Modeling Snapshot of Composite WS Execution by Colored Petri Nets

Yudith Cardinale[1], Marta Rukoz[2], and Rafael Angarita[1]

[1] Departamento de Computación, Universidad Simón Bolívar,
Caracas, Venezuela 1080
[2] LAMSADE, Université Paris Dauphine
Université Paris Ouest Nanterre La Défense
Paris, France
{yudith,rangarita}@ldc.usb.ve, marta.rukoz@lamsade.dauphine.fr

Abstract. The global transactional property of a Transactional Composite Web Service (TCWS) allows recovery processes if a Web Service (WS) fails during the execution process. The following actions can be performed if a WS fails: retry the faulty WS, substitute the faulty WS, or compensate the executed WSs. In consequence, these fault-tolerance mechanisms ensure the atomicity property of a TCWS with an all-or-nothing endeavor. In this paper, we present a formal definition of a checkpointing approach based in Colored Petri-Nets (CPNs) properties, in which the execution process and the actions performed in case of failures rely on unrolling processes of CPNs. Our checkpointing approach allows to relax the atomic transactional property of a TCWS in case of failures. The all-or-nothing transactional property becomes to the *something-to-all* property. A snapshot of the most possible advanced partial result is taken in case of failures and it is returned to the user (user gets *something*), providing the possibility of restarting the TCWS from an advanced execution state to complete the result (user gets *all* later), without affecting its original transactional property. We present the execution algorithms with the additionally capacity of taking snapshot in case of failures and experimental results to show the reception of partial outputs due to the relaxation of the *all-or-nothing* property.

1 Introduction

Web Service (WS) technology has gained popularity in both research and commercial sectors, based on the Semantic Web approach which makes part of the Web 3.0. With machine intelligence, users can resolve complex problems that require the interaction among different tasks. One of the major goals of the Web 3.0 is to support automatic and transparent WS composition and execution allowing a complex user request to be satisfied by a Composite Web Service (CWS), in which several WSs work together to resolve the complex query [2].

Automatic selection, composition, and execution of CWSs are issues that have been extensively treated in the literature by guaranteeing functional requirements (i.e., the set of input attributes provided in the query and the attributes that will be returned as output) and *QoS* criteria (e.g., response time

Z. Lacroix, E. Ruckhaus, and M.-E. Vidal (Eds.): RED 2012, LNCS 8194, pp. 23–44, 2013.

and price) [3–5, 11–13, 16, 20]. In some cases, transactional properties (e.g., atomic, compensable or not) are also considered to ensure fault-tolerant execution with Transactional Composite Web Services (TCWSs) [6, 7, 10, 14]. In a TCWS all its component WSs have transactional properties, which in turn could define an aggregated transactional property.

Because WSs can be created and updated on-the-fly, the execution system needs to dynamically detect changes during run-time, and adapt execution to the availability of the existing WSs. In this sense, TCWS becomes a key mechanism to cope with challenges of open software in dynamic changing environments to ensure that the whole system remains in a consistent state even in presence of failures [23].

Even if all component WSs of a Composite WS are transactional, the composition itself could be not transactional (e.g., a WS with an atomic but non-compensable transactional property, cannot be followed by another WS whose transactional property does not ensure a successful execution; if the second one fails, the first one cannot be compensated). Thus, to ensure the transactional property of a TCWS, the WSs selection process is made according to their transactional properties and their execution order. In this context, failures during the execution of a TCWS can be repaired by backward or forward recovery processes. Backward recovery implies to undo the work done until the failure and go back to the initial consistent state (before the execution started), by rollback and compensation techniques. Forward recovery tries to repair the failure and continues the execution; retry and substitution are some techniques used.

In both backward and forward recovery processes, the atomic (all-or-nothing) transactional property is comply to ensure system consistency. However, backward recovery means that users do not get the desired answer to their queries and forward recovery could imply long waiting time, because of the invested time to repair failures, for users to finally get the response. For some queries, partial responses may have sense for users; thus, they need alternative recovery strategies that provide this facility in case of failures.

In previous works, we provided the definition of backward recovery (compensation process) and forward recovery (retry and substitution) approaches [8, 9] based on Colored Petri Nets (CPNs) formalism. In CPNs, transitions represent WSs, places represent input/output WS attributes, and colors are used to represent transactional properties of transitions and types of values in places. In [8] unrolling algorithms of CPNs to control the execution and backward recovery were presented. This work was extended in [9] to consider forward recovery based on WS replacement; formal definitions for WSs substitution process, in case of failures, were presented. In [9], we also proposed an Executor architecture, independent of its implementation, to execute a TCWS following our proposed fault-tolerant execution approach. In [1, 18], we have presented implementations of our fault-tolerant approaches. In [1], we present FaCETa, a framework which implements the backward and forward recovery proposed in [8, 9]. In [18], we present the framework FaCETa*, an extension of FaCETa, in which the fault-tolerant approach is extended with checkpoints, i.e., in case of failures,

the execution state of a TCWS is checkpointed and the execution flow goes on as much as it is possible, therefore users can have partial responses and later restart the execution of the TCWS.

The contribution of this paper is focused in formally defining the checkpointing approach, based also in CPN properties, as a way to relax the atomic transactional property (all-or-nothing) of a TCWS in case of failures. It means that, instead of all-or-nothing, users can have a *something-to-all*. If a failure occurs, a snapshot that contains the execution state of the most possible advanced partial result is taken and it is returned to the user (user gets *something*). The checkpointed TCWS can be re-started from an advanced point of execution (snapshot) to complete the desired result (user gets *all* later), without affecting its aggregated transactional property. We also present the execution algorithms with the additionally capacity of taking snapshot in case of failures, the extended framework incorporating checkpointing facilities, and experimental results to show the results of the prototype implementation of the checkpointing mechanism.

This paper is organized as follows. Section 2 recalls some important concepts and formal definitions necessary for the understanding of this work, such as Web Service composition and their properties and execution. Section 3 introduces an alternative fault-tolerance approach as a way to relax the *all-or-nothing* transactional property to a *something-to-all* property. Section 4 presents the formal definitions to allow the execution of the checkpointing mechanism in case of failure. Section 5 presents the overall architecture of our extended framework and some results showing the reception of partial results in case of failure. Section 6 discusses related work in the field of checkpointing for TCWSs. Finally, Section 7 presents our conclusions.

2 Preliminaries

This Section recalls some important concepts and formal definitions about Web Service composition and their properties and execution.

2.1 Web Service

A Web Service, ws, consists of a finite set of operations, denoted as $ws = \{op_i, i = 1..n\}$, with $op_i = (I_i, O_i, Q_i, T_i)$, where $I_i = \{I_{i1}, I_{i2}, ..\}$ is a set of input attributes of op_i, $O_i = \{O_{i1}, O_{i2}, ..\}$ is a set of output attributes whose values are produced by op_i, $Q_i = \{Q_{i1}, Q_{i2}, ...\}$ is a set of QoS values of op_i for a set of QoS criteria $\{q_1, q_2, ...\}$, and $T_i \in \{p, a, c, cr, pr, ar\}$ is the transactional property of op_i (transactional properties are defined in Section 2.3). In this work, without loss of generality, we consider that ws has only one op_i and we use the term ws to denote the op of ws.

