Buletinul Stiintific al Universitatii "Politehnica" din Timisoara, ROMANIA
Seria AUTOMATICA si CALCULATOARE
PERIODICA POLITECHNICA, Transactions on AUTOMATIC CONTROL and COMPUTER SCIENCE
Vol.47 (61), 2002, ISSN 1224-600X

Business Process Modeling in INSPIRE Using Petri Nets

Costin Badica[*], Marius Brezovan[**] and Chris Fox[***]

[*] Department of Computer Science, King's College London, Strand, WC2R 2LS, London, UK
E-Mail: badica@dcs.kcl.ac.uk

[**] Faculty of Automatics, Computers and Electronics, University of Craiova, str.Lapus, nr.5, RO-1100, Craiova, Romania
E-Mail: Marius.Brezovan@aeic.ro

[***] Department of Computer Science, University of Essex, CO4 3SQ, Colchester, UK
E-Mail: foxcj@essex.ac.uk

***Abstract** – This paper introduces a notation for business process modeling and shows how it can be formally interpreted in terms of Petri nets. Petri nets have a quite respectable research community, which is 35 years old. However, they were only recently proposed for business process modeling. This is probably due to the fact they are often claimed to be "too complex" for this task. Nevertheless, they are quite well understood and the theory behind them is well developed, so we think they have a good potential for business process modeling, but more work needs to be done. In this paper we show that Petri nets can help in understanding formally the business process modeling notation developed in the Inspire project. This understanding can act as a basis for a future work on formal analysis of business process models developed with the INSPIRE tool. The Inspire project (IST-10387-1999) aims to develop an integrated tool-set to support a systematic and more human-oriented approach to business process re-engineering.*

***Keywords:** business process, Petri net, formal specification.*

I. INTRODUCTION

An increased interest in applying information technology to the field of business processes has been manifested during the last decade, both in research and commercial communities, as a response to slogans like business process re-engineering or total quality management. This is proven by the large number of papers published on the subject, the large number of emerging standards and proposals for representing different aspects of business processes and the large number of software tools that support tasks like business process modeling, design, analysis or simulation.

Informally, by a *business process* we mean a process that is carried out in an organization in order to achieve the organization business objectives. Because organizations are very complex artifacts, it has been claimed that carefully developed models are necessary for describing, analyzing and/or enacting the underlying business processes ([1]). A business process model is usually expressed by means of a graphical notation that must be able to capture all the relevant information from the model and additionally must facilitate the analysis of the modeled process by static verification and/or dynamic simulation.

Analysis of business process models is very important in the context of *business process re-engineering* (BPR hereafter), because the task of BPR is to evaluate the current processes with the goal of radically revising them, in order to accommodate their improvement to new organizational needs or goals.

The INSPIRE project (IST-10387-1999) aims to develop an integrated tool-set to support a systematic and more human-oriented approach to BPR. A central aspect in INSPIRE was the development of a notation for representing business process models that is both easy to use and understand by the project partners (consultants and developers; note that consultants are not usually IT experts) and also sufficiently rigorous to allow static verification and dynamic simulation -- essential tasks within a BPR context ([2]).

The notation of business process models employed in INSPIRE combines features of IDEF0 ([3]) and IDEF3 ([4]). We started with an IDEF0-based notation and then enhanced it with facilities for representing the dynamics of a business process, based on the process schematics employed in IDEF3 ([4]).

This paper introduces the Inspire notation and shows how this notation can be formally understood by mapping it to a special class of Petri nets - Place/Transition nets (P/T nets hereafter). Other techniques proposed in the literature for formally understanding business process models are based on flownomial expressions ([5]), process algebras ([6]) and knowledge-based systems ([7]). P/T nets have already been used for workflow modeling and verification ([8]). Our

work can be seen as an attempt to generalize the results from [8] which are restricted to workflows modeled as single-input/single–output systems.

The paper is structured as follows: section II gives an informal introduction of the notation and the rationale behind it; section III shows how we can describe our models in a formal way; section IV describes the mapping of the INSPIRE notation to P/T nets; section V concludes the paper.

