Robust Client/Server Shared State Interactions of Collaborative Process with System Crash and Network Failures

Lei Wang*, Andreas Wombacher*, Luís Ferreira Pires*, Marten J. van Sinderen* and Chihung Chi†

*Centre for Telematics and Information Technology, University of Twente

†CSIRO, Australia

Contact email: {wangl, a.wombacher, l.ferreirapires, m.j.vansinderen}@ewi.utwente.nl, chihungchi@gmail.com

Abstract—With the possibility of system crashes and network failures, the design of robust client/server interactions for collaborative process execution is a challenge. If a business process changes state, it sends messages to relevant processes to inform about this change. However, server crashes and network failures may result in loss of messages. In this case, the state change is performed by one party, resulting in state/behavior inconsistencies and possibly deadlocks. Our basic idea to solve the problem is to cache the response (in a synchronous interaction) if the state of the process instance has changed by the request message. The possible state inconsistencies are recognized and compensated by state-caching and retrying failed interactions. By doing this work, we have learnt the possible failures caused by system crashes and network failures. Our results make it possible to build robust interactions by cached-based process transformations.

Keywords—Business Process; BPEL; State Synchronization; Service Interaction;

I. INTRODUCTION

Electronic data interchange has grown significantly in the last decade. Often data interchange is based on processes run by different parties exchanging messages to coordinate the execution of the business process. With the possibility of system crashes and network failures, the design of robust client/server interactions for collaborative process execution is a challenge. In general a state inconsistency is not detected by a partner’s workflow engine, as can be seen from a screen dump of an error after a system crash of an orchestration engine such as Apache ODE (see Fig. 1). Fig. 2a illustrates the problem with a ticket selling process. Multiple client instances (*client1*, *client2*) are submitting order messages (*order1*, *order2*). At state *c1*, *client1* crashes after submitting *order1* without receiving *result1*. Some operations can be safely repeated. A request that has this property is called “idempotent” [1]. However, the ticket subscription operation described above does not have this property. First, server state changes to *s2* but *client1* does not change its state. Second, the server further changes its state to *s2’* after interaction with *client2*.

Standard technical solutions are reliable messaging protocols or business transactions. However, these solutions require additional infrastructure components or changes in the process respectively. Our aim is to transform the process

Figure 1. Apache ODE State Synchronization Errors

Figure 2. Idea of Caching Response Message

at a particular party to provide additional guarantees with regard to system crashes and network failures. In previous work [2], [3] we have considered coordination scenarios where the effects of the state changes in the collaboration do not affect other collaborations. In this paper we are focusing a server instance collaborating with multiple client instances, where one collaboration may affect another collaboration. Our basic idea is that whenever the state of the business process changes, the response message is cached. As shown in Fig. 2b, after a state change from *s1* to *s2*, the *ticket process* caches *result1*. When *client1* re-submits *order1* after recovery, the *ticket process* uses cached *result1* as response to achieve state consistency. The *state* of a business process is described by the values of the process variables. In this paper, in order to identify process *state* as a subset of the process variables, we model processes using Petri Nets (CPNTools, <http://cpntools.org>) to abstract the data dependency. We propose state identification criteria to the formal model. We propose to (automatically) extend the processes into synchronization-enabled counterparts via process transformations. The transformation is done in such a way that in the resulting processes possible state inconsistencies are recognized and compensated by state-caching and these

Figure 3. Multiple Processes Instances Shared State Types

processes retry failed interactions based on the contents of the cache.

We assume that in the case of server crashes or network failures, the state of the business process can be restored once recovered. This is a reasonable assumption, since most available business process engines, such as Apache ODE, work in this way. We choose WS-BPEL [4] as an illustrative process specification language because as an OASIS standard, it is widely used by enterprises. However, our mechanisms are applicable to other process specification languages which support similar workflow patterns [5].

This paper is further structured as follows. Section II investigates failures caused by network and system crashes. Section III presents our formalization of WS-BPEL processes using Petri Nets. Section IV proposes state determination criteria based on the formalization. Section V discusses the implementation of our cache-based process transformation. Section VI evaluates our mechanism. Section VII discusses related work and Section VIII gives our conclusions.