2.2 Composite Web Service

A Composite Web Service, described as $CWS = \{ws_i, i = 1..m\}$, is a combination of several WSs to produce more complex services that satisfy more complex

user requests. It concerns *which* and *how* WSs are combined to obtain the desired results. A CWS can be represented in structures such as workflows, graphs, or Petri Nets indicating, for example, the control flow, data flow, WSs execution order, and/or WSs behavior. The structure representing a CWS can be manually or automatically generated. Users can manually specify *how* functionality of WSs are combined or a COMPOSER can automatically decide *which* and *how* WSs are combined, according the desired query. In both cases, the execution of a CWS is carried out by an EXECUTOR, that decides *which* WSs comply each functionality manually specified by users and invokes them, or invokes the WSs automatically decided by the COMPOSER. In this paper, we represent a CWS by a colored Petri Net as it is established in Definition 1 and suppose that it was generated automatically by a COMPOSER [6].

2.3 Transactional Properties

The transactional property (TP) of a WS allows to recover the system in case of failures during the execution. A single WS is transactional (denoted as TWS), if when it fails, it has no effect at all. The most basic transactional property that implements this characteristic is **pivot** (p): A TWS is called **pivot** (p) WS, if its effects remain forever and cannot be semantically undone once it has completed successfully. For a CWS, it is transactional, named Transactional Composite WS (TCWS), if when it fails, its partial effects can be semantically undone. In this case, the basic transactional property is called **atomic** (a): a TCWS is **atomic** (a), if the effects of the TCWS remain forever and cannot be semantically undone once it has completed successfully. There exist other transactional properties for TWS and TCWS, which complement transactionality. A TCWS or TWS can be associated with another TCWS or TWS which can semantically undo its successfully execution; in this case, the TCWS or TWS is called **compensatable** (c). A TCWS or TWS can be combined with a retriable property, which guarantees a successfully termination after a finite number of invocations. In this case, we obtain **pivot retriable** (pr), **atomic retriable** (ar), and **compensatable retriable** (cr) WSs. WSs that provide transactional properties are useful to guarantee reliable TCWSs execution and to ensure the whole system consistent state even in presence of failures. Failures during the execution of a TCWS can be supported according to the TP of its component WSs by a forward recovery process, in which the failure is repaired to allow the failed WS to continue its execution or by a backward recovery process, wherein its partial effects are semantically undone.

2.4 User Query

We define a query in terms of functional conditions, expressed as input and output attributes; QoS constraints, expressed as weights over criteria; and the required global TP as follows. A query Q is a 4-tuple $Q = (I_Q, O_Q, W_Q, T_Q)$, where:

- I_Q is a set of input attributes whose values are provided by the user,

- O_Q is a set of output attributes whose values have to be produced by the system,
- $W_Q = \{(w_i, q_i) \mid w_i \in [0,1]$ with $\sum_i w_i = 1$ and q_i is a QoS criterion$\}$, and
- T_Q is the required transactional property: $T_Q \in \{T_0, T_1\}$; If $T_Q = T_0$, the system guarantees that a semantic recovery can be done by the user. If $T_Q = T_1$, the system does not guarantee that the result can be compensated. In any case, if the execution is not successful, nothing is changed on the system and its state is consistent.

2.5 Execution Control

A TCWS, which answers and satisfies a query Q, is modeled as an acyclic marked CPN, denoted CPN-EP_Q, as following.

Definition 1 Transactional Composite Web Service. *A* TCWS *is a 4-tuple* (A, S, F, ξ)*, where:*

- A *is a finite non-empty set of places, corresponding to input and output attributes of the* WSs*;*
- S *is a finite set of transitions corresponding to the set of* WSs $\in$ TCWS *;*
- $F : (A \times S) \cup (S \times A) \rightarrow \{0,1\}$ *is a dataflow relation indicating the presence (*1*) or the absence (*0*) of arcs between places and transitions defined as follows:* $\forall s \in S$, $(\exists a \in A \mid F(a,s) = 1) \Leftrightarrow$ *(a is an input place of s) and* $\forall s \in S$, $(\exists a \in A \mid F(s,a) = 1) \Leftrightarrow$ *(a is an output place of s);*
- ξ *is a color function such that* $\xi : S \rightarrow \Sigma_S$*, with* $\Sigma_S = \{p, pr, \boldsymbol{a}, \boldsymbol{a}r, c, cr\}$ *representing the* TP *of* $s \in S$*.*

According to CPN notation, we have that for each $x \in (A \cup S)$, $({}^\bullet x) = \{y \in A \cup S : F(y,x) = 1\}$ is the set of its predecessors, and $(x^\bullet) = \{y \in A \cup S : F(x,y) = 1\}$ is the set of its successors.

We suppose that a TCWS is well constructed, i.e., its component WSs satisfy the transactional rules presented in Table 1. Let us illustrate the rules in Table 1 with the following examples. If the TP of a ws_i is p or a, another ws_j, whose TP is pr, ar, or cr can be executed after ws_i (sequential execution, rule 1); ws_i can be executed in parallel with a ws_k with TP cr (rule 2). This rules guaranteeing that the resulting TCWS satisfies the transactional properties presented in section 2.3 [10] .

Definition 2 Marked CPN-EP_Q. *A marked CPN-*EP_Q *is a pair (*TCWS*,*M*), where* TCWS$=(A, S, F, \xi)$ *and* M *is a function which assigns tokens (values) to places such that* $\forall a \in A$, $M(a) \in N$*.*

Given a user query $Q = (I_Q, O_Q, W_Q, T_Q)$, a marked CPN-$EP_Q = ((A, S, F, \xi), M)$ satisfies Q if:

- $\forall x \in I_Q$, $\exists a \in A$ such that a is an input place.
- $\forall x \in O_Q$, $\exists a \in A$ such that a is an output place.

Table 1. Transactional rules of [10]

Transactional property of a WS	Sequential compatibility	Parallel compatibility
p, $\boldsymbol{a}$	$pr \cup \boldsymbol{ar} \cup cr$ (rule 1)	cr (rule 2)
pr, $\boldsymbol{a}r$	$pr \cup \boldsymbol{ar} \cup cr$ (rule 3)	$pr \cup \boldsymbol{ar} \cup cr$ (rule 4)
c	Σ_S (rule 5)	cr (rule 6)
cr	Σ_S (rule 7)	Σ_S (rule 8)

– let $M_Q = \{\forall a \in (A \cap I_Q),\ M(a) = 1$ and $\forall a \in (A - I_Q),\ M(a) = 0\}$ and $M_F = \{\forall a \in (A \cap O_Q),\ M(a) \geq 1\}$, the initial and final marking, respectively; there exist a firing sequence σ, such that: $M_Q \xrightarrow{\sigma} M_F$ and such that transitions of σ represent a TCWS whose components satisfy the transactional rules, locally optimize the QoS and $\forall s_i \in \sigma \mid \xi_Q(s) \in \{c, cr\}$ if $T_Q =' compensatable'$, and $\forall s_i \in \sigma \mid \xi_Q(s) \in \{pr, ar, cr\}$ if $T_Q =' retriable'$.

The marking of a CPN-EP_Q represents the current values of attributes that have been produced either for some component WSs or by the user, and that can be used for others component WSs to be invoked. A Marked CPN denotes which transitions can be fired.

In order to finally resolve the query Q, the given TCWS has to be executed by invoking its component WSs according to the execution flow depicted by the CPN representing the TCWS(i.e. CPN-EP_Q). In fact, during the execution, CPN-EP_Q represents the execution plan of TCWS. As the composition process presented in [9], the execution process is controlled by an unrolling algorithm over CPN-EP_Q.

