A Top-Down Petri Net-Based Approach for Dynamic Workflow Modeling*

Piotr Chrząstowski-Wachtel[1]**, Boualem Benatallah[2], Rachid Hamadi[2], Milton O'Dell[3], and Adi Susanto[2]

[1] Institute of Informatics, Warsaw University
Banacha 2, PL 02-097 Warszawa, Poland
pch@mimuw.edu.pl
[2] School of Computer Science and Engineering
The University of New South Wales, Sydney NSW 2052, Australia
{boualem,rhamadi,adis}@cse.unsw.edu.au
[3] Justwin Technologies Pty Ltd
7-9 West Street Suite I.20, Level 1, North Sydney NSW 2060, Australia
modell@justwin.com

Abstract. A top-down approach for workflow design is proposed in the framework of Petri net theory. Simple but powerful refinement rules are proposed that guarantee soundness of the resulting workflow nets. The refinement process supports the definition of regions, which are parts of the workflow that correspond to logistically related items. Exception handlers can be associated to regions. Defining regions helps determining the impact areas of the unexpected events during workflow execution.

1 Introduction

Workflow management systems are used for controlling the execution of business processes. These processes typically consist of multiple activities, which have to be performed in a valid sequence [1,2]. Dynamic workflows deal with changes during the workflow execution. Some of them are unpredictable, and it is hence necessary to allow flexible modeling and run-time maintenance. The main problem here is to restrict the impact area of unexpected exceptions. Recovery regions [3] form a partition of the workflow into areas to which the reaction for the exceptions is limited to. A flat structure of regions was considered in [3]. Since exceptions are of different importance, it is reasonable to make the reactions structured so that we achieve two goals. First, only the minimum part of the workflow is affected. Second, provide a mechanism for handling exceptions at

* This work is partially supported by an ARC SPIRT grant "Managing Changes in Dynamic Workflow Environments" between UNSW, QUT, and Justwin Technologies and by an internal research grant No. BW/ALG/01/2002 of PJWSTK financially supported by KBN in Poland.

** Also with Polish-Japanese Institute of Information Technology, Koszykowa 86, PL 02-008 Warszawa, Poland.

W.M.P. van der Aalst et al. (Eds.): BPM 2003, LNCS 2678, pp. 336–353, 2003.

different levels of the design that often reflect different levels of management. Certain recovery decisions will be limited to appropriate levels of management. If the design is made accordingly, the levels will be determined at design phase.

A hierarchical Petri net approach will be presented in this paper. The key idea is to design a workflow in a top-down manner, starting from a single place and performing refinements using a given set of rules. The refinement rules guarantee that the workflow will enjoy the desired soundness property [4]. The refinement rules are chosen to be simple enough to use and powerful enough to represent many common workflow patterns identified in [5].

The paper is structured as follows. Section 2 describes the design of workflows using basic refinement rules. Section 3 deals with non-refinement rules, i.e., communication and synchronization, as well as with soundness property. The recovery regions are presented in Sect. 4. Section 5 describes the *HiWorD* (HIerarchical WORkflow Design) tool. Finally, Sect. 6 concludes the paper.

2 Designing Workflows by Refinement

Workflows may represent significantly large business processes, having hundreds or thousands of actions to be performed upon completion. As usual, when dealing with complicated processes, it is important to follow a structured design model.

Our approach will concentrate on building a hierarchical workflow model based on Petri nets, a model used in concurrency theory, which has proven to support efficiently the hierarchization concept. Since the structure of the top-level tasks in most workflows is not too complicated, we decide to split the design process into two phases. First, the design of the overall workflow structure, which will be restricted by some rules allowing to define tasks in a safe way on top levels. Then specifying the details of the task description at a lower level, allowing some primitives, which will make the design safe, in the sense that it will not cause deadlocks [6,7].

It is tempting to use simpler models as they are easy to analyze and implement, but it is undesirable to limit the expressive power of the model. Our approach aims at making the workflow structured as in structured programming languages. A method similar to procedural encapsulation together with the idea of procedure nesting will be used for determining the workflow integral parts.

At some stage we let the designer define *regions*. They represent the encapsulation concept in the workflow design. We can think of them as *milestones* of workflow execution, i.e., the completion of region execution should result in reaching a state of execution, that typically once completed will not be re-done. Since our approach aims at unexpected exceptions, we associate with each region exception handlers that restore the workflow execution within a region once an exception is raised. Due to the nested structure of regions, different recovery levels can be described. A good design will reflect the management structure and support the decisions to be taken at an appropriate level.

We will illustrate the approach using the stepwise refinement method. Places in Petri net are considered not static states, but rather as representing the state

of execution of a part of the workflow. Beginning with a single place, being the coarsest view of the workflow description (i.e., the root of the refinement tree), the basic transformations allow to model the sequences of actions, the choice between two or more tasks, and the parallel split. The five basic refinement transformations are shown in Fig. 1.