II. ANALYSIS OF PROCESS STATE TYPES AND SYNCHRONIZATION FAILURES

A. Process State Types

Each process instance synchronizes its state with partner process instances via messages. Thus, the state information is “shared” implicitly between multiple process instances. How state information is shared [6] depends on the service interaction patterns [7] of the client and server processes. From the client’s point of view, one client instance can interact with one server instance (1-1) or many server instances (1-n). From the server point of view, one server instance can interact with one client instance (1-1) or many

Table I
FAILURE SCHEME

Type of failure	Description
Crash failure	Working correctly until it halts.
Omission failure	Fails to respond to incoming requests.
Timing failure	Response lies outside the specified time interval.
Response failure	Response is incorrect.
Arbitrary failure	Produce arbitrary responses.

Figure 4. Synchronization Failure Analysis for Shared, Static State Type

client instances (n-1). From a global point of view, we take a combination, as is visualized in Fig. 3. In type a, the state information is “shared” between clients. The number of server instances is “static” (could be one or more, but it is a fixed number at runtime). This state information type is named *shared, static*. In type b, the state information is shared between “multiple” server instances but “private” to each client instance. In type c, the state information is “private” to the requester-responder pair. In type d, the state information is “shared” between all instances. We name this state information type *multiple, private*.

B. Process Synchronization Failure Analysis

One of the developed failure schemes is shown in Tab. I [1]. With regards to client/server interactions with system crashes and network failures, we focus on “crash failure”, “omission failure” and “timing failure”. However, “arbitrary failure” (also called “Byzantine failure”) is more like a security issue and out of the scope of this work. In this paper, we propose a solution for the synchronization failure of state type *shared static* (Fig. 3a).

The UML sequence diagram of the synchronization for the state type *shared, static* is presented as Fig. 4. Multiple initiator process instances (A1, A2) synchronize with the responder process instance B. The possible failure points for a synchronization between A1 and B are marked as $X_{fp1} \sim X_{fp6}$. The failure points X_{fp1} , $X_{fp3} \sim X_{fp6}$ are discussed in our previous work [2], [3]. We look into failure point X_{fp2} . If A1 fails after sending $m1$, this is an omission failure because $m2$ cannot be received by A1. If A1 re-sends $m1$, the processes will not synchronize, since the interaction between A2 and B has already changed the state of B. This failure is referred in this paper as *pending*.

Figure 5. Solution Overview

request failure.

III. MODELING: BUSINESS PROCESSES TO PETRI NETS

An overview of our solution of failures is shown in Fig. 5. Given a business process, we infer state change for all synchronous process operations. We model the business process as Petri Nets and further generate the Occurrence Graph/Automaton models. By applying proposed criteria to the Petri Nets and Occurrence Graph/Automaton models, we detect whether a state change happens. For all process operations that change the process state, we do process transformation. The transformation is done in such a way that:

- For a new request coming from client, server caches and replies the response message .
- For the same synchronization request sent multiple times from the same client (which implies a client failure happens), the server process replies with the cached response.

We assume that each message is uniquely identifiable. This is a reasonable assumption because in a real business scenario, e.g., order information submitted with the same product will have different id fields and payment information is submitted with different timestamps.

We formalize WS-BPEL process as Petri Nets with the denotation of data flow. WS-BPEL models using Petri Nets have been reported in the literature, however, each approach has its particular focus and hardly fits our needs. For example, [8] focuses on control flow modeling thus state information is implicit. [9], [10], [11] address activity stops and correlation errors, which are not relevant and therefore unnecessarily complicate our formalism. Thus, we propose a simplified Petri Nets formalism. The Petri Net structure of each WS-BPEL activity has one start place and one sink place. The net structure of each activity can be nested or concatenated with each other, which is the semantic of WS-BPEL structured activities.

In order to improve readability, we use the two conventional notations to present reading or writing of process variables by activities. As shown in Fig. 6a, the Petri Net representation of an activity reading a process variable V is that the transition takes a token from the place and then puts a token back. We use dashed arrow as a graphical

Figure 6. Convention for Reading and Writing of BPEL Process Variables

Figure 7. The Petri Net Model for Basic Activities

convention. As shown in Fig. 6b, the Coloured Petri Net representation of an activity writing a process variable V is that the transition takes a token $v1$ out from the place and then puts another token $v2$ into it. We use double arrows in this convention. We use Petri Nets without Coloured extension since we do not need to distinguish $v1$ from $v2$.

