Distributed Change Region Detection in Dynamic Evolution of Fragmented Processes

Ahana Pradhan and Rushikesh K. Joshi

Department of Computer Science and Engineering
Indian Institute of Technology Bombay,
Powai, Mumbai-400076, India.
Email:{ahana,rkj}@cse.iitb.ac.in

Abstract. Change regions approximate dynamic instance non- migratability through schema based approach. In a distributed process, each node may have a partial view of the structure of the whole process. The distributed structure-fragments may evolve independently at runtime. Established centralized algorithms to compute change regions cannot be directly applied to such a scenario, due to absence of a centralized view. A fully distributed algorithm working on distributed fragments to solve this problem is presented. First, a new centralized change region computation algorithm is developed and proved correct as a basis for the distributed approach. The distributed algorithm is itself presented as a Hierarchical Colored Petri net. An application case is also illustrated.

Keywords: Change region, Distributed process evolution, Dynamic migration, Fragmented processes, WF-nets

1 Introduction

Dynamic process migration approaches address the problem of adapting a running workflow process instance of an old schema correctly to a new schema. The instance after migration needs to be *consistent*. In the context of adaptive systems, this problem is referred to as the *state transfer problem* [1]. Between a pair of nets the problem is seen as instance to instance migration. In the domain of business processes, this problem is scaled up to a problem for a mass of workflow instances, so that all active automated instances are evolved while they are under execution in a workflow engine. Many theoretical approaches for dynamic workflow evolution have been developed since the mid-nineties including the approaches by Ellis et al. [2], Van der Aalst et al. [3] and Sun et al. [4]. Recent BPM solutions include support for process flexibility, as in YAWL [5] and Aristaflow [6], which provide centralized approaches for dynamic migration.

Dynamic evolution is inevitable in an ever-changing business environment, yet the technical support for the same is far from mature, especially for distributed processes. As pointed out by Protogeros et al. [7], due to non-solidified

protocols among the parties, problems such as indefinite waits, incorrect invocations, which are typical to distributed systems may result, in addition to inconsistency. Autonomous evolution of parties pose additional challenges of conflicts.

The area of evolving distributed processes has been in focus quite recently due to current trends of automation in the inter-organizational collaborative processes. Recently Zaplata et al. [8] point out that the root of the difficulty in making changes in a distributed process is the *fragmentation of the schema*, due to which, the global view of the process is lost. The *change region* based approaches of (i) decentralized instance migration by Cicirelli et al. [9], and (ii) dynamic evolution of fragmented workflow process by Hens et al. [10] adopt a centralized process in the distributed setup. Therefore, a fully distributed approach of dynamic change remains to be theorized, which is the focus of our paper. The difference between the algorithm presented in this paper and the work of Hens et al. is that our algorithms compute the change region in a fully distributed environment without needing the centralized change manager.

However, the adaptation of such a theory in practice will need to face further challenges of many practical issues such as lack of consensus among different execution ends, contradictory independent changes, and unexpected delays in communication. Nevertheless, an algorithmic approach to evolve a fragmented process contributes a solution to the core of the bigger problem structure. The paper develops an algorithm for the distributed computation of the change region for a fully fragmented process schema of structured workflows. In this model, an execution node does not posses the knowledge of the full schema and the structural changes on other nodes. The change region is computed by making local decisions and through communication of minimal information. Such a change region determines migratability of the running instances, different parts of which may be live under different workflow engines in the distributed nodes.

In the paper, firstly, a formalization of the change region is presented for structured workflows, followed by the novel concept of *conjoint tree* (Section 2), which is used in formulating an algorithm for computing change region on centralized process schema (Section 3). Over a schema fragmentation model (Section 4), the centralized technique is adopted to develop and prove the distributed computation of change region (Sections 5 and 6 respectively). A practical example is also discussed to illustrate the applicability of approach.

Reason to use Change Regions: *Change region* provides a structural approximation of non-migratability given a migration net pair. To elaborate, change region is a region in the old schema, from where the old state cannot be transferred into the new schema within the framework of the chosen consistency model. When an instance state is outside such a region, its old state is a valid state in the new schema, and hence, consistent migration is ensured. Thus, change regions when known ahead of execution permit smooth maneuvering of transfer of process control-flow into a new schema. This approach was formally introduced by Van der Aalst [11] to eliminate the cost of instance-by-instance state-space exploration by replacing it with one-time computation of change regions.

Related Work on Change Regions: A preliminary notion of *change region* was given by Ellis et al. [2] in their early work on dynamic workflow change. Considering *marking reachability* as the consistency criteria, which is followed in this paper, the notion of change region was introduced by Van der Aalst [11]. The term *dynamic change region* was defined as the subnet in the old net, which, when is completely unmarked in a marking, guarantees consistent state transfer of that marking into the new net. Their work presented an algorithm to identify the smallest single-entry-single-exit (SESE) block (sometimes referred to as the minimal-SESE region) covering the *structural changes* performed on the net as the change region. Several other approaches to compute the minimal-SESE change region are found in the literature. Gao et al. [12] perform the same objective on Special Net Structure (SNS) models. Sun et al. [4] present a variant of the algorithm given by Van der Aalst [11] as an attempt to obtain a smaller region. Zou et al. [13] use the process structure tree of workflow graphs [14] to compute SESE change regions. In contrast, our approach does not require the region to be a strict SESE-block, and hence is applicable to fragmented nets where SESE-blocks may not even exist.

Related Work on Fragmented Process Migration: The recent literature in distributed process migration uses the minimal-SESE regions originally proposed for centralized workflows. Cicirelli et al. [9] adopt it for *decentralized migration*. Whereas distributed execution of different tasks at different physical locations are permitted, a centralized view on the whole process is required in their approach for computing the change region. Their mechanism facilitates independent migrations in the execution ends concurrent to each other, rather than mandating the whole instance migration at once. The recent works of Hens et al. [10], [15] apply the SESE approach to evolve fragmented processes, where the change region is first computed on the global process. For applying the knowledge of change region to migrate instances, a centralized change manager is considered to coordinate among the fragments of the live instances. The change manager constructs global state views of the running instances by collecting information from the live fragments, and then supervises the dynamic migration. Summarily, these approaches consider independent migrations based on scattering of blocks in distributed environment, but do not compute the change region through a fragmented schema devoid of a centralized view. The algorithm we present considers distributed decision making in presence of fragmented schema.

2 The Foundations

Block-Structured WF-nets Application of Petri nets for workflow modeling was introduced by Van der Aalst [16] through the class of Workflow nets, or WF-nets. WF-nets are elementary nets, model control flow of workflow schemas, whereas a marked WF-net models a workflow instance. A WF-net has a single source place and a single sink place. A token only in the source place is the initial marking and a token only in the sink place is the terminal marking. A

sound WF-net, once started with its initial marking, always reaches the terminal marking, and does not have any dead intermediate state or dead transition.

