Graceful Interruption of Request-Response Service Interactions*

Mila Dalla Preda, Maurizio Gabbrielli, Ivan Lanese,
Jacopo Mauro, and Gianluigi Zavattaro

Lab. Focus, Department of Computer Science/INRIA, University of Bologna, Italy
{dallapre,gabbri,lanese,jmauro,zavattar}@cs.unibo.it

Abstract. Bi-directional request-response interaction is a standard communication pattern in Service Oriented Computing (SOC). Such a pattern should be interrupted in case of faults. In the literature, different approaches have been considered: WS-BPEL discards the response, while Jolie waits for it in order to allow the fault handler to appropriately close the conversation with the remote service. We investigate an intermediate approach in which it is not necessary for the fault handler to wait for the response, but it is still possible on response arrival to gracefully close the conversation with the remote service.

1 Introduction

Service-oriented computing (SOC) is a programming paradigm based on the composition of services, computational entities available on the net. According to WSDL [9], the standard for describing web service interfaces, services can be invoked according to two main modalities: one-way and request-response. In one-way communication a message is sent to a remote service. In request-response communication a message is sent and an answer is waited for before continuing the computation.

Interaction with remote services may incur in errors of different kinds: the remote service may disconnect, messages may be lost, or a client may interrupt the interaction with a remote service exactly in between the request and the corresponding response. To avoid that such an error causes the failure of the whole application, error handling techniques have been developed. They are commonly based on the concept of fault handler and compensation. A fault handler is a piece of code devoted to take the application to a consistent state after a fault has been caught. A compensation is a piece of code devoted to undoing the effect of a previous activity because of a later error.

As an example, consider a hotel reservation service. A reservation can be canceled, but if it is not annulled the cost of one night will be charged in case of no show. If the trip has to be annulled, the compensation for the hotel reservation has to be executed, thus canceling the reservation and avoiding the cost of a no show.

Jolie [3] is a language for programming service-oriented applications. Jolie request-response invocation establishes a strong connection between the caller and callee, thus it should not be disrupted by faults. To this end, callee faults are notified to the caller that can thus manage them. Symmetrically, in case of caller faults the answer from the

* Partly funded by the projects EU FP7-231620 HATS and ANR-2010-SEGI-013 AEOLUS.

G. Kappel, Z. Maamar, H.R. Motahari-Nezhad (Eds.): ICSOC 2011, LNCS 7084, pp. 590–600, 2011.

callee is waited for and used during recovery. This allows, in particular, to compensate successful remote activities which are no more needed because of the local fault. This is the case of the hotel reservation above.

WS-BPEL [8], a main standard in the field, has a different approach: in case of caller faults execution can continue without waiting for the response, and the response is discarded upon arrival. In particular, it is not possible to write code that will be executed upon receipt of the response. The Jolie approach allows for programming safer applications. The fact that the request-response pattern is not disrupted by errors has been proved in [3], by relying on SOCK [4,2], a calculus defining the formal semantics of Jolie. A nasty side effect of the Jolie approach is that the client has to wait for answers of request-response invocations before proceeding in its execution. This slows down the caller execution. For instance, referring to the hotel reservation example, the client cannot continue its operations before the answer from the hotel has been received and (s)he gets stuck whenever the answer is lost. This drawback is unacceptable for programming applications over the net. Such a kind of problem is normally solved using timeouts, but they are not available in Jolie. Also, they are not easy to mimic.

We propose here a new approach to error handling in Jolie, allowing on one side to compensate undesired remote side effects, and ensuring on the other side that local computation is not slowed down in case of late answers. In particular, this new approach allows to easily program timeouts. We also extend the approach to deal with concurrent invocations of multiple services, as needed for implementing speculative parallelism.

2 SOCK

We first introduce SOCK [4], the calculus that defines the semantics of Jolie [3] programs, and then we extend it to account for request-response and multiple request-response service invocations. SOCK is suitable for illustrating our approach since it has a formal SOS semantics, it provides request-response as a native operator, and it has a refined approach to error handling.

