Deriving Robust Distributed Business Processes with Automated Transformations of Fallible Component Processes

Lei Wang,
Luís Ferreira Pires
and Marten J. van Sinderen

CTIT, University of Twente,
the Netherlands
Emails: {l.wang-1, l.ferreirapires, m.j.vansinderen}@utwente.nl

Andreas Wombacher

Achmea, the Netherlands
Postbus 866
3700 AW Zeist
Email: andreas.wombacher@achmea.nl

Chi-Hung Chi

CSIRO, Australia
3-4 Castray Esplanade,
Hobart, Tasmania, 7000
Email: chihungchi@gmail.com

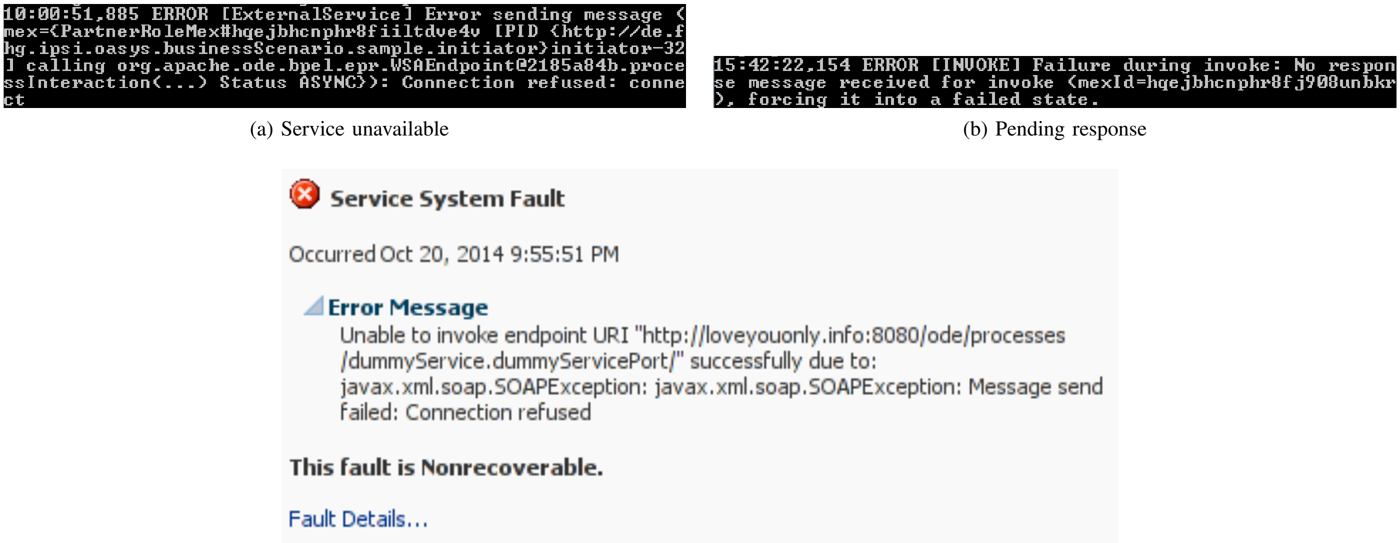
***Abstract*—Due to the possibility of system crashes and network failures, the design of robust interactions for collaborative business processes is a challenge. If a process changes state, it sends messages to other relevant processes to inform them about this change. However, server crashes and network failures may result in a loss of messages. In this case, the state change is performed by only one process, resulting in global state/behavior inconsistencies and possibly deadlocks. Our idea to solve this problem is to (automatically) transform the original processes into their robust counterparts. The robust initiator process re-tries failed interactions by resending the request message. The robust responder then replies with the possibly lost response message, without processing the request again. We illustrate our solution using a subset of Web Services Business Process Execution Language (WS-BPEL). A WS-BPEL process is modeled using a so called Nested Word Automata (NWA), to which we apply our transformation solution and on which we perform correctness proof. We have also analyzed the performance of our prototype implementation. In our previous work, we assumed that a certain pre-defined interaction follows the failed interaction. In this work, we lift this limitation by allowing an arbitrary behavior to follow the failed interaction, making our solution more generally applicable. This paper is an extension of our previous paper [1]. The additional contents of this paper is the following.**

***Keywords*–robust; collaborative processes; control flow; WS-BPEL; interactions; system crash; network failure; automata.**

I. Introduction

The electronic collaboration of business organizations has grown significantly in the last decade. Often data interchange is based on processes run by different parties exchanging messages to synchronize their states. If a process changes state, it sends messages to other relevant processes to inform them about this change. However, server crashes and network failures may result in a loss of messages. In this case, the state change is performed by one process, resulting in global state/behavior inconsistencies and possible deadlocks. In general, a state inconsistency is not recovered by the process engine that executes the process. This can be seen from a screen dump of errors after a system crash of the process engines such as Apache ODE and Oracle BPM, as is shown in Figure 1. Figure 1a shows that in the Apache ODE process engine the initiator sends the message to an unavailable server. Figure 1b shows that in the Apache ODE process engine the initiator sends a request message, and the responder crashes without sending the response message. Figure 1c shows the Oracle process engine 12c, which sends message to an unavailable server (see Figure 1).

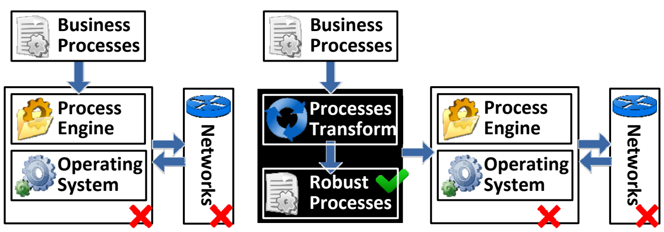
Figure 2a shows that normally, a business process is deployed to a process engine, which runs on the infrastructure services (OS, database, networks, etc.), where system crashes and network failures may happen. Our solution to recover from failures is to transform business processes into their robust counterparts, as shown in Figure 2b. The robust process is deployed on the unmodified infrastructure services and is recoverable from some interaction failures caused by system crashes and network failures. Our solution has the following properties: (1) the application protocols are not modified. We do not modify the message format nor message sequence, e.g., by adding message fields that are irrelevant for the application logic or adding acknowledge messages to the original message sequence. The service autonomy is kept in that if one party transforms the process according to our approach and the other party does not, they can still interact with each other, although without being able to recover from system crashes and network failures. (2) the process transformation is transparent for process designers. (3) the solution conforms to the process language specification. We use the standard process language constructs without extending the language to avoid making the robust process depend on a specific engine. In this paper, we illustrate our solution using WS-BPEL. WS-BPEL is a language for specifying business process behavior



(a) Service unavailable (b) Pending response

(c) Service unavailable

Figure 1. Interaction failures



(a) Interaction failures (b) Robust process transformation

Figure 2. Our idea to cope with failures.

based on web services. As an OASIS standard, it is widely used by enterprises. However, other process languages may be applicable as long as they support similar workflow patterns.

