `WarehouseState` = **record** `control`: `WarehouseControl`;

As all identifiers, the `WarehouseID` is an integer: **colset** `WarehouseID` = `int`;

And the `WarehouseControl` is an enumerated type with (in this case) only one value: `WarehouseControl` = **with** `Warehouse0`;

Function *Comp* (defined in Fig. 5(d) and 5(e)) transforms a process participant declaration (resp. a class) in the static view into a set of the type and function declarations needed to define and handle component State recording the participant state (resp. the states) of all class instances. The set of the states of the instances/objects of a class is realized by a list of pairs, made of an object identity and an object state.

For example, type `Warehouses` is defined as a list of pairs of `WarehouseID` and `WarehouseState`: **colset** `Warehouses` = **list product** `WarehouseID` * `WarehouseState`.

The function corresponding to an operation `op` of a class in the static view is defined by looking either at the method associated with `op` in the static view, in case of business object classes and of `<<aux>>` operation of workers and system classes, whereas for the other operations of the workers and system classes they are defined using the state machines associated with that class. By looking at the state machine transitions, it will be possible to know how these operation calls modify the attribute values and the control state. In particular, our mechanism defines functions set to set a value inside a record (e.g., “`State.set.CLIENT s id`” sets field `CLIENT` to `id` in state `s`), as well as functions to get a value from the record, and to update it. The definitions of these set, get and upd functions are omitted here; their definition for the EC example can be found in [ACR13].

Finally, the *TrType* function translates a UML type into its corresponding CPN ML type. This function is defined by cases below.

- *TrType*(string) = `STRING`
- *TrType*(boolean) = `BOOL`
- *TrType*(integer) = `int`
- *TrType*(Class) = `ClassID`, where `Class` is the name of a UML class appearing in the static view
- *TrType*(Datatype) = `DatatypeVal`, where `Datatype` is the name of a UML datatype appearing in the static view

4.2 Expressions

We give here the translation of the expressions of **EXP** into CPN ML expressions, since they will appear in the activity diagrams as conditions on the arcs leaving the merge nodes, as well as in the action nodes. We define below by cases the translation function *TrE*(Exp, *s*), that associates a CPN ML with an OCL expression Exp, given the current state *s*.

- $TrE(X, s) = \#X(s)$, if X is a participant of the process, ($\#X$ is the CPN ML operation selecting a record type component), e.g. CLIENT is translated to $\#CLIENT(s)$;
- $TrE(C, s) = C$, if C is a primitive data type constant;
- $TrE(op(Exp_1, \dots, Exp_n), s) = op'(TrE(Exp_1, s), \dots, TrE(Exp_n, s))$, if op is an operation of a primitive type, op' will be either op itself or it will be defined case by case in case of name mismatch between the operations on the UML primitive types and the corresponding ones of CPN ML;
- $TrE(op(Exp_1, \dots, Exp_n), s) = op(TrE(Exp_1, s), \dots, TrE(Exp_n, s))$, if op is an operation of a datatype defined in the static view;
- $TrE(Exp.op(Exp_1, \dots, Exp_n), s) = op(s, TrE(Exp, s), TrE(Exp_1, s), \dots, TrE(Exp_n, s))$, if op is an operation of a class defined in the static view of kind query.

For example, the translation of the guard $[RES = true]$ in Fig. 2 using function TrE results in the CPN ML expression $[\#RES(s) = true]$. And the OCL expression $CARRIER.deliver(PACK)$ is translated to $deliver(s, \#CARRIER(s), \#PACK(s))$.

4.3 Initial Process Execution State

In order to translate a business process into CPNs, and specifically define the initial execution state of the process itself, we also need a specific list of individual participants. Recall that the names in the participant list part of the process model are roles, not specific individuals.

If n is the number of participants and data not marked by $\langle\langle out \rangle\rangle$, we call a *business process instantiation* a list of n ground OCL expressions defined using the data type defined in the static view, and the constructors of the classes in the static view itself (operations stereotyped by $\langle\langle create \rangle\rangle$).

Given the business process instantiation (i.e., a list of ground expressions G_1, \dots, G_n), the function *Initialize* returns the CPN ML expression defining the initial state, where the participants not marked by $\langle\langle out \rangle\rangle$ are initialized with the values determined by the process instantiation. The other ones are initialized with some standard default values depending on their type, and the components corresponding to the objects of the various classes just contain the states of the objects appearing in the process instantiation. Hence, we have:

val initState = *Initialize*(C_1, \dots, C_n);

Initialize is defined using TrE (details can be found in Appendix).

