Robust collaborative process interactions under system crash and network failures

Lei Wang*

CTIT,
University of Twente,
P.O. Box 217,
7500 AE Enschede, The Netherlands
E-mail: wangl@ewi.utwente.nl
*Corresponding author

Andreas Wombacher

CTIT,
University of Twente,
P.O. Box 217,
7500 AE Enschede, The Netherlands
and
Nspyre,
Dillenburgstraat 25-3,
5652 AM Eindhoven, The Netherlands
E-mail: andreas.wombacher@nspyre.nl

Luís Ferreira Pires and Marten J. van Sinderen

CTIT,
University of Twente,
P.O. Box 217,
7500 AE Enschede, The Netherlands
E-mail: l.ferreirapires@utwente.nl
E-mail: m.j.vansinderen@utwente.nl

Chi-Hung Chi

Computational Informatics – Hobart, CSIRO,
3-4 Castray Esplanade, Hobart, Tasmania, 7000, Australia
E-mail: 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 its state, it sends messages to the 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 the sending process in isolation, resulting in state/behaviour inconsistencies among processes and possibly undistinguished deadlocks. Our basic idea to solve this problem is to cache the response (in a synchronous request-response interaction) if the state of the process instance has changed by the request message. The possible state inconsistencies are recognised and compensated by state-caching and by retrying failed interactions.

Keywords: robust; business process; WS-BPEL; state synchronisation; data dependency; service interaction; system crash; network failure; recovery; Petri nets.

Reference to this paper should be made as follows: Wang, L., Wombacher, A., Pires, L.F., van Sinderen, M.J. and Chi, C-H. (2013) 'Robust collaborative process interactions under system crash and network failures', *Int. J. Business Process Integration and Management*, Vol. 6, No. 4, pp.326–340.

Biographical notes: Lei Wang is pursuing his PhD at the EEMCS Department, University of Twente. He received his Masters degree of Software Engineering from Tsinghua University in 2012. He has published in international conferences such as EDOC and SCC. His current research interests include recovery of web services interaction under system crashes and network failures.

Andreas Wombacher is a management consultant at Nspyre. He received his Master and PhD from the Technical University of Darmstadt. He gathered professional experience at IBM (Germany), Fraunhofer Gesellschaft (Germany), the University of Twente (Netherlands), and the Swiss Federal Institute of Technology in Lausanne (EPFL). His research interests are in the area of service oriented architectures, distributed data management, and data processing and analysis with a focus on sensor networks always under the perspective of processes. He has been involved in several organisation and programme committees.

Luís Ferreira Pires is an Associate Professor at the University of Twente, The Netherlands. He holds a BSc in Electronics (1983), MSc in Electrical Engineering (1989) and PhD in Electrical Engineering (1994). Most recently he has been working on the application of model-driven engineering (MDE) and semantic web technologies to context-aware applications and services, and to service composition. He has contributed to various national and international research projects, and co-authored more than 100 research papers in international conferences, workshops and journals. He is currently Director of the 'Business Information Technology' Education Programme of the University of Twente.

Marten J. van Sinderen is an Associate Professor at University of Twente, The Netherlands, where he also leads the research area 'Service Science' at the Centre for Telematics and Information Technology. His research focuses on the design of networked information systems, particularly next-generation collaborative enterprises and smart system applications. He has participated in several inter/national research projects and coauthored over 100 research papers. He is a member of the editorial boards of *Enterprise Information Systems Journal* and *Service Oriented Computing and Applications Journal*. He holds a PhD in Computer Science.

Chi-Hung Chi is currently a science leader in the Computational Informatics Division, CSIRO, Australia. He received his PhD from Purdue University. After working for Philips Research Laboratories and for IBM Poughkeepsie, he returned back to academia as a Professor from 1993 to 2012, starting first with Chinese University of Hong Kong, then National University of Singapore and later Tsinghua University. He has published more than 200 papers in international journals and conferences and holds six US patents. His current research areas include cognitive computing, behaviour analytics, service and cloud computing.

This paper is a revised and expanded version of a paper entitled 'Robust client/server shared state interactions of collaborative process with system crash and network failures' presented at 10th IEEE Intl. Conf. on Services Computing (SCC), Santa Clara, CA, USA, July 2013.

1 Introduction

The electronic collaboration of business organisations has grown significantly in the last decade. Often data interchange is based on processes run by different parties exchanging messages to synchronise their states. 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 process engine (a software executes business process). This can be seen from a screen dump of an error after a system crash of an process engine such as Apache ODE (see Figure 1). Figure 1(a) shows the case where the client sends the message to a unavailable server. Figure 1(b) shows the case where the responder crashes without sending the response message. Figure 2(a) illustrates the problem with a ticket selling process and multiple client processes. At runtime, each client process may have multiple instances. Multiple client instances (*client1*, *client2*) submit order messages (*order1*, *order2*). The *client1* process may crash after submitting the *order1* message without receiving the *result1* response message. At a certain state *s1*, the *ticket process* receives the *order1* message, changes its state to s2 and sends the *result1* response. However, the response is not received by the *client1* due to the crash. The *client2* process submits message *order2* to the *ticket process* afterwards. The *ticket process* changes its state to *s2'*. Now, the *client1* process re-submits the order after recovery. By re-processing the same order at state *s2'*, the ticket process will reply with a different *result'*, which may incur state inconsistency. Some operations can be safely repeated. A request that has this property is called 'idempotent' (Tanenbaum and van Steen, 2002). For example, a request asking for weather information can be repeated as many times as possible. However, the ticket subscription operation described above that receives the order submission does not have this property. First, the ticket process state changes to *s2*, but *client1* does not change its state accordingly. Second, the ticket process 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 to provide improved reliability with regard to system crashes and network failures. In previous work (Wang et al., 2012a, 2012b) we have considered coordination scenarios where the effects of the state changes in one collaboration do not affect other collaborations. In this paper, we focus on a server process instance collaborating with multiple client process instances, where one collaboration may affect another collaboration. Our basic idea to solve the problem is that whenever the state of a business process changes, the response message is cached. As shown in Figure 2(b), 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 restore 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 (CPN Tools, 2012) to extract the data dependencies. We propose state identification criteria and we represent them in the formal model. The original processes can be (automatically) transformed into their synchronisation-enabled counterparts via process transformations. The transformation is done in such a way that in the resulting processes possible state inconsistencies are recognised and compensated by state-caching, and these processes retry failed interactions based on the contents of the cache.