II. A NOTATION FOR BUSINESS PROCESSES

A. *The Basic Black Box Model*

IDEF0 is a technique used to produce a function model of a new or existing system or subject area. The result of applying the IDEF0 technique is an IDEF0 model of the system. An important assumption stated in the IDEF0 standard ([3]) is: only that which is explicitly stated is necessarily implied. As a corollary, what is not (explicitly) prohibited is (implicitly) allowed. This shows that starting from an IDEF0-based notation and extending it in a consistent way is a correct approach. In our case, the extension is based on IDEF3 ([4]).

The modeling elements of IDEF0 are (i) *boxes* and (ii) *arrows*. *Boxes* represent functions defined as activities, processes or transformations and *arrows* represent data or objects related to functions. A *box* describes what happens in a designated function. A *box* has a set of inputs $i_1, \ldots, i_m$ and a set of outputs $o_1, \ldots, o_n$. We consider controls as a special kind of inputs. Mechanisms ([3]) are not considered in this paper, although the notation employed by the INSPIRE tool supports them. In INSPIRE we are using the term *flow* instead of *arrow*, thus highlighting the data-flow flavor of IDEF0.

The crucial thing is, how we are to interpret these boxes? The IDEF0 standard is very vague with respect to this. Basically it says that in order to produce any subset of the outputs $o_1, \ldots, o_n$ any subset of the entries $i_1, \ldots, i_m$ may be required. In order to understand this statement, we must first recognize that one intuitive interpretation in mathematical terms of an IDEF0 box is a function taking *m* inputs and producing *n* outputs. This function actually describes how we can compute the outputs on the basis on the given inputs. However, the "how" component is not explicitly modeled. Eventually, it is just suggested by the verb phrase that names the box.

Let us now return to the basic interpretation as stated in the IDEF0 standard ([3]). If our boxes are always modeling functions, it means that all the outputs will be produced if all the inputs are present. This turns out to be a very restrictive assumption, especially if the IDEF0 method is used in the early stage of requirements capture and specification. Obviously, we would not like to be very restrictive during this stage. Moreover, we would like to be able to model situations when even not all the inputs are present, at least some outputs will be produced. If we abstract away from the actual domains of values for the inputs and outputs and from how the outputs are actually produced, we finally arrive at an interpretation of boxes as Boolean relations taking *m* inputs and producing *n* outputs. So formally, a box should be interpreted as an unspecified relation $rel \subseteq \{0,1\}^m \times \{0,1\}^n$.

So, our boxes are in fact black boxes describing the dependency of subsets of outputs on subsets of inputs. A dependency is a pair of Boolean strings that just says that it is possible for the underlying activity modeled by the box to produce the specified outputs when the specified inputs are available. For example, the pair (011,100) says that it is possible to produce only output 1 if only inputs 2 and 3 are present.

As for the arrows, they should be interpreted as flows transporting data or object items from producers to consumers. However, special attention must be paid to branching arrows (forks and joins).

According to [3], a fork is an arrow from source to use that is divided into two or more arrows. Because the IDEF0 standard is too vague here, we make the following assumption: a fork indicates the fact that the item placed onto the arrow from source to use will be made available to all of its destinations. In this case all the arrows will have the same label. If the fork is not intended to model this situation, than it must be replaced with a single input/multiple outputs activity. We shall see in the next section that forks are just a form of syntactic sugar of the notation. They are not really needed for the theoretical investigation of the notation.

Also, [3] states that a join is a point where two or more arrows from source to use are merged into a single arrow. Merging is different from splitting because it is difficult to imagine that it could happen outside an activity. That is why we have chosen to model joins as "dummy" multiple inputs/single output activities.

For example, in a manufacturing company we find a business process for material acquisition. It takes material requests and produces purchase orders and payment authorizations. It contains a sub-process for handling the material requests that takes material requests and produces validated requests. The company has a list of available suppliers, but is must be prepared to find and handle new potential suppliers. So, there is an additional input to handle new supplier requirements and an additional output to produce new supplier packages. The process is represented at an abstract level in figure 1. The IDEF0 technique allows the presentation of a model at different levels of detail. The process in figure 1 is detailed in figure 2. Figures 1 and 2 are also called *IDEF0 diagrams* ([3]).