The standard default values are: 0 for int, false for BOOL, 0 for ClassID corresponding to the null object (since the object identities will be strictly positive integers), and nil the empty list for the list types.

val emptyState = {

$E_1 = def1, \dots, E_n = defn, class_1s = nil, \dots, class_ms = nil$ };

val initState = **let**

val (s, V_1) = $TrE(G_1, emptyState)$

val $s_1 = State.set_E_1\ s\ V_1$

```

...
val (sr1, Vr1) = TrE(Gr, sr-1)
val sr = State.set_Er sr1 Vr1
in sr end;

```

5 Conclusion and Future Work

In this work, we define precise business models, where the activity diagrams are inductively defined using a set of patterns combined in a modular way. Hence, we characterize a set of commonly used behaviors in activity diagrams. Moreover, our patterns provide the designer with guidelines, thus avoiding common modeling errors. Our second contribution is to provide the activity diagrams built using these patterns with a formal semantics using colored Petri nets, hence allowing the use of automated verification techniques.

Implementation. Following our algorithm, we implemented (manually) the EC example into the CPN Tools model checker [JK09]. This results in a CPN containing 24 places, 25 transitions and about 500 lines of CPN ML code; the detailed CPN description can be seen in Fig. 8, and the CPN Tools model is available online⁵. (One can see that, contrarily to what we stated in Section 3, a few transitions have type not UNIT but BOOL. This is a trick due to the fact that CPN Tools does not allow reading global variables in guards; hence, we first read the variables in the previous place, and then the guard checks this value.) Such an implementation allows for automated verification techniques; among the properties are for example the fact that the various final nodes may be reached in any case, and hence that the process is well-formed. Automatizing the translation process from a precise activity diagram to a CPN using model-driven methods and technologies does not raise any particular theoretical problem, and is the subject of ongoing work.

Future Works. Among directions for future research is the comparison of our semantics given in terms of CPNs where the process execution state is modeled by colored tokens, with existing (partial) semantics, such as [KH10] and [GRR10] (that has been a source of inspiration for our work). Furthermore, integrating accept and timed events to our approach is an interesting direction of research. Finally, we aim at finding the properties relevant for the business process, and providing guidelines to prove them.

Also note that the resulting CPN (including the functions) may be simplified in some cases. First, some places and transitions added by the translation may be unnecessary. This is the case, e.g., of a decision/merge pattern with only one activity on the left side, and one on the right side ($n = m = 1$). In that case, the only activity synchronizing in the merge is the left one; hence, the transition “*merge*₁” in Fig. 4(j), as well as the place below, are unnecessary. Second, some functions could

⁵ <http://lipn.univ-paris13.fr/~andre/activity-diagrams-patterns/>

be simplified for similar reasons. These simplifications, that are beyond the scope of this paper, could help to speed up the automated verification of the resulting CPN.

Acknowledgment. We wish to thank Michael Westergaard for his kind help when using CPN Tools, and anonymous reviewers for their helpful comments.

References

- ACK12. Étienne André, Christine Choppy, and Kais Klai. Formalizing non-concurrent UML state machines using colored Petri nets. *ACM SIGSOFT Software Engineering Notes*, 37(4):1–8, 2012. [2](#)
- ACR13. Étienne André, Christine Choppy, and Gianna Reggio. Activity diagrams patterns for modeling business processes (report version). Available at <http://lipn.fr/~andre/adp/>, 2013. [11](#), [15](#)
- BKO10. Dominik Birkmeier, Sebastian Kloeckner, and Sven Overhage. An empirical comparison of the usability of BPMN and UML activity diagrams for business users. In *ECIS 2010*, 2010. [2](#)
- BM07. Simona Bernardi and José Merseguer. Performance evaluation of UML design with stochastic well-formed nets. *Journal of Systems and Software*, 80(11):1843–1865, 2007. [2](#)
- Bör07. Egon Börger. Modeling workflow patterns from first principles. In *ER*, volume 4801 of *LNCS*, pages 1–20. Springer, 2007. [2](#)
- CDR⁺11. Francesco Cerbo, Gabriella Doderio, Gianna Reggio, Filippo Ricca, and Giuseppe Scanniello. Precise vs. ultra-light activity diagrams – An experimental assessment in the context of business process modelling. In *Product-Focused Software Process Improvement*, volume 6759 of *LNCS*, pages 291–305. Springer, 2011. [2](#)
- CPM06. William R. Cook, Sourabh Patwardhan, and Jayadev Misra. Workflow patterns in Orc. In *COORDINATION*, volume 4038 of *LNCS*, pages 82–96. Springer, 2006. [2](#)
- DSP11. Salvatore Distefano, Marco Scarpa, and Antonio Puliafito. From UML to Petri nets: The PCM-based methodology. *IEEE Transactions on Software Engineering*, 37(1):65–79, 2011. [2](#)
- Erl07. Thomas Erl. *SOA Principles of Service Design*. The Prentice Hall Service-Oriented Computing Series from Thomas Erl, 2007. [2](#)
- FELR98. Robert B. France, Andy Evans, Kevin Lano, and Bernhard Rumpe. Developing the UML as a formal modelling notation. In *Computer Standards and Interfaces: Special Issues on Formal Development Techniques*, pages 297–307. Springer-Verlag, 1998. [2](#)
- GHJV95. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. [6](#)
- GRR10. Hans Grönniger, Dirk Reiss, and Bernhard Rumpe. Towards a semantics of activity diagrams with semantic variation points. In *MODELS*, volume 6394 of *LNCS*, pages 331–345. Springer, 2010. [2](#), [17](#)
- JK09. Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009. [2](#), [9](#), [17](#)
- KH09. Frank Alexander Kraemer and Peter Herrmann. Automated encapsulation of UML activities for incremental development and verification. In Andy Schürr and Bran Selic, editors, *MoDELS*, volume 5795 of *LNCS*, pages 571–585. Springer, 2009. [3](#)
- KH10. Frank Alexander Kraemer and Peter Herrmann. Reactive semantics for distributed UML activities. In *FMOODS/FORTE*, volume 6117 of *LNCS*, pages 17–31. Springer, 2010. [3](#), [17](#)

