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 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) 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 retrimable (\( r \)), if \( s \) guarantees a successfully termination after a finite number of invocations; the retrimable property can be combined with properties \( p \) and \( c \) defining pivot retrimable (\( pr \)) and compensatable retrimable (\( cr \)) 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 (\( \bar{a} \)), 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 retrimable (\( r \)), if all its WSs component are retrimable; the retrimable property can be combined with properties \( \bar{a} \) and \( c \) defining atomic retrimable (\( \bar{a}r \)) and compensatable retrimable (\( cr \)) 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 retrimable and can be invoked again until success or a backward recovery is possible (i.e.,
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, $T_1$ if $TP$ of TCWS is in $\{\tilde{a},\tilde{ar}\}$ or $T_0$ 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 $I_Q = \{i \mid i \in Onto_A \text{ is an input attribute}\}, V_Q = \{(i, Op, V_i) \mid i \in I_Q, Op \in \{=, \neq\}\}$, and $V_i$ is a value whose domain depends on $i$. $O_Q = \{o \mid o \in Onto_A \text{ is an output attribute whose value has to be produced by the system}\}, W_Q = \{(w_i, q_i) \mid w_i \in [0, 1]\text{, with } \sum w_i = 1\text{ and } q_i \text{ is a } QoS\text{ criterion}\}$, 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 $(A, S, F, \xi)$, 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: $\forall s \in S, (\exists a \in A \mid F(a, s) = 1) <\Leftrightarrow (a \text{ is an input place of } s)$ and $\forall s \in S, (\exists a \in A \mid F(s, a) = 1) <\Leftrightarrow (a \text{ is an output place of } s)$.
- $\xi$ 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 $\Sigma_A = \{I, \tilde{a}, \tilde{ar}, c, cr\}$ and $C_S : S \rightarrow \Sigma_S$, is a color function such that $\Sigma_S = \{p, pr, \tilde{a}, \tilde{ar}, c, cr\}$.

The WCW 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 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 $(A, S, F, \xi)$, 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: $\forall s \in S, (\exists a \in A \mid F(a, s) = 1) <\Leftrightarrow (a \text{ is an input place of } s)$ and $\forall s \in S, (\exists a \in A \mid F(s, a) = 1) <\Leftrightarrow (a \text{ is an output place of } s)$; this relation establishes the input and output execution dependencies among WSS component.
- $\xi$ is a color function such that $\xi : S \rightarrow \Sigma_S$ and $\Sigma_S = \{p, pr, \tilde{a}, \tilde{ar}, c, cr\}$ represents the CP 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 $Q = (A, S, F, \xi)$, is a 4-tuple $(A', S', F^{-1}, \zeta)$, 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: $\forall s \in S, TP(s) \in \{c, cr\}, \exists s' \in S' \text{ which compensate } s$.
- $F^{-1} : (A \times S) \cup (S \times A) \rightarrow \{0, 1\}$ is a flow relation establishing the restoring order in a backward recovery defined as: $\forall s' \in S' \text{ associated to } s \in S, (\exists a' \in A' \text{ associated to } a \in A \mid F^{-1}(a', s') = 1) <\Leftrightarrow F(s, a) = 1 \text{ and } \forall s' \in S', (\exists a' \in A' \mid F^{-1}(a', s') = 1) <\Leftrightarrow F(a, s) = 1$.
- $\zeta$ is a color function such that $\zeta : S' \rightarrow \Sigma_{\delta}$ and $\Sigma_{\delta} = \{I, R, E, C, A\}$ 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 $= (A, S, F, \xi, M)$ 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, $(x) = \{y \in A \cup S : F(y, x) = 1\}$ is the set of its predecessors, and $(x^*) = \{y \in A \cup S : F(x, y) = 1\}$ is the set of its successors. Now we can define fireable transitions.
Definition 6: Fireable CPN transition. A marking $M$ enables a transition $s$ if all its input places contain tokens such that $\forall x \in (s), M(x) \geq card(x)$.

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-TCW S 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 $ws_4$ fails; in this case, the system can not be recovery because $TP(ws_3) = pr$ do not allow compensation.

In the BRCPN-TCW S, 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 $s'$ if all its input places contain tokens such that $\forall a' \in (s'), M(a') \neq 0 \land \zeta(s') \notin \{A,C\}$.

IV. EXECUTER: FAULT-TOLERANT EXECUTION CONTROL

Once a CPN-TCW S is and its corresponding BRCPN-TCW S are generated by the COMPUSER, 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-TCW S. To support backward recovery, it is necessary to keep the trace of the execution on the BRCPN-TCW S. To start the unfolding algorithm, the CPN-TCW S is marked with the Initial Marking: an initial token is added to places representing inputs of $Q$ $(\forall a \in (A \cup I_Q), M(a) = 1, \forall a \in (A \cup I_Q), M(a) = 0)$ and the state of all transitions in BRCPN-TCW S is set to initial $(\forall s' \in S', \zeta(s') \leftarrow I)$. The firing of a transition in CPN-TCW S 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-TCW S 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-TCW S 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 $(\forall a \in (s), M(x) = 0)$, if the $TP(s) \in \{c, cr\}$, the state of its corresponding $s'$ in BRCPN-TCW S 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 $(\forall x \in (s'), M(x) = M(x) + 1)$, and the state of its corresponding $s'$ in BRCPN-TCW S (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 $(pr, a_r, cr)$, $s$ 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-TCW S is guided by a unfolding algorithm of its associated BRCPN-TCW S. When a WS represented by a transition $s$ fails, the unfolding process over CPN-TCW S is halted and a backward recovery is initiated with the unfolding process over BRCPN-TCW S by marking it with its Initial Marking: a token is added to places representing inputs of $M(a') = card(x)$, 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-TCW S.