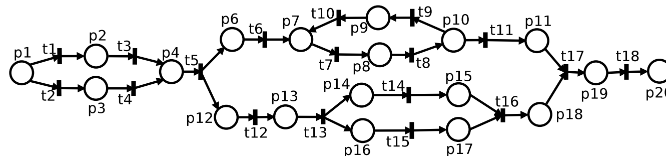
The scope of the work is characterized by ECWS-nets, a class of structured WF-nets composed of the basic primitives of sequence, exclusive-choice, concurrent blocks and iterative blocks. The blocks are sound by construction. The following ECWS is a grammar built over a convenient "string" based representation to ease specification and programming. More importantly, ECWS precisely represents the class of nets targeted in the paper. ECWS includes place labels and iterative blocks over the CWS language introduced in our earlier work [17].

Net → *Pnet*
Pnet → **Place**
  | *Pnet* **Trans** **Place**
  | *Pnet* **Trans** *loop* **Trans** *Pnet*
  | *Pnet* **Trans** *and* **Trans** *Pnet*
  | *Pnet* *xor* *Pnet*

loop → *{ Pnet } { Tnet }*
xor → *[Tnet] [Tnet]*
  | *[Tnet] xor*
and → *(Pnet) (Pnet)*
  | *(Pnet) and*
Tnet → **Trans** | **Trans** *Pnet* **Trans**

In the ECWS, the branches in an exclusive-choice blocks are enclosed in square brackets such as in [A][B], whereas round brackets are used for enclosing concurrent block such as in (A)(B), where A and B are ECWS subnets. A loop with execution path A BA BA ... is expressed with curly brackets as {A}{B}. Sequence of net-elements or blocks does not use spatial delimiters. As an example, the ECWS specification for the net in Fig. 1 is specified in the figure itself.

Consistency Criteria and Instance Migratability An instance state is a marking in the WF-net model of a given process schema. Migratability of an instance determines whether the corresponding state of the old workflow is also *reachable* in the new workflow. This state to state migration correspondence is termed as the *consistency criteria*, which was used by earlier researchers [11], [9], [15]. The consistency criteria ensures that, an old workflow instance can carry on to follow the new schema from its current state without having to encounter any control-flow error till termination. In business terms, the question becomes, *'can the process migrate safely and correctly with same status, place labels reflecting the status'.* As an illustration, in some context of a technical conference process, one can think of a state *Paper Accepted* as not migration equivalent to state *Paper Accepted, Author Registration Confirmed.*



p1 [t1 p2 t3][t2 p3 t4] p4 t5 (p6 t6 {p7 t7 p8 t8 p10}{t9 p9 t10} t11 p11)(p12 t12 p13 t13 (p14 t14 p15)(p16 t15 p17) t16 p18) t17 p19 t18 p20

Fig. 1. Example of a Structured WF-net

A marking in the old net is non-migratable if it is unreachable in the new net from the corresponding initial marking. In this context, a change region is a schema based approximation of the non-migratability, i.e. any marking having a place in the change region is considered to be non-migratable.

In contrast with the SESE change regions discussed previously, we define the change region as a set of places instead of subnets with transitions included. This definition simplifies the collection in terms of places only without any loss of accuracy of the region. The transitions are thus overlooked for investigation of migratability, since marking reachability requires the knowledge of only the places. The definition of consistency and change region are as follows considering a WF-net as a tuple (P, T, F) of sets of places, transitions and the arcs.

Let the old WF-net be $N = (P, T, F)$ and the new net be $N' = (P', T', F')$ with the respective initial markings M_0 and M_0', each consisting of only the source place of the respective nets of N and N'. Also, let $\mathbb{R}(m)$ be the set of markings reachable from a marking m, where each marking is a set of places.

Definition 1 Consistency: *A marking $M \subseteq P$ in the old WF-net N is consistent with a marking $M' \subseteq P'$ in the new WF-net N' iff $M = M'$.*

Definition 2 Migratability: *A marking M is migratable iff $M \in \mathbb{R}(M_0)$ and $M \in \mathbb{R}(M_0')$. A non-migratable marking M is reachable in the old net but not in the new net, i.e., $M \in \mathbb{R}(M_0)$ and $M \notin \mathbb{R}(M_0')$.*

Definition 3 Change Region: *A subset of places in a given WF-net N defines the change region denoted by $CR(N, N')$ for the old net N against a new WF-net N', if it covers all non-migratable markings. In other words, the change region includes at least one place from every non-migratable marking, i.e., $M \in \mathbb{R}(M_0)$ and $M \notin \mathbb{R}(M_0') \Rightarrow M \cap CR(N, N') \neq \emptyset$, i.e. non-migratable marking $\Rightarrow$ place overlap with change region.*

Following the definition, by taking contrapositive, we can observe that a marking in the old net that does not have a place in common with the change region is directly migratable. It can be noted that the above definition of change region permits overestimates since it permits a marking M to be part of change region and yet be reachable in the new net i.e., $M \cap CR(N, N') \neq \emptyset$ in the old net and $M \in \mathbb{R}(M_0')$ in the new. Ease of computation is at the cost of overestimates.

Conjoint Tree (C-tree), Generator of Concurrent Submarking (GCS): A *C-tree* of a given net is a tree capturing concurrency of sets of places covering the entire net. Consequently, it embeds all reachable markings. A node in the tree holds places which are component of a sequential branch. The nested structure of AND-blocks is captured through inner *C-block* elements (denoted by □ symbol) in a parent node, pointing to child C-trees. One *C-block* points to a single AND-block. Notably, places in XOR-blocks, loops and sequences do not form a hierarchy on their own. Construction of the C-tree of a net can be easily achieved from its ECWS specification. It can be noted that, a C-tree is

Algo. 1: Centralized Computation of Change Region

Input: C-trees C of N, C' of N'
Output: Subsets CR and $safe$ of places in N w.r.t. N'

```
1  CR_1 ← places(C) − places(C')
2  CR_2 ← places(root of C') ∩ places(C except the root)
3  CR_3 ← places(root of C) ∩ places(C' except the root)
4  CR ← CR_1 ∪ CR_2 ∪ CR_3
5  safe ← places(root of C) − CR
6  while CR ∪ safe ≠ places(C) do
7      pick the next p from set places(C) − (CR ∪ safe)
8      LC_p ← places(GCS(p, C)) − places(GCS(p, C'))
9      CR ← CR ∪ LC_p
10     CB_p ← {q|p, q co-occur in a node in both C and C'}
11     if root C-block in GCS(p, C) has less number of branches than root C-block
       in GCS(p, C') then CR ← CR ∪ {p} ∪ CB_p
12     else safe ← safe ∪ {p} ∪ CB_p
```

not a process tree as in [18], since it captures only the nesting of concurrency. Fig. 2 shows the C-tree of the ECWS-net given in Fig. 1. A marking can be constructed from the C-tree by selecting one place from each node, by covering all concurrent branches w.r.t. the node. For example, $\{p_1\}$, $\{p_9, p_{12}\}$ are valid markings, but $\{p_{11}, p_{16}\}$ and $\{p_1, p_{16}\}$ are not. Given a C-tree C and a place p in it, GCS(p,C) is only the subtree in C that is concurrent to p and excluding p. For example, GCS of p_{16} in Fig. 1 is shown in Fig. 2. Subtree GCS(p,C) is obtained by removing from C-tree C those places in the entire ancestry and postgenitus of place p that are not concurrent to p, including p. Thus, GCS retains only the places which can be concurrently marked with p in a reachable marking.

