Fault Tolerant Execution of Transactional Composite Web Services: An Approach

Yudith Cardinale
Departamento de Computación y Tecnología de la Información
Universidad Simón Bolívar
Caracas, Venezuela
Email: yudith@ldc.usb.ve

Marta Rukoz
LAMSADE, Université Paris Dauphine
Université Paris Ouest Nanterre La Défense
Paris, France
Email: marta.rukoz@lamsade.dauphine.fr

***Abstract*—We propose an approach for efficient, fault tolerant, and correct distributed execution of Transactional Composite Web Services (TCWSs), based on Colored Petri-Net (CPN) formalism. We extend a previous COMPOSER in order it generates, besides a TCWS represented by a CPN, another CPN representing the compensation order for backward recovery. We present an EXECUTER, which ensures correct execution flow and backward recovery by following unfolding processes of the CPNs. We present the formalization and algorithms of the TCWS execution and compensation processes.**

Keywords*-*Transactional Composite Web Services; Fault Tolerant Execution; Compensation; Backward Recovery.

I. INTRODUCTION

With the advent of Web 3.0, machines should contribute to users needs, by searching for, organizing, and presenting information from the Web which means, user can be fully automated on the Internet. One of the major goals of Web 3.0 is to make automatic and transparent to users the Web Service (WS) selection and composition to form more complex services. This process (executed by a COMPOSER) is normally based on functional requirements (i.e., the set of input attributes bounded in the query, and the set of attributes that will be returned as output), QoS criteria (e.g., response time and price), and transactional properties (e.g., compensable or not), producing Transactional Composite WSs (TCWSs). A TCWS is formed by many WSs and we call these WSs as components of the TCWS (WSs component). A TCWS should satisfy functional and transactional properties required by the user [1], [2], and it can be represented in a structure such as graph or Petri-Nets indicating the control flow and the WSs execution order.

In [2], we present such a COMPOSER. A brief description of this COMPOSER is presented in section III.

The contribution of this paper is focussed in two aspects. First, we extend our previous COMPOSER in order it automatically generates, besides the TCWS, another CPN representing the compensation order for a backward recovery process. Second, we specify an approach for efficient fault tolerant execution of TCWS; this approach is implemented in an EXECUTER. In the EXECUTER approach, the deployment of a TCWS will be carried on by following unfolding algorithms of CPNs representing the TCWS and its corresponding compensation flow in case of failures. The EXECUTER approach provides a *correct and fault tolerant execution* of TCWSs by: *(i)* ensuring that sequential and parallel WSs will be executed according the execution flow depicted by the TCWS; and *(ii)* in case of failures, leaving the system in a consistent state by executing a backward recovery with the CPN representing the compensation process. We formalize the TCWS execution problem and the backward recovery based on CPN properties. We also present the execution and compensation algorithms.

II. WSs TRANSACTIONAL PROPERTIES

A transactional property of a WS allows to recover the system in case of failures during the execution. In the related literature (see survey [3]), the most used WS transactional properties are the following. Let s be a WS: s is **pivot** (p), if once s successfully completes, its effects remain forever and cannot be semantically undone, if it fails, it has no effect at all; s is **compensatable** (c), if it exists another WS, s', which can semantically undo the execution of s; s is **retriable** (r), if s guarantees a successfully termination after a finite number of invocations; the retriable property can be combined with properties p and c defining **pivot retriable** (pr) and **compensatable retriable** (cr) WSs.

The Transactional Property (TP) of a Composite WS (CWS) can be derived from the properties of its WSs component and from their execution order (sequential or parallel). El Haddad et al. [4] extended the previous described transactional properties and adapted them to CWSs in order to define TCWSs as follows. Let cs be a TCWS: cs is **atomic** ($\vec{a}$), if once all its WSs component complete successfully, their effect remains forever and cannot be semantically undone, if one WS does not complete successfully, all previously successful WSs component have to be compensated; cs is **compensatable** (c), if all its WSs component are compensatable; cs is **retriable** (r), if all its WSs component are retriable; the retriable property can be combined with properties $\vec{a}$ and c defining **atomic retriable** ($\vec{a}r$) and **compensatable retriable** (cr) TCWSs.

According to these definitions, a TCWS must be constructed in such a way that if, at run-time, one of its WS component fails, then either it is retriable and can be invoked again until success or a backward recovery is possible (i.e.,

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-171-7

all successfully executed WSs have to be compensated).

III. Fault-Tolerant TCWS Composer

This section briefly describes our Composer [2] and the proposed extension in order to consider backward recovery. We formalize the WS composition problem by using Colored Petri-Nets (CPN), where WS inputs and outputs are represented by places and WSs with their transactional properties are represented by colored transitions.