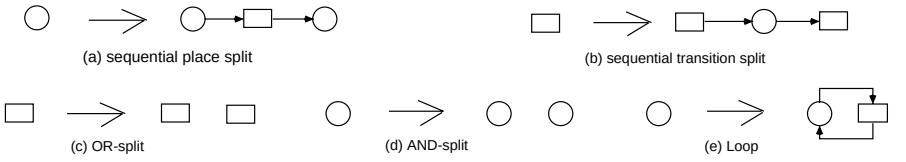

Fig. 1. Basic Refinement Rules

For all the rules displayed in Fig. 1, except the first one, we require that they can be applied only if there is at least one input and at least one output arc associated with the node to make the transformation valid. Moreover, we presume that all the input arcs of the node are copied to all the resulting entry nodes, and that analogous rule applies to the output arcs.

2.1 Workflow Refinement Rules

The first two pairs are dual, while the Loop-split rule does not have its dual counterpart. Observe that we do not want to introduce a dual rule of refining a transition, since we would face the problem of marking the input place. We would like to mark the whole workflow net (WF net) with only one token in the input place. This would mean, that the looping place would block its transition from being ever executed.

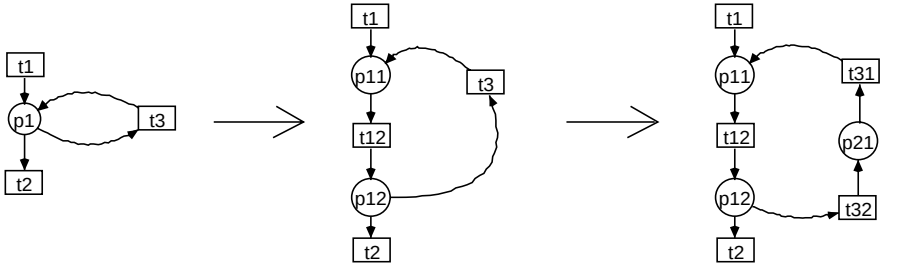

Fig. 2. Further Loop Refinement

The Loop-split rule allows us to construct loops having only one input and one output, both being places. When performing a sequential split, the place

that the loop was created from will be refined and the input and the output of the loop will be separated as in Fig. 2. We can clearly see, that when we refine the initial loop place, the actions that occur will be performed in any loop turn (action t_{12} in this case). When we refine the loop transition, we are in fact designing actions that should be performed between the failed loop-exit-tests and re-doing the loop again (t_{31}, t_{32}). So they serve for the re-initialization of the loop.

It should be noted that not all WF nets can be constructed using the proposed rules. The following theorem allows to specify most important features of such nets.

Theorem 1. *If we start with a single place and apply the above transformations under the constraint that, except for the sequential place split, all the transformations may be applied only if a node has at least one input and at least one output arc, then the resulting WF net* $\mathcal{N} = \langle P, T, \rightarrow \rangle$ *enjoys the following properties:*

1. *There is exactly one place* $p_{in} \in P$ *such that* $\forall x \in P \cup T \; p_{in} \rightarrow^* x$. *We call* p_{in} *the workflow input,*
2. *There is exactly one place* $p_{out} \in P$ *such that* $\forall x \in \; P \cup T \; x \rightarrow^* p_{out}$. *We call* p_{out} *the workflow output,*
3. *The number of incoming arcs in the WF net is equal to the number of outgoing arcs,*
4. *The WF net* $\mathcal{N}$ *is a free-choice net, and*
5. *The WF net* $\mathcal{N}^{-1}$ *resulting by reversing the arcs is a WF net, which is obtainable by the same set of refinements.*

Proof. The induction on the number of transformations made completes the proof of each of these properties. □

Recall, that a Petri net is a free-choice net if and only if for every two transitions, if they share any input place, then all their inputs are the same [8]. This condition guarantees that either both transitions are enabled or none of them is enabled under any marking.

2.2 On the Expressiveness of Refinement Rules

The proposed transformations are ready to express quite a variety of typical situations that occur in workflow design. However, there are some restrictions, that is, not every Petri net with one input and one output place (WF nets are assumed to have this property) can be generated using these rules.

Consider for instance the WF nets depicted in Fig. 3. These nets cannot be obtained by applying the rules given in Fig. 1. The net in Fig. 3(a) cannot be obtained because the total number of arcs being inputs to transitions is 4 and being outputs is 3, hence violating Theorem 1(3). The net in Fig. 3(b) cannot be obtained because it is not a free-choice net, thus violating Theorem 1(4).

In fact, to describe intuitively what is the main restriction on the class of WF nets obtained by the above rules, it should be noted that, although both

parallel splits and synchronizations are allowed, the following constraint must be satisfied: *what has been split must be synchronized later*. Similarly *what has constituted a simple exclusive choice must be later merged*. Recall, that although the net of Fig. 3(a) violates this principle (an exclusive choice if followed by a parallel merge), the net of Fig. 3(b) one does not. There is a triple OR-split at the top and a triple OR-join at the bottom, as well as three AND-splits, each of them followed by an AND-join. The problem with this net is that, although the numbers of splits and joins of the same type are the same, the OR- and AND-splits and joins are shuffled so that we obtain a sequence: OR-split followed by AND-split followed by OR-join followed by AND-join. In our model, nesting is allowed but not shuffling.

