The International Symposium on Advances in Transaction Processing (ATP)

**FACETa*: Checkpointing for Transactional Composite Web Service Execution based on Petri-Nets**

Marta Rukoz¹, Yudith Cardinale², Rafael Angarita²

¹LAMSADE, Université Paris Dauphine
Université Paris Ouest Nanterre La Défense
Paris, France

²Universidad Simón Bolívar, Departamento de Computación y Tecnología de la Información,
Apartado 89000, Caracas 1080-A, Venezuela
mrukoz@lamsade.dauphine.fr, {yudith, rangarita}@ldc.usb.ve

**Abstract**

Failures during the execution of Transactional Composite Web Services (TCWSs) can be repaired by forward or backward recovery processes, according to the component WSs transactional properties. In previous works, we presented TCWS fault tolerant execution approaches relying on WSs replacement, on a compensation protocol, and on unrolling processes of Colored Petri-Nets (CPNs) to support forward and backward recovery. We represent a TCWS and its corresponding backward recovery process by CPNs. Even though these recovery processes ensure system consistency, backward recovery means that users do not get the desired answer to their queries and forward recovery could imply long waiting time for users to finally get the desired response. In this paper, we present an alternative fault tolerant approach in which, in case of failures, the unrolling process of the CPN controlling the execution of a TCWS is checkpointed and the execution flow goes on as much as it is possible. In this way, users can have partial responses as soon as they are received and can re-submit the checkpointed CPN to re-start its execution from an advanced point of execution (checkpoint). We present the checkpointing algorithm integrated to our previous work.

© 2012 Published by Elsevier Ltd. Open access under CC BY-NC-ND license.

**Keywords:** Checkpointing, Composite Web Services, Transactional Properties, Fault Tolerant Execution, Petri-Nets, Colored Petri-Nets

1. Introduction

In distributed software contexts, Web Services (WSs) that provide transactional properties are useful to ensure reliable execution and system consistent state even in presence of failures. For example, a pivot WS guarantees no effect at all if it fails during its execution and if it successfully finishes, its effects are permanent. WS composition implies the construction of more complex Transactional Composite WS (TCWS), in which several transactional WSs work together to respond a user query. Even if all component WSs of a Composite WS are transactional, the composition itself could be no transactional (e.g., a pivot WS cannot be

---

This work was supported by the Franco-Venezuelan CNRS-FONACIT project N°22782

1877-0509 © 2012 Published by Elsevier Ltd. Open access under CC BY-NC-ND license.
doi:10.1016/j.procs.2012.06.115
followed by another pivot WS, if the second one fails the first one cannot be compensated). Thus, to ensure the transactional property of a TCWS, the selection process is made according to the transactional properties of its component WSs and their execution order. TCWS becomes a key mechanism to cope with challenges of open-world software. Indeed, TCWSs have to adapt to the open, dynamically changing environment, and unpredictable conditions of distributed applications, where remote services may be affected by failures and availability of resources [1]. The control flow and the order of WSs execution is generally represented with a structure, such as workflows [2, 3], graphs [4], or Petri-Nets [5].

Failures during the execution of TCWSs can be repaired by forward or backward recovery processes, according to the component WSs transactional properties [6]. Backward recovery allows to undo the work done until the failure and go back to the initial consistent state (before the execution started); it is based on rollback and compensation techniques. Forward recovery tries to repair the failure and continues the execution; retry and substitution are some techniques used. Although these recovery processes ensure system consistency, backward recovery means that users do not get the desired answer to their queries and forward recovery could imply long waiting time for users to finally get the response because of the invested time to repair failures. 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 a previous work [5], we presented an automatic WS composition approach based on QoS and transactional user requeriments. In this approach the directory of transactional WSs is modeled by colored Petri Nets (CPN) describing the data flow relation among all WSs. Giving a query, a TCWS and its compensation process, if any, can be automatically produced. We represented both TCWS and its compensation process by acyclic CPNs. In [6, 7] we formalized, FaCETA, a fault tolerant execution control mechanism to execute such TCWS. In [6] unrolling algorithms of CPNs to control the execution and backward recovery were presented. This work was extended in [7] to consider forward recovery based on WS replacement; formal definitions for WSs substitution process, in case of failures, were presented. In [7], we also proposed an Executer framework to execute a TCWS following our proposed fault tolerant execution approach. We consider the component WSs can suffer silent or stop failures (a WS does not finish because a crash occurred during its execution).

In this paper, we present FaCETA*, an extension of FaCETA approach in which, in case of failures, the unrolling process of the CPN controlling the execution of a TCWS is checkpointed and the execution flow goes on as much as it is possible. In this way, users can have partial responses as soon as they are received and can re-submit the checkpointed CPN to re-start its execution from an advanced point of execution (checkpoint). We present the checkpointing algorithm integrated to our Executer framework. When a user submits a TCWS to be executed by our Executer, he/she can ask for the checkpointing facility. Otherwise, only backward or forward recovery are executed in case of failures.