Definition 3 Fireable Transition. *A marking M enables a transition s (to invoke the ws it represents) iff all its input places contain tokens such that* $\forall x \in ({}^\bullet s)$, $M(x) \geq card({}^\bullet x)$.

To start the execution algorithm, the CPN-EP_Q is marked with the *Initial Marking* and some transitions become fireable. When a transition is fireable, it can be fired according to the firing rules (see definition 4). The firing of a transition of a CPN-EP_Q corresponds to the execution of a WS, let us say s, which participates in the composition.When s finishes, other transitions become fireable, and so on.

Definition 4 CPN-EP_Q Firing Rules. *The firing of a fireable transition s for a marking M defines a new marking M', such that: all tokens are deleted from its input places ($\forall x \in {}^\bullet s$, $M(x) = 0$) and the* WS *s is invoked. These actions are atomically executed. After* WS *s finishes correctly, tokens are added to its output places ($\forall x \in (s^\bullet)$, $M(x) = M(x) + 1$).*

Note that during the execution, in CPN-EP_Q a transition is fireable (its corresponding WS can be invoked) only if all its predecessor transitions have been

fired (each input place has as many tokens as WSs produce them or one token if the user provides them) and several transitions can be fireable at the same time. In this way, the execution control, followed according CPN-EP_Q, respect sequential and parallel executions, which in turn keeps the global transactional property.

Once a WS is executed, its input places are unmarked and its output places (if any) are marked. We illustrate these definitions with the example shown in Figure 1, where $Q = (I_Q, O_Q, W_Q, T_Q)$ with $I_Q = \{a_1, a_2\}$ and $O_Q = \{a_5, a_6, a_7\}$. Note that ws_3 needs two tokens in a_3 to be invoked; this data flow dependency indicates that it has to be executed in sequential order with ws_1 and ws_2, and can be executed in parallel with ws_4. Note that a_3 is produced by ws_1 and ws_2, ws_1 was already executed and it produced a token on a_3, and ws_2 is still running. Even if ws_3 could be invoked with the values produced by ws_1, if ws_3 is fired, it will be executed in parallel with ws_2; however, it could be possible that transactional properties of ws_2 and ws_3 dictate that they have to be executed in sequential order as the data flow indicates. Then, ws_3 has to wait for all its predecessors transitions to finish in order to be invoked. Once ws_2 finishes, ws_3 and ws_4 can be executed in parallel.

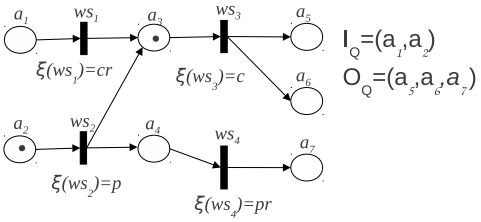

Fig. 1. Example of Fireable Transitions

2.6 Fault Tolerant Execution Control

The global TP of a TCWS allows recovery processes if a WS s fails during the execution process. In previous works, we have presented a recovery mechanism [9] based on TP properties of its component WSs. In these works, if a WS s fails, the following actions are executed:

- if $TP(s)$ is **retriable** (pr, $\boldsymbol{ar}$, cr), s is re-invoked until it successfully finishes (forward recovery);
- otherwise, another Transactional substitute WS, s^*, is selected to replace s and the unrolling algorithm goes on (trying a forward recovery);
- if there not exists any substitute s^*, a backward recovery is needed, i.e., all executed WSs must be compensated in the inverse order they were executed; for parallel executed WSs, the order does not matter. The compensation flow is represented by a backward recovery CPN (BRCPN-$TCWS_Q$),

which depicts the inverse order of the execution flow. The corresponding BRCPN-$TCWS_Q$ for a TCWS can be automatically generated by the same COMPOSER that built the TCWS.

These actions guarantee the atomicity (all-or-nothing) property of a TCWS. In the next section we present a checkpointing mechanism allowing to relax this property by returning *something* in case of failures. Then, a re-execution from an advanced execution state is possible to finally get the final desired response (user finally gets *all*).

3 Relaxing All-or-Nothing Transactional Properties by Checkpointing Mechanisms

Approaches previously presented in [8, 9] provide fault-tolerant mechanisms relying on WSs replacement, on a compensation protocol, and on unrolling processes of CPNs. Although these recovery processes ensure system consistency, they represent an "all-or-nothing" approach, since users either receive full answer to their queries or they do not get any answer (in case of failure, partial answers, if any, are undone). In this section we present an alternative fault-tolerant approach as a way to relax this transactional property to a *something-to-all* property. In case of failures, the unrolling process of the CPN controlling the execution of a TCWS is checkpointed and the execution flow continues as much as possible. In consequence, users can receive partial responses (*something*) as soon as they are produced and resubmit the checkpointed CPN to restart its execution from an advanced point of execution and finish the TCWS (to get *all*), without affecting the original transactional property.

For this purpose, when a WS associated to a fireable transition t fails, the execution control, instead of executing backward recovery, it saves the subnet of CPN-EP_Q that could not be executed. For that, the inputs and output places, and valid attributes (attributes already produced) of transition t are saved, and the same attributes are saved recursively for any other transition that depends on t.

The checkpointing mechanism is illustrated in the following. The marked CPN-EP_Q depicted in Figure 2 is the state when ws_4 fails and the unrolling of the CPN-EP_Q continues to allow the execution of all the WSs not affected by the failure of ws_4. The only WSs affected by this failure are ws_7 and ws_8; therefore, assuming that there will not be more failures, the output attribute corresponding to a_{10} will be obtained.

Figure 3 shows the execution state when all the WSs not affected by the failure were executed and the a_{10} value was received. Red places and transitions in Figure 3 represent the part of the marked CPN-EP_Q involved in the execution restart process, called CPN-$check_{Q'}$, (the associated WSs of these transitions were not executed because they need values produced by the failed WS or any other that depend of it). The red tokens in Figure 3 represent the values already produced during the normal execution (these tokens will be the initial marking of the CPN-$check_{Q'}$).

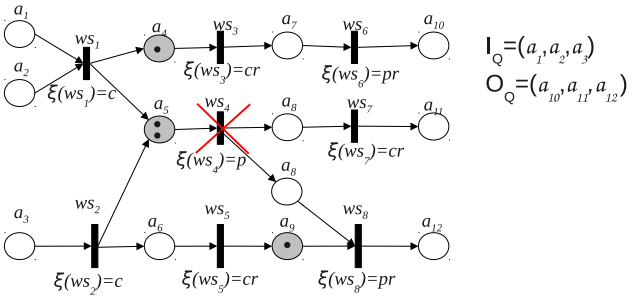

Fig. 2. Marked CPN-TCWS$_Q$ when ws_4 fails

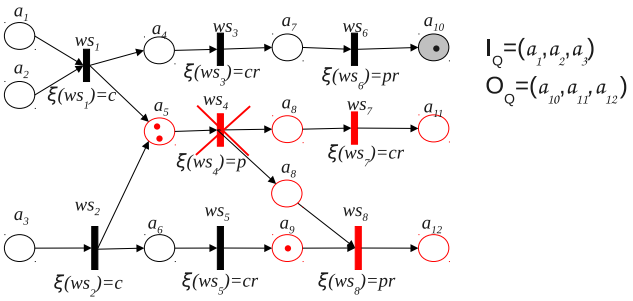

Fig. 3. Marked CPN-TCWS$_Q$ just before checkpointing

Figure 4 represents the CPN-$check_{Q'}$. Note that the initial marking contains tokens representing values produced during the CPN-EP_Q execution, whilst a_{11} and a_{12} are the only output attributes expected as a result of the CPN-$check_{Q'}$ execution. Note that ws_8 has to wait only for one of its two inputs, since it had already received a_9 before the execution was restarted.