Definition 9: BRCPN-TCW S 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 $(\forall x \in (s'), M(x) = M(x) - 1)$ and tokens are added to its output places $(\forall x \in (s'), M(x) = M(x) + 1)$.

We illustrate a backward recovery in Figure 2. The marked CPN-TCW S depicted in Figure 2(a) is the state when $ws_4$ fails, the unfolding of CPN-TCW S is halted, and the initial marking on the corresponding BRCPN-TCW S is set to start its unfolding process (see Figure 2(b)), after $ws_4$ and $ws_7$ are fired and $ws_7$ is abandoned before its invocation, a new marking is produced (see Figure 2(c)), in which $ws_1$ and
\( ws_Q \) 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\( _Q \)](image)

V. EXEUCTER 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, WS 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, TCWS 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\( _Q \): \( ws_{EE} \), the first transition providing the inputs referenced in \( Q \) (\( I_Q \)) and \( ws_{EE} \), the last transition consuming the outputs (\( O_Q \)); similar dummy transitions are added to BRCPN-TCWS\( _Q \) with inverse data flow relation (\( ws_{EE} \) and \( ws_{EE} \)); 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 \( ws_{EE} \) and \( ws_{EE} \) 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 \( ws_{EE} \). 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 it to running the state of its corresponding transition in BRCPN-TCWS\( _Q \)

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-171-7 161
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 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 retrievable, 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 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 with the Initial Marking (i.e., adds tokens to places representing inputs of BRCPN-TCWS and inputs of the faulty WS), and sends control tokens to ENGINE THREADS representing successors of $ws_{EEf}$. Once the rest of ENGINE THREADS receive the message compensate, they apply the firing rules in BRCPN-TCWS (see Def. 9). The compensation process finishes when $ws_{EEf}$ 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 ($ws_{EEf}$ becomes fireable) the EXECUTION ENGINE notifies all ENGINE THREADS predecessors of $ws_{EEf}$ 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 WS 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: $Q = (I_Q, O_Q, W_Q, P_Q)$, the user query – see Def. 1

Input: CPN-TCWS$_Q = (A, S, F, E)$, a CPN allowing the execution of a TCWS– see Def. 3

Input: BRCPN-TCWS$_Q = (A', S', F', \zeta)$, a CPN representing the compensation flow of TCWS– see Def. 4

Input: $OWS_i$: Ontology of WSs

Output: $OV_Q$: List of values of $o$ | $o \in O_Q$

begin

1. **Initial phase:**

begin

1. Insert $ws_{EEf}$ in CPN-TCWS$_Q$ | $((ws_{EEf})^* = I_Q)$ \& ($((ws_{EEf})^* = )$)

2. Insert $ws_{EEf}$ in BRCPN-TCWS$_Q$ | $((ws_{EEf})^* = A'$ \& ($((ws_{EEf})^* = )$)

3. Insert $ws_{EEf}$ in CPN-TCWS$_Q$ | $((ws_{EEf})^* = )$ \& ($((ws_{EEf})^* = )$)

4. Insert $ws_{EEf}$ in BRCPN-TCWS$_Q$ | $((ws_{EEf})^* = )$ \& ($((ws_{EEf})^* = )$)

5. $∀a ∈ CE_i$ ($\zeta(a') = )$

6. $∀a ∈ (A \cup I_Q), M(a) = 1$ \& $∀a ∈ (A - I_Q), M(a) = 0$

7. $∀a ∈ S, o(s') = I$.

8. $∀a ∈ S | (ws \neq ws_{EEf})$ \& $∀a ∈ S | (ws \neq ws_{EEf})$

9. Send $Pred Succ_{EE}$ from $ws_{EEf}$ to $ws_{EEf}$

10. Send $Pred Succ_{EE}$ from $ws_{EEf}$ to $ws_{EEf}$

11. Send $Pred Succ_{EE}$ from $ws_{EEf}$ to $ws_{EEf}$

12. $∀ws ∈ S | (ws \neq ws_{EEf})$ \& $∀ws \neq ws_{EEf}$

13. Send values of $I_Q$ to $ws_{EEf}$

14. Execute Final phase;

end

2. **Final phase:**

begin

15. repeat

16. Wait Result from $(*ws_{EEf})$;

17. if message compensate is received then

18. Execute Compensation Phase \(*$ this phase is shown in

19. Algorithm 3$^*$;

20. Exit Final phase;

else

21. Set values to $OV_Q$;

22. until $∀ws \in S | (ws \neq ws_{EEf})$ \& $∀ws \neq ws_{EEf}$

23. $∀ws \neq ws_{EEf}$

24. Send $Finish$ message to $(*ws_{EEf})$;

25. Return $OV_Q$;

26. end

end

$∀ws$ 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\(*$.

---

**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, ar) 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
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 need 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.
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 to 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