WS-BPEL activities is divided into two categories: basic and structured activities.

A. Basic Activities

Fig. 7a shows the Petri Net of a *receive* activity. Places $c1$ and $c2$ are the input and output control places. In order to express the *receive* semantic of WS-BPEL, the transition takes a token out from the msg place and “writes” to the place $v1$. Similarly, we have modeled basic activities *reply*, *assign*, and *invoke* as shown in Fig. 7b, to Fig. 7d, respectively.

We denote the data flow as a subset of the arcs annotated in bold. The data flow of the *assignment* activity (Fig. 7c, denoted as bold arcs) is from place $v1$ (and $v2$) to the transition *assg*, then to the place $v3$.

B. Structured Activities

The Petri Net of an *if* activity is presented in Fig. 8. Places $c1$ to $c6$ model the control flow. In WS-BPEL, the condition of an *if* activity is an expression, e.g. $\$v1 < \$v2$. The process variables that appear in the condition expression are modeled as places p_v1 , p_v2 in Petri Nets. The positive (negative) evaluation of the condition results in the execution of *true* (*false*) branch of the WS-BPEL process, which is modeled as a hierarchical transition *body_true* (*body_false*) and is initialized by firing transition *cond_true* (*cond_false*). In the Petri Net model, the transitions *cond_true* and *cond_false* “read” the places p_v1 and p_v2 . A token in

Figure 8. The Petri Net Model for *if* Activity

Figure 9. The Data Flow Path of *if* Activity

the place *in_true* (*in_false*) represents that the modeled WS-BPEL is executing the *true* (*false*) branch. We name these two places as control boundary indication places.

The data flow (denoted as bold arcs) starts from the “reading” of places *p_v1* (and *p_v2*) by the transition *cond_true* (*cond_false*) to the control boundary indication place *in_true* (*in_false*). The evaluation of values of variables in a condition determines the variables that are changed because it determines the branch to be chosen. Thus the process variables changed inside of the *if* branches should depend on the conditional variables. We model this as an “read” of control boundary indication place by the assignment transition that hierarchically nested in *if*. As illustrated in Fig. 9, which shows a true branch of an *if* activity. The transition *assg* is the Petri Net model of an assignment activity. By the application of the rule, we add an “read” of the indicator place *in_true* by the transition *assg*. Then the data dependency path representing that *v3* depends on *v1* and *v2* can be generated.

The idea of modelling system crash (network failure) is to use a transition which takes a token out from places modeling control flow (message channel) and puts a token into corresponding place which represents failure. Due to page limitations, the models of other structured activities and failures are not presented in this paper, which can be found from our technical report [12].

IV. STATE DETERMINATION CRITERIA

A. Inbound Message Activity

In order to identify the synchronous operation boundaries, we show the concept of Inbound Message Activity (IMA) from WS-BPEL. IMAs are activities in which messages are received from partners, and consists of:

- *receive*: receive message from partners.

Figure 10. Snippet of Ticket Process

Figure 11. Criterion Automaton of Read Before Write

- *pick*: based on the type of message received or a timeout, one execution branch is chosen.

Other types of IMAs like event handlers are out scope of this paper. The control boundary of a synchronous process operation starts with an IMA and ends with a *reply* activity.

We will use a ticket subscribing process to illustrate our criteria to identify process state variables. As shown in Fig. 10, the core of the process is a *pick* activity. Three *onMessage* handlers are nested inside the *pick* activity for the corresponding message type: “subscribe” for the subscription operation; “revoke” for the ticket revoke operation and “termination” to end the business process. The *pick* activity is nested in a *while* activity, allowing the process operations “subscribe” and “revoke” to be executed multiple times.

B. Inside process operation criteria

The following criteria is used only inside the control boundary of a process operation.