The first net represents a design structure that we want to avoid since it causes a deadlock. Similar difficulties would be faced in the dual case of a parallel split followed by a merge, resulting in a generation of a new token. The latter situation is at first glance less unsafe. However, the second net represents a kind of normal situation. Since we don't want to rule out this kind of behavior, we will consider such transformations possible on a lower level that will allow us to represent arrangements of this kind.

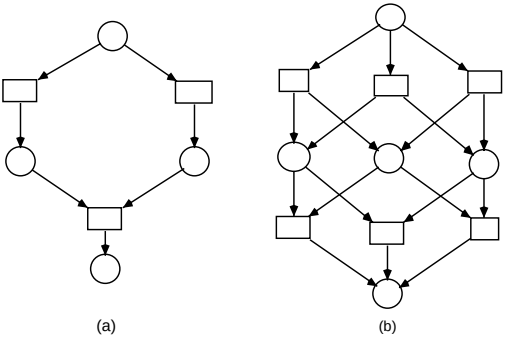

Fig. 3. Nets that can not be obtained by using the basic refinement rules

The proposed rules correspond to the first five patterns[1] identified by van der Aalst et al. [5], but with the restriction that the parallel split must be followed by the parallel synchronization, while the exclusive choice must be followed by a simple merge. This restriction is considered not to be a deficiency. Rather it introduces a sound constraint that prevents us from making obvious design errors. At the management level the above rules will suffice for a large amount of cases. What we gain is the guarantee that the resulting WF net is correct with respect to the *soundness conditions* as identified in [4].

[1] http://www.tm.tue.nl/it/research/patterns/.

Definition 1. *A sound WF net is a Petri net with places p_{in} and p_{out} that satisfies the following conditions:*

- *For each token put in p_{in}, one and only one token eventually appears in p_{out},*
- *When the token appears in p_{out}, all other places are empty, and*
- *For each transition (i.e, task), it is possible to move from the initial state to a state in which this transition is enabled.*

Theorem 2. *If we start with a WF net consisting of a single place with no transitions, and perform only the proposed transformations, then the resulting WF net is sound.*

Proof. The induction on the number of refinements performed completes the proof. □

2.3 Refinement Tree

In the next sections we will provide the rules that allow us to construct basic WF nets in order to support elaborated patterns like communication or synchronization between concurrent parts of the net. In order to do so, it is desirable to define the notion of concurrency in such nets precisely, and to have a fast algorithm for determining if two nodes are truly concurrent. For this purpose, we use the notion of *refinement tree*, which will be introduced in this section. We show that the refinement tree will be defined in a unique way for each net obtained by a sequence of basic refinements.

Note that, after applying several times the transformation rules, we obtain a tree structure reflecting the order in which the transformations were made. The places and transitions of the net will label the leaves of the tree, while the internal nodes will be labeled by the transformation names. Let us assume that the tree is ordered, so that we can identify the role of each node in case of non-symmetrical (sequential) transformations. The root of the tree is initially labeled by the initial place we start our design from. Since each transformation rule is of the form $\langle \text{node} \rangle ::= \langle \text{net} \rangle$, when we apply any of these rules, the node from the left hand side of the transformation description already exists in the tree being its leaf. We then create its children labeling them by the resulting nodes, and marking the new parent node by the transformation name. The tree is uniquely determined by the order in which the transformations were made.

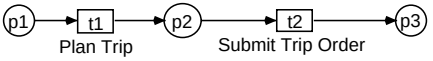

Fig. 4. A Sequence of Two Transitions

Since the refinement tree will be later used for characterization of sound transformations, which we are going yet to introduce, it would be desirable to

associate a unique tree to every WF net that resulted as a sequence of canonical refinements. So far this is not the case. To illustrate that let us consider the WF net of Fig. 4, for which three non-isomorphic trees can be created, as depicted in Fig. 5.

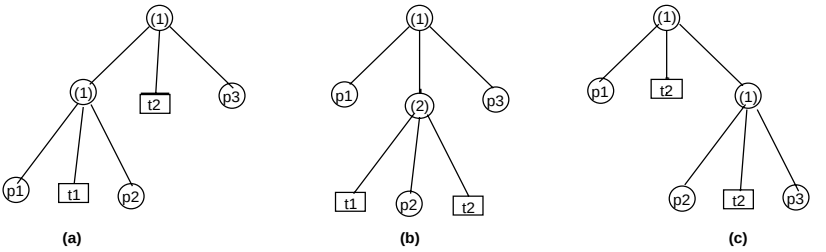

Fig. 5. Three Non-isomorphic Trees for the Net from Fig. 4

The refinement trees always have internal nodes labeled by the refinement rules, and the leaf nodes by the transitions and places. The trees are ordered, so that from such trees we are able to reconstruct the refinement process. The reason for which we cannot make the opposite (draw the tree when looking at the net) is that we are too accurate in performing sequences of refinements of the same kind. We distinguish too many details. Instead, we would prefer to look at such sequences of refinements as one-level refinements. In case of the example of Fig. 4, there should be one place sequential split node with five children: $p1$, $t1$, $p2$, $t2$, and $p3$, hence having only two levels instead of three.