4 Modeling Checkpointing Based on Petri-Net Formalism

We extend the CPN-EP_Q unrolling execution process to take into account checkpoints in case of failure by modifying definitions 1, 2, 3, and 4; additionally, new definitions regarding to the checkpointing mechanism are presented. In a general way, these definitions express the idea presented in the following paragraph.

If a WS fails, its corresponding transition informs its successors about the failure, thus they know they are not going to receive the values corresponding to the outputs of the failed WS. If one transition is notified that one or more of its predecessor transitions will not be able to produce its output values, it still waits until all its predecessor transitions have finished. Therefore, it is possible for transitions to receive their required input values partially. Then, it informs its

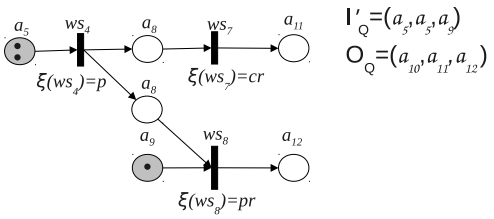

Fig. 4. Marked CPN-$check_{Q'}$

successor transitions about its inability to invoke its corresponding WS. When the unrolling process has finished, one or more O_Q attributes will have faulty values, whilst the rest will have correct values. At this point, a snapshot representing the non-successfully executed part of the CPN-EP_Q is saved. User is provided with the possibility to restart the execution later, executing only the previously failed WSs and the WSs that were never fired for execution. Note that it is possible that none of the O_Q attributes has a correct value, however at least the snapshot represents an advanced execution state.

Using CPNs, information can be modeled by tokens and the type of information can be modeled by the color of those tokens. We define the following colors associated to places in order to model the unrolling process for checkpointing.

- Valid (v): if a token belonging to a place has color v, it means that the WS that produced its value was executed successfully.
- Invalid (i): if a token belonging to a place has color i, it means that the WS that produces its value was not executed successfully; i.e., the WS supposed to produce its value failed or it was not executed because one of its WSs predecessors failed.

The following definitions allow the execution of the checkpointing mechanism in case of failure.

Definition 5 Transactional Composite Web Service. *A* TCWS *is a 4-tuple* (A, S, F, ξ)*, where:*

- A *is a finite non-empty set of places, corresponding to input and output attributes of the* WSs*;*
- S *is a finite set of transitions corresponding to the set of WSs* $\in$ TCWS *;*
- $\forall s \in S$, $(\exists a \in A \mid F(a,s) = 1) \Leftrightarrow$ *(a is an input place of s) and* $\forall s \in S$, $(\exists a \in A \mid F(s,a) = 1) \Leftrightarrow$ *(a is an output place of s);*
- ξ *is a color function such that* ξ*:* $C_A \cup C_S$ *with* C_A*:* $A \rightarrow \sum_A$*, a color function such that* $\sum_A = \{v, i\}$ *representing, for* $a \in A$*, either the success or failure of its predecessor transitions, and* $C_S : S \rightarrow \sum_S$*, a color function such that* $\sum_S = \{p, pr, \boldsymbol{a}, \boldsymbol{ar}, c, cr\}$ *represents the* TP *of* $s \in S$ $(TP(s))$*.*

Definition 6 Marked Executable CPN-EP_Q. *A marked CPN-EP_Q=(A, S, F, ξ) is a pair (CPN-EP_Q,M), where M is a function which assigns tokens (values) to places such that* $\forall a \in A,\ M(a) \subseteq \{\emptyset, Bag(\sum_A)\}$, *where Bag corresponds to a set which can contain several occurrences of the same element. The marking of a CPN represents the current state of the system, i.e., the set of attributes produced correctly by the system and/or signals indicating failures.*

A transition s is fireable when all its predecessor transitions have added a token to their output places (input places of s). If all of them are valid tokens we said that s is fireable for execution, otherwise (i.e., at least one of them is an invalid token) we said that s is fireable for checkpointing. During the unrolling process for the execution of a TCWS all the predecessor places of s will have the required tokens for the invocation of s. In case of failures, some of these tokens will be invalid, as it is shown in the following definition.

Definition 7 Fireable Transition. *A marking M enables a transition s for execution iff all its input places contain tokens such that* $\forall x \in ({}^\bullet s),\ card(M(x)) \geq card({}^\bullet x)\ \wedge\ M(x) \subseteq Bag(\{v\})$. *A marking M enables a transition s for checkpointing iff all its input places contain tokens such that* $(\forall x \in ({}^\bullet s),\ card(M(x)) \geq card({}^\bullet x))\ \wedge\ (\exists x \in ({}^\bullet s),\ \{i\} \in M(x))$.

Definition 8 CPN-EP_Q Firing Rules. *The firing for execution of a fireable transition s for a marking M defines a new marking M′, such that: all tokens are deleted from its input places* ($\forall x \in {}^\bullet s,\ M(x) = 0$) *and the* WS *s is invoked. These actions are atomically executed. After* WS *s finishes, tokens are added to its output places* ($\forall x \in (s^\bullet),\ (M(x) \leftarrow M(x) \cup \{v\})$). *The firing for checkpointing of a fireable transition s for a marking M, defines a new marking M′, such that: s, its inputs and outputs places, and its valid input attributes are saved; and tokens are added to its output places* ($\forall x \in (s^\bullet),\ (M(x) \leftarrow M(x) \cup \{i\})$). *These actions are also atomically executed.*

Definition 9 Local Snapshot. *A Local Snapshot is the set of data representing the state of a transition in a CPN. It contains:*

- *Inputs_ws represents the information about input attributes required by s to become fireable. For each predecessor place of s it contains a set of pairs token-value, where token contains either v or i depending on whether the attribute was generated correctly or not, and value contains the actual received value iff the attribute was generated correctly (value will be empty iff the token value is i). Inputs_ws will be empty if the transition did not consume the value of any of its predecessor places, or it can contain the value consumed prior to the checkpointing. If s was executed unsuccessfully, then $InputsNeeded\_ws$ will contain all the input values required by s;*
- *$Results_{ws}$ represents the output attributes of s iff s was executed successfully.*

Definition 10 Global Snapshot. *A Global Snapshot (GS) is the set of data necessary to restart the execution of a* TCWS. *A GS contains the union of all*

local snapshots, which is the CPN-check$_{Q'}$ containing the information of the part of the TCWS *to be restarted, the user query Q, and the I'_Q necessary to restart the execution.*

5 Framework Architecture

In [1], we presented a framework which implements the backward and forward recovery proposed in [8, 9]. In [18], we present a proposal to extend our framework for considering checkpointing mechanisms. In this section, we first present a deeper description of the overall architecture of our extended framework and a detailed explanation of the fault-tolerance algorithms incorporated to the framework. Finally, we present some results showing the reception of partial results in case of failure, which relaxes the *all-or-nothing* property to *something-or-nothing*.