In the following we present the three layers in which SOCK is structured, omitting the aspects that are not central for us (see [4] for a detailed description).

Service behavior layer. The service behavior layer describes the actions performed by services. Actions can be operations on the state or communications. Services are identified by the name of their operations, and by their location.

SOCK error handling is based on the concepts of *scope*, *fault*, and *compensation*. A scope is a process container denoted by a unique name. A fault is a signal raised by a process when an error state is reached. A compensation is used either to smoothly stop a running activity in case of an external fault, or to compensate the activity after its successful termination (this encompasses both WS-BPEL termination and compensation mechanisms). Recovering mechanisms are implemented by exploiting processes called *handlers*. We use *fault handlers* and *compensation handlers*. They are executed to manage respectively internal faults and external faults/compensation requests.

SOCK *syntax* is based on the following (disjoint) sets: Var, ranged over by x, y, for variables, Val, ranged over by v, for values, $\mathcal{O}$, ranged over by o, for one-way operations, $Faults$, ranged over by f, for faults, and $Scopes$, ranged over by q, for

Table 1. Service behavior syntax with faults

$P, Q ::= \overline{o}@l(\boldsymbol{y})$	output	$o(\boldsymbol{x})$	input
$x := e$	assignment	$P; Q$	sequence
$P \vert Q$	parallel comp.	$\sum_{i \in I} o_i(\boldsymbol{x}_i); P_i$	external choice
$if\ \chi\ then\ P\ else\ Q$	det. choice	$while\ \chi\ do\ (P)$	iteration
$\mathbf{0}$	null process	$\{P : \mathcal{H} : u\}_{q_\perp}$	active scope
$\mathrm{inst}(\mathcal{H})$	install handler	$\mathrm{throw}(f)$	throw
$\mathrm{comp}(q)$	compensate	$\langle P \rangle$	protection

scope names. Loc is a subset of Val containing locations, ranged over by l. We denote as SC the set of service behavior processes, ranged over by $P, Q, \ldots$. We use $q_\perp$ to range over $Scopes \cup \{\perp\}$, whereas u ranges over $Faults \cup Scopes \cup \{\perp\}$. Here $\perp$ is used to specify that a handler is undefined. $\mathcal{H}$ denotes a function from $Faults$ and $Scopes$ to processes (or $\perp$). The function associating P_i to u_i for $i \in \{1, \ldots, n\}$ is $[u_1 \mapsto P_1, \ldots, u_n \mapsto P_n]$. Finally, we use the notation $\boldsymbol{k} = \langle k_0, k_1, ..., k_i \rangle$ for vectors.

The syntax of service behavior processes is defined in Table 1. A one-way output $\overline{o}@l(\boldsymbol{y})$ invokes the operation o of a service at location l, where $\boldsymbol{y}$ are the variables that specify the values to be sent. Dually, in a one-way $o(\boldsymbol{x})$, $\boldsymbol{x}$ contains the variables that will receive the communicated values. Assignment, sequence, parallel composition, external and deterministic choice, iteration, and null process are standard.

We denote with $\{P\}_q$ a scope named q executing process P. An active scope has instead the form $\{P : \mathcal{H} : u\}_{q_\perp}$, where $\mathcal{H}$ specifies the defined handlers. Term $\{P\}_q$ is a shortcut for $\{P : \mathcal{H}_0 : \perp\}_q$, where $\mathcal{H}_0$ evaluates to $\perp$ for all fault names and to $\mathbf{0}$ for all scope names. The argument u is the name of a handler waiting to be executed, or $\perp$ if there is no such handler. When a scope has failed its execution, either because it has been killed from a parent scope, or because it has not been able to manage an internal fault, it reaches a zombie state. Zombie scopes have $\perp$ as scope name. Primitives $\mathrm{throw}(f)$ and $\mathrm{comp}(q)$ respectively raises fault f and asks to compensate scope q. $\langle P \rangle$ executes P in a protected way, i.e. not influenced by external faults. Handlers are installed into the nearest enclosing scope by $\mathrm{inst}(\mathcal{H})$, where $\mathcal{H}$ is the required update of the handler function. We assume that $\mathrm{comp}(q)$ occurs only within handlers, and q can only be a child of the enclosing scope. For each $\mathrm{inst}(\mathcal{H})$, $\mathcal{H}$ is defined only on fault names and on the name of the nearest enclosing scope. Finally, scope names are unique.