A user query Q is defined in terms of functional conditions expressed as input (I_Q) and output (O_Q) attributes belonging to an ontology, QoS constraints expressed as weights over criteria, and the required global transactional property expressed as, T1 if TP of TCWS is in $\{\vec{a},\vec{a}r\}$ or T0 if TP of TCWS is in $\{c, cr\}$. More formally:

Definition 1: **Query.** Let $Onto_A$ be the integrated ontology (many ontologies could be used and integrated). A Query Q is a 4-tuple (I_Q, O_Q, W_Q, T_Q), where $I_Q = \{i \mid i \in Onto_A$ is an input attribute$\}$, $V_Q = \{\ (i, Op, v_i) \mid i \in I_Q$, Op is an operator ($Op \in \{=, \in\}$), and v_i is a value whose domain depends on i $\}$, $O_Q = \{o \mid o \in Onto_A$ is an output attribute whose value has 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 the result can be compensated. In both cases, if the execution is not successful, no result is reflected to the system, i.e., nothing is changed on the system.

The WSs Registry is represented by a Web Service Dependence Net ($WSDN$) modeled as a CPN containing all possible interactions among WSs. More formally.

Definition 2: **WSDN.** A WSDN 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 in the registry such that $A \subset Onto_A$;
- S is a finite set of transitions corresponding to the set of WSs in the registry;
- $F : (A \times S) \cup (S \times A) \rightarrow \{0, 1\}$ is a flow 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 $\xi : C_A \cup C_S$ with: $C_A : A \rightarrow \Sigma_A$, is a color function such that $\Sigma_A = \{I, \vec{a}, \vec{a}r, c, cr\}$ representing, for $a \in A$, either the TP of the CWS that can produce it or the user input (I), and $C_S : S \rightarrow \Sigma_S$, is a color function such that $\Sigma_S = \{p, pr, \vec{a}, \vec{a}r, c, cr\}$ representing the TP of $s \in S$.

The WS composition problem is solved by a Petri-Net unfolding algorithm which embeds the QoS-driven selection within the transactional service selection. To start the Composer unfolding algorithm, the $WSDN$ is marked with tokens on places representing the input attributes (these marks represent the initial marking). At the end, the unfolding algorithm will define the CPN representing the composition that satisfies the **Query**. The transactional property of the resulting CWS is derived from the transactional properties of its WSs component and the structure of the CPN. Thus, the result of the composition process is a CPN corresponding to a TCWS whose WSs component locally maximize the QoS and globally satisfy the required functional and transactional properties. Formally, we say:

Definition 3: CPN-$TCWS_Q$. A CPN-$TCWS_Q$ 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 WSs in the TCWS such that $A \subset Onto_A$;
- S is a finite set of transitions corresponding to the set of WSs in the TCWS;
- $F : (A \times S) \cup (S \times A) \rightarrow \{0, 1\}$ is a flow 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); this relation establishes the input and output execution dependencies among WSs component.
- ξ is a color function such that ξ: $S \rightarrow \Sigma_S$ and $\Sigma_S = \{p, pr, \vec{a}, \vec{a}r, c, cr\}$ represents the TP of $s \in S$ ($TP(s)$).

For modeling TCWS backward recovery, our Composer can be easily extended in order it can generate a backward CPN , that we called BRCPN-$TCWS_Q$, associated to a CPN-$TCWS_Q$ as follows:

Definition 4: BRCPN-$TCWS_Q$. A BRCPN-$TCWS_Q$, associated to a given CPN-$TCWS_Q$=(A, S, F, ξ), is a 4-tuple (A', S', F^{-1}, ζ), where:

- A' is a finite set of places corresponding to the CPN-$TCWS_Q$ places such that: $\forall a' \in A'\ \exists a \in A$ associated to a' and a' has the same semantic of a.
- S' is a finite set of transitions corresponding to the set of compensation WSs in CPN-$TCWS_Q$ such that: $\forall s \in S$, $TP(s) \in \{c, cr\}, \exists s' \in S'$ which compensate s.
- $F^{-1} : (A \times S) \cup (S \times A) \rightarrow \{0, 1\}$ is a flow relation establishing the restoring order in a backward recovery defined as: $\forall s' \in S'$ associated to $s \in S$, $\exists a' \in A'$ associated to $a \in A \mid F^{-1}(a', s') = 1 \Leftrightarrow F(s, a) = 1$ and $\forall s' \in S'$, $\exists a' \in A' \mid F^{-1}(s', a') = 1 \Leftrightarrow F(a, s) = 1$.
- ζ is a color function such that $\zeta : S' \rightarrow \Sigma'_S$ and $\Sigma'_S = \{I, R, E, C, A\}$ represents the execution state of $s \in S$ associated to $s' \in S'$ (I: initial, R: running, E: executed, C: compensate, and A: abandoned).

The marking of a CPN-$TCWS_Q$ or BRCPN-$TCWS_Q$ represents the current values of attributes that can be used for some WSs component to be executed or control values indicating the compensation flow, respectively. A Marked CPN denotes which transitions can be fired.