3 Dynamic Migration of Centralized Workflows

The centralized algorithm (Algo. 1) works on the C-trees of the old and the new nets and produces the change region as a set of places. The algorithm is designed based on the principle that it is sufficient to include in the change region at least

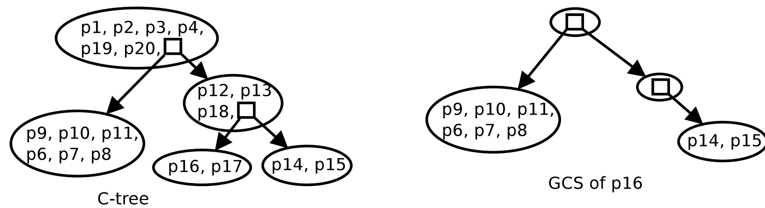


Fig. 2. C-tree for the ECWS-net shown in Fig. 1

one place from every non-migratable marking. Intuitively, it works by finding and including five kinds of places in the change region listed below.
C1: Places which are deleted from the old net (line 1)
C2: Places which are concurrent in the old but not the new net (line 2)
C3: Places which are concurrent in the new but not in the old net (line 3)
C4: Places inside a concurrent block which lose their concurrency w.r.t. at least one place inside the same concurrent block (line 9)
C5: Places inside a concurrent block that gain additional concurrency (line 11)

The algorithm to implement the above scheme is given in Listing Algo 1. It uses two functions `places` and `GCS`. Function `places(n)` returns the set of places present in the net structure n (which can be identified by a full C-tree, its subtree, or a GCS). In the algorithm, sets CR and $Safe$ are respectively used to contain the places inside and outside the change region. Line 4 builds up the initial set for CR following C1-3 by collecting the places which are either removed or which either gain or lose concurrency. Line 5 builds up the initial set $Safe$ collecting in it the places which are non-concurrent in both the nets.

The while loop iterates over all the unmarked nodes of the C-tree by picking the next place p (line 7) in each iteration. In each iteration, more than one places may be marked as members of either set CR or of set $Safe$. The loop terminates when both the sets together cover the entire set of places (condition on line 6).

The loop continues to grow set CR by collecting the places which are concurrent relative to p in the old but not in the new net (lines 8,9). Set LC_p gives the set of places of *lost concurrency* w.r.t. p. LC_p is computed by subtracting from the set of places involved in the concurrent submarkings of p in the old net the set of places involved in non-concurrent submarkings in the new net (line 8). These are added into set CR (line 9).

Set CB_p gives the set of places which are in the same *concurrent branch* (i.e. the same C-tree node) as p in both the nets, and have remained intact in the node with p after the structural changes from the old net to the new net (line 10). If p experiences an increased degree of concurrency by addition of a whole new branch in the concurrent block, p and CB_p are included in set CR (line 11). Otherwise, p and the places in set CB_p are marked as safe (line 12). It can be noted that, sets LC_p and CB_p are disjoint.

A case may arise where the root C-block in `GCS(p,C)` has same or more number of branches than that of the root C-block in `GCS(p,C')` where p and CB_p are involved in non-migratable markings due to local rearrangements or removal of places from the rest of the concurrent places. Both these cases are covered by inserting the responsible places into set CR on lines 1, 2 and 9. Thus, marking p and CB_p as safe (line 12) optimizes set CR in such a case.

Proof of Correctness: The correctness of the algorithm is proved by showing that its output set CR satisfies Def. 3 of change region. Assume the contrary, that there is a non-migratable marking of which not a single place belongs to set CR, i.e. $\exists M \in \mathbb{R}(M_0)$, $M \cap CR = \emptyset$, $M \notin \mathbb{R}(M'_0)$. If marking M is not migratable, it must satisfy one of the cases in Table 1. These are the exhaustive

Table 1. All Cases of Non-migratable Markings

Case No.	Structure of non-migratable marking M	Structure of markings in N' that overlap with M	Concurrency of p in New
(i)	$\{p\}$: Non-concurrent	no marking includes p	Absent (p is deleted)
(ii)	$\{p, ...\}$: Concurrent	no marking includes p	Absent (p is deleted)
(iii)	$\{p\}$: Non-concurrent	$\{p, ...\}$, but $\{p\}$ not available	Concurrent (p moves in an AND-block)
(iv)	$\{p, ...\}$: Concurrent	$\{p\}$, but $\{p, ...\}$ not available	Non-concurrent (p moves out of AND-block)
(v)	$\{p, ...\}$: Concurrent	$\{p, ...\}$, but same set not available	Concurrent (changes in other branches)

cases of non-migratability when seen through concurrency. The only case that it does not list is that of the combination *non-concurrent* $\times$ *non-concurrent*, which is migratable (line 5), since it is a case of a standalone place reachable in both.

It can be seen from the concurrency properties that for cases (i)-(iv), place $p \in M$ is put in set CR by line 1-4 of the algorithm (C1-3), thereby leading to a contradiction. In case (v), as per the assumption, M has been declared non-migratable, therefore an M' which includes the common member p must follow one of the three possibilities: (i) $M' \subset M$, (ii) $M \subset M'$, (iii) M' is neither subset nor a superset of M but $M \cap M' \neq \emptyset$. It can be noted that the case $M' = M$ is contrary to the assumption. In possibility (i) and (iii), M' misses at least one member q of M. Therefore, place q is concurrent with p in N, but is not so in N' though it may be present elsewhere in net N'. Therefore, q must have been put in set CR by line 9 of Algo. 1 (C4), leading to a contradiction. Possibility (ii) implies an increased degree of concurrency for p. In other words, p requires at least one additional places to be marked along with along with it in the new net N'. In line 11 of the algorithm, p is put in set CR since it has an increased number of concurrent branches in the GCS subtree of the new C-tree w.r.t. that of the old C-tree, thereby implying its increased degree of concurrency (C5), which leads to a contradiction. Hence the proof.

Cost of the Algorithm: The cost of our algorithm involves a series of set operations (lines 1-5), and a loop iterating over all places in worst case. Inside the iteration, a few set operations are involved. The complexity of the C-tree based change region algorithm is $O(n^2 \ log \ n)$ if n is the number of places.

4 The Model of Fragmentation for Distributed Schema

In a distributed environment, a workflow schema is assumed to be fragmented among different execution nodes. No single end of execution has total knowledge of the process schema, i.e., no single participant possesses a global view of the workflow state at any point of time. Local changes to different fragmented

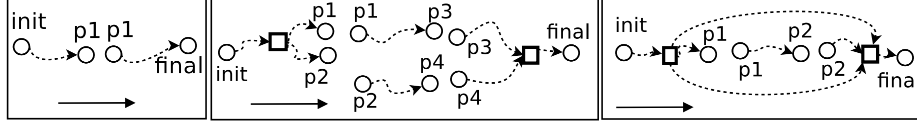


Fig. 3. Fragmentation Scheme

schema result in evolutionary changes to the whole process schema. Due to the fragmented nature of the schema and this localization of changes, simple local application of the algorithm discussed previously is not sufficient. The distributed adaptation of the algorithm develops around a model of fragmented workflows and sharing of local change region information among the fragments.