The service behavior layer *semantics* generates all the transitions allowed by the process behavior, specifying the constraints on the state that have to be satisfied for them to be performed. The state is a substitution of values for variables. We use σ to range over substitutions, and write $[\boldsymbol{v}/\boldsymbol{x}]$ for the substitution assigning values in $\boldsymbol{v}$ to variables in $\boldsymbol{x}$. Given a substitution σ, $\mathrm{Dom}(\sigma)$ is its domain.

Let Act be the set of labels of the semantics, ranged over by a. We use structured labels of the form $\iota(\sigma : \theta)$ where ι is the kind of action while σ and θ are substitutions containing respectively the assumptions and the effects on the state. We also use the unstructured labels $th(f), cm(q, P), inst(\mathcal{H})$. We use operator $\boxplus$ for updating the handler function:

$$(\mathcal{H} \boxplus \mathcal{H}')(u) = \begin{cases} \mathcal{H}'(u) \text{ if } u \in \mathrm{Dom}(\mathcal{H}') \\ \mathcal{H}(u) \text{ otherwise} \end{cases}$$

Table 2. Standard rules for service behavior layer ($a \neq th(f)$)

$$\text{(ONE-WAYOUT)}\quad \overline{o}@l(\boldsymbol{x}) \xrightarrow{\overline{o}(\boldsymbol{v})@l(\boldsymbol{v}/\boldsymbol{x}:\emptyset)} \mathbf{0} \qquad \text{(ONE-WAYIN)}\quad o(\boldsymbol{x}) \xrightarrow{o(\boldsymbol{v})(\emptyset:\boldsymbol{v}/\boldsymbol{x})} \mathbf{0}$$

$$\text{(ASSIGN)}\ \frac{\mathrm{Dom}(\sigma) = \mathrm{Var}(e) \quad [\![e\sigma]\!] = v}{x := e \xrightarrow{\tau(\sigma:v/x)} \mathbf{0}} \qquad \text{(IF-THEN)}\ \frac{\mathrm{Dom}(\sigma) = \mathrm{Var}(\chi) \quad [\![\chi\sigma]\!] = true}{if\ \chi\ then\ P\ else\ Q \xrightarrow{\tau(\sigma:\emptyset)} P}$$

$$\text{(SEQUENCE)}\ \frac{P \xrightarrow{a} P'}{P;Q \xrightarrow{a} P';Q} \qquad \text{(PARALLEL)}\ \frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q} \qquad \text{(CHOICE)}\ \frac{o_i(\boldsymbol{x}_i) \xrightarrow{a} Q_i \quad i \in I}{\sum_{i\in I} o_i(\boldsymbol{x}_i); P_i \xrightarrow{a} Q_i; P_i}$$

STRUCTURAL CONGRUENCE

$$P \mid Q \equiv Q \mid P \qquad P \mid \mathbf{0} \equiv P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad \mathbf{0}; P \equiv P \qquad \langle \mathbf{0} \rangle \equiv \mathbf{0}$$

Intuitively, handlers in $\mathcal{H}'$ replace the corresponding ones in $\mathcal{H}$. We also use $\mathrm{cmp}(\mathcal{H})$ to denote the part of $\mathcal{H}$ dealing with compensations.

The SOCK semantics is defined as a relation $\rightarrow \subseteq SC \times Act \times SC$. The main rules for standard actions are in Table 2, while Table 3 defines the fault handling mechanism.