Definition 5: **Marked** CPN. A marked CPN=(A, S, F, ξ) is a pair (CPN,M), where M is a function which assigns tokens (values) to places such that $\forall a \in A$, $M(a) \in N$.

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

Definition 6: **Fireable CPN transition.** A marking M enables a transition s iff all its input places contain tokens such that $\forall x \in ({}^\bullet s), \wedge\ M(x) \geq card({}^\bullet x)$.

Note that a transition is actually fireable if on each input place there are as many tokens as predecessor transitions produce them. This condition and the fact CPN-$TCWS$ is acyclic, guaranty that a transition is fireable only if all its predecessor transitions have been fired. Then, sequential WSs execution is controlled by input and output dependencies. If several transitions are fireable, all of them are fired (i.e., the corresponding WSs are executed in parallel). Hence, the sequential or parallel execution condition affecting the global TP is ensured. Figure 1 illustrates this definition. 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 if ws_2 and ws_3 were executed in parallel, it could be possible that ws_3 finishes successful and ws_2 fails; in this case, the system can not be recovery because $TP(ws_3) = pr$ do not allow compensation.

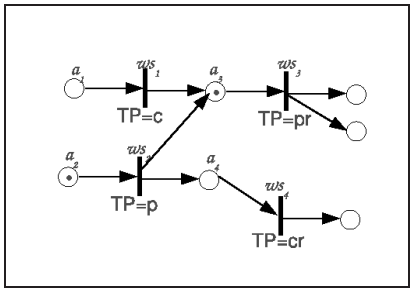

Figure 1. Example of Fireable Transitions

In the BRCPN-$TCWS$, a transition color represents the execution state of its corresponding compensable WS. A compensation transition can be fired only if the corresponding WS is not being abandoned or compensated (Def. 7).

Definition 7: **Fireable compensation transition.** A marking M enables a transition s' iff all its input places contain tokens such that $\forall a' \in ({}^\bullet s'),\ M(a') \neq 0 \wedge \zeta(s') \notin \{A, C\}$.

IV. Executer: Fault-Tolerant Execution Control

Once a CPN-$TCWS_Q$ and its corresponding BRCPN-$TCWS_Q$ are generated by the Composer, an Executer has to deploy the execution of the TCWS. The execution control of a TCWS is guided by a unfolding algorithm of its corresponding CPN-$TCWS_Q$. To support backward recovery, it is necessary to keep the trace of the execution on the BRCPN-$TCWS_Q$. To start the unfolding algorithm, the CPN-$TCWS_Q$ is marked with the *Initial Marking*: an initial token is added to places representing inputs of Q ($\forall a \in (A \cap I_Q), M(a) = 1$, $\forall a \in (A - I_Q),\ M(a) = 0$) and the state of all transitions in BRCPN-$TCWS_Q$ is set to *initial* ($\forall s' \in S',\ \zeta(s') \leftarrow I$). The firing of a transition in CPN-$TCWS_Q$ corresponds to the execution of a WS (or CWS), let say s, which participates in the composition. While a compensatable s is executing, the state of its corresponding s' in BRCPN-$TCWS_Q$ is set to *running* ($\zeta(s') \leftarrow R$). Then, when s finishes, it is considered that the transition was fired, others transitions become fireable, the state of its corresponding s' is set on *executed* ($\zeta(s') \leftarrow E$), and the following firing rules are applied.

Definition 8: **CPN-$TCWS_Q$ 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 ($\forall x \in {}^\bullet s$, $M(x) = 0$), if the $TP(s) \in \{c, cr\}$, the state of its corresponding s' in BRCPN-$TCWS_Q$ is set to *running* ($\zeta(s') \leftarrow R$), and the WS s is invoked. These actions are atomically executed. After WS s finishes, tokens are added to its output places ($\forall x \in (s^\bullet)$, $M(x) = M(x) + 1$), and the state of its corresponding s' in BRCPN-$TCWS_Q$ (if it exists) is set to *executed* ($\zeta(s') \leftarrow E$). These actions are also atomically executed.

In case of failure of a WS s, depending on the $TP(s)$, the following actions could be executed:

- if $TP(s)$ is retriable (pr, $\vec{a}r$, cr), s is re-invoked until it successfully finish (forward recovery);
- otherwise, 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.

In order to consider failures, the compensation control of a CPN-$TCWS_Q$ is guided by a unfolding algorithm of its associated BRCPN-$TCWS_Q$. When a WS represented by a transition s fails, the unfolding process over CPN-$TCWS_Q$ is halted and a backward recovery is initiated with the unfolding process over BRCPN-$TCWS_Q$ by marking it with its *Initial Marking*: a token is added to places representing inputs of BRCPN-$TCWS_Q$ ($\forall a' \in A' \mid {}^\bullet a' = \emptyset$, $M(a') = 1$), tokens are added to places representing inputs of s ($\forall a \in {}^\bullet s$, $M(a') = card({}^\bullet x)$, and other places has no tokens. Then, fireable compensation transitions defined in Def. 7 and the firing rules defined in Def. 9 guide the unfolding process of BRCPN-$TCWS_Q$.