- KT10. Fabrice Kordon and Yann Thierry-Mieg. Experiences in model driven verification of behavior with UML. In *Monterey Workshop*, volume 6028 of *LNCS*, pages 181–200. Springer, 2010. 2
- MGT09. Ahmed Mekki, Mohamed Ghazel, and Armand Toguyeni. Validating time-constrained systems using UML statecharts patterns and timed automata observers. In *VECoS*, pages 112–124. British Computer Society, 2009. 3
- PBA⁺08. Daniela Cascini Peixoto, Vitor Alcântara Batista, Ana Paula Atayde, Eduardo Borges Pereira, Rodolfo Ferreira Resende, and Clarindo Isaíá Pádua. A comparison of BPMN and UML 2.0 activity diagrams. In *Simpósio Brasileiro de Qualidade de Software*, 2008. Available at <http://homepages.dcc.ufmg.br/~cascini/>. 2
- RLR11. Gianna Reggio, Maurizio Leotta, and Filippo Ricca. Precise is better than light: A document analysis study about quality of business process models. In *First International Workshop on Empirical Requirements Engineering (EmpiRE)*, pages 61–68, 2011. 6
- RRS⁺11. Gianna Reggio, Filippo Ricca, Giuseppe Scanniello, Francesco Cerbo, and Gabriella Doderio. A precise style for business process modelling: Results from two controlled experiments. In *Model Driven Engineering Languages and Systems*, volume 6981 of *LNCS*, pages 138–152. Springer, 2011. 2, 3
- UML. OMG unified language superstructure specification(formal). version 2.4.1, 2011-08-06. <http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/>. 2, 5
- Wor. Workflow Patterns Initiative. Workflow patterns home page. <http://www.workflowpatterns.com>. 2, 7, 8
- ZL10. Shao Jie Zhang and Yang Liu. An automatic approach to model checking UML state machines. In *SSIRI (Companion)*, pages 1–6. IEEE Computer Society, 2010. 2

6 Appendix

6.1 Full Translation of the Static View

We give here the full translation of the precise model of the business process EC, introduced in Fig. 1 and 2, instantiated as follows.

CLIENT \rightarrow mkClient(false, mkName("X"))

(a client named X using the credit card to pay)

EC \rightarrow mkECommerce() (it has no registered clients)

WH \rightarrow mkWarehouse()

CARRIER \rightarrow mkCarrier()

CC \rightarrow mkCreditCard()

PP \rightarrow mkPaypal()

PACK, ORDER, ANS, and RES are not considered by the process instantiation, since they are marked by $\langle\langle\text{out}\rangle\rangle$, that means that they will be created during the process execution.

6.2 Translation of the Static View and the Participant List

Here, we present the result of the translation of the static view and of the participant lists of the precise model of the EC business process introduced in Section 2, that amounts to the definition of the data structure State together with the definitions of all needed data structures and of all the functions acting over it. We use here the CPN ML syntax.

We often add the signature of the various functions in italics. Indeed, although they are not part of the CPN ML language, they improve readability.