Rule ONE-WAYOUT defines the output operation, where $\boldsymbol{v}/\boldsymbol{x}$ is the assumption on the state. Rule ONE-WAYIN corresponds to the input operation: it makes no assumption on the state, but it specifies a state update. The other rules in Table 2 are standard. The internal process P of a scope can execute thanks to rule SCOPE in Table 3. Handlers are installed in the nearest enclosing scope by rules ASKINST and INSTALL. According to rule SCOPE-SUCCESS, when a scope successfully ends, its compensation handlers are propagated to the parent scope. Compensation execution is required by rule COMPENSATE. The actual compensation code Q is guessed, and the guess is checked by rule COMPENSATION. Faults are raised by rule THROW. A fault is caught by rule CATCHFAULT when a scope defining the corresponding handler is met. Activities involving the termination of a sub-scope and the termination of internal error recovery are managed by the rules for fault propagation THROW-SYNC, THROW-SEQ and RETHROW, and by the partial function $killable$. Function $killable$ computes the activities that have to be completed before the handler is executed and it is applied to parallel components by rule THROW-SYNC. Moreover, function $killable$ guarantees that when a fault is thrown there is no pending handler update. This is obtained by making $killable(P, f)$ undefined (and thus rule THROW-SYNC not applicable) if some handler installation is pending in P. The $\langle P \rangle$ operator (described by rule PROTECTION) guarantees that the enclosed activity will not be killed by external faults. Rule SCOPE-HANDLE-FAULT executes a handler for a fault. A scope that has been terminated from the outside is in zombie state. It can execute its compensation handler thanks to rule SCOPE-HANDLE-TERM, and then terminate with failure using rule SCOPE-FAIL. Similarly, a scope enters the zombie state when reached by a fault it cannot handle (rule RETHROW). The fault is propagated up along the scope hierarchy. Zombie scopes cannot throw faults any more, since rule IGNORE-FAULT has to be applied instead of RETHROW.

Table 3. Faults-related rules for service behavior layer ($a \neq th(f)$)

$$\text{(SCOPE)}\ \frac{P \xrightarrow{a} P' \quad a \neq inst(\mathcal{H}), cm(q', \mathcal{H}')}{\{P : \mathcal{H} : u\}_{q_\perp} \xrightarrow{a} \{P' : \mathcal{H} : u\}_{q_\perp}} \qquad \text{(INSTALL)}\ \frac{P \xrightarrow{inst(\mathcal{H})} P'}{\{P : \mathcal{H}' : u\}_{q_\perp} \xrightarrow{\tau(\emptyset:\emptyset)} \{P' : \mathcal{H}' \boxplus \mathcal{H} : u\}_{q_\perp}}$$

$$\text{(ASKINST)}\ \mathrm{inst}(\mathcal{H}) \xrightarrow{inst(\mathcal{H})} \mathbf{0} \qquad \text{(THROW)}\ \mathrm{throw}(f) \xrightarrow{th(f)} \mathbf{0} \qquad \text{(COMPENSATE)}\ \mathrm{comp}(q) \xrightarrow{cm(q,Q)} Q$$

$$\text{(SCOPE-SUCCESS)}\ \{\mathbf{0} : \mathcal{H} : \perp\}_q \xrightarrow{inst(\mathrm{cmp}(\mathcal{H}))} \mathbf{0} \qquad \text{(SCOPE-HANDLE-FAULT)}\ \{\mathbf{0} : \mathcal{H} : f\}_{q_\perp} \xrightarrow{\tau(\emptyset:\emptyset)} \{\mathcal{H}(f) : \mathcal{H} \boxplus [f \mapsto \perp] : \perp\}_{q_\perp}$$

$$\text{(COMPENSATION)}\ \frac{P \xrightarrow{cm(q,Q)} P', \mathcal{H}(q) = Q}{\{P : \mathcal{H} : u\}_{q'_\perp} \xrightarrow{\tau(\emptyset:\emptyset)} \{P' : \mathcal{H} \boxplus [q \mapsto \mathbf{0}] : u\}_{q'_\perp}}$$