B. Adding Glass Box Views

Sometimes it is useful to be able to constrain the dependency of the outputs from the inputs. One way of doing this is by attaching to each black box a glass-box view based on IDEF3. IDEF3 is a technique used to produce a dynamic model of the system. IDEF3 can produce two views of the system: a process-centered view and an object-centered view. Here we are considering only the process-centered view. The main building blocks of the process-centered view of IDEF3 are (i) *units of behaviors*, (ii) *links* and (iii) *junctions*. *Units of behavior* represent types of happenings (events, acts, etc.), *links* represent temporal relations between units of behavior, and *junctions* are used to specify the logic of process branching. Within the INSPIRE tool we are using the term *connector* instead of *junction*, so we shall use this term hereafter.

Fig. 1. A business process for processing material requests

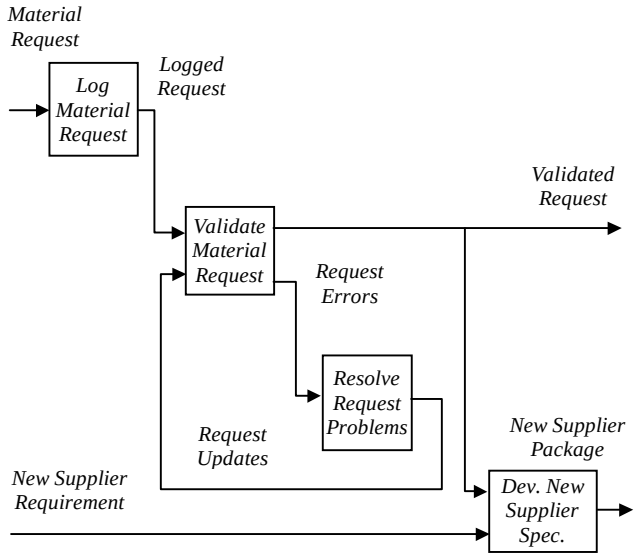

Fig. 2. A more detailed presentation of the process in figure 1

A glass-box view contains a unit of behavior, a tree of input connectors and a tree of output connectors. There are one-to-one mappings between the inputs of a black box and the leaves of its input tree and between the outputs of a black box and the leaves of its output tree. The unit of behavior models the ‘‘instantiation” of the activity. The input tree models the logic of selecting the inputs participating in the activity, and the output tree models the logic of generating the outputs produced by the activity.

Let us consider the business process in figure 2. We can associate glass-box views to activities *Validate Material Request* and *Resolve Request Problems*. *Validate Material Request* may take a *Logged Request* or *Request Updates* and may produce either a *Validated Request* or *Request Errors*. *Dev New Supplier Spec* takes both a *Validated Request* and a *New Supplier Requirement* to be able to produce a *New Supplier Package*. This is modeled in figure 3.

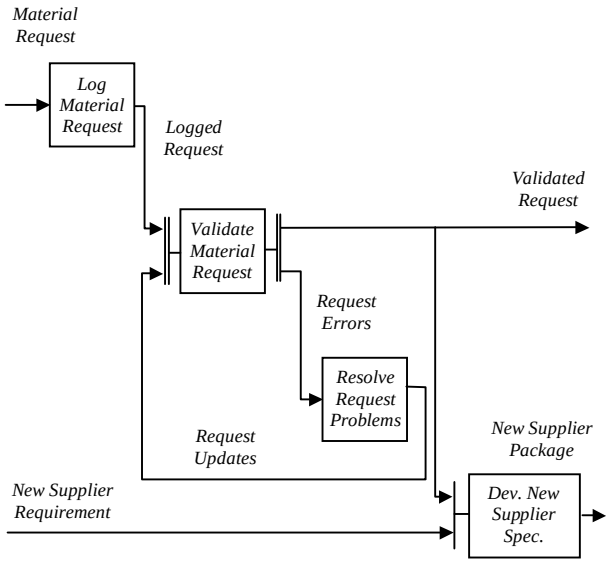

Fig. 3. The result of adding glass-box views to the process in figure 2

It is important to notice that the introduction of glass box views can simplify the IDEF0 level of the notation by eliminating forks. For example, the fork on the *Validated Request* arrow can be incorporated into the tree of output connectors of activity *Validate Material Request* by adding an output *and* connector. In this way this activity will have the representation in figure 4. Notice the presence of the new output *Validated Request*’.