Due to the use of lists, we recall classical notations for list. The empty list is denoted by nil, whereas the concatenation of an element e at the beginning of a list l is denoted by $e :: l$.

colset State = record			
State			
CLIENT : ClientID	*	EC : ECommerceID	*
WH : WarehouseID	*	CARRIER : CarrierID	*
CC : CreditCardID	*	PP : PaypalID	*
ORDER : OrderID	*	PACK : PackageID	*
ANS : BOOL	*	RES : BOOL	*
clients : Clients	*	eCommerces : ECommerces	*
warehouses : Warehouses	*	carriers : Carriers	*
creditCards : CreditCards	*	paypals : Paypals	*
orders : Orders	*	packages : Packages;	*

Clients

colset Clients = **list product** ClientID * ClientState
*upClient: Clients * ClientID * ClientState → Clients*
This function updates the state of a client in an element of type Clients.

```
fun upClient(nil,ci,cs) = (ci,cs)::nil
| upClient((ci',cs')::cls,ci,cs) =
  if ci'=ci then
    (ci,cs)::cls
  else
    (ci',cs')::upClient(cls,ci,cs);
```

*getClient: Clients * ClientID → ClientState*
This function gets the state of a client from an element of type Clients; it is assumed that there is a state associated with the passed ClientID.
fun getClient((ci',cs')::cls,ci) =
if ci'=ci **then** cs' **else** getClient(cls,ci);

ECommerces

colset ECommerces =
list product ECommerceID * ECommerceState
upECommerce:
*ECommerces * ECommerceID * ECommerceState →*
ECommerces
fun upECommerce(nil,ei,es) = (ei,es)::nil
| upECommerce((ei',es')::ecs,ei,es) =
if ei'=ei **then**
 (ei,es)::ecs
else
 (ei',es')::upECommerce(ecs,ei,es);

getECommerce:
*ECommerces * ECommerceID → ECommerceState*
fun getECommerce((ei',es')::ecs,ei) =
if ei'=ei **then** es' **else** getECommerce(ecs,ei);

Packages

colset Packages =
list product PackageID * PackageState
upPackage:
*Packages * PackageID * PackageState → Packages*
fun upPackage(nil,pi,ps) = (pi,ps)::nil
| upPackage((pi',ps')::pks,pi,ps) =
if pi'=pi **then**
 (pi,ps)::pks
else
 (pi',ps')::upPackage(pks,pi,ps);

```

getPackage: Packages * PackageID → PackageState
fun getPackage((pi',ps')::pks,pi) =
if pi'=pi then ps' else getPackage(pks,pi);

```

Paypals

```

colset Paypals =
list product PayPalID * PaypalState
upPaypal:
Paypals * PayPalID * PaypalState → Paypals
fun upPaypal(nil,pi,ps) = (pi,ps)::nil
| upPaypal((pi',ps')::pps,pi,ps) =
    if pi'=pi then
        (pi,ps)::pps
    else
        (pi',ps')::upPaypal(pps,pi,ps);

```

```

getPaypal: Paypals * PayPalID → PaypalState
fun getPaypal((pi',ps')::pps,pi) =
if pi'=pi then ps' else getPaypal(pps,pi);

```

Warehouses

```

colset Warehouses =
list product WarehouseID * WarehouseState
upWarehouse:
Warehouses * WarehouseID * WarehouseState →
    Warehouses
fun upWarehouse(nil,wi,ws) = (wi,ws)::nil
| upWarehouse((wi',ws')::whs,wi,ws) =
    if wi'=wi then
        (wi,ws)::whs
    else
        (wi',ws')::upWarehouse(whs,wi,ws);

```

```

getWarehouse:
    Warehouses * WarehouseID → WarehouseState
fun getWarehouse((wi',ws')::whs,wi) =
if wi'=wi then ws' else getWarehouse(whs,wi);

```

Carriers

```

colset Carriers =
list product CarrierID * CarrierState
upCarrier: Carriers * CarrierID * CarrierState → Carriers
fun upCarrier(nil,ci,cs) = (ci,cs)::nil
| upCarrier((ci',cs')::cas,ci,cs) =
    if ci'=ci then
        (ci,cs)::cas