This paper is an extension of our previous paper [1]. The additional contents of this paper is the following.

1) We analyze the possible synchronization failures in more detail.
2) More Nested Word Automata models of WS-BPEL activities are described inthis paper,suchas, “invoke”, “receive” and “reply” activities.
3) Omitted information on the correctness criteria and correctness evaluation process is presented in this paper.
4) Process transformation complexity of our soultion is analyzed.

This paper is based on our previous work [2][3][4][5], where we assumed that a certain pre-defined interaction follows the failed interaction, i.e., the only sequence control is assumed that the further interaction is sequentially following the failed interaction. In this paper, we lift this limitation by allowing an arbitrary behavior to follow the failed interaction, making our solution more generally applicable. We support conditional control flow and loops and their arbitrary combination as possible further interaction after interaction failure. The structure of the paper is the following: Section II analyzes possible interaction failures. Section III proposed our process transformation-based solution. Section IV validates our solution. Section V discusses related work and Section VI concludes our paper.

II. Interaction Failure Analysis

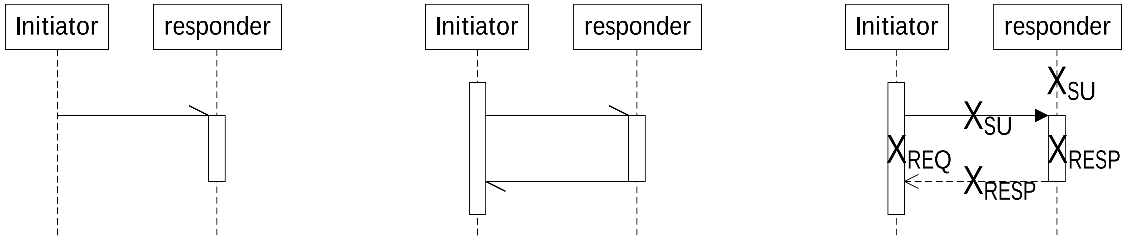
This section analyzes possible interaction failures of collaborative processes caused by system crashes and network failures.

A. *Process Interaction Patterns*

Process interaction failures are specific to interaction patterns. In [6], 13 interaction patterns are identified. In this paper, we focus on the *send*, *receive* and *send-receive* patterns. This limitation is not severe because more complex patterns can be composed using these basic interaction patterns. Figure 3a shows an initiator that sends a one-way message to a responder. The initiator behavior corresponds to the *send* pattern, while the responder behavior corresponds to the *receive* pattern. In pattern *send-receive* in Figure 3b the initiator combines one *send* and one *receive* pattern. We call this pattern asynchronous interaction in the sequel of the paper. In Figure 3c, the initiator starts a synchronous interaction by sending a request and getting a response, which characterize the *send-receive* pattern.

B. *Process Interaction Failures*

Table I shows a failure classification scheme [7]. Crash failure, omission failure and timing failure are considered in this work. Crash failure is referred as *system crashes* in this paper. Omission failure and timing failure occur when the network fails to deliver the messages (in a specified time interval) and are referred to as *network failures* in this paper. However, response failures due to a flaw in the process design and arbitrary failure, also referred to as Byzantine failure, which is more of a security issue, are out of the scope of this work.



(a) *send* and *receive* (b) *send-receive* case I (c) *send-receive* case II

Figure 3. Process interaction patterns.

TABLE I. Failure Scheme.

Type of failure	**Description**
Crash failure	Server halts, bus is working correctly until it halts.
Omission failure	Server fails to respond to incoming requests.
Receive omission	A server fails to receive incoming messages.
Send omission	A server fails to send messages.
Timing failure	Server's response lies outside the specified time interval.
Response failure	Server's response is incorrect.
Value failure	The value of the response is wrong.
State transition failure	The server deviates from the correct flow of control.
Arbitrary failure	Server may produce arbitrary responses at arbitrary times.

Interaction failures caused by system crashes and network failures are *pending request failure*, *pending response failure* and *service unavailable* [3]. As all failures possible in the interaction patterns of Figure 3a and Figure 3b are covered by Figure 3c, we look only into the interaction failures of the interaction pattern in Figure 3c. *Service unavailable* (marked as X_{SU}) is caused by a responder system crash or a network failure of the request message delivery. At process level, the initiator is aware of the failure through a catchable exception of the process implementation language. *Pending request failure* (marked as X_{REQ}) is caused by initiator system crashes after sending a request message. The initiator is informed of the failure after restart, e.g., through catchable exceptions. However, the responder is not aware of the failure, so that it replies with the response message and continues execution. *Pending response failure* (marked as X_{RESP}) is caused by a responder system crash or a network failure of the response message delivery. In both cases, the responder replies with the response message (after a restart if the responder system crashes) and continues execution. However, the connection gets lost and the initiator cannot receive the response message. The initiator is aware of this failure after a timeout.

C. Assumptions concerning the failure behavior

Due to the heterogeneous infrastructure, e.g., different process engine implementations or network environments, we have to make the following assumptions concerning the failure behavior of the infrastructure:

1) Persistent execution state. The state of a business process (e.g., values of process variables) are kept persistent and survive system crashes.

2) Atomic activity execution (e.g., invoke, receive, reply). A system crash means that the execution is stopped only after the previous activity is finished and the next activity has not started. A restart means that execution resumes from the previous stopped activity. These assumptions correspond with the default behavior of the most popular process engines, such as Apache ODE or Oracle BPEL Process Manager (released as a component of Oracle SOA Suite). In Apache ODE's term, this is named as persistent processes in their default configuration. Otherwise, this configuration can be modified to "in-memory" at deployment time [8]. For Oracle BPEL Process Manager, this is named as "durable" processes, otherwise is named as "transient" processes. By default all the WS-BPEL processes are durable processes and their instances are stored in the so called dehydration tables, which survives system crashes [9].

3) Network Failures interrupt the established network connections and the messages that are in transit get lost.

III. Process Transformation Based Solution

Figure 4 shows our approach based on process transformation. The transformation is implemented at an abstract level by using NWA (Nested Word Automata) [10]. NWA is an extension of the classical automata, which maintains the nested syntax of WS-BPEL process. Furthermore, its automata based definition, which facilitates the description of reliable interaction principles, correctness proof and complexity analysis formalized. Figure 4 shows that the process transformation is divided into three steps: *Transform1* transforms a business process into a NWA, *Transform2* transforms the NWA to the NWA' of a robust process, and finally, *transform3* generates the robust process from the NWA'.