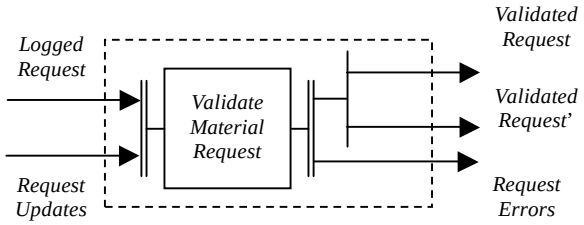

Fig. 4. Incorporating a fork into a tree of output connectors

If a fork appears on an input interface flow on a diagram, it will be replaced with a single input/multiple output dummy activity with the tree of output connectors consisting of a single output *and* connector.

III. FORMALIZING THE INSPIRE NOTATION

In order to describe the mapping of the INSPIRE notation to P/T nets we have first to describe the INSPIRE notation in a precise and unambiguous way.

Let *AN* be a set of activity names and let *FN* be a set of flow names.

A *tree of input connectors t* together with its set *Leaves*(*t*) of *leaves* is defined as follows:
i) if $f \in FN$ then *f* is a tree of input connectors and *leaves* $(t) = \{f\}$;
ii) if *Ts* with $|Ts| \geq 2$[1] if a set of tree of input connectors then *iconn*(*Ts*) with *iconn* ∈ {*iand, ixor*} is a tree of input connectors with $Leaves(iconn(Ts)) = \cup_{u \in Ts} Leaves(u)$

Tree of output connectors are defined similarly. The only difference is that their roots are *oand* and *oxor* instead *iand* and *ixor*.

Additionally we allow *undefined* trees of connectors, for the case when we do not attach a glass box view to an activity. They are denoted by *iundef*(*Fs*) and *oundef*(*Fs*) where $Fs \subseteq FN$. We define *Leaves*(*iundef*(*Fs*)) = *Leaves*(*oundef*(*Fs*)) = *Fs*.

For a tree of connectors *t* we define the following:
i) a predicate *Leaf*(*t*) which is true iff *t* is a leaf;
ii) a function *Subtrees*(*t*) that is defined iff *Leaf*(*t*) = *false* and that returns the set of subtrees of *t*;
iii) a function *Root*(*t*) that is defined iff *Leaf*(*t*) = *false* and that returns the root of *t*.

An *activity* is a triple *act* = (*a, it, ot*) such that $a \in AN$, *it* is a tree of input connectors and *ot* is a tree of output connectors. We define *ActivityName*(*act*) = *a*, *InputTree*(*act*) = *it* and *OutputTree*(*act*) = *ot*. Any two distinct activities have different names, i.e. if $act_1 \neq act_2$ then $a_1 \neq a_2$.

For each activity *act* we define its set of *inputs Is*(*act*) = *Leaves*(*InputTree*(*act*)) and its set of *outputs Os*(*act*) = *Leaves*(*OutputTree*(*act*)). Additionally, we require that inputs and outputs are distinct, i.e. $Is(act) \cap Os(act) = \emptyset$.

A set *A* of activities is called a *diagram* iff any two distinct activities have no common inputs and no common outputs. Formally:
i) $(\forall\ act_1 \in A)\ \forall\ i \in Is(act_1))\ (\forall\ act_2 \in A \setminus \{act_1\})\ (i \notin Is(act_2))$
ii) $(\forall\ act_1 \in A)\ (\forall\ o \in Os(act_1))\ (\forall\ act_2 \in A \setminus \{act_1\})\ (o \notin Os(act_2))$

Notice that any set with a single activity is a diagram.