Figure 1 Apache ODE state synchronisation errors, (a) service unavailable (b) pending response (see online version for colours)

```
11:45:35,885 ERROR [ExternalService] Err
or sending message (mex={PartnerRoleMex#
bcdhagnmojq8ehhkscud3u [PID {http://de.f
hg.ipsi.oasys.businessScenario.sample.in
itiator}initiator-24] calling org.apache
.ode.bpel.epr.WSAEndpoint@1085z73a.proce
ssInteraction(...) Status ASYNC}): Conne
ction refused: connect
```

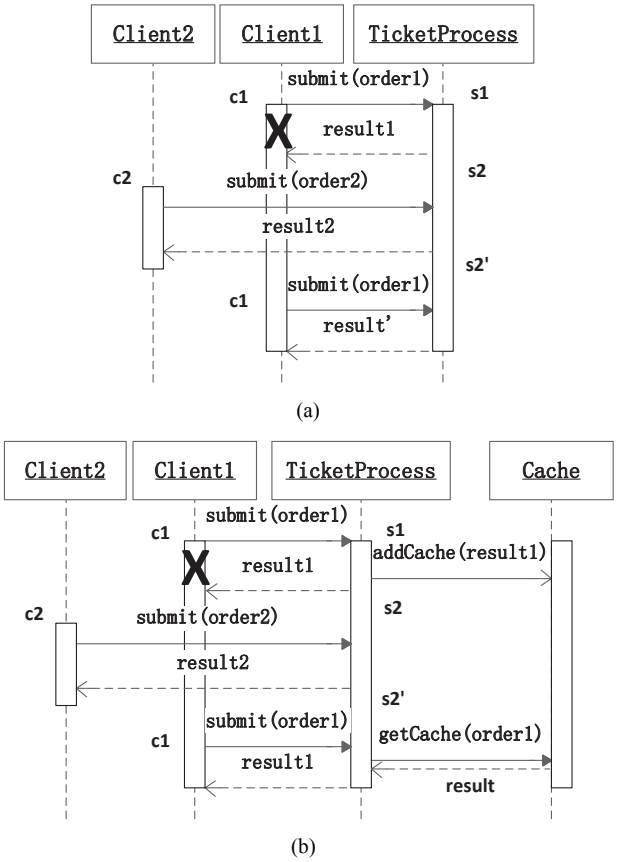
(a)

```
15:42:22,154 ERROR [INVOKE] Failure during in
voke: No response received for invoke (mexId=
hqejbhcnphr8fj908unbkr), forcing it into a fa
iled state.
```

(b)

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 (Apache Software Foundation, 2011) and Oracle SOA Suite (Oracle Fusion Middleware, 2011), work in this way. We assume that each message is uniquely identifiable. This is a reasonable assumption for a real business scenario, since e.g., each order information that is submitted for some product has a different identifier and each payment information is submitted with a different timestamp. We choose WS-BPEL (2007) 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 that support similar workflow patterns (van der Aalst et al., 2003).

Figure 2 Caching response message



This paper is an extension of our previous paper (Wang et al., 2013). The additional contents of this paper is the following.

1 we analyse the possible synchronisation failures in more detail
2 more Petri net models of WS-BPEL activities are described in this paper, such as, 'while' and 'pick' activities
3 the formal failure model is presented
4 omitted information on the correctness criteria and correctness evaluation process is presented in this paper.

This paper is further structured as follows. Section 2 investigates failures caused by network failures and system crashes. Section 3 presents our formalisation of WS-BPEL processes using Petri nets. Section 4 proposes state determination criteria based on the formalisation. Section 5 discusses the implementation of our cache-based process transformation. Section 6 evaluates our mechanism. Section 7 discusses related work and Section 8 gives our conclusions.

2 Analysis of process state types and synchronisation failures

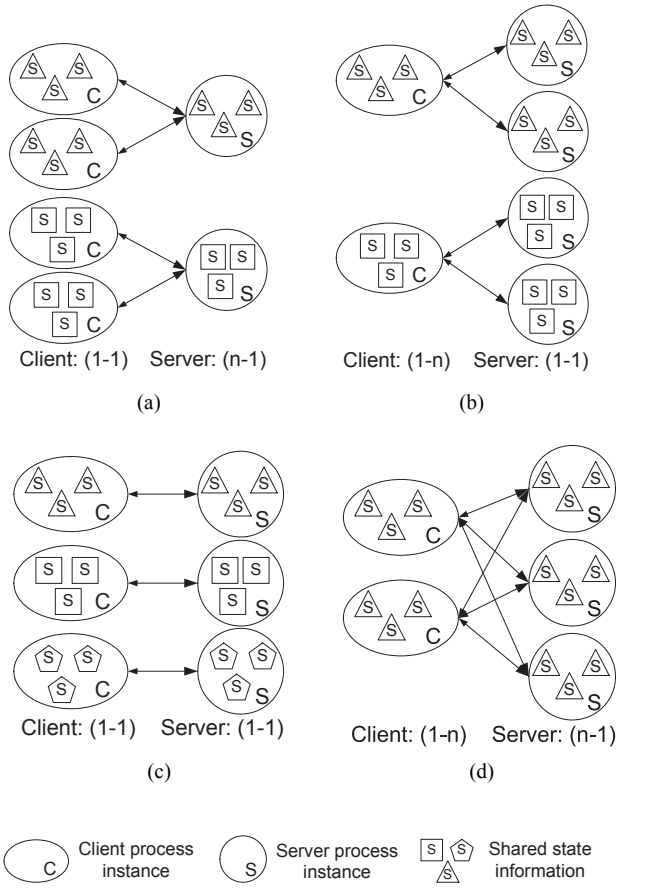
2.1 Process state types

At runtime, state information is propagated and shared between multiple process instances. Each process instance synchronises its state with partner process instances via messages. Thus, state information is 'shared' implicitly between multiple process instances. How state information is shared (Atkinson and Bostan, 2009) depends on the service interaction patterns (Barros et al., 2005) 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 client instances (n-1). From a global point of view, we take the combination types as shown in Figure 3, and visualised in Figure 4. In Figure 4(a), the state information is 'shared' between clients. One client instance interacts with one server instance (1-1), while globally one server instance interact with multiple client instances (n-1). 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 Figure 4(b), the state information is shared between 'multiple' server instances, but 'private' to each client instance. Each client instance interacts with multiple server instances (1-n). Each server instance interacts with multiple client instances (n-1). In Figure 4(c), the state information is *private* to the requester-responder pair. Each initiator process instance is dedicated to synchronise its state with a single responder instance (1-1). In Figure 4(d), the state information is *shared* between all instances. Each client instance interacts with multiple server instances (1-n). Each server instance interacts with one client instance (1-1). We name this state information type *multiple, private*.

Figure 3 State information types

	Client	*Server*	*Shared state types*	*Name*
a)	1-1	n-1	C^n – S	Shared, static
b)	1-n	n-1	C^m – S^n	Shared
c)	1-1	1-1	C – S	Private
d)	1-n	1-1	C – S^n	Multiple, private

Figure 4 Multiple processes instances shared state types, (a) shared static(b) multiple private (c) private (d) shared (e) legend



2.2 Process synchronisation failure analysis

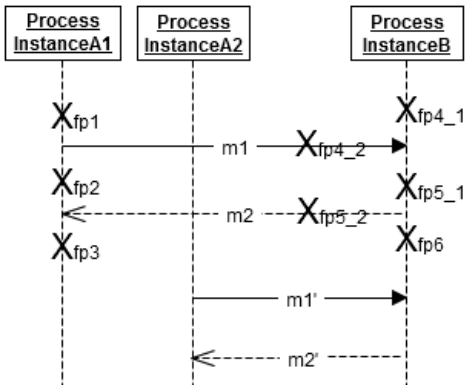
We consider the failure scheme shown in Figure 5 (Tanenbaum and van Steen, 2002). With regards to client/server interactions with system crashes and network failures, we focus on 'crash failure', 'omission failure' and 'timing failure'. 'Arbitrary failure' (also called 'Byzatine failure') is more like a security issue. 'Response failure' is triggered by the flaw of the process design. 'Arbitrary failure' and 'Response failure' are out of the scope of this work. In our previous work (Wang et al., 2012a, 2012b), the state synchronisation failure of state type *private* [Figure 4(c)] has been considered. In this paper, we propose a solution for the synchronisation failure of state type *shared static* [Figure 4(a)]. This subsection analyses possible synchronisation failures for the state information type *shared static*.