1) *Read before write*: The process variable should be read first and written afterwards. Formally, in Fig. 11, this criterion is presented as an automaton with the alphabet {*read(v)*, *write(v)*, *}, where *read(v)* and *write(v)* denote the reading and writing of the process variables *v* respectively. State 0 denotes the initial state. State 1 is the state in which the process variable *v* is read but not being written and State 2 is the accepted state which represents that the variable *v* is read first and written afterwards.

We discuss the use of the criteria automaton to check the Petri Net model in Section V.

2) *Circular Dependency*: The data flow denoted by the bold arcs in the Petri Net representation of the places should form a cycle, and the place representing the variable should be included in this cycle. The Petri Net model of the operation “subscribe” of the ticket process is shown in Fig. 12. The data flow path *true*, *inT*, *assg2*, *sub*, *assg1*, *ticket*, *true* forms a cycle, where two places representing variables can be found: *sub* and *ticket*, which considered as state variables.

Figure 12. Petri Net of Subscribe Operation of the Ticket Process

Figure 13. Automaton Model of Cross Process Operation Criteria

C. Cross-Process Operation Criteria

If a variable v has its value written inside the operation and read outside the operation afterwards, v should be considered as a state variable. Without loss of generality, for a specific synchronous process operation, say, the subscribe ticket process operation, we can construct a criteria automaton $\{q_0, Q, F, \Sigma, \delta\}$, with the alphabet $\Sigma = \{IMA_subscribe, OMA_subscribe, r_history, w_history\}$ for a process variable \$history. IMA_subscribe represents the *receive* activity. OMA_subscribe represents the *reply* activity. r_history is an assignment activity that reads the value of \$history and w_history is an assignment activity that writes the value of \$history. We define state set Q to contain five states, indexed from 0 to 4. The initial state q_0 is state 0. The final state set is $\{4\}$. Fig. 13 shows the automaton constructed in this way. The transition function δ is specified as follows:

- From state 0: IMA_subscribe leads to state 1; Stay in state 0 otherwise.
- From state 1: OMA_subscribe leads to state 0; w_history leads to state 2; Stay in state 1 otherwise.
- From state 2: OMA_subscribe leads to state 3; Stay in state 2 otherwise.

Figure 14. Architecture of Prototype Implementation

- From state 3: w_history leads to state 0; r_history leads to state 4. Stay in state 3 otherwise.
- From state 4: Stay in state 4 for any element of Σ .

V. IMPLEMENTATION DETAILS

The architecture of our prototype implementation is shown in Fig. 14. We implemented the state determination criteria proposed in Section IV in the State Dependency Analysis module to determine the state information. The result is used to decide whether to trigger the process transformation. The Process Transformation module performs the actual process transformation to cache the response message to achieve robust client/server interaction.

A. State Dependency Analysis Module

At the bottom layer is the CPN Simulation Module and the Automaton Class Library. The CPN Simulation Module generates the Occurrence Graph model from the Petri Net model. Inside this module the Access/CPN Class Library provides the Petri Net simulation support and the Graph Search Library provides graph representation support. The Occurrence Graph generation algorithm implemented in the State Space Generation Module is presented below.

```

1  Init : Queue :  $\mathbf{Q} \leftarrow \text{Empty}$ ,
2  add init marking  $\mathbf{m0}$  to Graph :  $\mathbf{G}$ 
3  Enqueue( $\mathbf{Q}, \mathbf{m0}$ )
4  while( $\mathbf{Q}$  is not empty) do
5    marking  $\mathbf{u} \leftarrow \text{Dequeue}(\mathbf{Q})$ 
6    for(each  $\mathbf{v}$  in directly reachable markings
       from  $\mathbf{u}$ ) do
7      if( $\mathbf{v}$  is not in  $\mathbf{G}$ ) then
8        Enqueue( $\mathbf{v}, \mathbf{G}$ )
9        add  $\mathbf{v}$  to  $\mathbf{G}$ 
10     add  $\langle \mathbf{u}, \mathbf{v} \rangle$  to  $\mathbf{G}$ 

```

In the middle layer, the occurrence graph is mapped to the automaton. Fig. 15 shows how Petri Nets concepts are mapped to automaton concepts. The Petri Net transitions are annotated with the names of the business activities, so when the Petri Net transition set is mapped to the automaton alphabet, an addition alphabet is required as

Figure 15. A Mapping from Occurrence Graph to Automaton

Figure 16. Cached Based Process Transformation Details

input. If the transition name is in the alphabet, the Petri Net transition is mapped to the corresponding automaton transition. If not, the Petri Net transition is mapped to an epsilon automata transition. We then transform the ϵ -NFA to DFA. Finally, we calculate the intersection of the DFA with the criteria automata in order to determine the necessary state information.