$$\text{(SCOPE-HANDLE-TERM)}\ \{\mathbf{0} : \mathcal{H} : q\}_\perp \xrightarrow{\tau(\emptyset:\emptyset)} \{\mathcal{H}(q) : \mathcal{H} \boxplus [q \mapsto \mathbf{0}] : \perp\}_\perp \qquad \text{(SCOPE-FAIL)}\ \{\mathbf{0} : \mathcal{H} : \perp\}_\perp \xrightarrow{\tau(\emptyset:\emptyset)} \mathbf{0}$$

$$\text{(PROTECTION)}\ \frac{P \xrightarrow{a} P'}{\langle P \rangle \xrightarrow{a} \langle P' \rangle} \qquad \text{(THROW-SYNC)}\ \frac{P \xrightarrow{th(f)} P', killable(Q, f) = Q'}{P|Q \xrightarrow{th(f)} P'|Q'} \qquad \text{(THROW-SEQ)}\ \frac{P \xrightarrow{th(f)} P'}{P; Q \xrightarrow{th(f)} P'}$$

$$\text{(CATCH-FAULT)}\ \frac{P \xrightarrow{th(f)} P', \mathcal{H}(f) \neq \perp}{\{P : \mathcal{H} : u\}_{q_\perp} \xrightarrow{\tau(\emptyset:\emptyset)} \{P' : \mathcal{H} : f\}_{q_\perp}} \qquad \text{(IGNORE-FAULT)}\ \frac{P \xrightarrow{th(f)} P', \mathcal{H}(f) = \perp}{\{P : \mathcal{H} : u\}_\perp \xrightarrow{\tau(\emptyset:\emptyset)} \{P' : \mathcal{H} : u\}_\perp}$$

$$\text{(RETHROW)}\ \frac{P \xrightarrow{th(f)} P', \mathcal{H}(f) = \perp}{\{P : \mathcal{H} : u\}_q \xrightarrow{th(f)} \langle \{P' : \mathcal{H} : \perp\}_\perp \rangle}$$

where

$killable(\{P : \mathcal{H} : u\}_q, f) = \langle \{killable(P, f) : \mathcal{H} : q\}_\perp \rangle$ if $P \not\equiv \mathbf{0}$
$killable(P \mid Q, f) = killable(P, f) \mid killable(Q, f)$
$killable(P; Q, f) = killable(P, f)$ if $P \not\equiv \mathbf{0}$
$killable(\langle P \rangle, f) = \langle P \rangle$ if $killable(P, f)$ is defined
$killable(P, f) = \mathbf{0}$ if $P \in \{\mathbf{0}, o(\boldsymbol{x}), \overline{o}@l(\boldsymbol{x}), x := e, if\ \chi\ then\ P\ else\ Q, while\ \chi\ do\ (P)$
$\sum_{i \in W} o_i(\boldsymbol{x}_i); P_i, \mathrm{throw}(f), \mathrm{comp}(q)\}$

Service engine layer. The service engine layer manages the service state and instances. A service engine Y can be a session $(P, \mathcal{S})$, where P is a service behavior process and $\mathcal{S}$ is a state, or a parallel composition $Y|Y$ of them. The service engine layer allows to propagate only labels such that the condition σ (if available) is satisfied by the current state, and applies to the state the state update ρ.

Services system layer. The service system layer allows the interaction between different engines. A service system E can be a located service engine $Y@l$ or a parallel composition $E \parallel E$ of them. The services system layer just allows complementary communication actions to interact, transforming them into internal steps τ, and propagates the other actions.

3 Request-Response Interaction Pattern

A request-response pattern is a bi-directional interaction where a client sends a message to a server and waits for an answer. When a server receives such a message, it elaborates the answer and sends it back to the client. In the literature there are two proposals to deal with a client that fails during a request-response interaction. The WS-BPEL approach kills the receive activity and, when the message arrives, it is silently discarded. In Jolie instead, clients always wait for the answer and exploit it for error recovery.