A fragment is a connected subnet having places and not transitions in its boundary. In fragmentation, boundary places are shared among fragments which keep them connected. If a place in a fragment has more than one incoming or outgoing transitions, all of them are included in the fragment. As a fallout, the boundary places can also be from an inner part of a SESE-block in the fragment. However, the boundaries must be singleton places with exactly one incoming and exactly one outgoing transition in the global net. Consequently, the scheme allows for sequential, concurrent and nested fragmentation as shown in Fig. 3. A *fragmentation* partitions the net into a set of *fragments*. Using the WF-net notation given in Section 2, these two notions are formally defined next.

Definition 4 Fragment: *A* fragment *f of a ECWS-net* $N = (P, T, F)$ *is a net* $f = (P_f, T_f, F_f)$ *which adheres to the following properties.*

1. A fragment is a subnet - $P_f \subseteq P, T_f \subseteq T, F_f \subseteq F$
2. The fragment subnet is a fully connected graph without local partitions.
3. A fragment includes all the pre-places and post-places of all its member transitions - $\forall t \in T_f$ s. t. $\bullet t \cup t\bullet \subseteq P_f$
4. Every input boundary place in a fragment must have exactly one pre-transition in the global net: $\forall p \in P_f \; \bullet p \cap T_f = \emptyset \Rightarrow |\bullet p \cap T| = 1$
5. Every output boundary place in a fragment must have exactly one post-transition in the global net: $\forall p \in P_f \; p\bullet \cap T_f = \emptyset \Rightarrow |p\bullet \cap T| = 1$
6. If a place in the fragment has multiple pre- or post-transitions, they belong to the same fragment - $\forall p \in P_f$, $|\bullet p| > 1 \Rightarrow \bullet p \subseteq T_f$, $|p\bullet| > 1 \Rightarrow p\bullet \subseteq T_f$.

A net is fragmented by creating multiple subnets such that when all the fragments are put together the original net is formed. Global source and sink are *not boundary* places. Fragments do not have common transitions. However, adjacent fragments need to have common boundary places to pass tokens around through fragments. The scheme can be applied to nets constructed by the ECWS grammar. Fig. 4 shows an example of a valid fragmentation of the net given in Fig. 1 in presence of a loop. The loop through place p_9 spans two fragments. The direction of token flow through the six fragments of this process is also shown in the figure. Fragments may be located on different machines. The scheme of

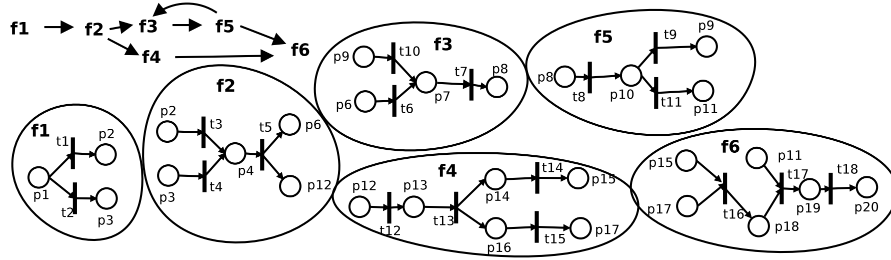


Fig. 4. A Valid Fragmentation of Net in Fig. 1

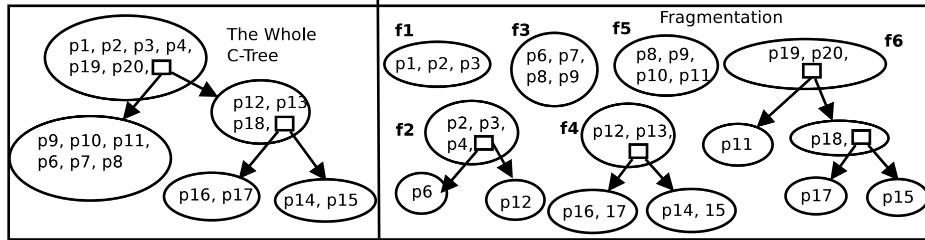


Fig. 5. Fragmentation of C-tree corresponding to Fig. 4

fragmentation prevents any overlap between two fragments apart from just the *singleton* boundary places as points of contacts.

Definition 5 Fragmentation: *A fragmentation of a given ECWS-net* $N = (P, T, F)$ *is a set* $R = \{f | f = (P_f, T_f, F_f)$ *is a fragment of* $N\}$*, such that it satisfies the following properties.*

1. All the fragments in the fragmentation together form the complete net - $P = \bigcup_{f \in R} P_f, T = \bigcup_{f \in R} T_f, F = \bigcup_{f \in R} F_f$,
2. No two different fragments in the fragmentation share any transition: $\forall f, f' \in R, f \neq f', T_f \cap T_{f'} = \emptyset$.

Fig. 5 shows the corresponding fragmentation of the global C-tree showing six fragmented C-trees. Only the tree nodes get partitioned into different fragments, and the C-blocks preserve the cardinality of their branching which is due to condition 3 of Def. 4 that makes it mandatory for all forking/joining element to hold all its forked/joined branches together. In other words, a C-block element may repeat in multiple partitions, but the branches of a C-block cannot be partitioned and at least one place must be present in every partition.

Though the C-trees of each of the fragments in the example are formed from the global C-tree, they can be constructed from the respective fragments independently. It can be noted here that a fragment is a subnet but may not be a complete WF-net by itself, as it can be observed in the case of fragments f_1 to f_6 in Fig. 4. When a local structural change is made to a fragment, it assumes