B. Process Transformation Module

As shown in Fig. 16a, a synchronous operation receives a message, does some processing and then replies. Our transformation is to replace the processing and *reply* by an *if* activity. The condition of the *if* activity checks whether the request message is cached. If it is cached, the process uses the cached response as reply. If the message is not cached, which implies that the message is sent for the first time, the message is processed. The response message is cached and replied.

The data structure of the cache is declared as an array of cached items. Each item is a $\langle \text{request}, \text{response} \rangle$ value pair. The cache structure is declared as an XSD definition in WSDL. In the WS-BPEL process, the cache is declared as a variable. Three cache operations are required:

- Given a request message, check whether the corresponding response message is cached.
- Given a request, get the corresponding response.
- Given a value pair of request and corresponding response messages, add it to the cache.

The cache data operation is implemented as XSLT transformations. An *assign* activity to check whether the request is cached is shown in the following WS-BPEL code:

```
<bpel:assign>
  <bpel:copy>
    <bpel:from>bpel:doXsltTransform(
      testCached.xsl,
      $cache, cacheItem,
      $request.payload)
    </bpel:from>
    <bpel:to>
      variable=foundCachedReques />
    </bpel:copy>
  </bpel:assign>
```

The from part of the assignment activity is the BPEL function `doXsltTransform()` with the request message and `$cache` as its parameters. Variable `$foundCachedReques` contains the result.

VI. EVALUATION

We evaluated our mechanisms in three aspects: their correctness, their performance overhead and the complexity of the process transformation.

A. Correctness Evaluation

To evaluate the correctness of our transformation, we started by proposing the correctness criteria, in the form of finite state automata. The alphabet Σ accepted by the automata is the set of sending and receiving messages. Then we model the transformed (in Fig. 16) business process to extract the automata model. We demonstrate correctness by showing that the automata model of the business process is subsumed by the criteria automata.

1) *Correctness Criteria:* For any message $M1$, we use the finite state automaton $\langle Q, \Sigma, \delta, q_0, F \rangle$ to formalize this correctness criteria. The global states of message sending and receiving status are modeled as the state set $Q = \{0, 1, 2\}$. The alphabet $\Sigma = \{sendM1, receiveM1\}$. *SendM1* models the behavior of sending message $M1$ and *receiveM1* models the behavior of receiving message $M1$, which $q_0 = 0$ is the initial state and $F = \{0, 2\}$ is the set of accepted states. The transition rules are visualized in Fig. 17a. A transition *sendM1* from state 0 to state 1 models the sending of message $M1$, a transition *sendM1* from state 1 to itself models that the message may be sent multiple times and a transition *receiveM1* from state 1 to state 2 represents that the message has been received.

The synchronous communication criteria should take into consideration both request and response messages. Informally, 1) a request may be sent multiple times until received; 2) a response message may be sent afterwards; 3) the sequence of 1) and 2) can be repeated multiple times until the response message is received. This criteria is formalized using the automaton shown in Fig. 17b. Details of the criteria can be found in our technical report [12].

2) *Evaluation Procedure:* Fig. 18 shows the correctness evaluation in three steps: first we prove that a business process can pass the correctness criteria when no failure

Figure 17. Correctness Criteria

Figure 18. Evaluation Procedure

Table II
AVERAGE PROCESS RESPONSE TIME DIFFERENT WORKLOAD

Origin	Trans	Overhead	Origin	Trans	Overhead
Workload $\lambda = 5$			Workload $\lambda = 10$		
313 ms	375 ms	62 ms	256 ms	440 ms	184 ms

happens, then we prove that the business process cannot pass the criteria if the pending request failure happens and finally we prove that the transformed business process fulfills correctness criteria when the pending request failure happens. The evaluation is done by our automata subsumption checking program. The transformation is applied to synchronous process interactions where pending request failure may happen. Due to page limitations, we omit the details of the evaluation in this paper.