We would like to contract the tree in case a parent and all its child nodes are labeled by the same transformation. In this case we would make all such children to become siblings of the parent by shifting them one level up. This operation reflects the extension of the rules to be n-ary instead of binary. For instance, we could define n-parallel split, n-parallel choice, and so forth. As both place sequence split and transition sequence split are reflecting a very similar kind of refinement, we will not distinguish between them, and treat them as one kind of refinement. The tree resulting from the original tree by such contraction will be called the *refinement tree*.

The height of the tree forms a basis for defining complexity of the workflow. From this point of view the workflow does not become more complicated if we add one more node to a sequence, but it does if we perform a parallel split in a sequence or sequentialize something that has been previously split.

To make the refinement tree canonical, hence unique for a given refinement net, we must define an equivalence relation on the refinement trees, so that non important differences in the construction will not lead to distinguishing the resulting two trees as different. The main problem is the importance of the order of children nodes depending on the kind of the father node. It turns out that if the sequential refinements take place, the order of children is important, as we

wish to distinguish between the first, and the last among the 3 resulting nodes, which are of the same kind and could be easily confused if the tree was unordered. At the same time, the other 3 refinement rules make the order irrelevant: there is no reason why we would have to say which of the nodes is first and which is the second child. They are identical at this stage. Let us define the equivalence relation on refinement tree recursively:

Definition 2. *Two trees T_1 with root v_1 and T_2 with root v_2 are equivalent iff one of the following conditions holds*

- *T_1 and T_2 are one-node trees and the types of the nodes is the same.*
- *If v_1 and v_2 are labeled by the sequential refinements, then the number of children of v_1 is the same as the number of children of v_2, and the corresponding children are equivalent.*
- *If v_1 and v_2 are labeled by OR-split, AND-split, or Loop-split, then there is a bijection between equivalent children of v_1 and v_2.*

It turns out that whenever different orders of refinements lead to the same net, then the contracted trees are identical for both of them.

Theorem 3. *If the WF net $\mathcal{N}$ resulted from a sequence of considered refinements then, no matter in which order they were made, the resulting refinement tree is uniquely determined.*

Proof. The proof will be an induction on the number of internal nodes n (each internal node represents one refinement). The theorem holds trivially for $n = 0$. Assume that $n > 0$ and for every $k < n$ all the nets with at most k internal nodes have isomorphic trees. Consider a WF net $\mathcal{N}$ having n nodes with two refinement trees of $\mathcal{N}$ representing different orders which led to the construction of $\mathcal{N}$. Each of the trees results as a sequence of refinements. Consider the last refinement made in the construction of T_1 and the last refinement of T_2. Let T_1' and T_2' be the trees before the last refinements made. Consider two cases:
Case 1. The areas touched by the last considered refinements are disjoint (i.e. none of them has a common node with the other one to share). Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be the nets corresponding to T_1' and T_2'. Let $\mathcal{N}_1'$ be the net resulting from $\mathcal{N}_1$ by undoing the possible refinement which was the last in $\mathcal{N}_2$. Similarly let $\mathcal{N}_2'$ be the net resulting from $\mathcal{N}_2$ by undoing the possible refinement which was the last in $\mathcal{N}_1$. Undoing these refinements is possible thanks to the assumption about the areas being disjoint. Now we got two isomorphic nets, and since the number of nodes is smaller than n, their corresponding refinement trees are also isomorphic. Now the only way to get the initial net $\mathcal{N}$ is to redo the two withdrawn refinements obtaining two isomorphic trees.
Case 2. The last refinement made in the construction of T_1 has a node in common with the last refinement made in the construction of T_2. In this case it is enough to verify that either the two last refinements were done in the same level, and both were of the same type (in particular sequential refinements of any type), in which case the order they were made is irrelevant, or they were indeed

refinement of different types, in which case we come to the contradiction, because no pair of different refinements is commutative. And we come to a contradiction: we cannot obtain the same (isomorphic) net. By not being commutative we mean, that starting from one node and performing any pair of different refinements (the only way to get the areas intersected) we always obtain non-isomorphic nets. This property can be checked manually, as there is a finite number of such pairs. □

Each WF net hence produces a uniquely determined (up to isomorphism) *refinement tree*. We will refer to this tree later, when defining new constructs, which allow us to have synchronization and communication between parallel parts. The tree will be used to determine the parallel parts of the WF net.

3 Non-refinement Rules

The refinement rules proposed so far are not sufficient for all situations that can occur during workflow design. We claim that they suffice for defining the region structure. Such structure is usually simpler than the detailed design, which may require more elaborate rules.

The key idea of this approach is to find a set of refinement rules that is sufficient to design typical workflow behavior. Workflows constructed using such rules: (i) enjoy all desired properties, such as soundness as mentioned in [7], (ii) have a structure easy to understand and maintain that reflects the management structure, and (iii) express most typical workflow patterns.