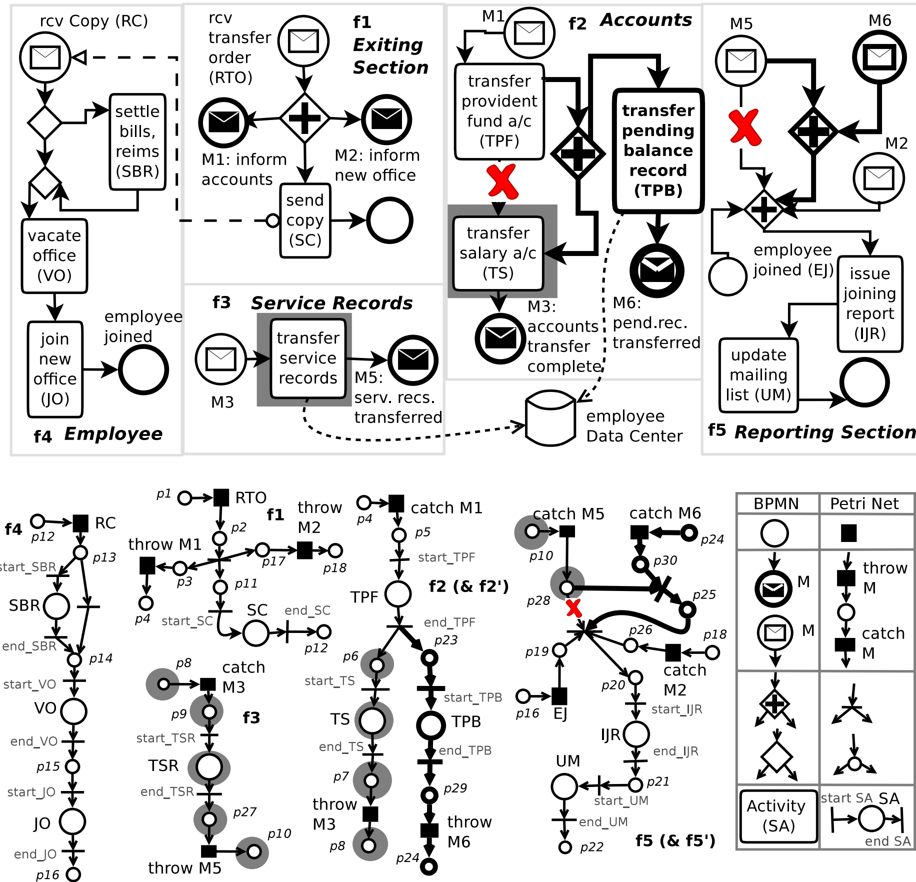


Fig. 6. BPMN process and its WF-net obtained by a BPMN to Petri net Mapping

that the change does not violate global well-formedness of ECWS structure and the local C-tree is available. The local C-trees can be constructed by adapting the C-tree construction of Section 2.

Practical Example of Fragmentation: Fig. 6 shows a distributed collaborative BPMN *employee transfer* process fragmented at five locations. When an employee is transferred from one administrative unit to another, this process is executed. The fragments correspond to five pools operated by five actors: (f_1) Exiting Section, (f_2) Accounts department, (f_3) Service records, (f_4) Employee, and (f_5) Reporting Section. Clearly, none of the fragments has the global view of the process. This process has now changed. In the new process, fragments *Accounts* (f_2) and *Reporting Section* (f_5) have changed their internal structure.

The fragmented ECWS-net model of the process is also depicted in the figure. Translation from BPMN to Petri net is a widely researched area [19], from

which we adopt a translation for basic primitives that are of interest to us as shown in the figure. It can be noted that a BPMN activity is modeled as a sequence of *start transition*, *a place* and an *end transition*, rather than as a single transition, in order to bring it in the fold of place-based change regions. When the corresponding place is a member of the change region, it implies that the real-world BPMN activity is in the change region. In the figure, the respective net elements that are not removed are shown in thin lines, deleted elements are crossed, and additions are shown in think lines.

5 The Distributed Algorithm

Schema Change Specification: Independent schema change specifications are given per fragment. Thus, individually they are local. They are together assumed to preserve the well-formedness of the global net. C-trees of the respective fragments are available locally. *Once a place is assigned into a fragment, the change specification does not move it to another fragment.* In other words, if a place is absent in a fragment after a structural change, it cannot be found in any other fragment, which makes it a deleted place. As a consequence, a boundary place in a fragment can not become an internal place in that fragment, since it would lead to removal of that place from the peer fragment that shares it. The algorithm uses the the following symbols: f is the old local fragment, f' represents the new fragment after the local structural change. If no change is performed on f, the value of f' is considered to be `null`.

A High-level Overview of the Algorithm: The algorithm is depicted as a Hierarchical Colored Petri net in Fig. 7. This and the rest of the models have been constructed using CPN Tools [20]. The algorithm proceeds in two phases. The first phase uses three modules. It starts with initiation round (IR), which implements a rendezvous among the fragments over change specifications. Next, module ICBBN computes the local change region and shares its boundary status (safe or unsafe) with peer fragments through event broadcasts. Incoming boundary status from peers is concurrently handled through module RcvBN. The second phase uses two modules. Here, the conflicts between local and peer decisions are resolved iteratively, till termination condition is reached.

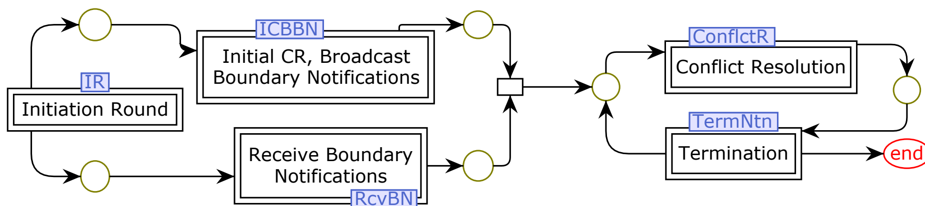


Fig. 7. Phases of the Algorithm

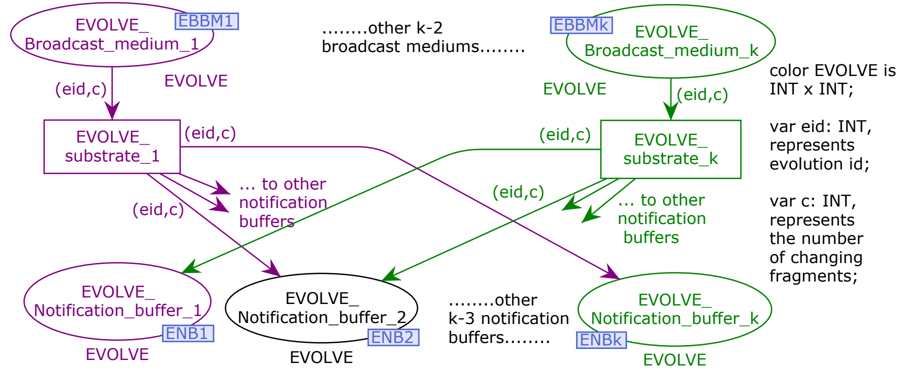


Fig. 8. CPN Model of the Substrate for `EVOLVE` event

Inter-Fragment Event Communication: Communication among the fragments is through a lossless publish-subscribe event-*substrate*. Between a pair of fragments event-messages are ordered. Every action in the algorithm logic is guarded by either the occurrence of an event or by a precondition, or by a combination of both. One instance of the algorithm runs on one fragment and this bundle is called a *Logical execution node*. Number k, the total number of logical execution nodes in the environment is known to every logical execution node. However, it is possible to combine multiple logical execution nodes on a single physical machine, which is a non-algorithmic deployment issue.

Each event type is assigned a color declaration. There are five event types, which are `EVOLVE`, `CR_B`, `SAFE_B`, `CHANGE` and `NOCHANGE`. Fig. 8 depicts the CPN model of the part of the substrate covering the connections for one event type `EVOLVE`. The substrate makes these connections for each event type. The event is published at place *broadcast medium*, and after the broadcast through the substrate, it is received at place *notification buffer* in a peer fragment. The bound variables on the arc inscriptions carry the parameters of the events. The substrate is thus a simple value passing CPN implementing point-to-point broadcast.