During the TCWS execution there exist two basic variants of execution scenarios for component WSs. In *sequential* scenario, WSs work on the result of previous services and cannot be invoked until previous services have finished. In *parallel* scenario, several services can be invoked simultaneously because they do not have data flow dependencies. The global TP of TCWSs is affected by the execution scenarios. Hence, it is mandatory to follow the same CPN unrolling algorithm taken by the Composer in order to ensure that sequential and parallel execution satisfies the global TP.

The execution of a TCWS in our framework (referenced as Executor) is managed by an Execution Engine and a collection of software components called Engine Threads, organized in a three-level architecture. Figure 5 depicts the overall architecture of our Executor. In the first level, the Execution Engine receives the TCWS and its corresponding $BRCPN$ (the compensation order), both represented by CPNs automatically generated by the Composer. It launches, in the second layer, an Engine Thread for each WS in the TCWS. Each Engine Thread is responsible for the execution control of its WS. They receive WS inputs, invoke their respective WS, and forward their results to their peers to continue the execution flow. Hence, the Execution Engine is responsible for initiating the Engine Threads and the unrolling algorithm, while Engine Threads are responsible for the actual invocation of WSs monitoring its execution, and forwarding results to its peers to continue the execution flow. In case of failure, all of them participate in the recovery process.

By distributing the responsibility of executing a TCWS across several Engine Threads, the logical model of our Executor enables distributed execution and it is independent of its implementation; i.e., this model can be implemented in a distributed memory environment supported by message passing or in a shared memory platform, e.g., supported by a distributed shared memory or tuplespace system. The idea is to place the Executor in different physical nodes (e.g., a high available and reliable computer cluster) from those where actual WSs are placed. Engine Threads remotely invoke the actual component WSs. The Execution Engine needs to have access to the WSs Registry, which contains the *WSDL* and *OWL-S* documents. The knowledge required at runtime

by each ENGINE THREAD (e.g., WS semantic and ontological descriptions, WSs predecessors and successors, and execution flow control) can be directly extracted from the CPNs in a shared memory implementation or sent by the EXECUTION ENGINE in a distributed implementation.

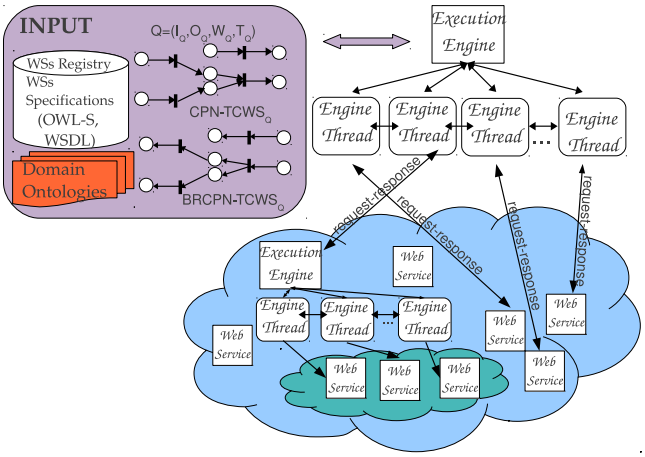

Fig. 5. Executor Architecture

Typically, WSs are distinguished in *atomic* and *composite* WSs. An atomic WS is one that solely invokes local operations that it consist of (e.g., *WSDL* and *OWL-S* documents define atomic WSs as a collection of operations together with abstract descriptions of the data being exchanged). A composite WS is one that additionally accesses other WSs or invokes operations of other WSs. We consider that transitions in the CPN, representing the TCWS to be executed, could be atomic WSs or TCWSs. Atomic WSs have its corresponding *WSDL* and *OWL-S* documents. TCWSs can be encapsulated into an EXECUTOR; in this case, the EXECUTION ENGINE has its corresponding *WSDL* and *OWL-S* documents. Hence, TCWSs may themselves become a WS, making the TCWS execution a recursive operation, as it is shown in Figure 5.

5.1 Checkpointing Algorithms

This section explains how the fault-tolerant execution control was extended in order to incorporate the checkpointing mechanism. The whole execution process is divided in several phases, in which the EXECUTION ENGINE and ENGINE THREADS can participate.

Initial Phase: Whenever an EXECUTION ENGINE receives a CPN-EP_Q and its corresponding BRCPN-EP_Q, it starts an ENGINE THREAD responsible for each transition in CPN-EP_Q, indicating to each one its predecessor and successor

transitions according to the CPN-EP_Q structure; this step means that the Execution Engine sends the part of the CPN-EP_Q that each Engine Thread concerns on; then it sends values of attributes in I_Q to Engine Threads in charge of WSs who receive them. In Algorithm 4, lines 1 to 14 describe these steps.

WS Invocation Phase: Once each Engine Thread is started, it waits until its inputs are produced. When an Engine Thread receives all the needed inputs, it invokes its corresponding WS. When a WS finishes successfully, the Engine Thread sends values of WS outputs to Engine Threads representing successors of its WS. This step emulates the firing rules in the CPN. Note that all fireable transitions can be invoked in parallel. If a WS fails during the execution, the *Checkpointing phase* is executed, in this case the Engine Thread sends faulty values to its successors to initiate the checkpointing process. When an Engine Thread receives at least one faulty value among its needed inputs, the *Checkpointing phase* is executed. Algorithm 3, lines 1 to 7 and Algorithm 4, line 17 to 18 describe these steps for Engine Thread and the Execution Engine, respectively.

Final Phase: This phase is carried out by both Execution Engine and Engine Threads. If the TCWS was successfully executed, the Execution Engine notifies all Engine Threads by sending the *finish* message, recalculates the Quality of TCWS in case some WSs were replaced, and returns the values of attributes in O_Q to the user. When an Engine Threads receives the *finish* message, it exits. In case that compensation is needed, the Execution Engine receives a *compensate* message, the process of executing the TCWS is stopped, and the compensation process is started by sending a *compensate* message to all Engine Threads. If an Engine Thread receives a *compensate* message, it launches the compensation protocol. If an Execution Engine receives a faulty value in at least one of the O_Q attributes, it executes the *Checkpointing phase*. Algorithm 3, lines 8 to 10, describe these steps for Engine Threads, and Algorithm 4, lines 15 to 21 describe these steps for Execution Engine.

Replacing Phase: This phase is carried out by an Engine Thread when a failure occurs during the execution of its WS. The Engine Thread tries to replace the faulty WS by a substitute and from candidates, it selects the best one according a quality function. According to the transactional property of TCWS, this phase should be executed until success or can be executed for a maximum number of times ($MAXTries$).

Compensation Phase: This phase, carried out by both Execution Engine and Engine Threads, is executed if a failure occurs in order to leave the system in a consistent state. The Engine Thread responsible of the faulty WS informs Execution Engine about this failure. The Execution Engine sends a message *compensate* to all Engine Threads and starts the compensation

process following a unrolling algorithm over BRCPN-$TCWS_Q$. Once the rest of Engine Threads receive the message *compensate*, they apply the firing rules in BRCPN-$TCWS_Q$ to follow the compensation process.

Checkpointing Phase: This phase is carried out by the Execution Engine and the Engine Threads who cannot invoke their corresponding WSs, because they are in the path of a failure. The Engine Thread sends faulty values to its successors, saves its state (snapshot), and sends it to the Execution Engine. The snapshot consists of values of input attributes (correct and faulty), the name of its WS, and successors. The correct values obtained in the input attributes of the failed transitions will be the I'_Q required to restart the execution of the TCWS. The Execution Engine saves the correct values of O_Q attributes, collects the snapshots of Engine Threads and return this partial response to the user along with the global snapshot, which is the part of CPN-EP_Q that could no be executed (PARTIAL-CPN-EP_Q). Algorithm 1 shows this phase for the Execution Engine and Engine Threads.