Figure 5 Failure scheme

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

The UML sequence diagram of the synchronisation for the state type *shared, static* is presented in Figure 6. Multiple initiator process instances (*A1*, *A2*) synchronise with the responder process instance *B*. The possible failure points for a synchronisation between *A1* and *B* are marked as $X_{fp1} \sim X_{fp6}$. Failure X_{fp1} does not affect the partner process because the initiator process (*A1*) has not started the synchronisation. Failures X_{fp3} and X_{fp6} do not affect the synchronisation because the failure occurs when the synchronisation is finished. The failure points X_{fp4} and X_{fp5} are regarded as *service unavailable failure* and *pending response failure* in our previous work (Wang et al., 2012a, 2012b) respectively. The service unavailable failure happens when the network fails to deliver the request message $m1$ (failure point X_{fp4_2}) or the ProcessInstanceB crashes before the message $m1$'s delivery (failure point X_{fp4_1}). The pending response failure happens when the ProcessInstanceB crashes before the delivery of response message (failure point X_{fp5_1}) or the network fails to deliver the response message (failure point X_{fp5_2}). These type of synchronisation failures can be solved using the synchronisation mechanisms proposed in Wang et al. (2012a, 2012b). In this work we focus on failure point X_{fp2}. If *A1* fails after sending *m1*, this is an omission failure because *m2* cannot be received by *A1*. If *A1* restores and re-sends *m1*, the processes will not synchronise, since the interaction between *A2* and *B* has already changed the state of *B*. This failure is referred in this paper as *pending request failure*.

Figure 6 Synchronisation failure analysis for *shared, static* state type



3 Modelling: business processes to Petri nets

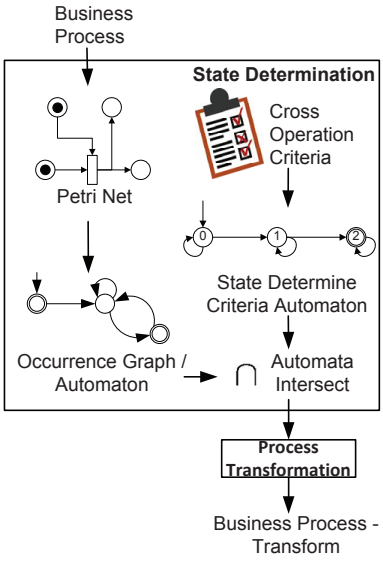
Collaborative business processes are autonomous, and furthermore it is difficult to change the communication protocol to solve the process synchronisation problem. Therefore we transform the business process to make it capable to synchronise its state on the notice of partner process failures. The transformation is done in such a way that in the resulting processes possible state inconsistencies are recognised and compensated by state-caching, and these processes retry failed interactions based on the contents of the cache.

An overview of our solution to failure is shown in Figure 7. Given a business process, we infer state change for all synchronous process operations. We model the business process as a Petri net and also generate the occurrence graph/automaton models. By applying proposed criteria to the Petri nets and occurrence graph/automaton models, we detect whether a sta te change happens. For all process operations that change the process state, we modify the original behaviour of the process in the following way:

1 for a new request coming from the client, the server caches and replies the response message

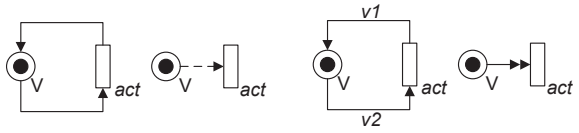
2 for the same synchronisation request sent multiple times from the same client (which implies that a client failure happened), the server process replies with the cached response.

Figure 7 Solution overview (see online version for colours)



We formalise WS-BPEL processes as Petri nets in which the dataflow is also annotated. 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. We use Petri nets to describe the underlying semantics of WS-BPEL and use them as a basis for our state determination criteria. 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, Ouyang et al. (2007) focuses on control flow modelling thus state information is implicit. Stahl (2005), Hinz et al. (2005) and Lohmann (2008) address activity stops and correlation errors, which are not relevant and therefore unnecessarily complicate our formalism. Thus, we propose a simplified Petri nets representation, in which 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 the structure of other activities, which is the semantics of WS-BPEL structured activities.

Figure 8 Convention for reading and writing of BPEL process variables, (a) read (b) write



(a) (b)

This Petri nets model is Petri net WS-BPEL processes, but its purpose is to allow the inference of data dependencies. Based on the Petri nets model generated, we will specify rules to identify data flows, based on which the criteria of determining state to be cached will be applied. In order to improve readability, we use the two conventional notations to denote the reading or writing of process variables by activities. As shown in Figure 8(a), the Petri net representation of an activity reading a process variable V is that the transition takes a token from the place that represents the variable and then puts a token back. We use dashed arrows as a graphical notition for this. As shown in Figure 8(b), 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 that represents the variable and then puts another token $v2$ into it. We use double arrows as a graphical for this. 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.

3.1 Basic activities

Figure 9(a) shows the Petri net of a *receive* activity, where places *c1* and *c2* are the input and output control places, respectively. In order to express the *receive* semantics of WS-BPEL, the transition takes a token out from the *msg* place and 'writes' to the place *v1*. Similarly, we have modelled basic activities *reply*, *assign*, and *invoke* as shown in Figure 9(b) to 9(d), respectively.

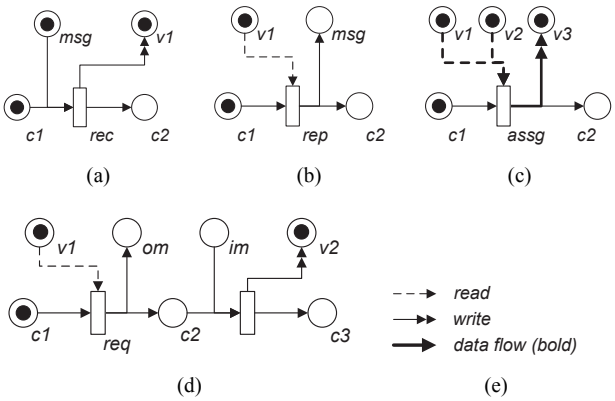
We denote data flow as a subset of the arcs annotated in bold. The data flow of the *assignment* activity [Figure 9(c), denoted as bold arcs] is from place *v1* (and *v2*) to the transition *assg*, then to the place *v3*.

3.2 Structured activities

The Petri net of an *if* activity is presented in Figure 10, where places $c1$ to $c6$ model the control flow. In WS-BPEL, the condition of an *if* activity is an expression, such as $\$v1 < \$v2$. The process variables that appear in the condition expression are modelled as places p_v1, p_v2 in our Petri nets. The positive (negative) evaluation of the condition results in the execution of the *true* (*false*) branch of the WS-BPEL process, which is modelled as a hierarchical transition *body_true* (*body_false*), and is initialised 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 the place *in_true* (*in_false*) represents that the modelled WS-BPEL executes the *true* (*false*) branch. We name this place *control boundary indication place*.

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 a 'read' of the control boundary indication place by the assignment transition that is hierarchically nested in the *if* construct. This is illustrated in Figure 11, which shows a true branch of an *if* activity. The transition *assg* is the Petri net representation of an assignment activity. The data flow generated by *if* activity model is the path from conditional variable p_v1 (and p_v2) to the transition *cond_true*, and then from the transition *cond_true* to the control boundary indicator place *in_true*. The data flow model generated for the assignment activity is from the places p_v4 and p_v5 (representing the process variables $v4$ and $v5$ which appear in the R-value part of the assignment) to the transition *assg*, and then from *assg* to the place p_v3 (representing the process variable $v3$ which appears in the L-value part of the assignment). By the application of the rule, we add a 'read' of the indicator place *in_true* by the transition *assg*, so that the data dependency path representing that *v3* depends on *v1* and *v2* can be generated.