The structural design we have proposed so far is simple and quite powerful, being able to represent many real life situations. However, there exist patterns, which can not be obtained in a hierarchical way. The key problem is the interaction between concurrently enabled activities. We distinguish such patterns, as they occur in many workflow designs. All these patterns do not follow the refinement schema, where a single node is being transformed into a more complicated net. The proposed patterns involve more than one node at a time, and build a net structure between them according to some restrictions that guarantee soundness. The soundness property is the one we would like to keep, even at a price of some limitations in design freedom.

In fact all the proposed patterns will have something in common. They will join in certain ways different parts of the WF net by introducing new transitions or places serving as bridges. The important thing is to restrict such auxiliary constructs to connect only parallel parts of the net, in order to avoid the situation where an active part of the net would interact with an inactive one. Another restriction is that such constructs should not introduce loops, as they could easily cause deadlock.

In the following, we denote by place-type nodes in the refinement tree places, sequential place splits, AND-splits, and Loop-splits, while transition-type nodes stand for sequential transition splits and OR-splits, as identified by the canonical refinements (see Fig. 1).

Definition 3. *Two nodes x_1 and x_2 (places or transitions) in a WF net obtained as a result of the canonical refinements lie in parallel threads iff in the refinement tree the only nodes on the path from x_1 to x_2 are the ones labeled by AND-split, sequential splits, or Loop-splits, but in this last case only followed by a place-type node.*

Therefore, there may be neither OR-splits nor Loop-splits followed by a transition-type node (transition, sequential transition split, or OR-split) on the path joining two nodes that we define being on parallel threads.

The notion of parallel threads will be helpful in defining sound patterns which allow us to join different parts of the WF net in a non-refinement manner. The key idea is that when two nodes are in parallel threads, then in all complete runs (from the token on p_{in} to the marking with a token on p_{out}) either both of them occur (place holds a token or transition fires) or none. Moreover, for all pairs of such nodes there exist such runs, in which each of them precedes the other one.

Theorem 4. *If a Loop-split followed by a transition-type node or an OR-split occurs on the path from x_1 to x_2 in the refinement tree, then there exists a complete run such that one and only one of x_1 and x_2 occurs on this run.*

Proof. When an OR-split occurs on the path from x_1 to x_2, then both of the nodes lie in different choices of one decision. Since the net is free-choice, when we make the choice in favour of one node, the other one cannot be activated. When a Loop-split followed by a transition-type node occurs on the path from x_1 to x_2 in the refinement tree, it means that x_2 occurred as a result of refining the transition, that closes the loop backwards. Any attempt to enforce the occurrence of this backward transition is hopeless, since we have free-choice decision whether to continue the loop or to leave it. Note, that the refinement of the Loop-place into a sequence produces part of the loop that will always be performed, so there is no need to exclude the Loop-splits on the path in general. □

The reverse statement is also valid as stated by the following result.

Theorem 5. *If neither a Loop-split followed by a transition-type node nor an OR-split occurs on the path from x_1 to x_2 in the refinement tree, then either both x_1 and x_2 or none of them occur in all complete runs.*

Proof. The sequential splits just carry tokens along the nodes that result after the split, so indeed a sequential split carries no danger that only one of x_1 or x_2 will occur in the complete run. Similarly the AND-split activates both places simultaneously. The Loop-split pattern followed by a place-type node plays the role of a part of the loop which will always be executed once a token occurs in the initial place after the refinement. So the entire sub-net that is the result of further refinement preserves the desired property. Induction completes the proof. □

These two theorems will form the basis for formulating the conditions for sound non-refinement patterns, which connect existing nodes in the WF net.

3.1 Communication

The first of the non-refinement patterns is *communication*, where one agent wants to send a message to another, and both are concurrently running. We assume here that all messages are important and that the receiver will wait for the expected message until it arrives. This construct is somewhat of different nature and should be used with extreme care. We will propose a condition that will allow a safe introduction of a new communication place, creating a buffer for message passing without loosing soundness condition.

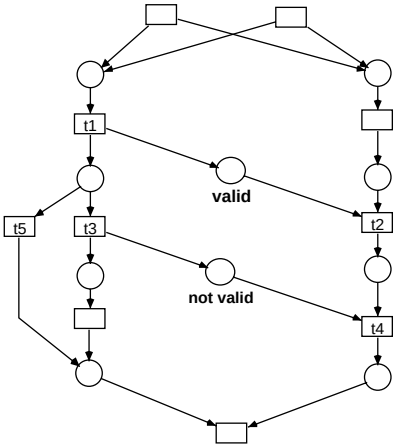

Fig. 6. Communication

In many cases, when the execution of parallel threads appears in a workflow, a communication should occur involving sending a message from one thread to another one. We assume that, the recipient of the message should wait until the message arrives. So we treat messages like signals that enforce synchronization between parallel parts of a workflow rather than information spread across the workflow diagram. In order to send a message, one needs to specify the sender and the receiver, being aware that they are indeed active agents. A deadlock can occur if the receiver is expecting a message from the sender that is not activated because, for instance, it was not chosen at an OR-split point.

At design phase, we will create the communication links by pointing at two partners of communication, but with an additional restriction. They should lie on parallel threads, and the introduced communication place should not close a cycle.