Here we present an intermediate approach: in case of failure we wait for the answer, but without blocking the computation. Moreover, when the answer is received we allow for the execution of a compensation activity. Let $\mathcal{O}_r$ be the set of request-response operations, ranged over by o_r. We define the request-response pattern in terms of the output primitive $\overline{o_r}@l(\boldsymbol{y}, \boldsymbol{x}, P)$, also called solicit, and of the input primitive $o_r(\boldsymbol{x}_1, \boldsymbol{y}_1, Q)$. When interacting, the client sends the values from variables $\boldsymbol{y}$ to the server, that stores them in variables $\boldsymbol{x}_1$. Then, the server executes process Q and, when Q terminates, the values in variables $\boldsymbol{y}_1$ are sent back to the client who stores them in variables $\boldsymbol{x}$. Only at this point the execution of the client can restart. If a fault occurs on the client-side after the remote service has been invoked, but before the answer is received, we allow the client to handle the fault regardless of the reply, so that recovery can start immediately. However, we create a receiver for the missing message in a fresh session so that, if later on the message is received, the operation can be compensated. The compensation is specified by the parameter P of the solicit operation. If instead a fault is raised on the server-side during the computation of the answer, the fault is propagated to the client where it raises a local fault. In this case there is no need to compensate the remote invocation, since we assume that this is dealt with by local recovery of the server.

Service behavior calculus - extension. We extend here the behavioral layer with the request-response and with few auxiliary operators used to define its semantics.

$\overline{o_r}@l(\boldsymbol{y}, \boldsymbol{x}, P)$	Solicit	$o_r(\boldsymbol{x}_1, \boldsymbol{y}_1, Q)$	Request-Response
$Exec(l, o_r, \boldsymbol{y}, P)$	Req.-Resp. execution	$Wait(o_r, \boldsymbol{y}, P)$	Wait
$\overline{o_r}!f@l$	Fault output	$Bubble(P)$	Bubble

$Exec(l, o_r, \boldsymbol{y}, P)$ is a server-side running request-response: P is the process computing the answer, o_r the name of the operation, $\boldsymbol{y}$ the vector of variables to be used for the answer, and l the client location. Symmetrically, $Wait(o_r, \boldsymbol{y}, P)$ is the process waiting for the response on client-side: o_r is request-response operation, $\boldsymbol{y}$ is the vector of variables for storing the answer and P is the compensation code to run in case the client fails before the answer is received. When a fault is triggered on the server-side, an error notification has to be sent to the client: this is done by $\overline{o_r}!f@l$, where o_r is the operation, f the fault and l the client location. If a fault occurs on client-side, we have to move the receipt operation to a fresh, parallel session, so that error recovery can start immediately. This is done by the primitive $Bubble(P)$, which allows to create a new session (a "bubble") executing code P. This primitive is the key element that allows a failed solicit to wait for a response outside its scope and potentially allowing its termination regardless of the arrival of the answer.

The semantics of the behavior layer is extended with the rules presented in Table 4 (the last rule refers to the engine). Function $killable$ is also extended, as follows:

Table 4. Request-response pattern and engine rules

(SOLICIT)
$$\overline{o_r}@l(\boldsymbol{y},\boldsymbol{x},P) \xrightarrow{\overline{o_r}(\boldsymbol{v})@l(\emptyset:\boldsymbol{v}/\boldsymbol{x})} Wait(o_r,\boldsymbol{x},P)$$

(REQUEST)
$$o_r(\boldsymbol{x},\boldsymbol{y},P) \xrightarrow{o_r(\boldsymbol{v})::l(\emptyset:\boldsymbol{v}/\boldsymbol{x})} Exec(l,o_r,\boldsymbol{y},P)$$

(REQUEST-EXEC)
$$\frac{P \xrightarrow{a} P'}{Exec(l,o_r,\boldsymbol{y},P) \xrightarrow{a} Exec(l,o_r,\boldsymbol{y},P')}$$