Figure 9 The Petri net model for basic activities, (a) receive (b) reply (c) assign (d) invoke (e) legend



(a) (b) (c) (d) (e)

Figure 10 The Petri net model for *if* activity

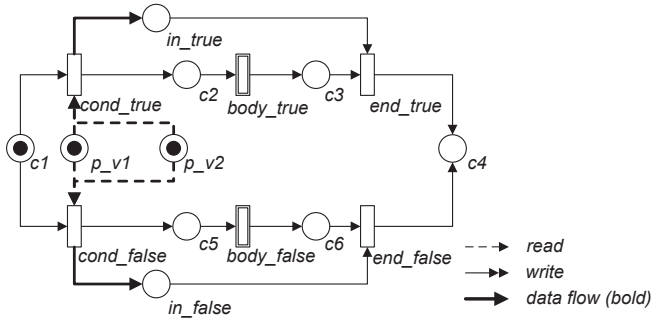
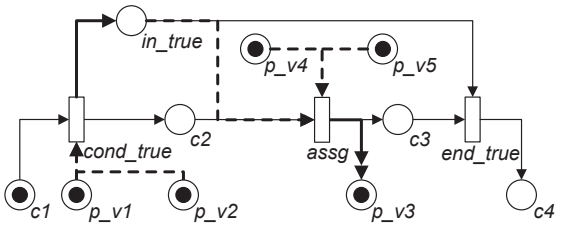
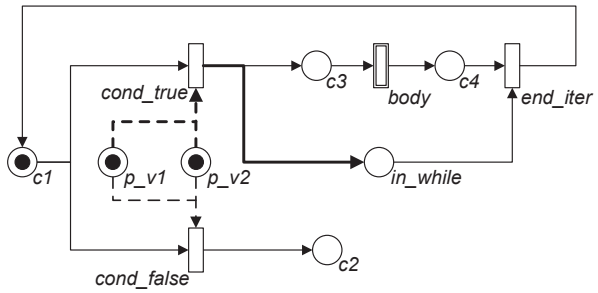


Figure 11 The data flow path of *if* activity



The Petri net model for a *while* activity is shown in Figure 12. Places $c1$ to $c4$ model the control flow. The variables ($v1$ and $v2$) which appear in the *while* conditional expression are modelled as places p_v1 and p_v2. At runtime, the evaluation result of the conditional expression will determine whether the $body$ of the $while$ iteration is executed or not. This is modelled as the transitions $cond\_true$ and $cond\_false$. These transitions 'read' the places p_v1 and p_v2.

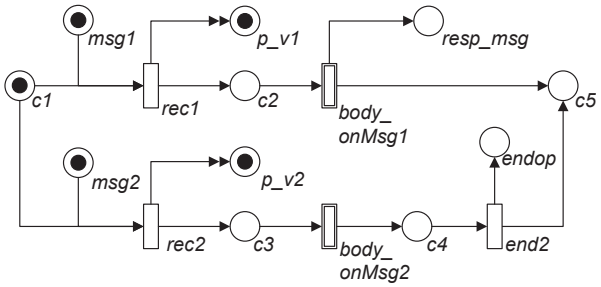
Figure 12 The Petri net model for *while* activity



The process variables could be changed inside the $while$ iteration, so a data dependency should be generated to indicate that these variables depend on the variables which occur in the $while$ condition ($v1$ and $v2$). This mechanism is similar with the if model, in which we use the place in_while to indicate that the WS-BPEL execution is inside the while iteration. This place works together with Petri net model of the $assignment$ activity to generate this dependency.

The Petri net model for a $pick$ activity is shown in Figure 13. Places $c1$ to $c5$ model the control flow of the *pick* activity.r Transitions $rec1$ and $rec2$ (could be more) model the receiving behaviour of each <*onMessage*> branch. Hierarchical transitions $body\_onMsg1$ and $body\_onMsg2$ model each of the $onMessage$ branches of the pick activity. If there is any $reply$ activity corresponding to the $onMessage$ branch, the output message corresponds to the $resp\_msg$. If there is no $reply$ for the $onMessage$ (the message $msg2$ is a one-way message without corresponding reply), we use the transition $end2$ to model the end of this branch to facilitate simulation. The timer-based event is not supported in our current version of the $pick$ model.

Figure 13 The Petri net model for *pick* activity



The Petri net specified in this section is used for state determination in Section 4. The idea of modelling system crash (network failure) is to use a transition which takes a token out from places modelling control flow (message channel) and puts a tolen into a corresponding place which represents failure. The Petri net that models failure is presented in Section 6.

4 State determination criteria

4.1 Inbound message activity

In order to identify the synchronous operation boundaries, we introduce the concept of Inbound Message Activity (IMA) in WS-BPEL. IMAs are activities in which messages are received from partners, and consists of the activities *receive* and *pick*. Other types of IMAs, like $eventhandlers$, are out scope of this paper. The control boundary of a synchronous process operation starts with an IMA and ends with a *reply* activity.

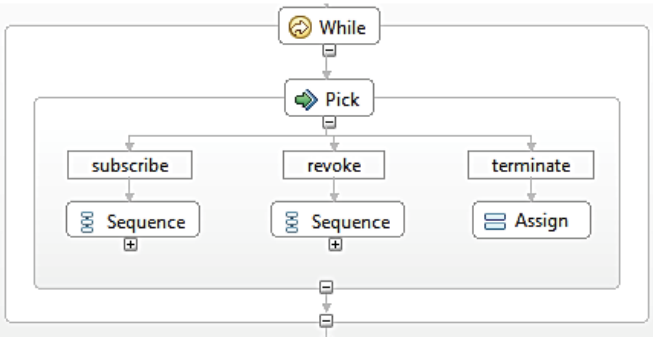
OMAs (outbound message activities) reply the response message, and consist of the activities *invoke* and *reply*.

IMAs and OMAs correspond to the begin and end of the control boundary of a synchronous process operation, respectively. If a state variable is identified for a synchronous process operation, we cache the response message. We will use a ticket subscribing process to illustrate our criteria to identify process state variables. As shown in Figure 14, 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.

4.2 Inside process operation criteria

Below we discuss the criteria used inside the control boundary of a process operation.

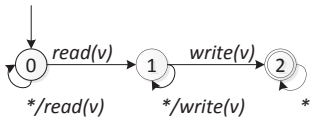
Figure 14 Snippet of ticket process (see online version for colours)



4.2.1 *Read before write*

The process variable should be read first and written afterwards. Formally, in Figure 15, 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 denotes the state in which the process variable v is read but not being written, and State 2 is the accepted state, which represents that variable v is read first and written afterwards.

Figure 15 Criterion automaton of read before write



We discuss the use of the criteria automaton to check the Petri net model in Section 5.

4.2.2 *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 Figure 16. 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 are considered as state variables.