```

```

    else
      (ci',cs')::upCarrier(cas,ci,cs);

  getCarrier: Carriers * CarrierID → CarrierState
  fun getWarehouse((ci',cs')::cas,ci) =
  if ci'=ci then cs' else getWarehouse(cas,ci);

                                CreditCards

  colset CreditCards =
  list product CreditCardID * CreditCardState
  upCreditCard:
  CreditCards * CreditCardID * CreditCardState → CreditCards
  fun upCreditCard(nil,ci,cs) = (ci,cs)::nil
  | upCreditCard((ci',cs')::ccs,ci,cs) =
    if ci'=ci then
      (ci,cs)::ccs
    else
      (ci',cs')::upCreditCard(ccs,ci,cs);

  getCreditCard: CreditCards * CreditCardID → CreditCardState
  fun getCreditCard((ci',cs')::ccs,ci) =
  if ci'=ci then cs' else getCreditCard(ccs,ci);

                                Orders

  colset Orders =
  list product OrderID * OrderState
  upOrder: Orders * OrderID * OrderState → Orders
  fun upOrder(nil,oi,os) = (oi,os)::nil
  | upOrder((oi',os')::ors,oi,os) =
    if oi'=oi then
      (oi,os)::ors
    else
      (oi',os')::upOrder(ors,oi,os);

  getOrder: Orders * OrderID → OrderState
  fun getOrder((oi',os')::ors,oi) =
  if oi'=oi then os' else getOrder(ors,oi);

```

The empty state is a state where the various components are set with the default value of their types. That is for example false for booleans, 0 for int, 0 for class ID (the identities of the existing objects will be integer greater than 0), nil for lists, etc.

```

val emptyState = {

```

```

    CLIENT = 0          EC = 0
    WH = 0             CARRIER = 0
    CC = 0             PP = 0
    ORDER = 0          PACK = 0
    ANS = false        RES = false
    clients = nil      eCommerces = nil
    warehouses = nil    carriers = nil
    creditCards = nil   paypals = nil
    orders = nil        packages = nil
  };

val initState = let
  val (s1,ppi) = mkPaypal(emptyState)
  val (s2,cci) = mkCreditCard(State.set_PP s1 ppi )
  val (s3,ci) = mkCarrier(State.set_CC s2 cci )
  val (s4,wi) = mkWarehouse(State.set_CARRIER s3 ci )
  val (s5,ei) = mkECommerce(State.set_WH s4 wi )
  val (s6,cli) =
    mkClient(State.set_EC s5 ei ,false,mkName("X"))
in State.set.CLIENT s6 cli end;

```

In the following we report all the definitions of the types with the relative functions used in the definition of State.

Name

```

colset NameVal = record cont: STRING;
mkName: STRING → NameVal - - constructor
fun mkName(str) = {cont=str};

```

Order

```

colset OrderID = int;
colset OrderState =
record orderer: NameVal * archived: BOOL;

mkOrder: State * NameVal → (State * OrderID)
- - constructor
fun mkOrder(s,n) =
  (State.set_Orders s
   upOrder(#orders(s),1,{orderer=n,archived=false},1);

```

Since we have only a unique order, we assume that its identity is 1. In the general case, we have to add to the state an extra component to get the first unused identity (recall that instances identities are numbers).

```

orderer: State * OrderID → NameVal
fun orderer(s,oi) = #orderer(getOrder(#orders(s),oi))

```

Recall from the CPN ML syntax that #orderer is the selector operation of records, returning the component name orderer.

```

archive: State * OrderID → State

```

```

fun archive(s,oi) =
let val n = #orderer(s,oi) in
  State.set_Orders s
    upOrder(#orders(s),oi,{orderer=n,archived=true}) end

```

Client

```

colset ClientID = int;
colset ClientState = record
  usingPaypal: BOOL * name: NameVal * control: ClientControl;
colset ClientControl = with Client0;

```

```

mkClient: State * BOOL * NameVal →
  (State * ClientID) - - constructor
fun mkClient(s,payp,n) =
  (State.set_Clients s,
    upClient(#clients(s),1,
      {usingPaypal=pp,name=n,control=Client0}),1)

```

```

sendOrder: State * ClientID * ECommerceID →
  (State * OrderID)
fun sendOrder(s,ci,ei) = mkOrder(s,name(s,ci))

```

```

name: State * ClientID → NameVal
fun name(s,ci) = #name(getClient(#clients(s,ci))

```

```

usingPaypal: State * ClientID → BOOL
fun usingPaypal(s,ci) =
  #usingPaypal(getClient(#clients(s,ci))

```

```

answers: State * ClientID → (State * BOOL)
fun answers(s,ci) = (s,true)
  Here, we assume that this client answers yes in any case.