Definition 9: **BRCPN-$TCWS_Q$ Firing rules.** The firing of a fireable transition (see Def. 7) s' for a marking M defines a new marking M', such that:

- if $\zeta(s') = I$, $\zeta(s') \leftarrow A$ (i.e., the corresponding s is abandoned before its execution),
- if $\zeta(s') = R$, $\zeta(s') \leftarrow C$ (in this case s' is executed after s finishes, then s is compensated),
- if $\zeta(s') = E$, $\zeta(s') \leftarrow C$ (in this case s' is executed, i.e., s is compensated),
- tokens are deleted from its input places ($\forall x \in {}^\bullet s'$, $M(x) = M(x) - 1$) and tokens are added to its output places ($\forall x \in (s'^\bullet)$, $M(x) = M(x) + 1$),

We illustrate a backward recovery in Figure 2. The marked CPN-$TCWS_Q$ depicted in Figure 2(a) is the state when ws_4 fails, the unfolding of CPN-$TCWS_Q$ is halted, and the initial marking on the corresponding BRCPN-$TCWS_Q$ is set to start its unfolding process (see Figure 2(b)), after ws'_3 and ws'_5 are fired and ws_7 is abandoned before its invocation, a new marking is produced (see Figure 2(c)), in which ws'_1 and

ws'_2 are both fireable and can be invoked in parallel. Note that only compensatable transitions have their corresponding compensation transitions in BRCPN-$TCWS_Q$.

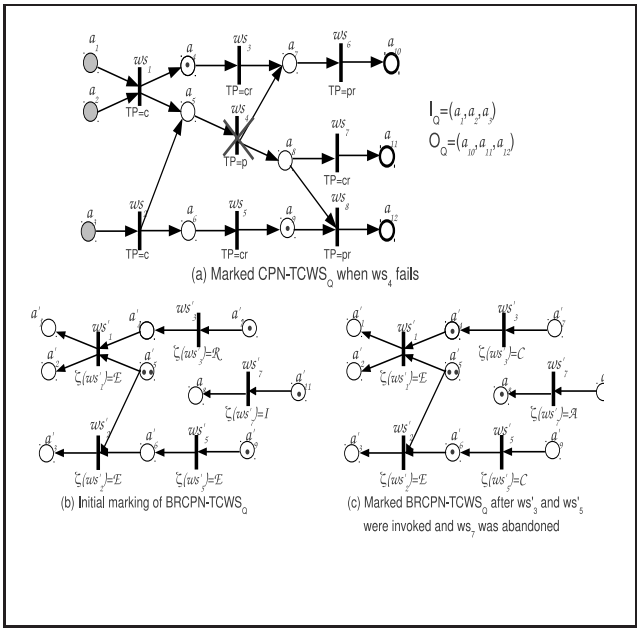

Figure 2. Example of BRCPN-$TCWS_Q$

V. EXECUTER APPROACH

In our approach, the execution of a TCWS is managed by an EXECUTER, which in turn is a collection of software components called EXECUTION ENGINE and ENGINE THREADS. One ENGINE THREAD is assigned to each WS in the TCWS. The EXECUTION ENGINE and its ENGINE THREADS are in charge of initiating, controlling, and monitoring the execution, as well as collaborating with its peers to deploy the TCWS execution. By distributing the responsibility of executing a TCWS across several ENGINE THREADS, the logical model of our EXECUTER 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. EXECUTION ENGINE and ENGINE THREADS are placed in different physical nodes from those where actual WSs are placed. ENGINE THREADS remotely invoke the actual WSs component. The EXECUTION ENGINE needs to have access to the WSs Registry, which contains the WSDL and OWLS documents. The knowledge required at run-time 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.

Typically, WSs are distinguished in *atomic* and *composite* WSs. An atomic WS is one that solely invokes local operations that it consists of (e.g., *WSDL and OWLS documents* define atomic WSs as collection of operations together with abstract descriptions of the data being exchanged). A composite WS is one that additionally accesses other WSs or, in particular, invokes operations of other WSs. Hereby, these additional involved WSs may be provided by different organizations and were registered in the Registry as a CWS (e.g., a *WS-BPEL document* defines CWSs by describing interactions between business entities through WS operations). In our case, we consider that transitions in the CPN, representing the TCWS to be executed, could be atomic WSs or CWSs (TCWSs in our case). Atomic WSs have its corresponding *WSDL and OWLS documents*. TCWSs can be encapsulated into an EXECUTER; in this case the EXECUTION ENGINE has its corresponding *WSDL and OWLS documents*. Hence, TCWSs may themselves become a WS, making TCWS execution a recursive operation.