A. Business Processes

We choose WS-BPEL [11] as process specification language in our work. However, other process languages may be applicable as long as they support similar workflow patterns [12]. A WS-BPEL process is a container where relationships to external partners, process data and handlers for various purposes and, most importantly, the activities to be executed are declared. As an OASIS standard, it is widely used by enterprises. We use Nested Word Automata (NWA) [10] to

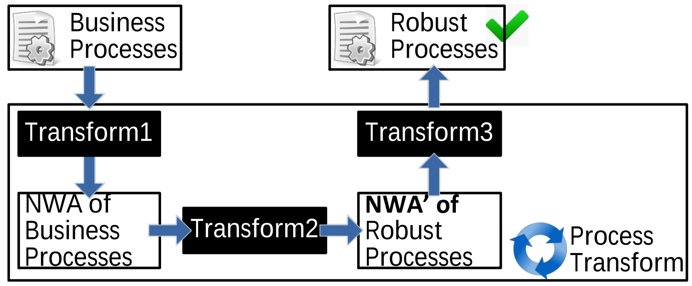


Figure 4. Process transformation

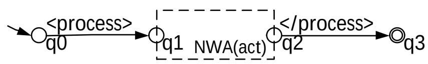


Figure 5. NWA model of a process.

describe the underlying semantics of WS-BPEL and use them as a basis for our formal evaluation. We choose NWA because we need to model the nested structure of WS-BPEL syntax. While traditional finite state automata can be used for describing all possible states of messages, and their sending and receiving sequences, they lack the capability of describing nested structures of activities.

An NWA is an automaton that has hierarchical nesting structures. Formally, an NWA A over an alphabet Σ is a structure $(Q, q_0, Q_f, P, p_0, P_f, \delta_c, \delta_i, \delta_r)$ consisting of

- a finite set of (linear) states Q,
- an initial (linear) state $q_0 \in Q$,
- a set of (linear) final states $Q_f \subseteq Q$,
- a finite set of hierarchical states P,
- an initial hierarchical state $p_0 \in P$,
- a set of hierarchical final states $P_f \subseteq P$,
- a call-transition function $\delta_c : Q \times \Sigma \mapsto Q \times P$,
- an internal-transition function $\delta_i : Q \times \Sigma \mapsto Q$, and
- a return-transition function δ_r: $Q \times P \times \Sigma \mapsto Q$.

The definition of Q, q_0, Q_f, δ_i corresponds to the definition of a finite state automata over an alphabet Σ [13]. The alphabet Σ represents all possible process behaviors, e.g., *<process>* $\in \Sigma$ represents the starting of a business process, $?m_i \in \Sigma$ represents receiving a message while $!m_j \in \Sigma$ represents sending a message. An internal transition $\delta_i(q_i, !m_i) = q_j$ represents that the process replies a message $!m_i$ at the state q_i and then enters the state q_j. The hierarchical states P, p_0, P_f are used to describe the nesting structure of an NWA. A *call transition* δ_c enters the nested automaton while a *return transition* δ_r leaves the nested automaton. The other transitions are also found in traditional automata, and called internal transitions. A nested structure is graphically represented as dashed box. The NWA model of a WS-BPEL process is shown in Figure 5. A call transition $\delta_c(q_0, \mathit{<process>}) = (q_1, p_a)$ starts from the initial state and a return transition $\delta_r(q_2, p_a, \mathit{</process>}) = q_3$ leads to the accepted state. The NWA model of an activity *NWA(act)* is nested within the NWA of the process. This is described by the hierarchical state p_a.

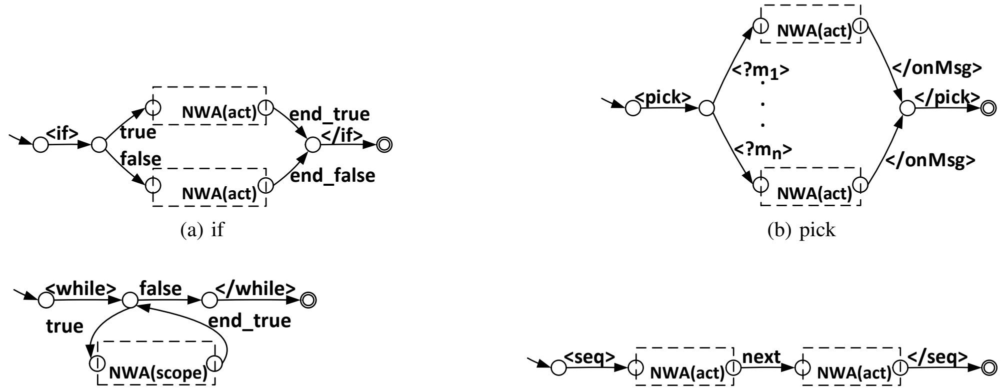
WS-BPEL activities are divided into two categories, namely *basic* and *structured* activities. The currently supported *structured* activities are *if*, *pick*, *while* and *sequence*, as shown in Figure 6. The conditional branch (*if*), repeat (*while*), and sequential (*sequence*) activities are modeled as shown in Figure 6a, 6c, 6d, respectively. Figure 6b shows the model of a *pick*, which is another case of conditional control flow that depends on the type of the incoming message. The *flow* (concurrent execution) or the other forms of loops (*RepeatUntil, ForEach*) are not considered now and will be considered in future work. Each structured activity model has exactly one call transition and one return transition to *enter* and *leave* its nested structure(s), which is represented as a dash box.

The models of *basic* activities are visualized in Figure 7. These models can be *nested* (to replace the dash box in Figure 6) in the models of structured activities. However, the model of *exit* is an exception. Once the transition *exit* fires, the NWA goes to a terminated state. From this state, the NWA is *deactivated* (no transition will leave this state), which breaks the well-nested structure.

B. Transformation Method

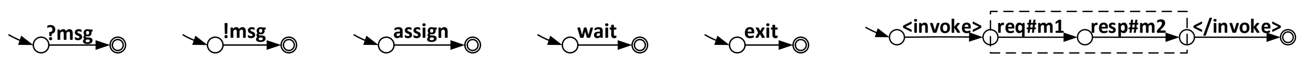
An operation that can be safely repeated is called idempotent [7]. Idempotent operations can be recovered by resending the request message. However, a request resent to non-idempotent operations (such as bank transfer operations) triggers potentially incorrect executions. Our solution for non-idempotent operation is that when a failure happens, a resent message is replied with a copy of the previous processing result.