[1] $|S|$ denotes the cardinality of set *S*

Let *A* be a diagram.
i) the set of *flows* of *A* is the union of the inputs and the outputs of the activities in *A*, i.e. $Flows(A) = (\cup_{act \in A} Is(act)) \cup (\cup_{act \in A} Os(act))$
ii) the set of *inputs* of *A* consists of those inputs of activities in *A* that are not outputs for any activity in *A*, i.e. $Inputs(A) = \{i \in FN \mid (\exists\ act_1 \in A)\ ((i \in Is(act_1)) \wedge (\forall\ act_2 \in A \setminus \{act_1\})\ (i \notin Os(act_2)))\}$
iii) the set of *outputs* of *A* consists of those outputs of activities in *A* that are not inputs for any activity in *A*, i.e. $Outputs(A) = \{o \in FN \mid (\exists\ act_1 \in A)\ ((o \in Os(act_1)) \wedge (\forall\ act_2 \in A \setminus \{act_1\})\ (o \notin Is(act_2)))\}$
iv) the set of *private flows* of *A* consists of those flows of *A* that are not inputs and nor outputs of *A*, i.e. $PrivateFlows(A) = Flows(A) \setminus (Inputs(A) \cup Outputs(A))$.

It is easy to prove that for any diagram *A* the set of inputs, outputs and private flows of *A* is a partition of the set of flows of *A*.

An important concept in INSPIRE is that of presenting a model at different levels of detail. This is based on the notion of *decomposition*. So it is important to know when a diagram may be a decomposition of an activity.

Let *A* a diagram and let $act \notin A$ be an activity. *A* is called a *decomposition* of *act* iff i) $Inputs(A) = Is(act)$ and ii) $Outputs(A) = Os(act)$.

Note that for any activity *act* the singleton set {*act*} is a trivial decomposition of *act*.

An important result is that we can "compose" decompositions in order to obtain more refined decompositions of an activity.

Proposition 1 Let act_0 be an activity, A_1 a decomposition of act_0, $act_1 \in A_1$ and A_2 a decomposition of act_1. If $Flows(A_1) \cap PrivateFlows(A_2) = \emptyset$ then $(A_1 \setminus \{act_1\}) \cup A_2$ is a decomposition of act_0.

The interpretation is that if we decompose further in a "clean" way an activity in a decomposition of an activity act_0, we obtain a more refined decomposition of act_0. Note that if A_2 is a trivial decomposition we obtain no refinement of act_0, i.e. the result of composing the decompositions A_1 and A_2 is A_1.

To model the tree-like structure of presenting a business process model at different levels of details in INSPIRE we need the notion of *decomposition structure*.

Let *A* be a set of activities. A *decomposition structure* rooted at $r \in A$ is a function $dec : A \rightarrow 2^A$ such that:
i) $(\forall\ x,y \in A)\ (x \neq y \Rightarrow dec(x) \cap dec(y) = \emptyset)$
ii) $\cup_{x \in A} dec(x) = A \setminus \{r\}$
iii) $(\forall\ x \in A)(dec(x) \neq \emptyset \Rightarrow dec(x)$ is a decomposition of $x)$
iv) $(\forall\ x \in A)\ (\forall\ y \in dec(x))\ Flows(dec(x)) \cap$

PrivateFlows(*dec*(*y*)) = ∅
Some comments are needed. Conditions i) and ii) ask for the *dec* function to define a tree structure on *A*. *r* is the root of the tree and it is also called the *top level activity* of *A*. Condition iii) asks for *dec*(*x*) to be a decomposition of *x*. Condition iv) asks for the context of decomposing *y* to be "clean", i.e. the diagram that contains *y* should not have any common flows with the private set of flows of the decomposition of *y*.

A final important concept is that of the *level of detail* in the presentation of a business process model. A business process model in INSPIRE has a tree structure. Clearly, the least refined level of detail of the process is the root of the tree, and the most refined level of detail of the process is the set of leaves of this tree. Between them there are many "intermediate" levels of details. An "intermediate" level of detail corresponds to a *cut* in the decomposition tree.