4.3 *Cross-process operation criteria*

If a variable *v* has its value written inside an operation and read outside the operation afterwards, *v* should 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 $\{q_0, Q, F, \sum, \delta\}$, with the alphabet $\sum = \{IMA\_subscribe, OMA\_subscribe, r\_history, w\_history\}$ 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 $\{4\}$. Figure 17 shows the automaton constructured in this way. The transition function δ is specified as follows:

1 From state 0: IMA_subscribe leads to state 1; Stay in state 0 otherwise.

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

3 From state 2: OMA_subscribe leads to state 3; Stay in state 2 otherwise.

4 From state 3: w_history leads to state 0; r_history leads to state 4. Stay in state 3 otherwise.

5 From state 4: Stay in state 4 for any element of $\sum$.

5 Implementation details

The architecture of our prototype implementation is shown in Figure 18. We implemented the state determination criteria proposed in Section 4 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.

5.1 *State dependency analysis module*

At the bottom layer is the CPN simulation module and the Automaton Class Library of our architecture in Figure 18. 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 : Q ⇐ Empty,
2  add init marking m0 to Graph : G
3  Enqueue(Q, m0)
4  while(Q is not empty) do
5    marking u ⇐ Dequeue(Q)
6    for(each v in directly reachable markings
       from u) do
7      if(v is not in G) then
8        Enqueue(v, G)
9        add v to G
10         add < u, v > to G
```

Figure 16 Petri net of subscribe operation of the ticket process

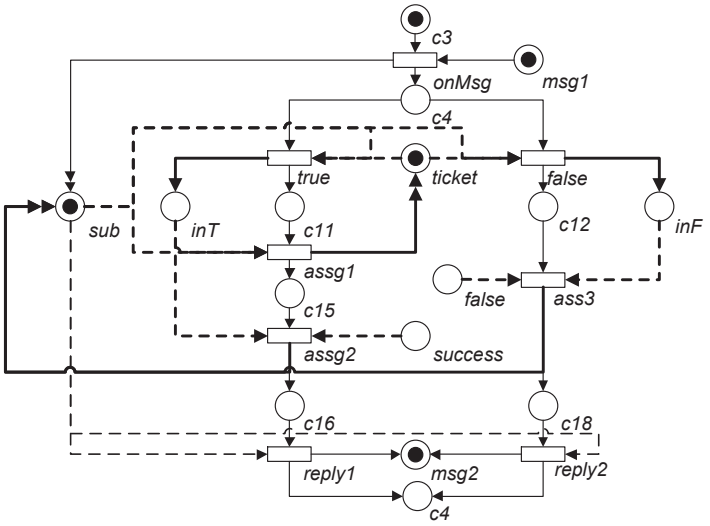


Figure 17 Automaton model of cross process operation criteria

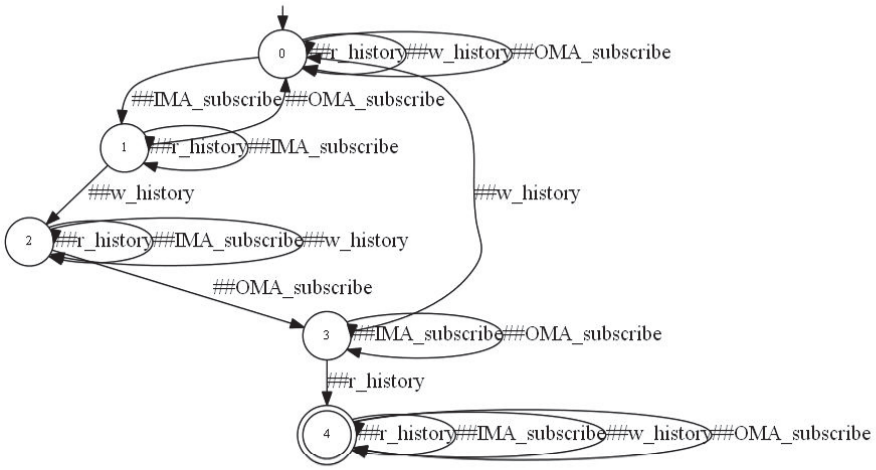
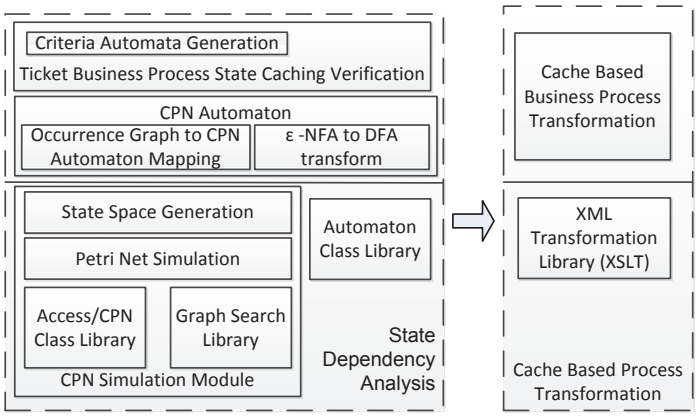


Figure 18 Architecture of prototype implementation



In the middle layer, the occurrence graph is mapped to the automaton. Figure 19 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 additional alphabet Σ' is required as input. If the transition name is in Σ', the Petri net transition is mapped to the corresponding automaton transition. If not, the Petri net transition is mapped to an ϵ automata transition. We then transform the non-deterministic finite automaton (NFA) containing the ϵ to a Deterministic Finite Automaton. Finally, we calculate the intersection of the DFA with the criteria automata in order to determine the necessary state information.

Figure 19 A mapping from occurrence graph to automaton (see online version for colours)

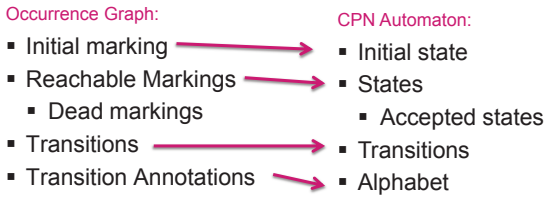
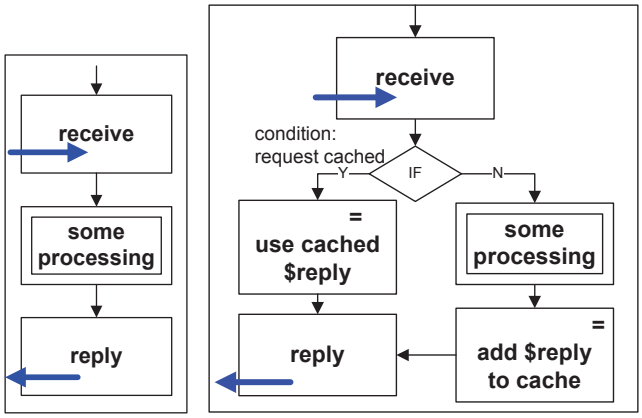


Figure 20 Cache-pased process transformation details (a) original (b) transformed (see online version for colours)



(a) (b)

5.2 *Process transformation module*

As shown in Figure 20(a), a synchronous operation receives a message, performs some processing and then replies. Our transformation replaces the processing and *reply* by an *if* activity, where the condition of the *if* activity checks whether the request message has been cached before. If the message is cached, the process uses the cached response as reply. If the message is not cached, which implies that the message was 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 <request, response> 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:

1 given a request message, check whether the corresponding response message is cached

2 given a request, get the corresponding response

3 given a value pair of request and corresponding response messages, add it to the cache.

The cache data operation is implemented as a XSLT transformation. 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:doXslTransform(
      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 doXslTransform() with the request message and $\$cache$ as its parameters. Variable $\$foundCachedReques$ contains the result.

6 Evaluation

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

6.1 *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 business process [in Figure 20(b)] to extract the automata model. We demonstrate correctness by showing that the automata model of the business process is subsumed by the criteria automata.

6.1.1 *Correctness criteria*

Informally, for any message M1, it could be sent multiple times due to failures, and ultimately received. We use the DFA $< Q, \Sigma, \delta, q_0, F >$ to formalise this correctness criteria. The global message sending and receiving states are modelled as the state set $Q = \{0, 1, 2\}$. The alphabet $\Sigma = \{sendM1, receiveM1\}$. $SendM1$ $(receiveM1)$ models the sending (receiving) of message $M1$. $q_0 = 0$ is the initial state and $F = \{0, 2\}$ is the set of accepted states. The transition rules are visualised in Figure 21(a). 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 be received.

The synchronous communication criteria should take into consideration both request and response messages. Informally,