2. FaCETA: A TCWS Executer with Backward and Forward Recovery Support

This section briefly describes FaCETA [6, 7]. Our TCWS fault tolerant execution approach is based on Colored Petri-Net (CPN) formalism, in which backward and forward recovery are executed transparently and automatically in case of failures. We define a Query $Q$ as a 4-tuple ($I_Q, O_Q, W_Q, T_Q$) in terms of functional conditions, expressed as inputs ($I_Q$ is a set of input attributes whose values are provided by the user) and outputs ($O_Q$ is a set of output attributes whose values have to be produced by the system); QoS constraints, expressed as weights over criteria ($W_Q = \{ (w_i, q_i) | w_i \in [0, 1] \text{ with } \sum_i w_i = 1 \text{ and } q_i \text{ is a } QoS \text{ criterion} \}$); and the required global transactional property ($T_Q$ is the required transactional property).

A TCWS, which answers and satisfies a Query $Q$, is automatically produced by a COMPOSER which is represented as an acyclic marked CPN$^1$, called CPN-TCWS $Q$, where WS inputs and outputs are represented by places and WSs, with their transactional properties, are represented by colored transitions –colors distinguish WS transactional properties [5]. The Initial Marking of CPN-TCWS $Q$ is dictated by the user inputs. In this way, the execution control is guided by a unrolling algorithm.

$^1$A marked CPN is a CPN having tokens in its places, where tokens represent that the values of attributes (inputs or outputs) have been provided by the user or produced by a WS execution.
As in [8] we use the following definition of individual WS transactional properties \((TP(s))\). Let \(s\) be a WS: \(s\) is **pivot** \((p)\), if once \(s\) successfully completes, its effects remain forever and cannot be semantically undone (compensated), 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 \(s\), even after \(s\) successfully completes; \(s\) is **retriable** \((r)\), if \(s\) guarantees a successful 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.

Regarding the global \(TP\) of TCWSs, we consider it is derived from the \(TP\) of its component WSs and their execution order (sequential or parallel) [8] as follows. Let \(tcs\) be a TCWS: \(tcs\) is **atomic** \((a)\), if once all its component WSs complete successfully, they cannot be semantically undone, if one component WS does not complete successfully, it is rollback and all previously successful component WSs if any, have to be compensated; \(tcs\) is **compensatable** \((c)\), if all its component WSs are compensatable; \(tcs\) is **retriable** \((r)\), if all its component WSs are retriable; the retriable property can be combined with properties \(a\) and \(c\) defining **atomic retriable** \((ar)\) and **compensatable retriable** \((cr)\) TCWSs. Note that an atomic CWS behaves as a pivot WS only if the first WS of the atomic CWS fails.

According to these transactional properties, we distinguish two possible recovery techniques:

- **Backward recovery**: it consists in restoring the state (or a semantically close state) that the system had at the beginning of the TCWS execution; i.e., all the successfully executed WSs, before the fail, must be compensated to undo their produced effects. All transactional properties \((p, a, c, pr, ar, and cr)\) allow backward recovery;

- **Forward recovery**: it consists in repairing the failure to allow the failed WS to continue its execution. Transactional properties \(pr, ar, and cr\) allow forward recovery.

In our **Executor** framework, the execution control of a TCWS is guided by a unrolling algorithm of its corresponding CPN-TCWS\(_Q\). A WS is executed if all its inputs have been provided or produced, i.e., each input place has as many tokens as WSs produce them or one token if the user provides them. Once a WS is executed, its input places are unmarked and its output places (if any) are marked.

In case a component WS fails, the global \(TP\) of CPN-TCWS\(_Q\) ensures that if its \(TP\) does not allow forward recovery, then all previous executed WSs could be compensated by a backward recovery process. For modeling TCWS backward recovery, we have defined a backward recovery CPN, called BRCPN-TCWS\(_Q\), associated to a CPN-TCWS\(_Q\) [6]. The component WSs of BRCPN-TCWS\(_Q\) are the compensation WSs, \(s'\), corresponding to all \(c\) and \(cr\) WSs in CPN-TCWS\(_Q\). The BRCPN-TCWS\(_Q\) represents the compensation flow, which is the inverse of the execution order flow. The compensation control of a TCWS is also guided by a unrolling algorithm. In [5, 6] we propose techniques to automatically generate both CPNs, CPN-TCWS\(_Q\) and BRCPN-TCWS\(_Q\).

If a failure occurs in an advanced execution point, a backward recovery may incur in high wasted resources. On the other hand, it is hard to provide a **retriable** TCWS, in which all its components are **retriable** to guaranty forward recovery. We proposed an approach based on WS substitution in order to try forward recovery [7]. When a WS fails, if it is not **retriable**, instead of backward recovery, a substitute WS is searched to be executed on behalf of the faulty WS. The protocol followed by our **Executor** in case of failure of a WS \(s\) depends on the \(TP(s)\) as follows:

- if \(TP(s)\) is **retriable** \((pr, ar, cr)\), \(s\) is re-invoked until it successfully finishes (forward recovery);
- otherwise, another Transactional Equivalent WS, \(s'\), is selected to replace \(s\) and the unrolling algorithm goes on (trying a forward recovery);
- if there not exist 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.