TCWS Execution and Backward Recovery

We present the four phases of the fault tolerant execution algorithm by pointing out which components of the EXECUTER are in charge of carrying on which task. Algorithms 1, 2, and 3 describe in detail all phases.

Initial phase: Whenever an EXECUTION ENGINE receives a CPN-$TCWS_Q$ and its corresponding BRCPN-$TCWS_Q$ (see Def. 3 and Def. 4), it performs the following tasks: *(i)* add two *dummy* transitions to CPN-$TCWS_Q$: ws_{EE_i}, the first transition providing the inputs referenced in Q (I_Q) and ws_{EE_f}, the last transition consuming the outputs (O_Q); similar *dummy* transitions are added to BRCPN-$TCWS_Q$ with inverse data flow relation (ws'_{EE_i} and ws'_{EE_f}); these transitions are represented by the EXECUTION ENGINE and have only control responsibilities to start the unfolding process and know when it is finished; *(ii)* mark the CPN-$TCWS_Q$ with the Initial Marking (i.e., add tokens to places representing the attributes in I_Q) and mark all transitions in BRCPN-$TCWS_Q$ in *initial* state; *(iii)* start an ENGINE THREAD responsible for each transition in CPN-$TCWS_Q$, except by ws_{EE_i} and ws_{EE_f}, indicating to each one its predecessor and successor transitions as CPN-$TCWS_Q$ indicates (for BRCPN-$TCWS_Q$ the relation is inverse) and the corresponding WSDL and OWLS documents (they describe the WS in terms of its inputs and outputs and who is the compensation WS, if it is necessary); and *(iv)* send values of attributes in I_Q to ENGINE THREADS representing successors of ws_{EE_i}. In Algorithm 1, lines 1 to 14 describe these steps.

WS Invocation phase: Once each ENGINE THREAD is started, it retrieves the corresponding WSDL and OWLS documents to extract information about the required inputs and to construct the invocation. It waits its WS becomes fireable to invoke it (see Def. 6). Whenever an ENGINE THREAD receives all the inputs needed it sets to *running* the state of its corresponding transition in BRCPN-$TCWS_Q$

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-171-7

and invokes its corresponding WS with its corresponding inputs. When the WS finishes successfully, the ENGINE THREAD changes to *executed* the state of its corresponding transition in BRCPN-$TCWS_Q$ and sends values of WS outputs to ENGINE THREADS representing successors of its WS. If the WS fails during the execution, if TP(WS) is retriable, the WS is re-invoked until it successfully finish; otherwise the *Compensation phase* has to be executed. In Algorithm 2, lines 1 to 7 describe this phase.

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 with a message *compensate*, marks the respective transition in BRCPN-$TCWS_Q$ to *compensate* state and sends control tokens to transitions successor of the compensation WS. The EXECUTION ENGINE sends a message *compensate* to all ENGINE THREADS, marks the BRCPN-$TCWS_Q$ with the Initial Marking (i.e., adds tokens to places representing inputs of BRCPN-$TCWS_Q$ and inputs of the faulty WS), and sends control tokens to ENGINE THREADS representing successors of ws'_{EE_f}. Once the rest of ENGINE THREADS receive the message *compensate*, they apply the firing rules in BRCPN-$TCWS_Q$ (see Def. 9). The compensation process finishes when ws'_{EE_i} becomes fireable. Algorithm 3 describe these steps for both EXECUTION ENGINE and ENGINE THREADS.

Final phase: This phase is carried out by both EXECUTION ENGINE and ENGINE THREADS. If the TCWS was successfully executed (ws_{EE_f} becomes fireable) the EXECUTION ENGINE notifies all ENGINE THREADS predecessors of ws_{EE_f} by sending *Finish* message and returns the values of attributes in O_Q to user. When ENGINE THREADS receive the *Finish* message, they backward this message to its ENGINE THREAD predecessors and return. In case compensation is needed, the EXECUTION ENGINE receives a message *compensate*, the process of executing the TCWS is stopped, and the compensation process is started by sending a message *compensate* to all ENGINE THREADS. If an ENGINE THREAD receives a message *compensate*, it launches the compensation protocol. Algorithm 1 (lines 15-18) and Algorithm 2 (lines 8- 10) describe this phase for EXECUTION ENGINE and ENGINE THREADS respectively.

In order to guarantee the correct execution of our algorithms, the following assumptions are made: *i)* the network ensures that all packages are sent and received correctly; *ii)* the EXECUTION ENGINE and ENGINE THREADS run in a reliable server, they do not fail; and *iii)* the WSs component can suffer silent or stop failures (WSs do not response because they are not available or a crash occurred in the platform); run-time failures caused by error in inputs attributes and byzantine faults are not considered.