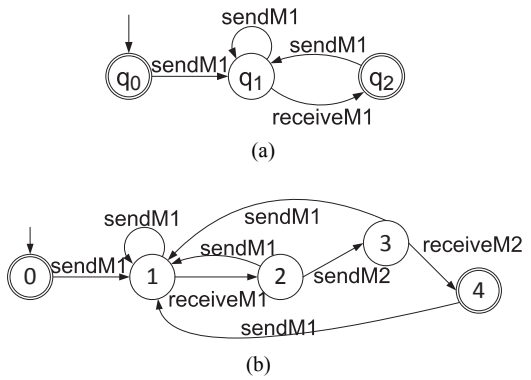
1 a request may be sent multiple times until received

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.

The criteria automatia is visulised in Figure 21(b). The state set is $\{Q = 0, 1, 2, 3, 4\}$. State 0 is initial state, States 1 and 2 represent that the request message $M1$ is sent and received, respectively, and States 3 and 4 represent that the response message $M2$ is sent and received, respectively. The alphabet $\Sigma = \{sendM1, receiveM1, sendM2, receiveM2\}$ models the behaviour of $sending$ and $receiving$ of request message $M1$ and response message $M2$. The transition rules are specified as the following. At state 0, transition $sendM1$ to state 1 represents that the request message $M1$ is sent. At state 1, transition $receiveM1$ to state 2 represents that the message $M1$ has been received. At state 2, transition $sendM2$ to state 3 represents that the response message $M2$ has been sent. At state 3, transition $receiveM2$ to state 4 represents that the response message $M2$ has been received. At states $1, 2, 3$, transition $sendM1$ to state 1 represents that the request message $M1$ can be sent multiple times due to synchronisation failure. At state 4, transition $sendM1$ to state 1 represents that the request message $M1$ can be sent multiples times due to a specific process design, for example, an $invoke$ activity nested inside a $while$ iteration.

Figure 21 Correctness criteria, (a) single message (b) synchronous request and response



6.1.2 Evaluation procedure

Figure 22 shows the correctness evaluation in three steps. First we prove that a business process can pass the correctness criteria when no failure 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.

In each step of evaluation, we start from modelling the business process as a Petri net. Then the Petri net is transformed into occurrence graph (automata model). Finally, we prove the correctness by proving that the automata of the business process is subsumed by the criteria automata.

The Petri net snippet of the ticket subscription operation is shown as Figure 23. Places $c1$ to $c3$ model the control flow of the initiator process, and $c4$ to $c7$ model the control flow of the responder process. The sending and receiving of the $order$ message are modelled by the transitions $send1$ and $receive1$, respectively. The sending and receiving of the $result$ message are modelled by the transitions $send2$ and $receive2$, respectively. The transtions $channel1$ and $channel2$ model the communication channel.

A system crash or network failure is modelled with a transition which takes a token out from places modelling control flow or messages, and puts a tolen into corresponding place which represents failure. The Petri net model of the synchronisation failure is shown as Figure 24. The *Service Unavailable* failure caused by a system crash is represented by the firing of transition *SU1* or *SU2*, thus a token is put into place *SU_SC* to indicate the failure state. The *Service Unavailable* failure caused by network failure is represented by the firing transition *SU3*, thus a token is put into place *SU_NF* to indicate the failure state. The *Pending Request* failure is modelled by transition *REQ*. The *Pending Response* failure caused by a system crash is modelled by the firing of transition *RESP1* or *RESP2*. The *Pending Response* failure caused by network failure is modelled by the firing of transition *RESP3* or *RESP4*. The places *RESP_SC* and *RESP_NF* are used to represent the corresponding failure states.

The Petri net model of our process that is able to recover from pending request failures is shown in Figure 25. On the initiator side, the message re-send behaviour is modelled as transition *reSend* to take a token from REQ and put a token in place $c1$. On the responder side, after the $order$ message has been received, we model our transformation as the Petri net 'between' $c5$ and $c6$. The firing of transition *ifCached* indicates that the request order message is cached. The $assign$ activity which assigns a cached response to the variable $result$ is modelled as transition *getCache*. The case in which the $order$ message is not cached is modelled as the transition *ifNotCached*. In this case, we model the behaviour of processing the order message.

We finish the correctness proof by checking that the occurrence graph (automaton) model of the transformed process is subsumed by the correctness criteria automata. The subsumption checking algorithm (Hopcroft et al., 2006) is implemented as a programme to check correctness.

6.2 Performance overhead evaluation

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

Figure 22 Evaluation procedure (see online version for colours)

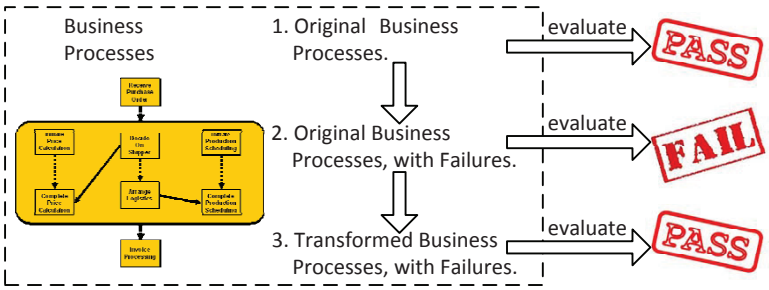


Figure 23 Petri net model of ticket subscription operation

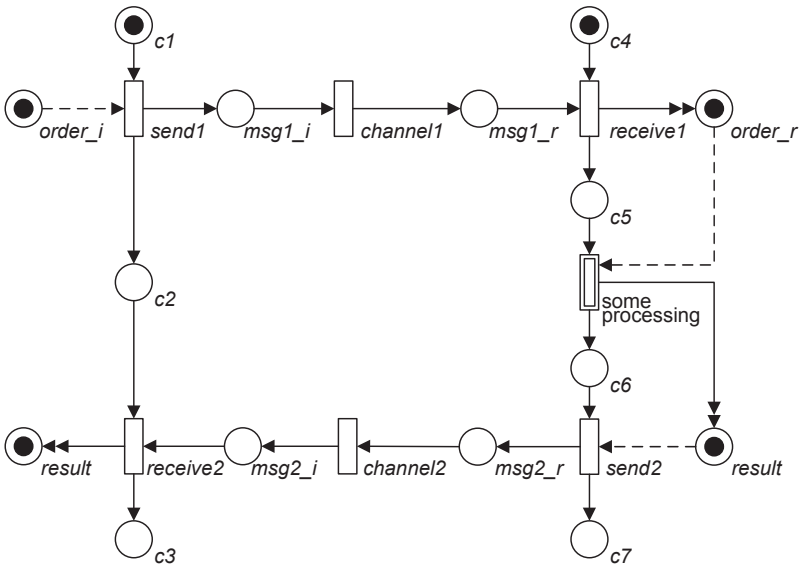


Figure 24 Synchronisation failure model

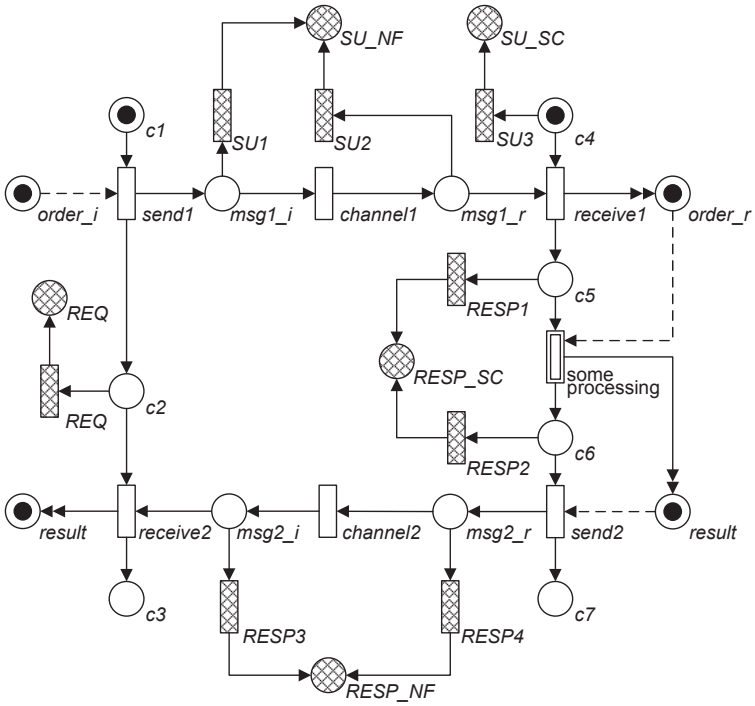
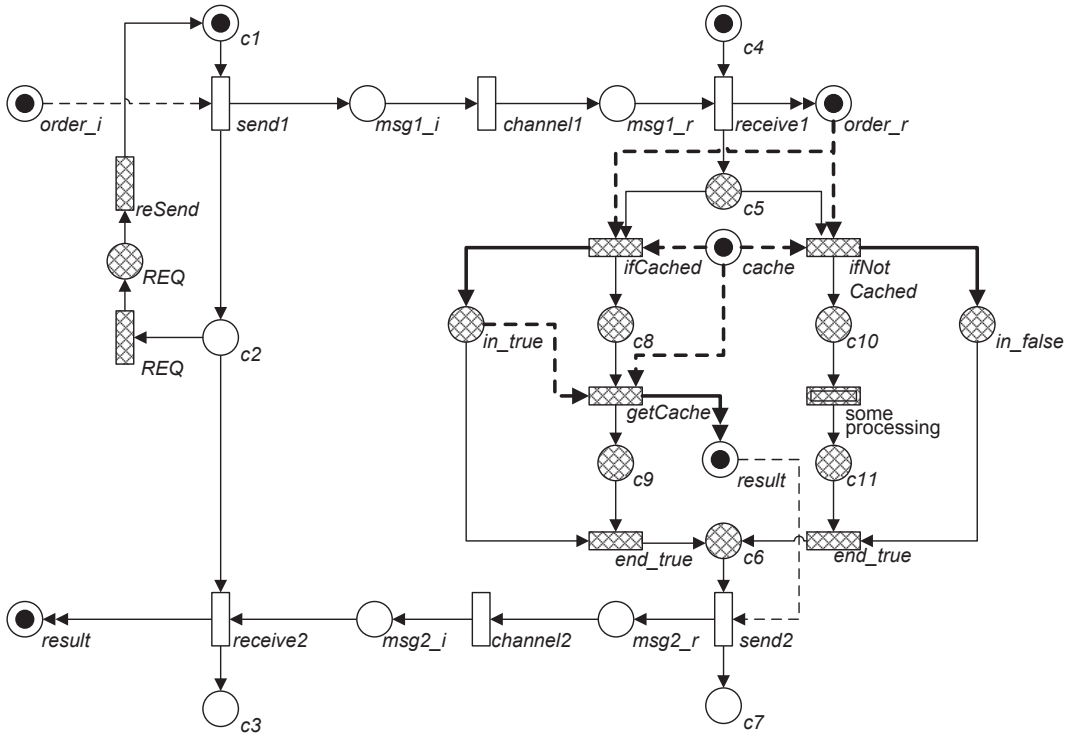