Restart Phase: This phase is carried out by the Execution Engine. First, all the required data is obtained from the previously saved global snapshot. Similar to the *Initial phase*, the Execution Engine starts an Engine Thread responsible for each transition in PARTIAL-CPN-EP_Q, it removes the valid tokens and values from failed transitions and builds I'_Q with those values, sends the I'_Q to the corresponding Engine Thread and the unrolling algorithm over PARTIAL-CPN-EP_Q is started by executing *Invocation phase* and *Final phase*. Algorithm 2 describes this phase for the Execution Engine; whilst the Engine Threads do not take any special action for this phase.

Algorithms for the Replacing and Compensation phases are not shown here for space reasons. They can be found in [8]. Figure 7 depicts the flow diagrams showing the phases previously described for the Execution Engine and Engine Threads, respectively.

5.2 Experimental Results

We developed a prototype of our proposed approach using Java 6 and MPJ Express 0.38 library to allow the execution in distributed memory environments. The deployment was made in a cluster of PCs: one node for the Execution Engine and one node for each Engine Thread needed to execute the TCWS. All PCs have the same configuration: Intel Pentium 3.4GHz CPU, 1GB RAM, Debian GNU/Linux 6.0, and Java 6. They are connected through a 100Mbps Ethernet interface.

We generated 80 TCWSs of sizes from 3 to 10 WSs. All those TCWSs were automatically generated by a composition process [6], from synthetic datasets comprised by 800 WSs with 7 replicas each, for a total of 6400 WSs. Replicas of WSs have different response times.

The OWLS-API 3.0 was used to parse the WS definitions and to deal with the OWL classification process.

Algorithm 1. Checkpointing

```
begin
    Execution Engine:
    begin
        Save received right values of O_Q;
        Collect Snapshots from Engine Threads
        (ETWS_ws);
        I'_Q ← correctvaluesfromallSnapshots;
        Build PARTIAL-CPN-EP_Q;
        Save PARTIAL-CPN-EP_Q as globalsnapshot;
        Return GlobalSnapshot reference;
    end
    Engine Threads:
    begin
        Send faulty values to Sucessors_ETWS_ws;
        Snapshot_ETWS_ws ← received right values and
        Sucessors_ETWS_ws;
        Send Snapshot_ETWS_ws to Execution Engine;
        Return /* the Engine Thread finishes */;
    end
end
```

Algorithm 2. Execution Engine Restart

```
Input: GS: a reference to a Global Snapshot
begin
    Execution Engine:
    begin
        Load Q, PARTIAL-CPN-EP_Q, BRCPN-EP_Q,
        OWS (Ontology of WSs), OV_Q (list of values of
        o | o ∈ O_Q), I'_Q, InputsNeeded from GS;
        /*I'_Q represents the correct values obtained before
        failure */
    end
    repeat
        Instantiate an ETWS_ws;
        Send Predecessors_ETWS_ws ←• (•ws);
        Send Successors_ETWS_ws ← (ws•)•;
        Send InputsNeeded_ETWS_ws; /*Inputs already
        received by the ETWS_ws*/
        /* each Engine Thread keeps the part of CPN-EP_Q
        and BRCPN-EP_Q which it concerns on*/
    until
    ∀ws ∈ S | (ws ≠ ws_EE_i) ∧ (ws ≠ ws_EE_f) ∧ ¬(∀a ∈
    InputsNeeded_ETWS_ws, M(a) = card(•a));
    Send values of I'_Q to ETWS_ws receiving them ;
    Execute Final phase;
end
```

Fig. 6. Checkpointing & Restart Algorithms

In order to test the checkpointing mechanism, a randomly selected WS fails during the execution of each TCWS allowing to continue the Petri Net unrolling and receive all the outputs that were not affected by the failure. We executed TCWSs comprised of 3, 4, 5, 6, 7, 8, 9, and 10 WSs. Each TCWS was executed 100 times, for a total of 8000 executions. Table 2 shows the different percentages of outputs received in presence of failures during the 100 executions and the

Algorithm 3. ENGINE THREAD Algorithm

Input: $Predecessors\_ETWS_{ws}$, predecessors WSs of ws
Input: $Successors\_ETWS_{ws}$, successors WSs of ws
Input: $WSDL_{ws}, OWLS_{ws}$, semantic web documents
Input: $MAXTries$: Max number of tries to replace a faulty WS
1 Invocation phase:
begin
  $InputsNeeded\_ETWS_{ws} \leftarrow getInputs(WSDL_{ws}, OWLS_{ws})$;
2 **repeat**
    Wait Result from ($Predecessors\_ETWS_{ws}$);
    Set values to $InputsNeeded\_ETWS_{ws}$;
  until $\forall a \in InputsNeeded\_ETWS_{ws}, M(a) = card(^\bullet a)$;
  /* all the predecessor transitions have finished */
  if $\exists a \in InputsNeeded\_ETWS_{ws} \mid M(a) \in Bag(\{e\})$ **then**
    /*one or more predecessors transitions have finished unsuccessfully
    Execute Checkpointing phase;
3 $success \leftarrow false$;
  $cantry \leftarrow true$;
  $tries \leftarrow 0$;
  $equivalents \leftarrow getEquivalents(WSDL_{ws}, OWLS_{ws})$;
4 $\zeta(ws') \leftarrow R$;
5 **repeat**
    Invoke ws;
    if *(ws fails)* **then**
      if $TP(ws) \in \{pr, ar, cr\}$ **then**
6
        Re-invoke WS;
      else
        Execute Replacing phase;
      /*forward recovery*/
    else
      Wait Result from ws;
      $\zeta(ws') \leftarrow E$;
      Remove tokens from inputs of ws;
      Send Results to $Successors\_ETWS_{ws}$;
      $success \leftarrow true$;
  until $(success) \vee (\neg cantry)$;
7 **if** $\neg success$ **then**
    if *checkpointing is enabled* **then**
      Execute Checkpointing phase;
    else
      Send *compensate* to EXECUTION ENGINE;
      $\zeta(ws') \leftarrow C$;
      Execute Compensation phase;
    /*backward recovery*/
  else
    Execute Final phase;
end
8 Final phase:
begin
9 Wait *message*;
  if *message is Finish* **then**
    Send *Finish* message to $Predecessors\_ETWS_{ws}$;
    Return;
  else
    if *message is Snapshot* **then**
      Send $ETWS_{ws}$ *snapshot* message to EXECUTION ENGINE;
10
      Return;
    else
      Execute Compensation phase;
end

Algorithm 4. EXECUTION ENGINE Algorithm

Input: $Q = (I_Q, O_Q, W_Q, T_Q)$, the user query
Input: CPN-$EP_Q = (A, S, F, \xi)$, a CPN representing a TCWS
Input: BRCPN-$EP_Q = (A', S', F^{-1}, \zeta)$, a CPN representing the compensation flow of the TCWS
Input: OWS: Ontology of WSs
Output: OV_Q: List of values of $o \mid o \in O_Q$
Output: $Quality_Q$: quality obtained after executing CPN-EP_Q