In the example of Figure 8a, the WS-BPEL snippet receives a message $m1$, performs some (non-idempotent) processing, then replies with a message $m2$. The next incoming messages could be $m3$ or $m4$. If the initiator sends request $m3$ or $m4$, this implies that the initiator has successfully received the response message $m2$. If due to an interaction failure, for example, the initiator crashes and fails to receive the response message $m2$, the initiator can recover by resending request message $m1$. Thus, the responder can be aware of whether failures have happened by inspecting the incoming message. If the incoming message is a resent message, this implies that a failure happened in the previous interaction. In this section, we will first present our formal method of calculating the set of failure-resent messages at a state. We then transform the responder process to use a copy of the previous result as response. Figure 8b shows that in order to make a copy of the response message, we use an *assign* activity to keep the result value in a process variable $\$copy$. In the *pick* activity, we add an *onMessage* branch to accept the resent message $m1$ and use the variable $\$copy$ as the response. However, a resent message could be sent multiple times before the response is ultimately received. We nest the *pick* activity in a *while* to cope



(a) if (b) pick (c) while (d) sequence

Figure 6. NWA model of WS-BPEL structured activities.



(a) receive (b) reply (c) assign (d) wait (e) exit (f) invoke

Figure 7. NWA model of WS-BPEL basic activities.

```
<sequence>
   <receive variable="m1" ... />
   <!-- some process -->
   <reply variable="m2" ... />
   <pick>
      <onMessage variable="m3".../>
      <onMessage variable="m4".../>
   </pick>
</sequence>
```

(a) example WS-BPEL snippet

```
<sequence>
   <receive variable="m1" ... />
   <!-- some process -->
   <reply variable="m2" ... />
   <assign>$copy := $m2</assign>
   <pick>
      <onMessage variable="m3".../>
      <onMessage variable="m4".../>
      <onMessage variable="m1"…>
         <reply variable="copy" … />
      </onMessage>
   </pick>
</sequence>
```

(b) transformation, step I

Figure 8. WS-BPEL example of our solution.

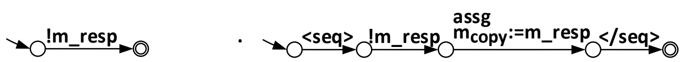
with the duplicate resent message. Our process transformation algorithm is presented as follows.

1) Responder Transformation Algorithm: For a WS-BPEL process, given its *NWA* model $(Q, q_0, Q_f, P, p_0.P_f, \delta_c, \delta_i, \delta_r)$ over the alphabet Σ, we assume that the alphabet that represents the response messages is Σ_{resp} and the alphabet that represents the request messages is Σ_{req}, thus $\Sigma_{req} \subseteq \Sigma$ and $\Sigma_{resp} \subseteq \Sigma$. The transformation algorithm is as Figure 9.

The algorithm iterates through all combinations of a state q, a request message $?m_{req}$ and a response message $!m_{resp}$. In line 2, we check if the message pair $(?m_{req}, !m_{resp})$ corresponds to the request and response for a synchronous operation and at state q, the response message $!m_{resp}$ is sent, represented by a transition $\delta_i(q, !m_{resp})$. This is the failure point that the response message may be lost due to interaction failures and where our transformation method applies. As defined in line 3, we first make a copy of the response message, as shown in Figure 10. The NWA model of *reply* activity in Figure 10a is replaced by an NWA model of a *sequence* activity in Figure 10b, in which a *reply* activity model and an *assign* activity model are nested. The *assign* activity model represents

1: **for all** $q \in Q$, $?m_{req} \in \Sigma_{req}$ **and** $!m_{resp} \in \Sigma_{resp}$ **do**
2:   **if** (m_{req}, m_{resp}) is a synchronous message pair **and** $\delta_i(q, !m_{resp})$ is defined in *NWA* **then**
3:     $save\_reply(q, !m_{resp})$
4:     N $\leftarrow$ $next\_receive(q, !m_{resp})$
5:     **for all** $?m_{next} \in N$ **and** $q_{next} \in Q$ **do**
6:       **if** $\delta_i(q_{next}, ?m_{next})$ is defined in *NWA* **then**
7:         $transform\_receive(q_{next}, ?m_{next})$
8:       **else if** $\delta_c(q_{next}, ?m_{next})$ is defined in *NWA* **then**
9:         $transform\_pick(q_{next}, ?m_{next})$
10:       **end if**
11:     **end for**
12:   **end if**
13: **end for**

Figure 9. Responder process transformation algorithm



(a) *reply* activity (b) Transformed *reply activity*

Figure 10. Responder process transformation, *reply* activity.

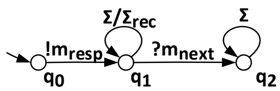


Figure 11. The automaton $A(!m_i, ?m_{next})$.

the copy of the reply message into the variable m_{copy}. In order to process the possible resent request message $?m_{req}$ due to the lost of the message $!m_{resp}$ sent at state q, we calculate the set of all possible next incoming messages, which is defined as $next\_receive(q, !m_{resp})$ in line 4. We construct an automaton $A(!m_{resp}, ?m_{next})$ as in Figure 11 to describe that a process replies with a message $!m_{resp}$ and waits for some possible next incoming message $?m_{next}$. $\delta(q_0, !m_{resp}) = q_1$ models the reply of the response message $!m_{resp}$. $\delta(q_1, \Sigma/\Sigma_{req}) = q_1$ represents some process execution in which no messages are received. $\delta(q_1, ?m_{next}) = q_2$ represents that the process receives an incoming message $?m_{next}$. $\delta(q_2, \Sigma) = q_2$ models any process execution. For the process *NWA* model, at some state q, a reply of a message $!m_{resp}$ is represented by an internal transition $\delta_i(q, m_{resp})$. We change the initial state of the process *NWA* model to from q_0 to q, and call this automaton *NWA*(q). Starting at q, after replying the message $!m_{resp}$, if one possible next incoming message is $?m_{next}$, then *NWA*$(q) \cap A(!m_{resp}, ?m_{next}) \neq \emptyset$, i.e., the process modeled by *NWA* has the behavior described by $A(!m_{resp}, ?m_{next})$.

The intersection operation $\cap$ between an *NWA* and an finite state automaton is defined to check whether the business process modeled by the *NWA* has the message sending and receiving behavior modeled by the automaton. The intersection operation is based on finite state automata. We "flatten" an *NWA* to a finite state automaton by skipping hierarchical information, described as follows. Given a NWA $(Q, q_0, Q_f, P, p_0, P_f, \delta_c, \delta_i, \delta_r)$ over the alphabet Σ, the "flattened" automaton is $A(Q, q_0, Q_f, \Sigma, \delta)$, where Q, q_0, Q_f and Σ are the same as the NWA, the transition function δ is defined as

1) $\delta(q_{i1}, a) = q_{i2}$, if the NWA has an internal transition $\delta_i(q_{i1}, a) = q_{i2}$.
2) $\delta(q_{c1}, a) = q_{c2}$, if the NWA has a call transition $\delta_c(q_{c1}, a) = (p, q_{c2})$.
3) $\delta(q_{r1}, a) = q_{r2}$, if the NWA has a return transition $\delta_r(q_{r1}, p, a) = q_{r2}$.