```

ECommerce

```

colset ECommerceID = int;
colset Names = list NameVal;
colset ECommerceState =
record cList: Names * control: ECommerceControl;
colset ECommerceControl = with ECommerce0;

```

```

mkECommerce: State →
  (State * ECommerceID) - - constructor
fun mkECommerce(s) =
  (State.set_ECommerces s,

```

```

upECommerce(#eCommerces(s),
  {clients= nil,control= ECommerce0},1,1)

```

```

checkRegistered: State * ECommerceID * NameVal →
  (State * BOOL)

```

```

fun checkRegistered(s,ei,n) = (s,mem(cList(s,ei),n))

```

Note that $\text{mem}(l,e)$ is a standard function from CPN ML that checks whether element e belongs to list l .

```

cList: State * ECommerceID → Names

```

```

fun cList(s,ei) = #cList(getClient(#clients(s,ei))

```

```

proposeRegistration: State * ECommerceID * ClientID →
  State

```

The definition of this operation depends on the state machine associated with the ECommerce class; here, we assume that it has the form of a perfect daisy.

```

fun proposeRegistration(s,ei,ci) = s

```

```

register: State * ECommerceID * NameVal → State

```

```

fun register(s,ei,n) =

```

```

let val oldEs = getECommerce(#eCommerces(s),ei) in

```

```

  State.set_ECommerces s

```

```

  upECommerce(#eCommerces(s),

```

```

    {cList= append(#cList(oldEs),n),

```

```

    control= #control(oldEs)},ei) end

```

Again, the definition of this operation depends on the state machine associated with the ECommerce class; here, we assume that it has the form of a perfect daisy.

```

archiveOrder: State * ECommerceID * OrderID → State

```

```

fun archiveOrder(s,ei,oi) =

```

```

let val n = orderer(s,oi) in

```

```

  State.set_Orders s

```

```

  upOrder(#orders(s),oi,

```

```

    {orderer=n,archived=true}) end;

```

Again, the definition of this operation depends on the state machine associated with the ECommerce class; here we assume that it has the form of a perfect daisy; however, to comply with the post condition, the state of the order is changed.

Warehouse

```

colset WarehouseID = int;

```

```

colset WarehouseState = record control: WarehouseControl;

```

```

WarehouseControl = with Warehouse0;

```

*mkWarehouse: State → (State * WarehouseID)*
-- constructor

```
fun mkWarehouse(s) =
(State.set_Warehouses s
  upWarehouse(#warehouses(s),
    {control= Warehouse0},1,1)
```

*prepare: State * WarehouseID * OrderID →*
*(State * PackageID)*

Again, the definition of this operation depends on the state machine associated with the Warehouse class.

```
fun prepare(s,wi,oi) = mkPackage(s)
```

Carrier

```
colset CarrierID = int;
colset CarrierState = record control: CarrierControl;
colset CarrierControl = with Carrier0 ;
```

*mkCarrier: State → (State * CarrierID)* *-- constructor*

```
fun mkCarrier(s) =
(State.set_Carriers s
  upCarrier(#carriers(s),{control= Carrier0},1,1)
```

*deliver: State * CarrierID * PackageID → State*

```
fun deliver(s,ci,pi) = s
```

The definition of this operation depends on the state machine associated with the Carrier class.

confirmDelivered:

*State * CarrierID * PackageID * ECommerceID → State*

```
fun confirmDelivered(s,ci,pi) = s
```

The definition of this operation depends on the state machine associated with the Carrier class.

CreditCard

```
colset CreditCardID = int;
colset CreditCardState = record control: CreditCardControl;
colset CreditCardControl = with CreditCard0;
```

*mkCreditCard: State → (State * CreditCardID)*

-- constructor

```
fun mkCreditCard(s) =
(State.set_CreditCards s
  upCreditCard(#creditCards(s),
    {control= CreditCard0},1,1)
```

*pay: State * CreditCardID * OrderID → State*

fun pay(s,ci,oi) = s

The definition of this operation depends on the state machine associated with the CreditCard class.

Paypal

colset PaypalID = int;

colset PaypalState = **record** control: PaypalControl;

colset PaypalControl = **with** Paypal0;

*mkPaypal: State → (State * PaypalID) - - constructor*

fun mkPaypal(s) =

(State.set_Paypals s

upPaypal(#paypals(s),{control= Paypal0},1,1)

*pay: State * PaypalID * OrderID → State*

fun pay(s,ppi,oi) = s

The definition of this operation depends on the state machine associated with the Paypal class.

Package

colset PackageID = int;

colset PackageState = **record** control: PackageControl;

colset PackageControl = **with** Package0;

*mkPackage: State → (State * PackageID) - - constructor*

fun mkPackage(s) =

(State.set_Packages s

upPackage(#packages(s),{control=Package0},1,1)

6.3 Full Translation of the Activity Diagram

The CPN resulting from the translation of the activity diagram modeling the behavior of the business process EC is shown in Fig. 6 and 7.

We give in Fig. 8 the whole CPN model as in CPN Tools.