Let *A* be a set of activities and let *dec* be a decomposition structure rooted at $r \in A$. A *cut* in *dec* is a subset of *A* with the following properties:
i) {*r*} is a cut in *dec*
ii) if $C \subseteq A$ is a cut in *dec* and $x \in C$, then $(C \setminus \{x\}) \cup dec(x)$ is a cut in *dec*.

A very important result that follows from proposition 1 is that any cut in a decomposition structure defines a decomposition of the root activity.

Proposition 2 Let *A* a set of activities, *dec* a decomposition structure rooted in $r \in A$ and *C* a cut in *dec*. Then *C* is a decomposition of *r*.

For example consider the process shown in figures 1, 2 and 3 and eliminating the fork as shown in figure 4. To avoid the use of long names, we rename the flows and activities: *Process Material Requests* = a_0, *Material Request* = i_1, *New Supplier Requirement* = i_2, *Validated Request* = o_1, *Validated Request'* = o_1', *New Supplier Package* = o_2, *Logged Request* = *t*, *Request Errors* = *t*, *Request Updates* = t_3, *Log Material Request* = a_1, *Validate Material Request* = a_2, *Resolve Request Problems* = a_3 and *Dev New Supplier Spec* = a_4. The formal representation of this process is:
$act_0 = (a_0, iundef(\{i_1, i_2\}), oundef(\{o_1, o_2\}))$
$act_1 = (a_1, i_1, t_1)$
$act_2 = (a_2, ixor(\{t_1,t_3\}),oxor(\{oand(\{o_1,o_1'\}),t_2\}))$
$act_3 = (a_3, t_2, t_3)$
$act_4 = (a_4, iand(\{o_1',i_2\}),o_2)$
Note that $\{act_1, act_2, act_3, act_4\}$ is a decomposition of act_0.

IV. MAPPING THE INSPIRE NOTATION TO P/T NETS

The result stated by proposition 2 in section III allows the study of the problem of mapping a cut to a P/T net. The result of this mapping will provide the semantics for the presentation of the business process model at the level of detail corresponding to the cut.

P/T nets have been intensely studied and have a well-established classic theory. They have already been used for modeling workflows ([8]). Moreover, there have been identified special classes of P/T nets suitable for an efficient static analysis ([9]).

A P/T net can be represented using three sets: i) the set of *places*; ii) the set of *transitions*; iii) the set of *arcs* $\subseteq$ (*places* $\times$ *transitions*) $\cup$ (*transitions* $\times$ *places*)

We shall describe the mapping by means of an algorithm for translating a cut into a P/T net. We assume that each activity of the cut has associated a glass box view. For each activity in the cut the algorithm translates the tree of input connectors and the tree of output connectors. The activity itself is translated into a transition. The flows are translated into locations. The translation of trees will produce new locations and transitions. Due to the lack of space we show only a part of the translation algorithm. However, it is sufficient to understand how the translation works.

The translation of a cut means translating each activity of the cut. The translation of an activity *act* is done according to the algorithm in figure 5.

```
procedure Activity2PN(act)
        l1 ← InputTree2PN(InputTree(act))
        l2 ← OutputTree2PN(OutputTree(act))
        trans ← NewTransition(ActivityName(act))
        transitions ← transitions ∪ { trans }
        arcs ← arcs ∪ { (l1,trans),(trans,l2)}
end
```

Fig. 5. Translating an activity

The translation of a tree of input connectors is done according to the algorithm in figure 6.

```
function InputTree2PN(IT)
        if Leaf(IT) then
                r ← NewPlace(IT)
                places ← places ∪ { r }
        else
                trees ← Subtrees(IT)
                r ← NewPlace(NewSymbol('l'))
                places ← places ∪ { r }
                rs ← InputTrees2PN(trees)
                if Root(IT) = 'iand' then
                        IAnd2PN(rs,r)
                else
                        IXor2PN(rs,r)
        return r
end
```

Fig. 6. Translating a tree of input connectors

The translations of *iand* and *ixor* connectors are shown in figures 7 and 8. The *oand* and *oxor* connectors are translated in a similar way excepting that the direction of the arc is reversed.