Initial phase:
begin
  Insert ws_{EE_i} in CPN-$EP_Q \mid ((ws_{EE_i})^\bullet = I_Q) \wedge ((^\bullet ws_{EE_i}) = \emptyset)$;
  Insert ws'_{EE_i} in BRCPN-$EP_Q \mid (^\bullet ws'_{EE_i} = \{a' \in A' \mid (a')^\bullet = \emptyset\}) \wedge ((ws'_{EE_i})^\bullet = \emptyset)$;
  Insert ws_{EE_f} in CPN-$EP_Q \mid ((ws_{EE_f})^\bullet = \emptyset) \wedge ((^\bullet ws_{EE_f}) = O_Q)$;
  Insert ws'_{EE_f} in BRCPN-$EP_Q \mid (^\bullet ws'_{EE_f} = \emptyset) \wedge ((ws'_{EE_f})^\bullet = \{a' \in A' \mid {}^\bullet a' = \emptyset\})$;
  $\forall a \in (A \cap I_Q)$, $M(a) = 1 \wedge \forall a \in (A - I_Q)$, $M(a) = 0$;
  /* Marks the CPN-EP_Q with the Initial Marking*/
  $\forall s' \in S'$, $\zeta(s') \leftarrow I$;
  /* the state of all transitions in BRCPN-EP_Q is *inicial* */
  repeat
    Instantiate an $ETWS_{ws}$;
    Send $Predecessors\_ETWS_{ws} \leftarrow^\bullet (^\bullet ws)$;
    Send $Successors\_ETWS_{ws} \leftarrow (ws^\bullet)^\bullet$;
    Send $WSDL_{ws}, OWLS_{ws}$; /* documentos semánticos */
    /* each ENGINE THREAD keeps the part of CPN-EP_Q and BRCPN-EP_Q which it concerns on*/
  until $\forall ws \in S \mid (ws \neq ws_{EE_i}) \wedge (ws \neq ws_{EE_f})$;
  Send values of I_Q to $(ws_{EE_i})^\bullet$;
  Execute Final phase;
end

Final phase:
begin
  repeat
    Wait Result from $(^\bullet(^\bullet ws_{EE_f}))$;
    if *message compensate is received* **then**
      Execute Compensation phase; /*this phase is shown in [8]*/ **Exit Final phase**;
    else
      Set values to OV_Q;
  until $(\forall o \in O_Q, M(o) = card(^\bullet o)$;
  /*o has a value an all predecessor transitions have finished*/
  if $\exists a \in OV_Q \mid M(o) \in Bag(\{e\})$ **then**
    /*one or more predecessors transitions of ws_{EE_f} have finished unsuccessfully*/
    Execute Checkpointing phase;
  else
    Send *Finish* message to $^\bullet(^\bullet ws_{EE_f})$;
    $Quality_Q \leftarrow recalculate\_Quality(S)$;/* Quality is recalculated in case some WSs were replaced */
    Return OV_Q,$Quality_Q$;
end

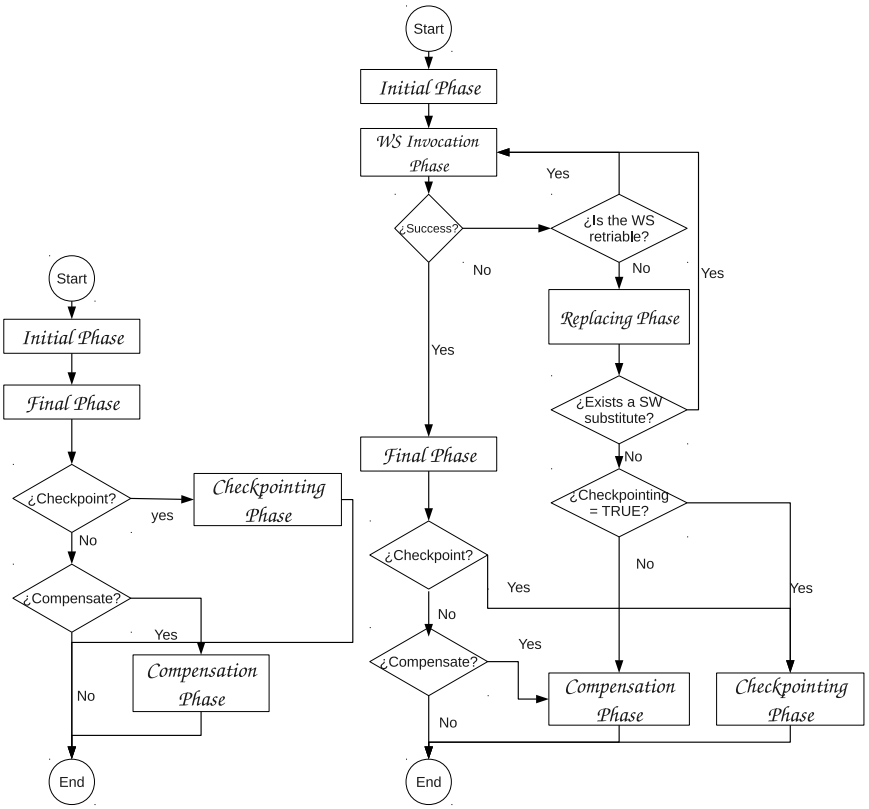

Fig. 7. Execution Engine & Engine Thread flow diagrams

number of WSs that will take part on the execution restart in order to get the 100% of the outputs. For example, for the TCWS of size 10, there was obtained the 91%, 86%, or 81% of the outputs during different executions, and there was not possible to execute a maximum of 3 WSs out of 10.

The *all-or-nothing* property is then relaxed to *something-to-all*, since it is possible to generate partial results and deliver them to the user, whilst the rest of the execution can be performed later; of course, it is up to the user to determine the usefulness of the partial results.

6 Related Work

Related work in the field of checkpointing for TCWSs is scarce. Prior works can be classified into two broad categories: works that require the user to specify the exact checkpointing location [15, 19, 21] and works that perform checkpointing in an automatic fashion[17, 22].

Table 2. Partial Outputs Results

TCWS **size**	**Output received (%)**	**WSs not executed after the failure**
03	72 - 58 - 43	1 - 2
04	78 - 67 - 56	1 - 2
05	82 - 73 - 64	1 - 2
06	85 - 77 - 70	1 - 2
07	87 - 80 - 74 - 60	1 - 2 - 3
08	88 - 82 - 77	1 - 2
09	90 - 85 - 79	1 - 2
10	91 - 86 - 81	1 - 2 - 3

The problem addressed in [15] is the strong mobility of CWSs; which is defined as the ability to migrate a running WS-BPEL process from a host to another to be resumed from a previous execution state. The proposed solution uses Aspect-Oriented Programming (AOP) in order to enable dynamic capture and recovery of a WS-BPEL process state. In [21] authors present a checkpointing approach based on Assurance Points (APs) and the use of integration rules. An AP is a combined logical and physical checkpoint, which during normal execution, stores execution state and invokes integration rules that check pre-conditions, post-conditions, and other application rule conditions. APs are also used as rollback points. Integration rules can invoke backward recovery to specific APs using compensation as well as forward recovery through rechecking preconditions before retry attempts or through execution of contingencies and alternative execution paths. APs together with integration rules provide an increased level of consistency checking as well as backward and forward recovery actions. This work does not specify the use of APs to restart the execution of the CWS later, or in another system. The goal of [19] is to provide a checkpointing scheme as the foundation for a recovery strategy for interorganizational information exchange. The authors adopt concepts from the mobile computing literature to decompose workflows into mobile agent-driven processes that will prospectively attach to web services-based organizational *docking stations*. This decomposition is extended in order to define logical points, within the dynamics of the entire workflow execution, that provide for locating accurate and consistent states of the system for recovery in case of a failure.