Consider the net example of Fig. 6. One can verify, that the transitions $t1$ and $t2$ lie on parallel threads. It would be perfectly valid to make a communication from $t1$ to $t2$, as well as the opposite way around. But we do not allow a communication to be performed between $t3$ and $t4$, because in the refinement

tree on the path leading from t_3 to t_4 there is an OR-split. In case a choice for $t5$ has been made in favor of $t3$, the transition $t4$ would be made dead. In fact the presence of an OR split on the path between two communicating agents is inherently bad, as the net is free choice and we can always make a wrong decision causing a deadlock or a trash token, which could possibly never be taken away.

Moreover, it should not be allowed to send a message to a node, which is our predecessor in the net, hence closing a cycle. But this will be excluded by the restriction we impose on the communication agents as defined below.

Definition 4. *Let t_1 and t_2 be two transitions. A place p joining t_1 and t_2 is a communication place iff the following conditions are satisfied:*

- *t_1 and t_2 lie on parallel threads, and*
- *Introducing the communication place does not close a cycle.*

To preserve soundness, both conditions should be satisfied. In case t_1 and t_2 are not in parallel threads, a scenario can be created in which soundness is lost.

3.2 Synchronization

It is often the case that two parallel threads should synchronize their activities before they are completed. We distinguish here between two cases: *symmetric synchronization* and *asymmetric synchronization*. The first one occurs when there are checkpoints in each of them that must be simultaneously reached. The asymmetric one occurs when one of them is privileged: it does not need to wait for the other one to proceed, while the other one can not go across the checkpoint if the first one has not reached it.

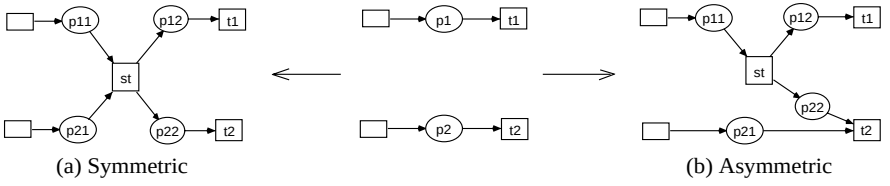

Fig. 7. Synchronization

Symmetric Synchronization. The symmetric synchronization can be obtained by the pattern depicted on Fig. 7(a). We are supposed to synchronize the parallel threads on places $p1$ and $p2$. Neither $t1$ nor $t2$ may fire unless the other one is enabled. In other words, we would like to enable both of them as soon as tokens appear on both $p1$ and $p2$. As a result of the synchronization, the synchronized places $p1$ and $p2$ are split and the synchronizing transition st fires as soon as tokens appear on their "left" halves, i.e., on $p11$ and $p21$. It is

assumed that, this transition fires without any delay as soon as it is enabled. As a result of firing st we obtain tokens on places $p12$ and $p22$. They enable transitions $t1$ and $t2$ as required.

Asymmetric Synchronization. The asymmetric synchronization can be obtained by the pattern depicted on Fig. 7(b). This time we want to let the upper thread proceed without waiting for the lower one, but the lower thread should wait until a token appears on $p1$ in the upper thread. This pattern, like the symmetric synchronization, should be made under the same condition as the communication pattern. In fact, communication involving the transition immediately preceding the synchronization place in the dominating thread could be an option for realizing this pattern.

Definition 5. *Let p_1 and p_2 be two places. A synchronization pattern may be added involving p_1 and p_2, as in Fig. 7, iff the following conditions are satisfied:*

- *p_1 and p_2 lie on parallel threads, and*
- *Introducing the synchronizing transition does not close a cycle.*

3.3 Soundness Property

All the introduced patterns have a similar condition: the only parts of a WF net that can interfere safely (without loosing soundness property) with each other by some additional constructs, are the ones that occur in parallel threads. Hence, the threads that were generated from an AND-split as a result of further refinement of places which were in the right-hand side of the refinement rule.

Theorem 6. *If we start with a single place and proceed with the canonical refinements as well as with the communication and synchronization patterns under the given restrictions, then the resulting WF net is sound.*

Proof. Let us concentrate on the communication pattern, as the proof for the synchronization patterns follows the same line of reasoning. Assume, that t_1 wants to send a message to t_2 involving communication pattern and both lie on parallel threads, and the introduced new place does not close a cycle. Theorem 4 says that either both transitions will be absent in a complete run (in this case soundness is trivially preserved) or both will be present. In the latter case the introduced place may enforce the order in which the transitions will be executed. If in all runs, in the original net, t_1 precedes t_2, then the same runs will be possible after introducing the communication place. If in all runs, in the original net, t_2 precedes t_1, then it means that, t_2 is a predecessor of t_2 in the net, hence introducing the communication place from t_1 to t_2 would close a cycle.
If in some runs t_1 precedes t_2 and in some other ones t_2 precedes t_1, then introducing the communication place will enforce the order in which they are executed, but will not cause a trash token or a deadlock. The token put on the communication place will be consumed by t_2. And the communication place cannot cause a deadlock, because it will withhold t_2 from firing until t_1 fires, and

since neither of them is a predecessor of the other one and the net is free choice, we can always execute the runs in which t_1 precedes t_2. □

Observe that, the conditions under which the communication and synchronization patterns were allowed are also necessary. The reader can verify, that if the involved nodes do not lie on parallel paths or the introduced net closes a cycle, then there is always a possibility of creating a scenario that causes deadlock or trash tokens.