procedure *IAnd2PN*(*rs*,*r*)
    trans ← *NewTrans*(*NewSymbol*('tr'))
    transitions ← *transitions* ∪ { *trans* }
    for each *x* ∈ *rs* **do**
        arcs ← *arcs* ∪ { (*x*,*trans*),(*trans*,*r*)}
end

Fig. 7. Translating an *iand* connector

procedure *IXor2PN*(*rs*,*r*)
    for each *x* ∈ *rs* **do**
        trans ← *NewTrans*(*NewSymbol*('tr'))
        transitions ← *transitions* ∪ { *trans* }
        arcs ← *arcs* ∪ { (*x*,*trans*),(*trans*,*r*)}
end

Fig. 8. Translating an *ixor* connector

The function *NewSymbol*(*Prefix*) generates a new symbol of a given prefix. The functions *NewPlace*(*Label*) and *NewTransition*(*Label*) generate a new place and a new transition with a given label.

It can be easily noticed that the complexity of the translation algorithm is linear in the number of activities plus the number of connectors in the cut.

The result of translating the process in figure 3, assuming that the fork has been eliminated as in figure 4, is shown in figure 9.

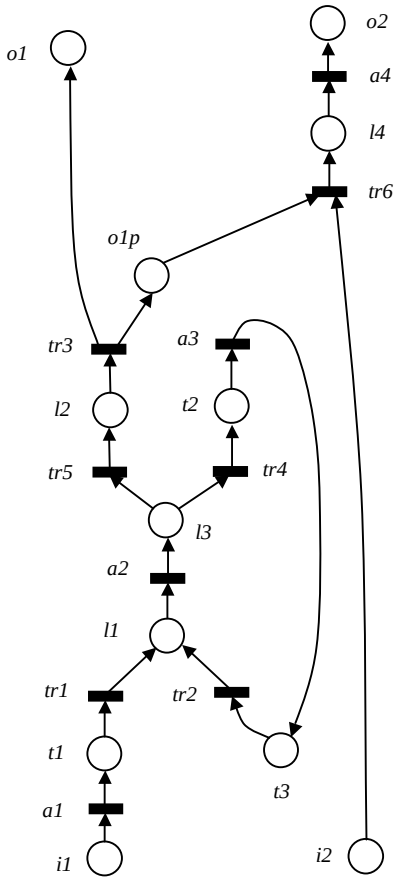

Fig. 9. The translation of the process in figure 3

V. CONCLUSIONS AND FUTURE WORK

P/T nets proved to be very useful in understanding the dynamics of the business processes modeled by the INSPIRE tool. The P/T nets generated can act as a basis for the static analysis and dynamic simulation of these processes.

ACKNOWLEDGMENTS

The work described here was supported by funding from the European Union as part of the Framework V INSPIRE project (IST-10387-1999).

REFERENCES

[1] M. Ould,, Business Processes: Modeling and Analysis for Re-engineering and Improvement, John Wiley & Sons, 1995.
[2] C. Fox, "The process representation module (specification)", INSPIRE (IST-1999-10387) Deliverable 2.1, 2000.
[3] Draft Federal Information Processing Standards Publication 183, Integration Definition for Function Modeling (IDEF0), 1993.
[4] R. Mayer et al., Information Integration for Concurrent Engineering (IICE): IDEF3 Process Description Capture Method Report, 1993.
[5] C. Badica, C. Fox, ″Modeling and verification of business processes″ unpublished.
[6] M. Schroeder, ″Verification of business processes for a correspondence handling center using CCS″, Proc. European Symposium on Validation and Verification of Knowledge Based Systems and Components, Oslo, Norway, pp.253-264, 1999.
[7] E. Yu, J. J. Mylopoulos, Y. Lesprance, "AI models for business process re-engineering", IEEE Expert, 11(4), pp.16-23, 1996.
[8] W.M.P. van der Aalst, "The Application of Petri nets to workflow management", The Journal of Circuits, Systems and Computers, 8(1), pp.21-66, 1998.
[9] J. Desel, J. Esparza, Free Choice Petri Nets, Cambridge Tracts in Theoretical Computer Science 40, Cambridge University Press, 1995.