Both call transitions and return transition are treated as flat transitions that the hierarchical state p is not considered. The intersection operation can be done between two finite state automata, as defined in [13].

We define the set of all possible next incoming messages as $next\_receive(q, !m_{resp}) = \{?m_{next} | ?m_{next} \in \Sigma_{req} \wedge$ *NWA*$(q) \cap A(!m_{resp}, ?m_{next}) \neq \emptyset\}$.

For all $?m_{next} \in next\_receive(q, !m_{resp})$ and $q_{next} \in Q$, if at the state q_{next} the next incoming message $?m_{next}$ is received, two cases of transition may be defined in *NWA*: in a model of a *receive* activity as an internal transition $\delta_i(q_{next}, ?m_{next})$ or in the model of a *pick* activity as a call transition $\delta_c(q_{next}, ?m_{next})$. For the first case (line 6), as shown in Figure 12a, the procedure $transform\_receive(q_{next}, ?m_{next})$ is introduced as follows. We replace the transition with a *pick* activity with two branches, as shown in Figure 12b. One *onMessage* branch models the receive of the resent message $?m_{req}$ and the reply of the result message m_{copy}. The other *onMessage* branch models the receive of the message $?m_{next}$, and after that we set the flag $success$ to $true$ to indicate that the previous interaction is finished successfully.

However, a possible loss of the response message m_{copy} triggers multiple resending of the request m_{req}. Therefore, the *pick* activity is defined in a *while* iteration so that multiple requests $?m_{req}$ can be accepted. Figure 12c shows that the *while* iteration ends when the flag $success$ is set to *true*.

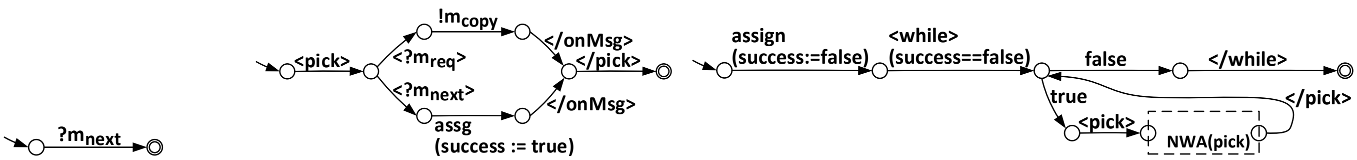
For the second case (line 8), as shown in Figure 13a, the message $?m_{next}$ is one of the messages in $m_1, ..., m_n$. Figure 13b shows that we then add a call transition $<?m_{req}>$ to model that the process accepts the resent message, and an internal transition $!m_{copy}$ to represent the reply using a copy of the previously cached result m_{copy}. In the other branches, the nested NWA(act) is replaced by the model of a *sequence* activity, in which we model the assignment of the flag variable *success* to *true*, followed by the original NWA(act). Similarly, in order to cope with a possible loss of the response message m_{copy}, the *pick* activity model is nested in a *while* iteration to handle multiple resent messages, as shown in Figure 12c. We do not directly prove the correntness of this algorithm, however, the correctness of the transformed processes by this algorithm have been proved in Section IV.

After the transformation, at some states the responder can receive more messages than the original process, because the resent message can be accepted and be replied. However, the request is not processed again. In this sense, we do not give malicious initiators any chance of jeopardizing the process by changing the sequence of requests or sending the same request multiple times.

C. Initiator Transformation

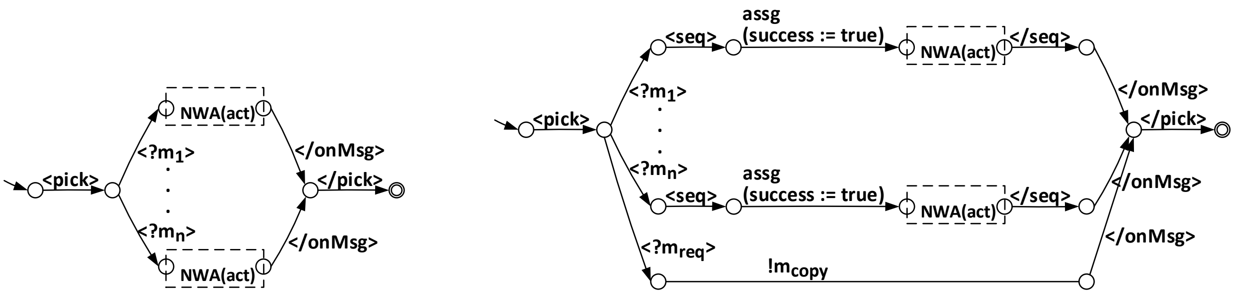
The initiator starts the interaction by executing the *invoke* activity. An *invoke* activity, which is shown as Figure 14a, is replaced by the model of a scope activity, which consists of an NWA of a fault handler and a *sequence* activity model. Nested in the *sequence* activity model there is the model of the original *invoke* activity, followed by an assignment of the $false$ value to a process variable w. The whole NWA of the *scope* activity is nested in a *while* activity model.

The whole model represents the process behavior of invocation. If failure happens and gets caught by a fault handler, the process waits for a specific time period and finishes the *scope* activity, then the outside *while* makes the *invoke* activity be executed again, until it finishes successfully and the variable w is assigned to $false$. The possible interaction failure is modeled as the transition *fail*. This is a reasonable failure model since that if a failure happens the control flow is deviated from the normal flow to the end of the scope to which a fault handler is attached, rather than leading the process to an exceptional end.



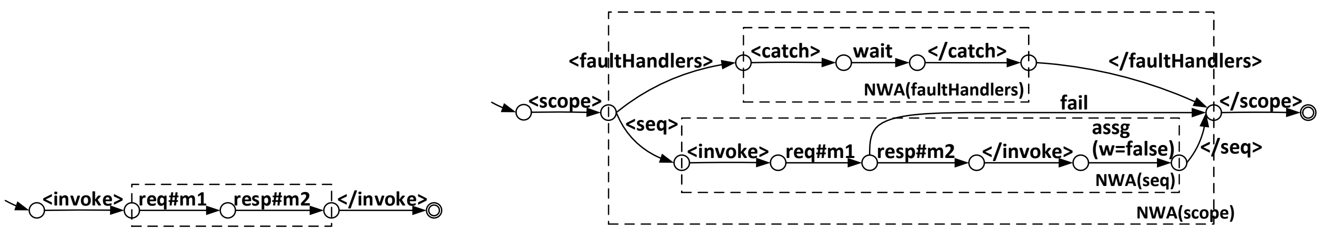
(a) Responder, *receive* activity (b) Transformed responder, part I (c) Transformed responder, part II

Figure 12. Responder process transformation, *receive* activity.