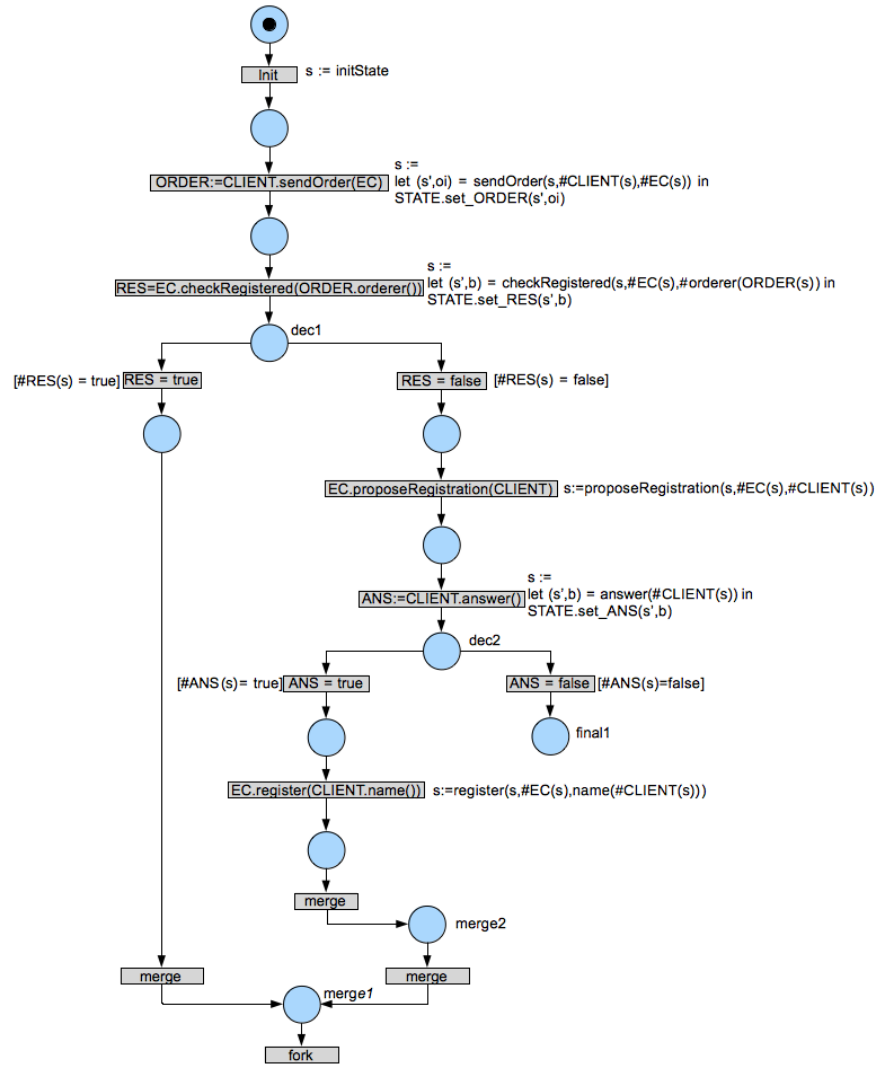


Fig. 6 Resulting colored Petri net (part 1)

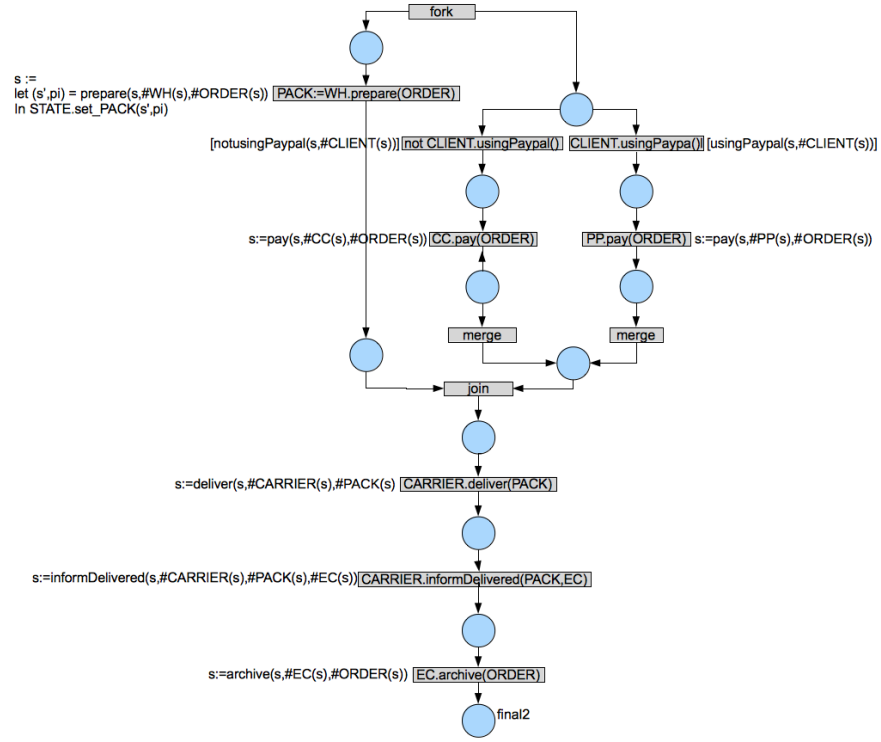


Fig. 7 Resulting colored Petri net (part 2)