Figure 25 Transformation to recover from pending request failure



We evaluated the performance overhead of our solution with different workloads. The requests sent per minute by the simulation client comply to a Possion distribution. We collect performance under two workloads, namely $\lambda = 5$ and $\lambda = 10$. We use these workloads because according to our tests under the available hardware and software configurations, higher workload exhausts the server resources. Each test run lasted for 60 minutes, but only the response time in the 30 minutes in the middle of this period have been considered (steady state).

Figure 26 Average process response time different workload

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

Under the workload of $\lambda = 5$, the performance overhead of our transformation mechanism is 62 ms. Under the workload of $\lambda = 10$, the performance overhead of our transformation mechanism is 184 ms. 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.

6.3 Process design complexity

We have implemented the process designed in Figure 20 using WS-BPEL. The synchronous interaction is presented with two activities (one *receive* and one *reply*). By applying our process transformation mechanism, the resulting process is as follows:

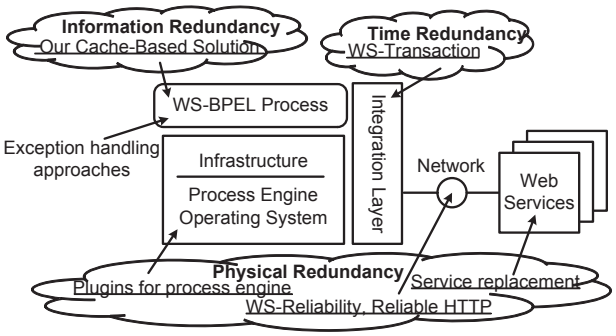
```
<receive ...>
<assign name="assg1" ... />
<if ...>
  <condition .../>
  <assign name="assg2" .../>
  <else>
    <!-- Some Processing -->
    <assign name="assg3" .../>
  </else>
</if>
<reply .../>
```

We add one structured activity and three basic activities. One assignment activity is used to check whether a request message was cached or not, while second assignment is used to get the cached response message and add it to the cache. The third activity 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.

7 Related work

Fault handling approaches (Russell et al., 2006; Lerner et al., 2010) require that the process designers are aware of possible failures and their recovery strategies. Alternatively, cache-based process transformations can be transparent to process designers. As described in Tanenbaumand van Steen (2002), the key technique for masking faults is to use redundancy. As shown in Figure 27, three kinds of redundancy are possible: information redundancy, time redundancy and physical redundancy.

Figure 27 Overview of related solutions



Physical redundancy-based solutions include Modafferi et al. (2006), Modafferi and Conforti (2006), Charfi et al. (2009), Fredj et al. (2008) and Cavallaro et al. (2009). Recovery mechanisms implemented as plug-ins for a WS-BPEL engine, such as Modafferi et al. (2006), Modafferi and Conforti (2006) and Charfi et al. (2009), strongly depend on a specific WS-BPEL engine. The approach to recovery presented in Fredj et al. (2008) and Cavallaro et al. (2009) consists of substituting a service with another one dynamically if a synchronisation error occurs. In Moser et al. (2008) and Moo-Mena et al. (2008, 2009), the QoS aspects of dynamic service substitution are considered. An alternative to avoid the loss of state synchronisation is to use reliable messaging. Message exchange is realised 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 (Todd et al., 2005), WS-RX (2009) 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 outsourced deployments where the infrastructure layer is out of control of the process designer. For example, in some cloud computing environments, user-specific network configuration capabilities to enhance state synchronisation are 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 synchronisation-enabled counterparts via process transformations.

Information-based redundancy is achieved based on replication. Our solution can be classified in this category since caching is a kind of replication. Time-based redundancy solutions include WS-Transactions. Transaction-based process recovery approaches, such as in the WS-AT (WSTX, 2009a) and WS-BA (WS-TX, 2009b) standards, require a central coordinator, in contrast with our approach, which is based on process transformations.

8 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 type. The challenge is how to cache process interaction messages in order to recover from failures. 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 implemented a prototype to desmonstrate our approach. As a next step, we will extend our work to include other types of shared states, particularly involving multiple process instances.

References

Apache Software Foundation (2011) *Apache ODE, the Orchestration Director Engine*, 1.3.5 edition [online] http://ode.apache.org/index.html.

Atkinson, C. and Bostan, P. (2009) 'Towards a client-oriented model of types and states in service-oriented development', in *13th IEEE Intl. EDOC Conference*, September, pp.119–127.

Barros, A., Dumas, M. and Hofstede, A.H.M. (2005) 'Service interaction patterns', in *Business Process Management*, Vol. 3649 of *Lecture Notes in Computer Science*, pp.302–318, Springer, Berlin, Heidelberg, September.