Algorithm 1: EXECUTION ENGINE Algorithm

Input: $Q = (I_Q, O_Q, W_Q, R_Q)$, the user query – see Def. 1
Input: CPN-$TCWS_Q = (A, S, F, \xi)$, a CPN allowing the execution of a TCWS– see Def. 3
Input: BRCPN-$TCWS_Q = (A', S', F^{-1}, \zeta)$, a CPN representing the compensation flow of TCWS– see Def. 4
Input: OWS: Ontology of WSs
Output: OV_Q: List of values of $o \mid o \in O_Q$
begin
1 **Initial phase**:
begin
2 Insert ws_{EE_i} in CPN-$TCWS_Q \mid ((ws_{EE_i})^\bullet = I_Q) \wedge ((^\bullet ws_{EE_i}) = \emptyset)$;
3 Insert ws'_{EE_i} in BRCPN-$TCWS_Q \mid (^\bullet ws'_{EE_i} = \{a' \in A' \mid (a')^\bullet = \emptyset\}) \wedge ((ws'_{EE_i})^\bullet = \emptyset)$;
4 Insert ws_{EE_f} in CPN-$TCWS_Q \mid ((ws_{EE_f})^\bullet = \emptyset) \wedge ((^\bullet ws_{EE_f}) = O_Q)$;
5 Insert ws'_{EE_f} in BRCPN-$TCWS_Q \mid (^\bullet ws'_{EE_f} = \emptyset) \wedge ((ws'_{EE_f})^\bullet = \{a' \in A' \mid {}^\bullet a' = \emptyset\})$;
6 $\forall a \in (A \cap I_Q)$, $M(a) = 1 \wedge \forall a \in (A - I_Q)$, $M(a) = 0$;
/* Mark the CPN-$TCWS_Q$ with the Initial Marking*/
7 $\forall s' \in S'$, $\zeta(s') \leftarrow I$;
/* state of all transitions in BRCPN-$TCWS_Q$ is set to *initial* */
8 **repeat**
9 Instantiate an $ETWS_{ws}$;
10 Send $Predecessors\_ETWS_{ws} \leftarrow^\bullet (^\bullet ws)$;
11 Send $Successors\_ETWS_{ws} \leftarrow (ws^\bullet)^\bullet$;
12 Send $WSDL_{ws}, OWLS_{ws}$; /* Semantic web documents */
/* each ENGINE THREAD keep the part of CPN-$TCWS_Q$ and BRCPN-$TCWS_Q$ which it concerns on*/
until $\forall ws \in S \mid (ws \neq ws_{EE_i}) \wedge (ws \neq ws_{EE_f})$;
13 Send values of I_Q to $(ws_{EE_i})^\bullet$;
14 **Execute Final phase**;
end
15 **Final phase**:
begin
16 **repeat**
Wait Result from $(^\bullet(^\bullet ws_{EE_f}))$;
if *message compensate is received* **then**
Execute Compensation Phase /* this phase is shown in Algorithm 3*/;
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 transition predecessors have finished*/
17 Send $Finish$ message to $^\bullet(^\bullet ws_{EE_f})$;
18 Return OV_Q;
end
/*Send instructions are necessary if ENGINE THREADS are executed in a distributed system, otherwise in a shared memory system, ENGINE THREADS can access directly CPN-$TCWS_Q$ to obtain this information*/
end

VI. RELATED WORK

There exist some recent works related to compensation mechanism of CWSs based on Petri-Net formalism [5]–[7]. The compensation process is represented by Paired Petri-Nets demanding that all WSs component have to be compensatable. Our approach considers other transactional properties (e.g., pr, cr, $\vec{a}r$) that also allow forward recovery and the compensation Petri-Net can model only the part of the TCWS that is compensable. Besides, in those works, the Petri-Nets are manually generated and need to be verified, while in our approach they are automatically generated.

Regarding the decentralized fault tolerant execution model, we can distinct two kinds of distributed coordination