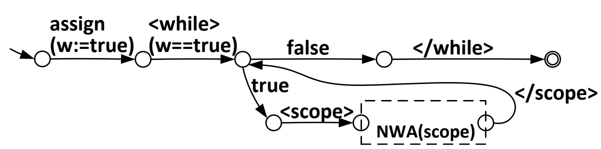


(a) Responder, *pick* activity (b) Transformed responder, part I

Figure 13. Responder process transformation, *pick* activity.



(a) Initiator (b) Transformed initiator, part I



(c) Transformed initiator, part II

Figure 14. Initiator process transformation.

1) Recoverable Assumption: Assume that $(?m_{req}, !m_{resp})$ is a pair of synchronous request and response messages, the process receives request message $?m_{req}$, then at state q, the process sends the response message $!m_{resp}$. However, if $?m_{req} \in next\_receive(q, m_{resp})$, then one of the next possible messages is still $?m_{req}$, in this case, the responder cannot distinguish a resent message due to a failure from a normal request message. Thus, we have to require that in the process design the condition $?m_{req} \notin next\_receive(q, !m_{resp})$ can be met. However, by following a few process design principles during the design of the original process, this condition can be met. An example is a split of message $?m_{req}$ into two different messages, $?m_{req1}$ and $?m_{req2}$ (for example, one message is used to send request, the other asks for results). The initiator sends the two messages back to back. If a responder receives $?m_{req1}$, then it waits for $?m_{req2}$, rather than waiting for $?m_{req1}$ again.

IV. EVALUATION

This section presents the correctness validation of our solution. We also evaluate the performance overhead of our prototype under different workloads. Finally, we analyze the complexity of the transformed process by comparing it with the original process.

A. Correctness Validation

The correctness proof shows that the solution introduced in Section III provides a robust process. The core of the proof is to define the correctness criteria for asynchronous and synchronous interactions and represent them such that they can be automatically evaluated.

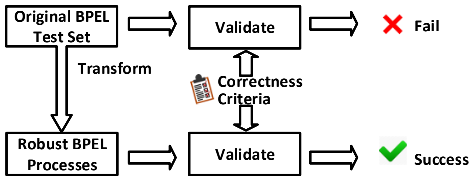


Figure 15. The setup of the correctness proof.

Figure 15 shows the setup of our correctness proof. We take the test set of 726 example WS-BPEL processes, which implement all possible Internet Open Trading Protocol (IOTP) interactions [14], and we transform them into the *NWA* model of the corresponding robust business process. The proof is finished by checking that the *NWA* process model is a subset of the correctness criteria, which are modeled as automata. Given an automaton $A(Q, \Sigma, \delta, q_0, F)$, each state in Q represents a state of the messages sending and receiving status. Set Σ models of all possible process behaviors, e.g., sending and receiving messages. δ is transition function: $Q \times \Sigma \to Q$ that models the state transition triggered by process execution, e.g., for states $q_i, q_j \in Q$ and $!m \in \Sigma$, $\delta(q_i, !m) \to q_j$ represents that at state q_i, the process replies with message $!m$ and then enters state q_j. The correctness criteria automata models the set of correct message sending and receiving sequences. We present the correctness criteria as follows.

1) Initiator Side Correctness Criteria: There are two criteria due to the interaction patterns: the criteria for a single message sending and the criteria for synchronous request and response message pair. In this paper, we discuss only the latter due to the page limitations. The automaton is visualized as Figure 16a. A correct interaction is regarded as, a request message $?m1$ can be sent at state q_0 and can be resent multiple times at state q_1 until a response message $!m2$ is received at state q_2. The correctness criteria automaton is formally defined as $\{Q, q_0, Q_f, \Sigma, \delta\}$, where

- $\Sigma = \{?m1, !m2\}$. $?m1$ and $!m2$ are the synchronous request and response messages.
- $Q = \{q_0, q_1, q_2\}$ and $Q_f = \{q_2\}$.
- For the transition $\delta(q_0, ?m1) = q_1$, the state changes from q_0 to q_1 on the input $?m1$ and stays on state q_1 on the input $?m1$ (transition $\delta(q_1, ?m1) = q_1$). This represents that a request could be sent multiple times due to failures. Transition $\delta(q_1, !m2) = q_2$ represents that if the response is received the initiator is in an accepted state and any transition is accepted (transition $\delta(q_2, \Sigma) = q_2$).

2) Responder Side Correctness Criteria: The property we want to validate is that any resent message can be accepted by the transformed process and replied. Given a process, assume that the synchronous request and response messages are m_j and m_i. If the transformed robust process model is NWA', for state q_i and transition $!m_i$, we have $?m_j \in next\_receive(q_i, !m_i)$. The idea behind it is that after the reply message $!m_i$ is sent, the robust process can accept possible resent messages due to failures. In this case, the response should be sent without reprocessing. The criteria are shown as Figure 16b.

The process control flows can be designed in arbitrary ways, and since we cannot exhaust all possibilities, we use a WS-BPEL test set that implements all possible IOTP interactions, which is a total of 726 BPEL processes. After the transformation of the test processes into automata, we apply the subset check to evaluate the correctness of the WS-BPEL test processes, i.e., we prove that for all processes and their *NWA* model and criteria automaton A, $NWA \in A$, i.e., all messages sending and receiving sequences are correct.

We take the process in Figure 8 to illustrate the correctness validation. An original WS-BPEL snippet shown in Figure 8a is transformed into the robust counterpart, and their automata models are shown as Figures 17a and 17b, respectively. At state q_2, where the message $!m_2$ is to be replied, the set of the possible next incoming messages of the responder is $next\_receive(q_2, !m2) = \{?m1, ?m3, ?m4\}$. The criteria for the synchronous request $?m1$ and the response $!m2$ is shown as Figure 18. First, we do a subset check to prove that the original is not a subset of the criteria. Actually, we can see that the message sequence $(.., ?m1, !m2, .., ?m1, !m2, ..?m3, ..)$ can be accepted by the criteria automaton. However, this sequence cannot be accepted by the model of the original process, since there is no transition defined for the second $?m1$. Second, we do a subset check to prove the transformed automata model is a subset of the criteria, i.e., all sending and receiving message sequences are correct.

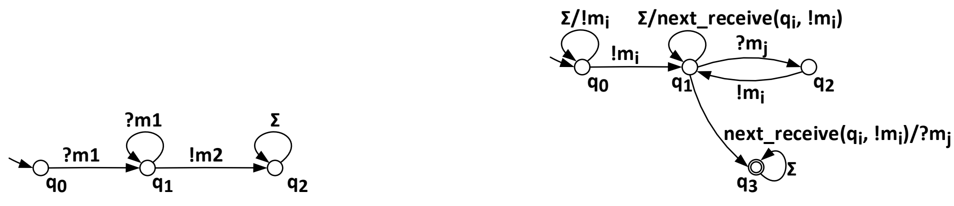
B. Performance Evaluation

In Figure 2 of the whole setup, if the infrastructure (OS, process engine, hardware and network configuration) is the same, performance depends mainly on the process design and the workload, i.e., $performance = Test(ProcessDesign, workload)$.