B. Performance Overhead Evaluation

In case the infrastructure (software, hardware and network configuration) is the same, the performance depends on the process design and the workload, i.e. $performance = Test(ProcessDesign, workload)$

We want to evaluate the performance overhead with different workloads. The requests sent per minute by the simulation client comply a Poisson Distribution. We collect performance under two workloads, namely $\lambda = 5$ and $\lambda = 10$. (However, according to our test under current hardware and software configurations, still higher workload will exhaust the server resource.) Each test run lasted for 60 minutes. Only the response time in the 30 minutes in the middle of this period have been considered (steady state).

Under the workload $\lambda = 5$, the performance overhead of our transformation mechanism is 62ms. Under the workload that $\lambda = 10$, the performance overhead of our transformation mechanism is 184ms. We conclude then that the performance overhead increases with the workload. However, we expect lower performance overhead when the infrastructure is scalable, like in a cloud environment.

Figure 19. Overview of Related Solutions

C. Process Design Complexity

We have implemented the process designed in Fig. 16 using WS-BPEL. The synchronous interaction is presented with two activities (one *receive* and one *reply*). By applying our process transformation mechanism, we add one structured activity and three basic activities. One assignment activity is used to check whether a request message was cached or not. The second assignment is used to get the cached response message and add it to the cache. The third one is used to cache the request message. In future work, the process transformation can be done automatically based on XML transformation techniques and thus transparently to process designers.

VII. RELATED WORK

Fault handling approaches, such as [13], [14], require that the process designers are aware of possible failures and their recovery strategies. Alternatively, cache based process transformations can be defined to add generic state synchronization behaviors to collaborative business processes. As described in [1], the key technique for masking faults is to use redundancy. As shown in Fig. 19, three kinds of redundancy are possible: information redundancy, time redundancy and physical redundancy.

Physical redundancy-based solutions include [15], [16], [17], [18], [19]. Recovery mechanisms implemented as plugins for a WS-BPEL engine, such as [15], [16], [17], strongly depend on a specific WS-BPEL engine. The approach to recovery presented in [18], [19] consists of substituting a service with another one dynamically if a synchronization error occurs. In [20], [21], [22], the QoS aspects of dynamic service substitution are considered. An alternative to avoid the loss of state synchronization is to use reliable messaging. Message exchange is realized at the technical level using standard communication protocols like HTTP (on the TCP/IP protocol stack). However, HTTP does not provide reliable messaging. Reliable messaging protocols such as HTTPR, WS-RX solve the problem by introducing a middle layer, which increases the complexity of the required infrastructure. We assume that server crashes and

network failures are rare events, and therefore extending the infrastructure introduces too much overhead. Further, adding a middle layer could turn out to be a problem for some out sourced deployment where the infrastructure layer is out of control of the process designer. For example, in some cloud computing environment, user specific network configuration to enhance state synchronization is not available. Another possibility is to design the process to deal with unreliable messaging. However, this makes the process design and the created model much more complicated. Instead we propose to (automatically) extend the original processes into synchronization-enabled counterparts via process transformations.

Information based redundancy is achieved based replication. Our solution is of this kind. Time based redundancy solutions include WS-Transactions. Transaction-based process recovery approaches, such as in WS-AT and WS-BA, require a central coordinator, in contrast with our approach, which is based on process transformations.

VIII. CONCLUSIONS

In this paper, we propose robust interaction mechanisms for collaborative business processes. We identify four ways in which state can be shared between multiple process instances. We look into possible interaction failures of the “shared static” state. The challenge is how to cache process interaction messages in order to recover. We transform the business process design into an automata model. The alphabet of the automata is the sending and receiving of messages and the reading and writing of process state. We define a criteria automata for identifying state changes that are worth caching. We implement our illustrative prototype. As a next step, we will extend our work to include other types of shared states by multiple process instances.