In contrast with works presented in [15, 19, 21], our checkpointing strategy is transparent to users and WS developers. They only have to ask for that facility, when a TCWS is submitted to be executed. As these works do, our strategy can be combined with backward and forward recovery techniques.

Recently research has been done in contrast to the checkpointing techniques wherein users have to specify the checkpointing location. In [22] authors propose a checkpointing policy which specifies that when a WS calls another WS, the calling WS has to save its state. The proposed checkpointing policy uses Predicted Execution Time (PET) and Mean Time Between Failures (MTBF), to decide on each WS invocation whether a checkpoint has to be taken or not.

For example, is a WS with PET < MTBF is called, then it is known that it will complete its execution within its MTBF and there is no need for checkpointing. In [17] the idea of checkpoints is rather to keep the execution history containing all successful operations, and at resume time, the system starts the workflow from the beginning but skips all operations that succeeded earlier.

As our approach, works described in [17, 22], proceed with checkpoints, without user intervention. In contrast, in our strategy, checkpoints are taken only in case of failures, so we do not increase the overhead while the execution is free of failures.

7 Conclusions and Future Work

In this work, we have presented a formal definition based on CPN properties for our checkpointing approach, providing an alternative to the previously presented all-or-nothing fault-tolerance mechanisms of WS retry, WS substitution, and compensation. The checkpointing mechanism defined in this paper allows to relax the all-or-nothing property to a *something-to-all* property. The idea is to execute a TCWS as much as possible (in the presence of failures) and then, taking a snapshot of that state. This mechanism allows users to receive partial answers as soon as they are produced (*something*) and provides the option of restarting the TCWS (to get *all* later) without losing the work previously done and without affecting the original transactional property. The formal definition was done by extending definitions of the CPN unrolling execution process and introducing new ones specific to checkpointing. We are currently working on an implementation comprising all our proposed fault-tolerance mechanisms in order to study and compare the performance among them and to provide a fully working real-world implementation.

References

1. Angarita, R., Cardinale, Y., Rukoz, M.: Faceta: Backward and forward recovery for execution of transactional composite ws. In: RED 2012 (2012)
2. Benjamins, R., Davies, J., Dorner, E., Domingue, J., Fensel, D., López, O., Volz, R., Wahler, A., Zaremba, M.: Service web 3.0, Tech. report, Semantic Technology Institutes International (2007)
3. Blanco, E., Cardinale, Y., Vidal, M.-E.: Aggregating Functional and Non-Functional Properties to Identify Service Compositions, pp. 1–36. IGI BOOK (2011)
4. Brogi, A., Corfini, S., Popescu, R.: Semantics-based composition-oriented discovery of web services. ACM Trans. Internet Techn. 8(4), 1–39 (2008)
5. Cardinale, Y., El Haddad, J., Manouvrier, M., Rukoz, M.: Web service selection for transactional composition. In: Int. Conf. on Computational Science (ICCS). Elsevier Science-Procedia Computer Science Series, vol. 1(1), pp. 2689–2698 (2010)
6. Cardinale, Y., El Haddad, J., Manouvrier, M., Rukoz, M.: CPN-TWS: A colored petri-net approach for transactional-qos driven web service composition. Int. Journal of Web and Grid Services 7(1), 91–115 (2011)

7. Cardinale, Y., El Haddad, J., Manouvrier, M., Rukoz, M.: Transactional-aware Web Service Composition: A Survey. IGI Global - Advances in Knowledge Management (AKM) Book Series, ch. 6, pp. 2–20 (2011)
8. Cardinale, Y., Rukoz, M.: Fault tolerant execution of transactional composite web services: An approach. In: Proc. of the Fifth Int. Conf. on Mobile Ubiquitous Computing, Systems, Services and Technologies, UBICOMM (2011)
9. Cardinale, Y., Rukoz, M.: A framework for reliable execution of transactional composite web services. In: Proc. of the Int. Conf. on Management of Emergent Digital EcoSystems, MEDES (2011)
10. El Haddad, J., Manouvrier, M., Rukoz, M.: TQoS: Transactional and QoS-aware selection algorithm for automatic Web service composition. IEEE Trans. on Services Computing 3(1), 73–85 (2010)
11. Hoffmann, J., Weber, I., Scicluna, J., Kaczmarek, T., Ankolekar, A.: Combining Scalability and Expressivity in the Automatic Composition of Semantic Web Services. In: Proc. of 8th Int. Conf. on Web Eng. (ICWE), pp. 98–107 (2008)
12. Hogg, C., Kuter, U., Munoz-Avila, H.: Learning Hierarchical Task Networks for Nondeterministic Planning Domains. In: The 21st Int. Joint Conf. on Artificial Intelligence, IJCAI 2009 (2009)
13. Ben Lakhal, N., Kobayashi, T., Yokota, H.: FENECIA: failure endurable nested-transaction based execution of composite Web services with incorporated state analysis. VLDB Journal 18(1), 1–56 (2009)
14. Liu, A., Li, Q., Huang, L., Xiao, M.: FACTS: A Framework for Fault Tolerant Composition of Transactional Web Services. IEEE Trans. on Services Computing 3(1), 46–59 (2010)
15. Marzouk, S., Maâlej, A.J., Jmaiel, M.: Aspect-oriented checkpointing approach of composed web services. In: Daniel, F., Facca, F.M. (eds.) ICWE 2010. LNCS, vol. 6385, pp. 301–312. Springer, Heidelberg (2010)
16. Mei, X., Jiang, A., Li, S., Huang, C., Zheng, X., Fan, Y.: A compensation paired net-based refinement method for web services composition. Advances in Information Sciences and Service Sciences 3(4) (2011)
17. Podhorszki, N., Ludaescher, B., Klasky, S.A.: Workflow automation for processing plasma fusion simulation data. In: WORKS 2007: Proceedings of the 2nd Workshop on Workflows in Support of Large-Scale Science, pp. 35–44. ACM, New York (2007)
18. Rukoz, M., Cardinale, Y., Angarita, R.: Faceta*: Checkpointing for transactional composite web service execution based on petri-nets. Procedia Computer Science 10, 874–879 (2012)
19. Sen, S., Demirkan, H., Goul, M.: Towards a verifiable checkpointing scheme for agent-based interorganizational workflow system "docking station" standards. In: Proceedings of the 38th Annual Hawaii International Conference on System Sciences, HICSS 2005, vol. 07, p. 165.1. IEEE Computer Society, Washington, DC (2005)
20. Thakker, D., Osman, T., Al-Dabass, D.: Knowledge-intensive semantic web services composition. In: Tenth Int. Conf. on Computer Modeling and Simulation, pp. 673–678 (2008)
21. Urban, S.D., Gao, L., Shrestha, R., Courter, A.: Achieving recovery in service composition with assurance points and integration rules. In: Meersman, R., Dillon, T.S., Herrero, P. (eds.) OTM 2010, Part I. LNCS, vol. 6426, pp. 428–437. Springer, Heidelberg (2010)
22. Vani Vathsala, A.: Article: Optimal call based checkpointing for orchestrated web services. International Journal of Computer Applications 36(8), 44–50 (2011)
23. Yu, Q., Liu, X., Bouguettaya, A., Medjahed, B.: Deploying and managing web services: issues, solutions, and directions. The VLDB Journal 17, 537–572 (2008)