Algorithm 2: ENGINE THREAD Algorithm

Input: $Predecessors\_ETWS_{ws}$, WS predecessors of ws
Input: $Successors\_ETWS_{ws}$, WS successors of ws
Input: $WSDL_{ws}, OWLS_{ws}$, semantic web documents
begin
Invocation phase:
begin
$InputsNeeded\_ETWS_{ws} \leftarrow getInputs(WSDL_{ws}, OWLS_{ws})$;
repeat
Wait Result from ($Predecessors\_ETWS_{ws}$));
Set values to $InputsNeeded\_ETWS_{ws}$;
until $\forall a \in InputsNeeded\_ETWS_{ws}, M(a) = card(^\bullet a)$;
/* a has a value and all transition predecessors have finished */
$success \leftarrow false$;
$compensate \leftarrow false$;
$\zeta(ws') \leftarrow R$;
repeat
Invoke ws;
if *(ws fails)* **then**
if $TP(ws) \in \{pr, ar, cr\}$ **then**
Re-invoke ws;
else
$compensate \leftarrow true$;
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 $(success) \vee (compensate)$;
if $compensate$ **then**
Send *compensate* to EXECUTION ENGINE;
$\zeta(ws') \leftarrow C$;
Execute Compensation phase;/* backward recovery: this phase is shown in Algorithm 3 */
else
Execute Final phase;
end
Final phase:
begin
Wait $message$;
if $message$ *is* $Finish$ **then**
Send $Finish$ message to $Predecessors\_ETWS_{ws}$;
Return;
else
Execute Compensation phase;
end
/* In a shared memory system $Predecessors\_ETWS_{ws}$ can be accessed as $^\bullet(^\bullet ws)$; $Successors\_ETWS_{ws}$ as $(ws^\bullet)^\bullet$; and $InputsNeeded\_ETWS_{ws}$ as $(^\bullet ws)$, because all ENGINE THREADS share the CPN-$TCWS_Q$ and none send is necessary */
end

Algorithm 3: Compensation Protocol

begin
EXECUTION ENGINE:
begin
$\forall a' \in A' \mid {}^\bullet a' = \emptyset, M(a') = 1 \wedge \forall a \in {}^\bullet s, M(a') = 1$;
/* Mark the BRCPN-$TCWS_Q$ with the Initial Marking*/ Send *compensate* to all ENGINE THREADS;
Send control values to $^\bullet(^\bullet ws'_{EE_f})$;
Wait control values from $((ws'_{EE_i})^\bullet)^\bullet$;
Return ERROR;
end
ENGINE THREADS:
begin
$ws' \leftarrow$ WS which compensates its WS;
if $\zeta(ws') = A \vee \zeta(ws') = C$ **then**
Send Control tokens to $Successors\_ETWS_{ws'}$;
else
$InputsNeeded\_ETWS_{ws'} \leftarrow getInputs(WSDL_{ws'}, OWLS_{ws'})$;
repeat
Wait Control tokens from $Predecessors\_ETWS_{ws'}$;
Set Control tokens to $InputsNeeded\_ETWS_{ws'}$;
until $(\forall a' \in InputsNeeded\_ETWS_{ws'}, M(a') \neq \emptyset)$;
/* Wait its corresponding ws' becomes fireable: a' has a control value and all transition predecessors have finished*/
if $\zeta(ws') = I$ **then**
$\zeta(ws') \leftarrow A$
if $\zeta(ws') = R$ **then**
Wait ws finishes;
Invoke ws';
$\zeta(ws') \leftarrow C$
;
if $\zeta(ws') = E$ **then**
Invoke ws';
$\zeta(ws') \leftarrow C$;
Send Control tokens to $Successors\_ETWS_{ws'}$;
Return /* ENGINE THREAD finishes */;
end
end

approach. In the first one, nodes interact directly. In the second one, they use a shared space for coordination. FENECIA framework [8] introduces WS-SAGAS, a transaction model based on arbitrary nesting, state, vitality degree, and compensation concepts to specify fault tolerant CWS as a hierarchy of recursively nested transactions. To ensure a correct execution order, the execution control of the resulting CWS is hierarchically delegated to distributed engines that communicate in a peer-to-peer fashion. FACTS [1], is another framework which extends the FENECIA transactional model. When a fault occurs at run-time, it first employs appropriate exception handling strategies to repair it. If the fault has been fixed, the TCWS continues its execution. Otherwise, it brings the TCWS back to a consistent termination state according to the termination protocol. In [9] a fault handling and recovery CWSs, in a decentralized orchestration approach that is based on continuation-passing messaging, is presented. Nodes interpret such messages and conduct the execution of services without consulting a centralized engine. However, this coordination mechanism implies a tight coupling of services in terms of spatial and temporal composition. Nodes need to know explicitly which other nodes they will potentially interact with, and when, to be active at the same time. They are frameworks to support users and developers to construct TCWS based on WS-BPEL, then they are not transparent.

In [10], [11] engines based on a peer-to-peer application architecture, wherein nodes are distributed across multiple computer systems, are used. In these architectures the nodes collaborate, in order to execute a CWS with every node executing a part of it. In [10], the execution is controlled by the component state and routing tables in each node containing the precondition and postprocessing actions indicating which components needs to be notified when a state is exited. In [11], the authors introduce service invocation triggers, a lightweight infrastructure that routes messages directly from a producing service to a consuming one, where each service invocation trigger corresponds to the invocation of a WS.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-171-7

Another series of works rely on a shared space to exchange information between nodes of a decentralized architecture, more specifically called a tuplespace. Using tuplespace for coordination, the execution of a (part of a) workflow within each node is triggered when tuples, matching the templates registered by the respective nodes, are present in the tuplespace. Thus, the templates a component uses to consume tuples, together with the tuples it produces, represent its coordination logic. In [12], [13] is presented a coordination mechanism where the data is managed using tuplespace and the control is driven by asynchronous messages exchanged between nodes. This message exchange pattern for the control is derived from a Petri net model of the workflow. In [14], an alternative approach is presented, based on the chemical analogy. The proposed architecture is composed by nodes communicating through a shared space containing both control and data flows, called the multiset. The chemical paradigm is a programming style based on the chemical metaphor. Molecules (data) are floating in a chemical solution, and react according to reaction rules (program) to produce new molecules (resulting data). As this approach, in our approach the coordination mechanism stores both control and data information independent of its implementation (distributed or shared memory). However, none of these works manage failures during the execution.