4 Recovery Regions

We will now apply the proposed top-down approach to the definition of *recovery regions* [3]. We propose another idea, that is, to use the refinement process as a natural method to define regions.

Regions represent parts of a workflow. Region elements are related to each other and can be identified as a whole activity to be performed, clearly indicating its origin and end. We want regions to be workflows themselves, enjoying all the workflow properties. We also associate with regions exception handlers. Each exception handler will determine the compensation procedure and entry points, so that some amount of work done so far can be saved. We would like to restart the execution of a region at the latest possible entry point. In the sequel, we introduce the encapsulation mechanism, which will allow us to define recoveries in a structured way.

4.1 Design Overview

At any step of the refinement process, we can request a place to become a region. To each region several exception handlers can be assigned. Each such compensation procedure will affect the execution of the region in the following way. The entire region's interior will be reset, and tokens will be deposited on certain places associated with the recovery transition. By resetting the region's interior, we mean stopping all the activities being executed, removing all the tokens from the interior of the region, and performing some compensation procedures as needed. Depositing tokens means that we specify the entry points, from which the execution of the region will be resumed. This can save some work done so far, however quite often it can happen that a default reset, i.e., putting a token on the region input place, will be performed. The default reset means we will start executing the whole region from the beginning.

Once a place p is refined to a region, all the subnet resulting from p forms the region's interior. The subtree of a node, that was called a region, represents the interior of a region. The place, being the first one in the left-to-right in-order traversal of the subtree associated with a region, is the input place of the region, while the last one is the output place of the region. Note that any region is a valid WF net by itself.

Regions represent typically the milestones of a workflow execution. Once passed, they will not usually be redone. The transitions between the regions

play an auxiliary role of transferring the control from one or more regions to some other ones. However, sometimes an unexpected event may necessitate to move backward and forward between regions, as parts of workflow will have to be redone or skipped. For example, if a transport of produced garments is stolen or destroyed, we must go back to the production phase, but usually after the design phase. On the other hand, if during quality assessment, for instance, we require three independent checks, and only two are ready by the deadline, we can decide to skip waiting for the last one to arrive and go to the next phase which form another region. Such events require triggering a recovery transition which can cause, for instance, stopping all the activities that concern the stolen or destroyed goods and initiating the production phase in the first example, or informing the quality assessment third party about abandoning the need for the review and preparing the unit which is responsible for the collection of the reviews to discard anything that comes from it in the second example. Similar procedures could be taken at any level of the description since every region is a workflow by itself. The only difference is that the management level is different and the communication is more local.

4.2 Example: Ordering Flight Tickets

We give here an example of workflow representing ordering flight tickets over the Internet to illustrate the concepts introduced. A customer plans her/his trip by specifying the various stages of the overall journey. Then s/he sends this information together with the list of all participants of the trip and the information about the credit card to be charged for the ordered tickets to the travel agent and waits for the submission of the electronic tickets as well as the final itinerary before preparing for the trip by making other arrangements such as hotel booking and car renting. When the travel agent receives the customer's trip order, s/he will determine the legs for each of the stages, submits these legs together with the information about the credit card to be charged to the airline company, and waits for the confirmation of the flights. This information is completed into an itinerary and sent to the customer. When the airline receives the tickets order submitted by the agent, the requested seats will be checked and, if available, assigned to the corresponding participants. After that, the credit card will be charged and the confirmation of the flights sent. Finally, once the travel agent receives confirmation, the airline sends the electronic tickets by e-mail to the customer.

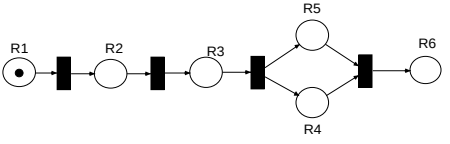

Fig. 8. Ordering Flight Tickets High-level Workflow

Starting from a single place and using a succession of refinement rules, the high-level workflow shown in Fig. 8 is obtained. Black rectangles stand for silent (or empty) transitions. By further refining places `R1` through `R6`, we obtain the flat workflow represented in Fig. 9. An example of regions associated within the workflow is also given in Fig. 9. Regions are depicted by dotted polygons, recovery transitions by dashed boxes, and recovery arcs (i.e., arcs linking regions to recovery transitions) by attaching two arrowheads instead of one to the end of the arcs. For presentation clarity, we restrict the detail of recovery transitions to region `R3` only. Within a region, for all other (unexpected) exceptions that do not match the expected exceptions, a default unexpected exception handler is raised. This can be, for instance, undoing the effects of the activities and resuming the execution of the region from the beginning. For region `R3` in Fig. 9, this is represented by the recovery transition `t3`.

Such decisions must be done at an appropriate level. If management levels are clearly identified then a natural workflow design decision would be to create regions in a way that reflects the management levels and associates certain privileges to trigger recoveries at each level. Consider a situation in which manager A, responsible for part α of the workflow, has a subordinate B, responsible for part β being a subpart of α. In this case A will be able to perform more general recoveries, involving parts of the workflow managed by B, because her/his recovery procedures can also reset the part β of the workflow. On the other hand, B can only handle exceptions for β without having permission to trigger recovery procedures outside β.