Cavallaro, L., Nitto, E. and Pradella, M. (2009) 'An automatic approach to enable replacement of conversational services', in *Service-Oriented Computing*, Vol. 5900, pp.159–174, Springer, Berlin, Heidelberg.

Charfi, A., Dinkelaker, T. and Mezini, M. (2009) 'A plug-in architecture for self-adaptive web service compositions', in *IEEE International Conference on Web Services*, July, pp.35–42.

CPN Tools (2012) *CPN Tools*, 3.4.0 edition, AIS Gruop, Eindhoven University of Technology, The Netherlands [online] http://www.cpntools.org.

Fredj, M., Georgantas, N., Issarny, V. and Zarras, A. (2008) 'Dynamic service substitution in service-oriented architectures', in *IEEE Congress on Services – Part I*, July, pp.101–104.

Hinz, S., Schmidt, K. and Stahl, C. (2005) 'Transforming bpel to petri nets', in *Business Process Management*, Vol. 3649 of *Lecture Notes in Computer Science*, pp.220–235, Springer, Berlin, Heidelberg.

Hopcroft, J.E., Motwani, R. and Ullman, J.D. (2006) *Introduction to Automata Theory, Languages, and Computation*, 3rd ed., July, Addison Wesley, Boston, MA, USA.

Lerner, B.S., Christov, S., Osterweil, L.J., Bendraou, R., Kannengiesser, U. and Wise, A.E. (2010) 'Exception handling patterns for process modeling', *IEEE Transactions on Software Engineering*, Vol. 36, No. 2, pp.162–183.

Lohmann, N. (2008) 'A feature-complete petri net semantics for ws-bpel 2.0', in Dumas, M. and Heckel, R. (Eds.): *Web Ser- vices and Formal Methods*, Vol. 4937 of *Lecture Notes in Computer Science*, pp.77–91, Springer, Berlin, Heidelberg.

Modafferi, S. and Conforti, E. (2006) 'Methods for enabling recovery actions in ws-bpel', in *On the Move to Meaning- ful Internet Systems: CoopIS, DOA, GADA, and ODBASE*, Vol. 4275 of *Lecture Notes in Computer Science*, pp.219–236, Springer, Berlin, Heidelberg.

Modafferi, S., Mussi, E. and Pernici, B. (2006) 'SH-BPEL: a self-healing plug-in for WS-BPEL engines', in *1st Workshop on Middleware for Service Oriented Computing, WM4SOC*, pp.48–53, ACM, NY, USA.

Moo-Mena, F., Garcilazo-Ortiz, J., Basto-Diaz, L., Curi-Quintal, F. and Alonzo-Canul, F. (2008) 'Defining a self-healing QoS-based infrastructure for web services applications', in *11th IEEE Intl. Conf. on Computational Science and Engineering Workshops*, July, pp.215–220.

Moo-Mena, F., Garcilazo-Ortiz, J., Basto-Diaz, L., Curi-Quintal, F., Medina-Peralta, S. and Alonzo-Canul, F. (2009) '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*, October, pp.163–169.

Moser, O., Rosenberg, F. and Dustdar, S. (2008) 'Non-intrusive monitoring and service adaptation for WS-BPEL', in *Proceedings of the 17th Intl. Conf. on World Wide Web, WWW*, pp.815–824, ACM, NY, USA.

OASIS Web Services Reliable Exchange (WS-RX) TC (2009) *Web Services Reliable Messaging (WS-Reliable Messaging*, February, OASIS [online] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-rx.

OASIS Web Services Transaction (WS-TX) TC (2009a) *Web Services Atomic Transaction (WS-Atomic Transaction)*, February, OASIS [online] http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec.html.

OASIS Web Services Transaction (WS-TX) TC (2009b) *Web Services Business Activity (WS-Business Activity)*, February, OASIS [online] http://docs.oasis-open.org/ws-tx/wsba/2006/06.

Oracle Fusion Middleware (2011) *Oracle SOA Suite*, 11gr1 (11.1.1.5.0) edition [online] http://www.oracle.com/technetwork/middleware/soasuite/overview/index.html.

Ouyang, C., Verbeek, E., van der Aalst, W.M., Breutel, S., Dumas, M. and ter Hofstede, A.H. (2007) 'Formal semantics and analysis of control flow in ws-bpel', *Science of Computer Programming*, Vol. 67, Nos. 2–3, pp.162–198.

Russell, N., Aalst, W. and Hofstede, A. (2006) 'Workflow exception patterns', in *Advanced Information Systems Engineering*, Vol. 4001 of *Lecture Notes in Computer Science*, pp.288–302, Springer, Berlin, Heidelberg.

Tanenbaum, A.S. and van Steen, M. (2002) *Distributed Systems: Principles and Paradigms*, 1st ed., Chapter 7, pp.366–368, Prentice Hall, New Jersey, NJ, USA.

Todd, S., Parr, F. and Conner, M. (2005) *A Primer for HTTPR.*, IBM, March.

Wang, L., Wombacher, A., Pires, L.F., van Sinderen, M.J. and Chi, C. (2012a) 'An illustrative recovery approach for stateful interaction failure of orchestrated processes', in *IEEE 16th International Enterprise Distributed Object Computing Conference Workshops (EDOCW)*, pp.38–41, September.

Wang, L., Wombacher, A., Pires, L.F., van Sinderen, M.J. and Chi, C. (2012b) 'A state synchronization mechanism for orchestrated processes', in *IEEE 16th Intl. EDOC Conference Workshops (EDOCW)*, pp.51–60, September.

Wang, L., Wombacher, A., Pires, L.F., van Sinderen, M.J. and Chi, C. (2013) 'Robust client/server shared state interactions of collaborative process with system crash and network failures', in *10th IEEE Intl. Conf. on Services Computing (SCC)*.

WS-BPEL (2007) *Web Services Business Process Execution Language*, 2nd ed., OASIS [online] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.