We use similar setup of our performance test with our previous performance tests [15], which is shown in Figure 19. We use the cloud infrastructure from Amazon EC2. The initiator and responder processes are deployed on two computing instances and we use a local client to collect the performance data. At the node *initiator*, the original and transformed initator processes are deployed. At the node *responder*, the original and transformed responder processes are deployed. At the node *performance data collector*, a local client is deployed.

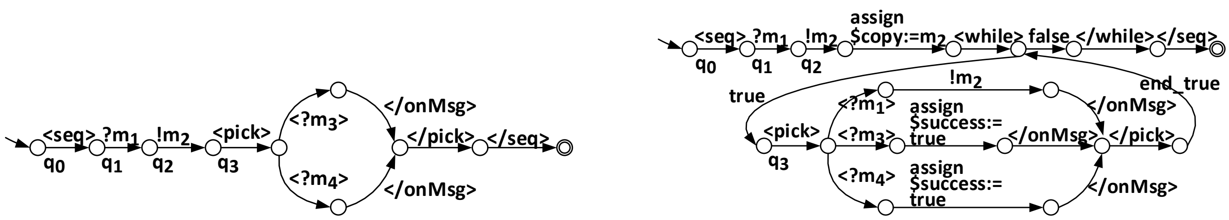
We evaluated the performance overhead of our transformed process under different workloads. The number of requests sent per minute by the local client complies with a Poisson distribution with parameters $\lambda = 5$ and $\lambda = 10$ requests per minute. We used these workloads because according to our tests under the available hardware and software configurations, higher workloads would exhaust the server resources. We use the open source Apache ODE process engine where an embedded Derby [16] database is used. The Amazon EC2 instance type is t1.micro with 1 vCPU and 0.594GiB memory. Each test run lasted for 60 minutes, but only the response times in the 30 minutes in the middle of this period have been considered (steady state).

The performance data is shown as Table II. Under the workload of $\lambda = 5$, the average performance overhead of our transformation mechanism is 92 ms. Under the workload of $\lambda = 10$, the average overhead is 130 ms. We conclude then that the performance overhead increases with the workload.



(a) Initiator (b) Responder

Figure 16. Correctness criteria of process transformation.



(a) Automaton model of the original process (b) Automaton model of the transformed process

Figure 17. Illustration of the correctness validation, robust *NWA* model of Figure 8.

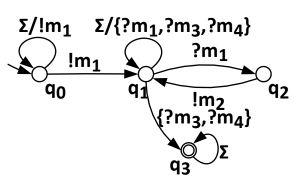


Figure 18. Correctness criteria for the illustrative process.

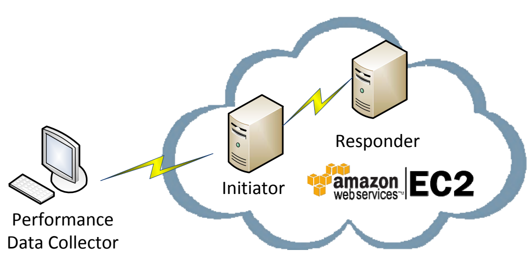


Figure 19. Setup of pperformance test.

TABLE II. PERFORMANCE OVERHEAD.

Origin	Trans	Overhead	Origin	Trans	Overhead
Workload $\lambda = 5$			Workload $\lambda = 10$		
287 ms	379 ms	92 ms	322 ms	452 ms	130 ms

However, we expect lower performance overhead when the infrastructure is scalable, like in a cloud environment.

C. Process Design Complexity

We have implemented the process designed in Figure 10, Figure 12 and Figure 13 using WS-BPEL. The number of activities before and after the process transformation is shown as Table III. The transformation is applied to one *reply* activity and the *receive* (or *pick*) activities in the set $next\_receive()$.

As an example, by applying our process transformations, a *reply* activity is replaced by one *sequence* structured activity and two basic activities: *reply* and *assign* nested in it.

```
<receive .../>
<assign name="assg1" ... />
<if ...>
  <condition .../>
  <assign name="assg2" .../>
  <else>
   <!-- Some Processing -->
   <assign name="assg3" .../>
  </else>
</if>
<reply .../>
```

V. Related Work

Solutions based on exception handling [17][18] is process-specific. WS-BPEL supports compensations of well-defined exceptions using exception handlers. However, elaborate process handler design requires process-specific knowledge of failure types and their related recover strategies. Alternatively, we try to ease the process designers from dealing with synchronization failures by a transparent process transformation from a given business process to its recovery-enabled counterpart.

A fault tolerant system can be built by coping with the occurrence of failures by applying redundancy [7]. Three kinds of redundancy are possible: information redundancy, time redundancy and physical redundancy. However, the existing solution either requires more effort of the business process designers, or additional infrastructure support, or both.

On physical layer, the robust solutions on process engine level [19][20] dependend on a specific process engine. We defined our solution based on the WS-BPEL building blocks without requiring extensions at the engine level. However, the

TABLE III. Number of activities before and after transformation.

	Before transformation	After transformation
Initiator	1 (*invoke*)	7 (2 *sequence*, 2 *assign*, 1 *while*, 1 *scope*, 1 *invoke*)
Responder	1 (*reply* in the set $next\_receive()$)	3 (1 *sequence*, 1 *reply*, 1 *assign*)
Responder	1 (*pick* with n *onMessage* branches in the set $next\_receive()$)	$2n + 5$ (1 *pick*, $n + 1$ *sequence*, 1 *reply*, $n + 1$ *assign*, 1 *while*)

transformed process can still be migrated to other standard process engines. Reliable network protocols such as HTTPR have been proposed to provide reliable synchronization. However, the deployment of these solutions increases the complexity of the network infrastructure. We assume that system crashes and network failures are rare events, thus extending the infrastructure may introduce too much overhead. Further, the solutions are not applicable in some outsourced deployment environments. For example, in some cloud computing environments, user-specific infrastructure configuration to enhance synchronization is not possible. Dynamic service substitution [21][22] is a way to perform recovery by replacing the target services by equivalent services. In [23][24], the QoS aspects of dynamic service substitution are considered. In our work, we do not change the business partners at runtime.

Information redundancy recovery is based on replication. Our cache-based process transformation is information redundant because a cache is a kind of replication. Time redundancy solutions include web services transactions. The WS-AT [25] standard specifies atomic web services transactions, while WS-BA [26] standard specifies relaxed transactions so that the participant can choose to leave the transaction before it commits. However, if a transaction rolls back, a process-specific compensation is required. Actually, transactions can deal with well-defined failures. The 2-phase commit distributed transaction protocol can not deal with system crash (referred to as *cite failure* in [27]). However, in a special case of process in which all participants send vote results to a coordinator, if the coordinator crashes before sending the vote results to any participant, all the participants are blocked and the final results of the transaction remain unknown.