Such exception handling would require channels for communication ready for transmitting recovery decisions and a language clearly indicating the actions to be performed, since recoveries should follow some predefined patterns. Typically, the regions' structure will create ideal opportunities for the organization of message transmissions making them hierarchical.

5 Tool Support

To illustrate the viability of the approach presented in this paper, we have developed *HiWorD* (HIerarchical WORkflow Designer), a hierarchical Petri net tool using Java [9]. The tool has been successfully used to hierarchically design workflow scenarios in a safe and effective way from our project industrial partner Justwin Technologies Pty Ltd[2]. We plan to integrate the modeling tool with Justwin Workflow engine.

HiWorD is an editor tool that can be used to provide an efficient way to design and display workflows, by using the concepts of refinement rules and regions. In addition to traditional workflow modeling, *HiWorD* allows, starting from a single place, the refinement of places and transitions to create hierarchical workflows. The tool supports also the concept of region. A region is considered as a refined place in *HiWorD* for which one or several recovery transitions are added and each recovery transition is associated with an exception handler.

[2] http://www.justwin.com/.

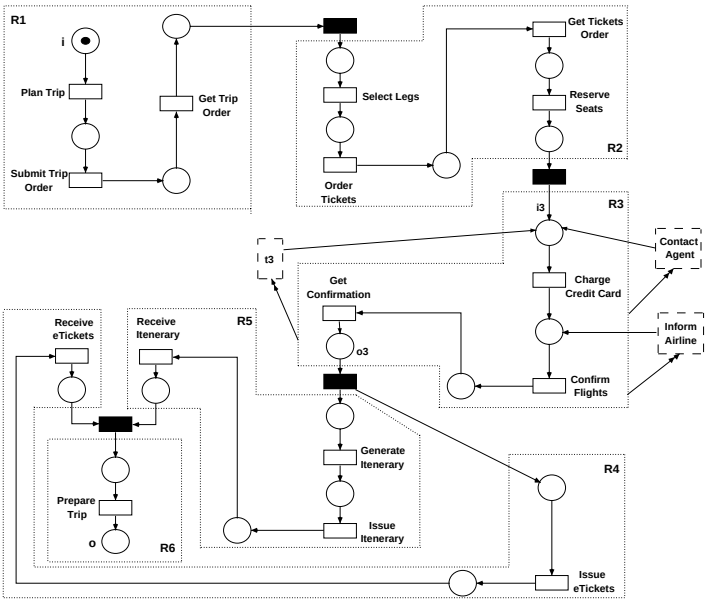

Fig. 9. Ordering Flight Tickets Workflow with Regions

This implementation has shown the validity of the approach and its advantages with regard to the traditional workflow design tools in handling and supporting hierarchical modeling. It should be noted that *HiWorD* can also be extended to interface with commonly used commercial workflow management systems such as Staffware and IBM MQSeries.

6 Conclusions

In this paper, we proposed a hierarchical model based on Petri nets to design workflows in a structured way. There are five basic refinement rules and three rules for adding communication and synchronization mechanisms. Sufficient conditions are proposed to ensure soundness after introducing any of the proposed constructs. Therefore, the workflow built according to the proposed rules is guaranteed to be a sound WF net with no risk of deadlock.

References

1. Georgakopoulos, D., Hornick, M., Sheth, A.: An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. Distributed and Parallel Databases **3** (1995)
2. WfMC: Workflow Management Coalition, Terminology and Glossary, Document Number WFMC-TC-1011. (1999) http://www.wfmc.org/standards/docs.htm/.

3. Chrzastowski-Wachtel, P.: Recovery Nets: Model for Dynamic Workflows. In: Proceedings of the 13th Workshop on Concurrency, Specification, and Programming, Humboldt University Report, Berlin, Germany (2002)
4. Aalst, W.v.d., Hee, K.v.: Workflow Management. Models, Methods and Systems. MIT Press, Cambridge, Massachusetts (2002)
5. Aalst, W.v.d., Hofstede, A.t., Kiepuszewski, B., Barros, A.: Workflow Patterns. Technical Report FIT-TR-2002-02, Queensland University of Technology, Brisbane, Australia (2002)
6. Aalst, W.v.d.: Verification of Workflow Nets. In Azema, P., Balbo, G., eds.: Proceedings of the Application and Theory of Petri Nets'97, Toulouse, France (1997)
7. Aalst, W.v.d.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers **8** (1998) 21–66
8. Desel, J., Esparza, J.: Free Choice Petri Nets. Volume 40 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, UK (1995)
9. Benatallah, B., Chrzastowski-Wachtel, P., Hamadi, R., O'Dell, M., Susanto, A.: HiWorD: A Petri Net-based Hierarchical Workflow Designer. In: Proceedings of the 3rd International Conference on Application of Concurrency to System Design (ACSD'03), Guimaraes, Portugal, IEEE Computer Society Press (2003)