Initiation Round (IR): As shown in Fig. 9, the initial round implements a rendezvous to bring every node into the change region computation. The round is initiated by all fragments where change is required. A structural change request arrives from some external user to every *changing fragment* in the form of event `INITIATE` in place *INB*. A change request carries (i) f' the new fragment, which is stored in place *NF*, (ii) count c of the number of fragments which are changing in this round of evolution, and (iii) *eid*, a global evolution id, which is an unused provision for withholding the next evolution till the current one completes.

A node receiving `INITIATE` publishes in place *EBM* a broadcast of `EVOLVE` carrying only the *eid* and count c. This broadcast brings about the involvement of non-changing fragments into the change region computation. In place *ENB*, every changing and non-changing fragment receives from peers a total of $c - 1$

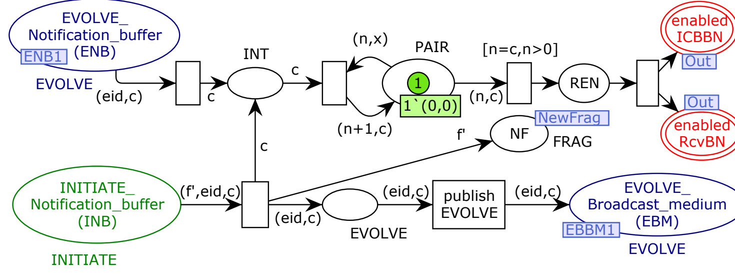


Fig. 9. Initiation Round (Module **IR**)

and c EVOLVE tokens respectively. Fig. 9 implements the common IR module unifying the rendezvous logic of both types of participants. The rendezvous is achieved in place REN when a total of c notifications are received. The value c is known only when the first notification arrives either from ENB or INB.

Initial CR, broadcast boundary notifications (ICBBN): As shown in Fig. 10, after the rendezvous, every fragment computes its local change region by applying function `computeCR` (which uses Algo. 1) given in Table 2 on inputs f and f' (old and new fragments). It produces a local result $CR \times SAFE$ (color $CRnSAFE$). From here, one branch computes local set CR and the other computes local set $SAFE$. After computing CR, the boundary places in local CR are identified as intersection of places in CR, and ShB, the known set of boundary places. The boundary places in CR and $SAFE$ are published through events CR_B and SAFE_B respectively, after which this module completes. It can be observed from the figure that the net puts tokens from CR (unsafe) and $SAFE$ back into their respective places since they are needed later in the algorithm.

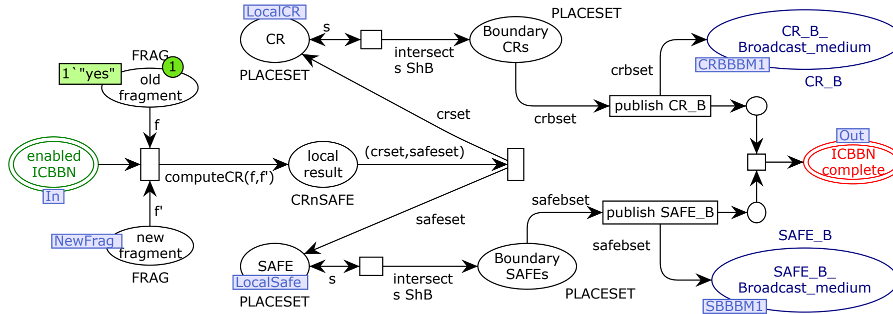


Fig. 10. Initial CR, broadcast boundary notifications (Module **ICBBN**)

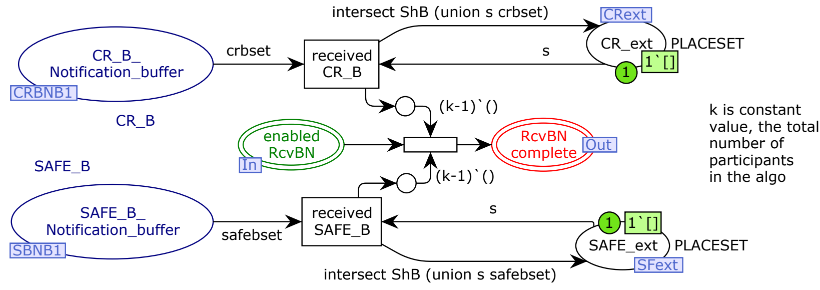


Fig. 11. Receive boundary notifications (Module **RcvBN**)

Receive boundary notifications (RcvBN): As shown in Fig. 11, set CR_ext stores the set of boundary places which are announced to be unsafe by peers. Whenever a set of unsafe boundary places is notified through the notification buffer, set CR_ext is updated. Similarly, set $SAFE_ext$ stores and updates the set of boundary places upon receiving `SAFE_B` notifications. Sets CR_ext and $SAFE_ext$ are used later. This module completes after it receives and processes notifications about safe and unsafe boundaries from each of $k-1$ peer fragments.

Conflict Resolution (ConflctR): The module is shown in Fig. 12. When this round is enabled, intersection of local safe ($SAFE$) and external unsafe (CR_ext) is computed. If this result is nil (arc 1\`[ ]), event `NOCHANGE` is published and a *no change* count is incremented (`NOCHANGE` counter), else the result (conflict set) is bound to variable a.

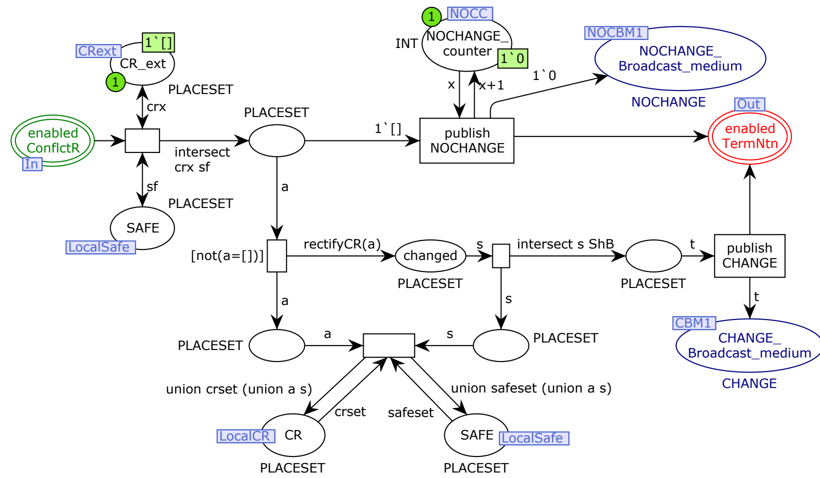


Fig. 12. Conflict Resolution (Module **ConflctR**)