REFERENCES

- [1] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*, 1st ed. Prentice Hall, 2002, ch. 7, pp. 366–368.
- [2] L. Wang, A. Wombacher, L. F. Pires, M. J. van Sinderen, and C. Chi, “A state synchronization mechanism for orchestrated processes,” in *IEEE 16th Intl. Enterprise Distributed Object Computing Conference (EDOC)*, 2012, pp. 51–60.
- [3] —, “An illustrative recovery approach for stateful interaction failure of orchestrated processes,” in *IEEE 16th EDOC Workshops*, 2012, pp. 38–41.
- [4] *Web Services Business Process Execution Language*, OASIS Standard, Rev. 2.0, 2007.
- [5] W. M. P. van der Aalst, et al., “Workflow patterns,” *Distributed and Parallel Databases*, vol. 14, pp. 5–51, Jul. 2003.
- [6] C. Atkinson and P. Bostan, “Towards a client-oriented model of types and states in service-oriented development,” in *IEEE 13th EDOC Conference*, 2009, pp. 119–127.
- [7] B. Alistair, et al., “Service interaction patterns,” in *Business Process Management*. Springer Berlin Heidelberg, 2005, vol. 3649, pp. 302–318.
- [8] C. Ouyang, et al., “Formal semantics and analysis of control flow in ws-bpel,” *Science of Computer Programming*, vol. 67, no. 23, pp. 162–198, Jul. 2007.
- [9] C. Stahl, “A petri net semantics for bpel,” ser. Informatik-Berichte. Institut für Informatik, 2005, no. 188.
- [10] S. Hinz, K. Schmidt, and C. Stahl, “Transforming bpel to petri nets,” in *Business Process Management*. Springer Berlin Heidelberg, 2005, vol. 3649, pp. 220–235.
- [11] N. Lohmann, “A feature-complete petri net semantics for ws-bpel 2.0,” in *Web Services and Formal Methods*. Springer Berlin Heidelberg, 2008, vol. 4937, pp. 77–91.
- [12] L. Wang, et al., “Robust client/server shared state interactions for collaborative process with system crash and network failures,” CTIT, University of Twente, Tech. Rep., 2013. [Online]. Available: <http://paperscc2013.appspot.com/SCC2013TechRepo.pdf>
- [13] N. Russell, W. Aalst, and A. Hofstede, “Workflow exception patterns,” in *Advanced Information Systems Engineering*. Springer Berlin Heidelberg, 2006, vol. 4001, pp. 288–302.
- [14] B. Staudt, et al., “Exception handling patterns for process modeling,” *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 162–183, 2010.
- [15] S. Modafferi, E. Mussi, and B. Pernici, “Sh-bpel: a self-healing plug-in for ws-bpel engines,” in *the 1st workshop on M4SOC*. NY, USA: ACM, 2006, pp. 48–53.
- [16] S. Modafferi and E. Conforti, “Methods for enabling recovery actions in ws-bpel,” in *On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE*. Springer Berlin Heidelberg, 2006, vol. 4275, pp. 219–236.
- [17] A. Charfi, T. Dinkelaker, and M. Mezini, “A plug-in architecture for self-adaptive web service compositions,” in *IEEE Intl. Conf. on Web Services*, 2009, pp. 35–42.
- [18] M. Fredj, N. Georgantas, V. Issarny, and A. Zarras, “Dynamic service substitution in service-oriented architectures,” in *IEEE Congress on Services - Part I*, 2008, pp. 101–104.
- [19] L. Cavallaro, et al., “An automatic approach to enable replacement of conversational services,” in *Service-Oriented Computing*. Springer Berlin Heidelberg, 2009, vol. 5900, pp. 159–174.
- [20] O. Moser, F. Rosenberg, and S. Dustdar, “Non-intrusive monitoring and service adaptation for ws-bpel,” in *the 17th Intl. Conf. on WWW*. ACM, 2008, pp. 815–824.
- [21] F. Moo-Mena, et al., “A diagnosis module based on statistic and qos techniques for self-healing architectures supporting ws based applications,” in *Intl. Conf. on CyberC.*, 2009, pp. 163–169.
- [22] —, “Defining a self-healing qos-based infrastructure for web services applications,” in *11th IEEE Intl. Conf. on Computational Science and Engineering Workshops*, 2008, pp. 215–220.