In case of failures, the execution control is still managed by our framework and the recovery techniques are applied automatically according the \(TP\) of the component WSs, without intervention of users.

3. **FaCETA***: Extending FaCETA with Checkpointing/Restart Support

This section explains how the fault tolerant execution control was extended in FaCETA* to incorporate the Checkpointing facility. The execution of a TCWS in FaCETA* is managed by an **Execution Engine** and
a collection of software components called Engine Threads, organized in a three-level architecture. In the first level, the Execution Engine receives the TCWS and the compensation order (both represented by CPNs) and the indication of the user regarding the checkpointing facility (ckp=true or ckp=false). 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 the respective WS, and forward their results to their peers to continue the execution flow. In case of failure, all of them participate in the backward, forward, or checkpointing recovery process. Actual WSs are in the third layer.

**Initial phase**: Whenever an Execution Engine receives a CPN-TCWS, its corresponding BRCPN-TCWS, and the user desire about checkpointing facility, it starts an Engine Thread responsible for each transition in CPN-TCWS, indicating to each one its predecessor and successor transitions according to the CPN-TCWS structure: this step means that Execution Engine sends the part of CPN-TCWS and BRCPN-TCWS 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.

**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, if TP(WS) is retrievable, the WS is re-invoked until it successfully finishes; otherwise the Engine Thread executes the Replacing phase. If replacing is not possible and checkpointing facility is enabled, the Checkpointing phase has to be executed, in this case the Engine Thread sends faulty values to its successors to initiate the checkpointing process. If replacing is not possible and checkpointing facility is not enabled, the only option left is to perform the Compensation phase. When an Engine Thread receives at least one faulty value among its needed inputs, the Checkpointing phase is executed.

**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 Finish message, recalculates the Quality of TCWS in case some WSs were replaced, and returns the values of attributes in O\(_Q\) to user. When an Engine Thread receives the Finish message, it exits. 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. If Execution Engine receives a faulty value in at least one of the O\(_Q\) attributes, it executes the Checkpointing phase.

**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. This phase 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. Once the rest of Engine Threads receive the message compensate, they apply the firing rules in BRCPN-TCWS 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 and saves its state (snapshot). 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 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-TCWS that could no be executed (PARTIAL-CPN-TCWS). 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-TCWS, sends the I\(_Q\) to the corresponding
ENGINE THREAD and the unrolling algorithm over PARTIAL-CPN-TCWS \( Q \) 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.

![Algorithm 1: Checkpointing Phase](image)

**Algorithm 2: Execution Engine Restart Phase**

```
Input: GS: a reference to a Global Snapshot
Begin
    \begin{itemize}
        \item \textbf{Execution Engine:}
            \begin{itemize}
                \item Load \( Q \), PARTIAL-CPN-TCWS\( Q \), BRCPN-TCWS\( Q \),
                OWS, O\( V \), Inputs Needed from GS;
                \end{itemize}
            \end{itemize}
        \item \textbf{Engine Threads:}
            \begin{itemize}
                \item \textbf{Invoke Final phase;}
            \end{itemize}
    \end{itemize}
```

![Fig. 1: Checkpointing & Restart Phases](image)

4. 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 [9, 10, 11] and works that perform checkpointing in an automatic fashion[12, 13].

The problem addressed in [9] 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 [10] 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 [11] 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. 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 some of these works, 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 [12] 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 \( \text{PET} < \text{MTBF} \) is called, then it is known that it will complete its execution within its MTBF and there is no need for checkpointing. In [13] 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, these works proceed with checkpoints, without user intervention. In contrast, in our strategy, checkpoints are taken only in case of failures, it means that we do not increase the overhead while the execution is free of failures.

5. Conclusions and Future Work

In this paper we have presented a checkpointing mechanism for FaCETa*, which is a framework for ensuring correct and fault tolerant execution order of TCWSs. To support failures, our previous framework FaCETa implements forward recovery by replacing the faulty WS and backward recovery based on a unrolling process over a CPN representing the compensation flow. In this paper, we have extended the fault tolerant execution control in FaCETa* with a checkpointing mechanism, allowing to continue the normal execution of the part of the TCWS not affected by that failure, and then, after the maximum possible number of WSs has been executed, the state of the CPN-TCWS is saved. This mechanism provides the possibility to restart the execution later, when the faulty WSs have been repaired, reducing the amount of lost work and the cost of execution by avoiding the execution of compensation and the re-execution of previously successfully executed WSs.

We are currently working on implementing the checkpointing mechanism for FaCETa* in a distributed shared memory platform in order to test the performance of the framework in centralized and decentralized platforms.

References