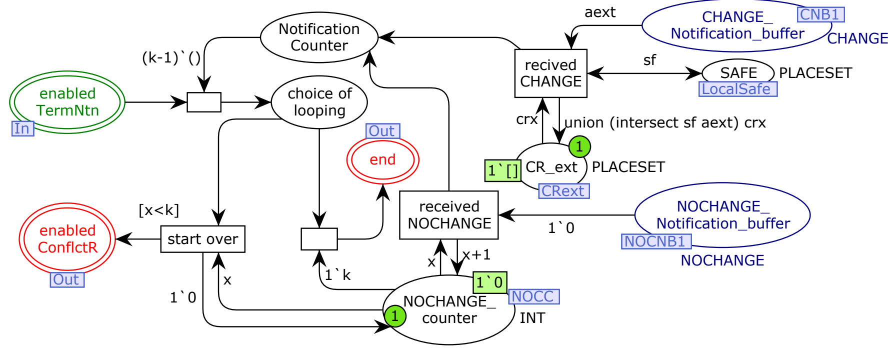


Fig. 13. Termination (Module **TermNtn**)

Table 2. Procedures used in the Algorithm

`computeCR(`f,f'`)`	`rectifyCR(`$Conflict_set$`)`
if $f' =$ `null` **then** $CR \leftarrow \{\}$; $Safe \leftarrow$ all places in f; **else** $CR, Safe \leftarrow$ output from Algo. 1 for f and f';	$CB_p \leftarrow \{ q \mid q \in Safe, p \in Conflict_set,$ q co-occur with p either in the same node or in a descender node in both C-trees of f and $f'\}$ **return** CB_p;

As a result of the conflict, some additional locally safe places may become unsafe, which are bound to variable s after they are identified by function `rectifyCR`. Boundary places in set s are published through event `CHANGE`. CR and $SAFE$ are recomputed using their previous values and a and s. The C-tree based algorithm of function `rectifyCR` is shown in Table 2, which has been detailed in the proof given in the next section.

Termination (TermNtn): Fig. 13 shows the corresponding CPN model. After `CHANGE` or `NOCHANGE` broadcast, the fragment waits for similar event notifications from every other node. If the node has broadcast `NOCHANGE` due to no conflict in the previous module and received $k-1$ `NOCHANGE` notifications (value 1`$k$ in NOCHANGE counter), the termination condition is reached. Otherwise, set $CR_ext$ is updated with the parameter places of `CHANGE` notifications and the conflict resolution phase is again re-triggered. Place *Notification counter* in the net keeps track of total number of `CHANGE` and `NOCHANGE` notifications received in one round. Place *choice of looping* provides an exclusive choice between termination of the algorithm and pursuing the conflict resolution round again.

Cost: The initiation round results in c broadcasts. After local computation of CR, the initial sharing of place status results in $2k$ broadcasts. The conflict resolution phase involves k broadcasts in each round, with at most s rounds, where s is the number of *safe* boundary places judged locally by `computeCR`s.

6 Correctness of Working Through Fragmentation

Lemma 1 proves that place deletion does not cause a conflict. Lemma 2 proves that change in concurrency is detected at least in one fragment. Due to fragmentation, the change in concurrency may not be visible in all the affected fragments due to lack of knowledge of the global GCS. If a fragment, locally safe, has changed concurrency globally, at least one of its boundary comes to know about the change through status notifications from sharing fragments if the boundary in the peer fragment is able to detect the change by local GCS. This gives a case of conflict in the fragment which is not able to see the global GCS, which causes the fragment to call procedure `rectifyCR`. This procedure includes the locally safe boundary in CR. Since the reason of conflict is known to be change in concurrency only, the places in the same AND-branch (same or deeper nesting) also gets affected in the same way as the boundary. Therefore, `rectifyCR` puts all of them in CR. Any newly unsafe boundary has to propagate this rectification in the same manner until all non-migratabilities get covered by this branch. Conflict resolution messages are propagated from one fragment to another until the conflict stabilizes.

Lemma 1 *The effect of deletion of a place in the change region in a fragment is fully covered by the fragment in which the place is deleted.*

Proof: If the deleted place is not shared between two fragments, it is inserted in the change region of only its container fragment by procedure `computeCR` which directly uses Algo. 1 on the fragment and its new replacement. If the deleted place is a boundary shared between two fragments, it is inserted into the change region by both the fragments by procedure `computeCR`. It is assumed that the new f's corresponding to the two fragments remain consistent in deleting the globally deleted place. Further, a boundary place does not become an internal place as per the assumptions. Consequently, deletion of a place in a fragment does not create a conflict with the change region locally identified by some other fragment. The lemma covers cases (i) and (ii) in Table 1.

Lemma 2 *If there is any change in concurrency of a place in the global net, some fragment detects it.*

Proof: The cases of change in concurrency are the three cases (iii), (iv), (v) in Table 1. We need to prove that a change in GCS of the place p in contention is locally detectable by at least one fragment in all these cases.

In case (iii), the global gained concurrency of p can happen by either by adding new concurrent branches and gateways around p, or by moving p in an existing AND-block globally. In the former sub-case, an existing transition t_1 upstream to p with its only post-place q (which can also be p) is expanded with additional post-places. Similarly, downstream transition t_2 with its only pre-place q' (which can also be p) additionally joins with new pre-place. As per the fragmentation property 3 of Def. 4, since a transition cannot be a boundary, t_1, q and the new fork-places are co-located, and t_2, q' and the new join-places are

co-located. So the respective fragments detect the change in concurrency for q and q' in `computeCR` after the initial rendezvous. This is because the procedure generates a non-empty GCS for q and q' in both their host fragments, though p may be located in those fragments (lines 11-12 of Algo. 1).

In the latter sub-case, a part of the the AND-block in reference must be in the same fragment as that of p since none of them can move out. Also, for p to move in the AND-block, either the fork or the join gateway of this AND-block must be in the same fragment, else the fragment becomes disconnected. Due to the existence of a concurrent gateway, `computeCR` creates a local non-empty GCS for p, which detects the gained concurrency (lines 11-12 of Algo. 1)

The argument for case (iv) is exactly a reverse one with the sub-cases of removal of AND-branches which need not be located in the same fragment, or by a movement of p out of an AND-block somewhere in a sequence without an enclosing AND-block in the same fragment. In this case, with the reverse argument for its both sub-cases, the procedure `computeCR` detects the loss of concurrency by observing differences in the GCS (lines 8-9 of Algo. 1).

Case (v) occurs when (a) p moves in an existing inner AND-block or an existing outer AND-block or jumps into another branch of the same block, or (b) the AND-block in which p occurs is modified by either removal or addition of branches anywhere in the nesting but by maintaining p to be concurrent. Thus we get two sub-cases. In the first sub-case, the part of the AND-block where p is moving to is in the same fragment as that of p. Hence, one of the fork and join gateways that p crosses must be in the fragment to keep the fragment connected, leading to detection of the change. In the second case, fragments containing either of fork or join gateways detect deletion and addition of branches.

Bounded Wait: Every broadcast is non-blocking. It is released immediately after the resource is generated. Asynchronous event communication is assumed to be handled by the substrate. Every event token is consumed in the respective modules in bounded time. Therefore, a cyclic deadlock through multiple nodes involving a broadcast and wait for notification does not arise. Also, the conflict resolution loop eventually terminates as discussed below.