Facing our approach against all these works, we overcome them because the execution control is distributed and independent of the implementation (it can be implemented in distributed or shared memory platforms), it efficiently executes TCWSs by invoking parallel WSs according the execution order specified by the CPN, and it is totally transparent to users and WS developers, i.e., user only provides its TCWS, that was automatically generated by the COMPOSER and no instrumentation/modification/specification is needed for WSs participating in the TCWS. while most of these works are based on WS-BPEL and/or some control is sitting closely to WSs and have to be managed by programmers.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a fault tolerant execution control mechanism for ensuring *correct and fault tolerant execution order* of TCWSs. Our approach ensures that the deployment of the TCWS will be carried on by following unfolding algorithms of CPNs representing the TCWS and the compensation process. We are currently working on extending the approach with forward recovery based on WS substitution. We are also implementing prototype systems to test the performance of the approach in centralized and decentralized platforms. Our intention is to compare both implementations under different scenarios (different characterizations of CPNs) and measure the impact of compensation and substitution on QoS.

REFERENCES

[1] A. Liu, Q. Li, L. Huang, and M. Xiao, "FACTS: A Framework for Fault Tolerant Composition of Transactional Web Services," *IEEE Trans. on Services Computing*, vol. 3, no. 1, pp. 46–59, 2010.

[2] Y. Cardinale, J. El Haddad, M. Manouvrier, and M. Rukoz, "CPN-TWS: A colored petri-net approach for transactional-qos driven web service composition," *International Journal of Web and Grid Services*, vol. 7, no. 1, pp. 91–115, 2011.

[3] ——, *Transactional-aware Web Service Composition: A Survey*. IGI Global - Advances in Knowledge Management (AKM) Book Series, 2011, pp. 116–141.

[4] J. El Haddad, M. Manouvrier, and M. Rukoz, "TQoS: Transactional and QoS-aware selection algorithm for automatic Web service composition," *IEEE Trans. on Services Computing*, vol. 3, no. 1, pp. 73–85, 2010.

[5] Y. Wang, Y. Fan, and A. Jiang;, "A paired-net based compensation mechanism for verifying Web composition transactions," in *The 4th Int. Conf. on New Trends in Information Science and Service Science*, 2010.

[6] F. Rabbi, H. Wang, and W. MacCaull, "Compensable workflow nets," in *Formal Methods and Software Engineering - 12th Int. Conf. on Formal Engineering Methods*, ser. LNCS, 2010, vol. 6447, pp. 122–137.

[7] X. Mei, A. Jiang, S. Li, C. Huang, X. Zheng, and Y. Fan, "A compensation paired net-based refinement method for web services composition," *Advances in Information Sciences and Service Sciences*, vol. 3, no. 4, May 2011.

[8] N. B. Lakhal, T. Kobayashi, and H. Yokota, "FENECIA: failure endurable nested-transaction based execution of compo site Web services with incorporated state analysis," *VLDB Journal*, vol. 18, no. 1, pp. 1–56, 2009.

[9] W. Yu, "Fault handling and recovery in decentralized services orchestration," in *The 12th International Conference on Information Integration and Web-based Applications & Services*, ser. iiWAS '10. ACM, 2010, pp. 98–105.

[10] D. M. Benatallah Boualem, Sheng Quan, "The self-serv environment for web services composition," *IEEE Internet Computing*, pp. 40–48, 2003.

[11] W. Binder, I. Constantinescu, and B. Faltings, "Decentralized orchestration of compositeweb services," in *The IEEE International Conference on Web Services*. IEEE Computer Society, 2006, pp. 869–876.

[12] D. Martin, D. Wutke, and F. Leymann, "Tuplespace middleware for petri net-based workflow execution," *Int. J. Web Grid Serv.*, vol. 6, pp. 35–57, March 2010.

[13] P. Buhler and J. M. Vidal, "Enacting BPEL4WS specified workflows with multiagent systems," in *The Workshop on Web Services and Agent-Based Engineering*, 2004.

[14] H. Fernandez, T. Priol, and C. Tedeschi, "Decentralized approach for execution of composite web services using the chemical paradigm," in *IEEE Int. Conf. on Web Services*, 2010, pp. 139–146.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-171-7