VI. Conclusion and Future Work

In this paper, we have identified three types of interaction failures caused by system crashes and network failures and we have proposed a process interaction failure recovery method to cope with system crashes and network failures. This paper is an extension of our previous work [2][3][4][5], where we assumed that a certain pre-defined interaction follows the failed interaction. In this paper, we lift this limitation by allowing an arbitrary behavior to follow the failed interaction, making our solution more generally applicable. The challenge is to accept the resent message due to failures with the arbitrary control flow of the responder process. We transformed the business process design into nested word automata model. At a state that models the reception of an incoming message, we add an additional transition to accept the resent message due to failure. We have proved the correctness of our process transformations and we implemented a prototype to test the runtime performance of our method. The transformation complexity of our solution is analyzed. Currently, the transformation process is semi-automatic. We have implemented the automatic transformation from a WS-BPEL process to the NWA model, however, the transformation of the NWA model to the robust counter part is manually. In future, we will automate the transformation process and we will investigate more complex process interaction patterns.

References

[1] L. Wang, A. Wombacher, L. Ferreira Pires, M. J. van Sinderen, and C. Chi, "Robust interactions under system crashes and network failures of collaborative processes with arbitrary control flows," in The Seventh International Conferences on Advanced Service Computing, SERVICE COMPUTATION, 2015.

[2] L. Wang, A. Wombacher, L. Ferreira Pires, M. J. van Sinderen, and C.-H. Chi, "An illustrative recovery approach for stateful interaction failure of orchestrated processes," in IEEE 16th EDOC Workshops, 2012, pp. 38–41.

[3] ——, "A state synchronization mechanism for orchestrated processes," in IEEE 16th Intl. EDOC Conf., 2012, pp. 51–60.

[4] ——, "Robust client/server shared state interactions of collaborative process with system crash and network failures," in 10th IEEE Intl. Conf. on Services Computing (SCC), 2013.

[5] ——, "Robust collaborative process interactions under system crash and network failures," Intl. J. of Business Process Integration and Management, vol. 6, no. 4, 2013, pp. 326–340.

[6] A. Barros, M. Dumas, and A. Hofstede, "Service interaction patterns," in Business Process Management, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, vol. 3649, pp. 302–318.

[7] A. S. Tanenbaum and M. van Steen, Distributed Systems: Principles and Paradigms, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2006, ch. 8, pp. 321–375.

[8] Apache ODE, "Create a process," https://ode.apache.org/creating-a-process.html#in-memory-execution.

[9] SOA Technology for beginners and learners, "Transient vs. durable bpel processes," http://ofmxperts.blogspot.nl/2012/11/transient-vs-durable-bpel-processes.html, Nov. 2012.

[10] R. Alur and P. Madhusudan, "Adding nesting structure to words," J. ACM, vol. 56, no. 3, May 2009, pp. 16:1–16:43.

[11] OASIS, Web Services Business Process Execution Language, 2nd ed., OASIS, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, [retrieved: Feb., 2015], Apr. 2007.

[12] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, "Workflow patterns," Distributed and Parallel Databases, vol. 14, no. 1, Jul. 2003, pp. 5–51.

[13] J. E. Hopcroft, Introduction to Automata Theory, Languages, and Computation, 3rd ed. Pearson Addison Wesley, 2007.

[14] J. Schiedung, "Analysing and modelling of IOTP transactions by CPNs and BPEL," Master's thesis, Darmstadt University of Technology, 2004.

[15] L. Wang, A. Wombacher, L. Ferreira Pires, M. J. van Sinderen, and C.-H. Chi, "A colloborative processes synchronization method with regards to system crashes and network failures," in the 29th Symp. on Applied Computing (SAC), 2014.

[16] Apache Software Foundation, "Ode database setup," http://ode.apache.org/databases.html.

[17] N. Russell, W. Aalst, and A. Hofstede, "Workflow exception patterns," in Advanced Information Systems Engineering, ser. Lecture Notes in Computer Science, E. Dubois and K. Pohl, Eds. Springer Berlin Heidelberg, 2006, vol. 4001, pp. 288–302.

[18] B. S. Lerner, S. Christov, L. J. Osterweil, R. Bendraou, U. Kannengiesser, and A. E. Wise, "Exception handling patterns for process modeling," IEEE Transactions on Software Engineering, vol. 36, no. 2, 2010, pp. 162–183.

[19] S. Modafferi, E. Mussi, and B. Pernici, “Sh-bpel: a self-healing plug-in for ws-bpel engines,” in the 1st workshop on Middleware for Service Oriented Computing. NY, USA: ACM, 2006, pp. 48–53.

[20] A. Charfi, T. Dinkelaker, and M. Mezini, “A plug-in architecture for self-adaptive web service compositions,” in IEEE Intl. Conf. on Web Services, Jul. 2009, pp. 35–42.

[21] M. Fredj, N. Georgantas, V. Issarny, and A. Zarras, “Dynamic service substitution in service-oriented architectures,” in IEEE Congress on Services - Part I, Jul. 2008, pp. 101–104.

[22] L. Cavallaro, E. Nitto, and M. Pradella, “An automatic approach to enable replacement of conversational services,” in Service-Oriented Computing, L. Baresi, C.-H. Chi, and J. Suzuki, Eds. Springer Berlin Heidelberg, 2009, vol. 5900, pp. 159–174.

[23] O. Moser, F. Rosenberg, and S. Dustdar, “Non-intrusive monitoring and service adaptation for ws-bpel,” in the 17th intl. conf. on World Wide Web. NY, USA: ACM, 2008, pp. 815–824.

[24] F. Moo-Mena, J. Garcilazo-Ortiz, L. Basto-Diaz, F. Curi-Quintal, S. Medina-Peralta, and F. Alonzo-Canul, “A diagnosis module based on statistic and qos techniques for self-healing architectures supporting ws based applications,” in Intl. Conf. on Cyber-Enabled Distributed Computing and Knowledge Discovery, Oct. 2009, pp. 163 –169.

[25] OASIS Web Services Transaction (WS-TX) TC, Web Services Atomic Transaction (WS-AtomicTransaction), http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec.html, [retrieved: Feb., 2015], OASIS Standard, Rev. 1.2, Feb. 2009.

[26] ——, Web Services Business Activity (WS-BusinessActivity), http://docs.oasis-open.org/ws-tx/wstx-wsba-1.2-spec-os/wstx-wsba-1.2-spec-os.html, [retrieved: Feb., 2015], OASIS Standard, Rev. 1.2, Feb. 2009.

[27] M. T. Ozsu, Principles of Distributed Database Systems, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007, ch. 12.