Termination: Let the initial global change region be $CR_{global} = \bigcup_{i=1}^{k} CR_i$, where CR_i and $SAFE_i$ are sets CR and $SAFE$ computed by `computeCR` in i^{th} fragment. The conflict resolution phase in node i either grows set CR_i by adding places through `rectifyCR` or keeps it the same. Therefore, set CR_{global} monotonically increases in every round till stabilizes (becomes non-increasing) in all fragments, when we know that the termination is reached. In other words, $SAFE_i$ is monotonically decreasing. At least one node broadcasts `CHANGE` in each round prior to the termination round. Since set $SAFE_i$ monotonically decreases, after a finite number of rounds there is no change in all nodes, at termination.

A Trace of the Algorithm: The above algorithm has been simulated in CPN tools on the example provided in Fig. 6. Currently, we have separate implemen-

tation of the local functions given in Table 2 in Java, which are not integrated with the CPN models. We have used results obtained from these functions as return results of functions `computeCR` and `rectifyCR` used in the CPN models that appear as arc inscriptions. In addition to the initial markings shown in the respective module nets, place *old fragment* (module ICBBN) in the non-changing fragments require initial marking `''null''` to run the simulation.

For the given net, fragments f_2 and f_5 are initiator participants, fragments f_1, f_3 and f_4 are participants in which nothing changes. In f_2, non-concurrent places p_6, p_8, TS, p_7 become concurrent. In f_5, places p_{10} and p_{28} gain additional concurrency. Hence, all of them are put in the change regions of their respective host fragments. Initially, f_3 has all its places locally safe. However, according to boundary status received from f_2 and f_5, it obtains conflict set $\{p_8, p_{10}\}$, then puts all its places in the change region by `rectifyCR`, and publishes `CHANGE` event with empty list as parameter. Every fragment therefore proceeds to another round of conflict resolution, where each of them finds no conflict. As a result, the algorithm requires two rounds of the second phase loop before termination. The final change region is depicted with shaded backgrounds in Fig. 6.

7 Conclusions

The paper presents a fully distributed algorithm to compute change region for fragmented processes in a dynamic evolution context. The algorithm works on structured WF-net models specified by the ECWS language. First, a distribution-friendly centralized approach to compute of change region was developed, and it was later fragmented to work in a fully distributed environment. The change region is seen as a set of places instead of the traditional connected subnet. This view enabled the distributed algorithm to use a strategy based on asynchronous status exchange. The distributed algorithm comprises three stages of initiation, basic local CR computation, and exchange of status of boundaries for adjustments to local change region till termination. The algorithm computes change region through fragmented processes without using node ids, and without requiring a centralized change manager, in contrast to the existing approaches for distributed processes. The algorithm was presented as a Hierarchical Colored Petri net, which is instantiated by every fragment. The applicability of the approach was highlighted through a fragmented BPMN collaborative process. The fragmentation rules are flexible enough for SESE-blocks to be fragmented. The approach may further be continued towards development and extensions such as *places jumping between nodes*, optimization of the static change region, integration of a distributed dynamic instance migration protocol possibly through a horizontal (concurrent) approach, optimization of messaging rounds without relaxing the condition of non-disclosure of the identities, and an algebraic proof that works on independent fragments to produce a result equivalent to the centralized computation albeit with certain overestimates.

References

1. Bruni, R., Corradini, A., Gadducci, F., Lafuente, A.L., Vandin, A.: A white box perspective on behavioural adaptation. In: Software, Services, and Systems. Springer (2015) 552–581
2. Ellis, C., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: Proc. of conf. on Organizational computing systems, ACM (1995) 10–21
3. van der Aalst, W.M., Basten, T.: Inheritance of workflows: an approach to tackling problems related to change. Theoretical Computer Science **270**(1) (2002) 125–203
4. Sun, P., Jiang, C.: Analysis of workflow dynamic changes based on petri net. Information and Software Technology **51**(2) (2009) 284 – 292
5. Adams, M., Ter Hofstede, A.H., Edmond, D., van der Aalst, W.M.: Implementing dynamic flexibility in workflows using worklets. BPMCenter Report **6**(06) (2006)
6. Lanz, A., Kreher, U., Reichert, M., Dadam, P.: Enabling process support for advanced applications with the aristaflow bpm suite. In: Proc. of the Business Process Management 2010 Demonstration Track, September 2010. (2010)
7. Protogeros, N., Tektonidis, D., Mavridis, A., Wills, C., Koumpis, A.: Fuse: A framework to support services unified process. In: Enterprise Interoperability III. Springer (2008) 209–220
8. Zaplata, S., Hamann, K., Kottke, K., Lamersdorf, W.: Flexible execution of distributed business processes based on process instance migration. Journal of Systems Integration **1**(3) (2010) 3–16
9. Cicirelli, F., Furfaro, A., Nigro, L.: A service-based architecture for dynamically reconfigurable workflows. Systems and Software **83**(7) (2010) 1148–1164
10. Hens, P., Snoeck, M., Poels, G., De Backer, M.: Process evolution in a distributed process execution environment. International Journal of Information System Modeling and Design **4**(2) (2013) 65–90
11. van der Aalst, W.M.: Exterminating the dynamic change bug: A concrete approach to support workflow change. Information Systems Frontiers **3**(3) (2001) 297–317
12. Gao, X., Wang, X., Yang, M., Liu, Y.: Workflow region recognition algorithm and its time complexity. Przeglad Elektrotechniczny **89**(1b) (2013) 184–186
13. Zou, J., Sun, H., Liu, X., Fang, K., Lin, J.: A hybrid instance migration approach for composite service evolution. In: Proceedings of the 2nd International Conference on Advanced Service Computing, Lisbon, Portugal. (2010) 153–159
14. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. Data & Knowledge Engineering **68**(9) (2009) 793–818
15. Hens, P., Snoeck, M., Poels, G., De Backer, M.: Process fragmentation, distribution and execution using an event-based interaction scheme. Journal of Systems and Software **89** (2014) 170–192
16. van der Aalst, W.M.: The application of petri nets to workflow management. Journal of circuits, systems, and computers **8**(01) (1998) 21–66
17. Pradhan, A., Joshi, R.K.: Catalog-based token transportation in acyclic block-structured wf-nets. In: PNSE. (2015) 287–307
18. Leemans, S.J., Fahland, D., van der Aalst, W.M.: Discovering block-structured process models from event logs – a constructive approach. In: Application and Theory of Petri Nets and Concurrency. Springer (2013) 311–329
19. Dijkman, R.M., Dumas, M., Ouyang, C.: Formal semantics and analysis of bpmn process models using petri nets. Queensland Univ. of Technology, Tech. Rep (2007)
20. Ratzer, A.V., Wells, L., Lassen, H.M., Laursen, M., Qvortrup, J.F., Stissing, M.S., Westergaard, M., Christensen, S., Jensen, K.: Cpn tools for editing, simulating, and analysing coloured petri nets. In: ICATPN, Springer-Verlag (2003) 450–462