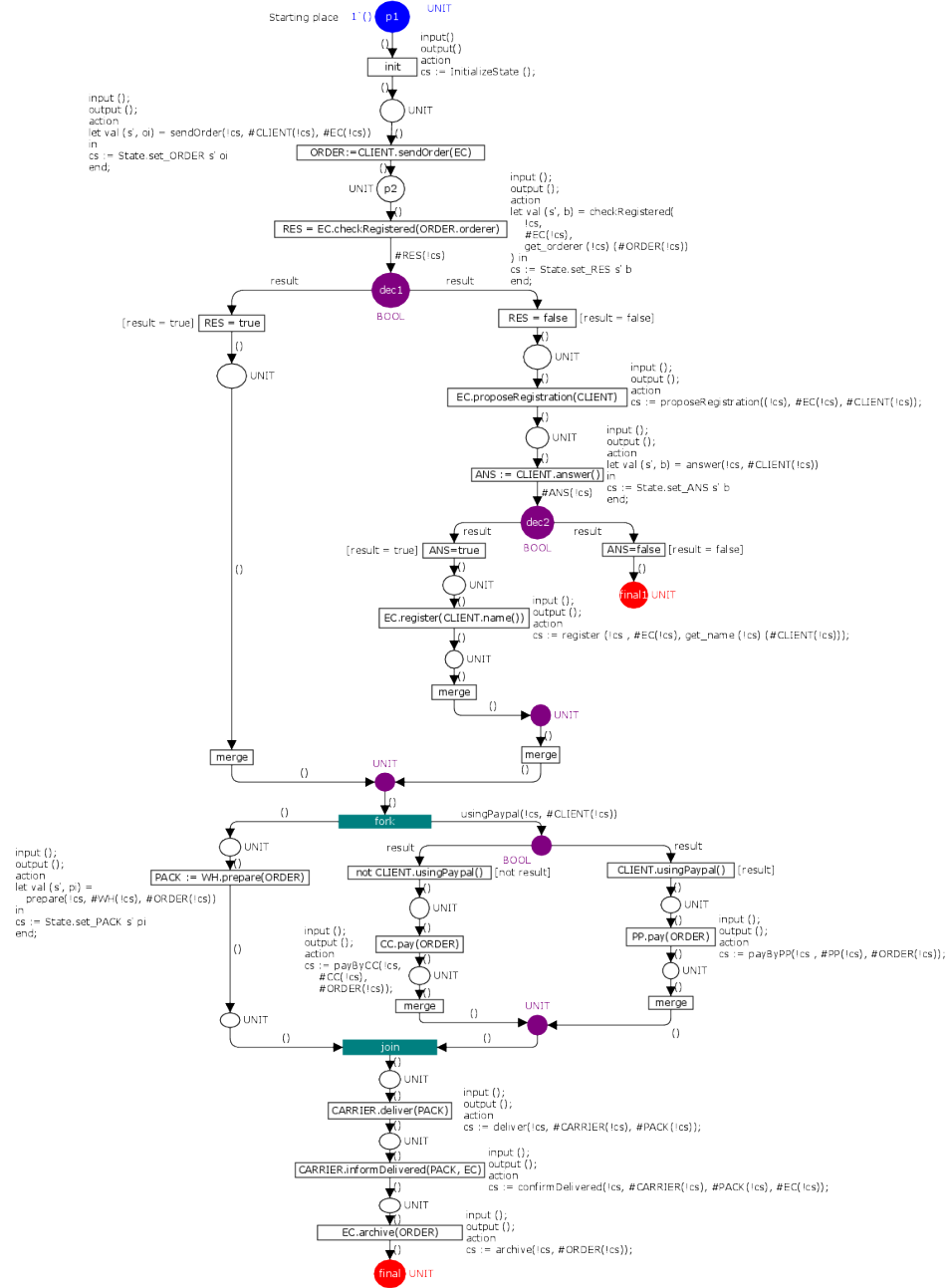


Fig. 8 Resulting colored Petri net (exported from CPN Tools)