(THROW-REXEC)
$$\frac{P \xrightarrow{th(f)} P'}{Exec(l,o_r,\boldsymbol{y},P) \xrightarrow{th(f)} P' \mid \langle \overline{o_r}!f@l \rangle}$$

(REQUEST-RESPONSE)
$$Exec(l,o_r,\boldsymbol{y},\mathbf{0}) \xrightarrow{\overline{o_r}(\boldsymbol{v})@l(\boldsymbol{v}/\boldsymbol{y}:\emptyset)} \mathbf{0}$$

(SOLICIT-RESPONSE)
$$Wait(o_r,\boldsymbol{x},P) \xrightarrow{o_r(\boldsymbol{v})(\emptyset:\boldsymbol{v}/\boldsymbol{x})} \mathbf{0}$$

(SEND-FAULT)
$$\overline{o_r}!f@l \xrightarrow{\overline{o_r}(f)@l(\emptyset:\emptyset)} \mathbf{0}$$

(RECEIVE FAULT)
$$Wait(o_r,\boldsymbol{x},P) \xrightarrow{o_r(f)(\emptyset:\emptyset)} \mathrm{throw}(f)$$

(CREATE BUBBLE)
$$Bubble(P) \xrightarrow{\tau(\emptyset:\emptyset)[\![P]\!]} \mathbf{0}$$

(ENGINE-BUBBLE)
$$\frac{P \xrightarrow{\tau(\emptyset:\emptyset)[\![Q]\!]} P' \quad Q \neq \mathbf{0}}{(P,\mathcal{S}) \xrightarrow{\tau} (P',\mathcal{S}) \mid (Q,\mathcal{S})}$$

- $killable(Exec(l,o_r,\boldsymbol{y},P),f) = killable(P,f)|\langle o_r!f@l\rangle$
- $killable(\,Wait(o_r,\boldsymbol{x},P),f) = Bubble(\,Wait(o_r,\boldsymbol{x},\mathbf{0});P)$
- $killable(\overline{o_r}!f@l,f) = \overline{o_r}!f@l$
- $killable(Bubble(P),f) = Bubble(P)$

Rules SOLICIT and REQUEST start a solicit-response operation on client and server side respectively. Upon invocation, the request-response becomes an active construct executing process P, and storing all the information needed to send back the answer. The execution of P is managed by rule REQUEST-EXEC. When P terminates, rule REQUEST-RESPONSE sends back an answer. This synchronizes with rule SOLICIT-RESPONSE on the client side.

A running request-response reached by a fault is transformed into a fault notification (see rule THROW-REXEC and the definition of function $killable$) on server side. Fault notification is executed by rule SEND-FAULT, and it interacts with the waiting receive thanks to rule RECEIVE-FAULT. When received, the fault is re-thrown at the client side.

A fault on client side instead gives rise to a bubble, creating the process that will wait for the answer in a separate session. The bubble is created by rule CREATE BUBBLE, and will be installed at the service engine level by rule ENGINE-BUBBLE. The label for bubble creation has the form $\tau(\emptyset:\emptyset)[\![P]\!]$, where P is the process to be run inside the new session. We will write $\tau(\emptyset:\emptyset)$ for $\tau(\emptyset:\emptyset)[\![\mathbf{0}]\!]$. The new receive operation inside the bubble has no handler update, since it will be executed out of any scope, and its compensating code P has been promoted as a continuation. In this way, P will be executed only in case of successful answer. In case of faulty answer, the generated fault will have no effect since it is in a session on its own.

Service engine calculus - extension. We have to add to the service engine layer a rule for installing bubbles: when a bubble reaches the service engine layer, a new session is started executing the code inside the bubble (rule ENGINE-BUBBLE in Table 4).

Service system calculus - extension. The service system calculus semantics is extended by allowing the labels for request-response communication to be matched.

Example. We present now an example of usage of the request-response primitive. A first solution for the hotel reservation example described in the introduction is:

```
CLIENT :== book_r@hotel_Imperial(⟨CC,dates⟩,⟨res_num⟩,
             annul@hotel_Imperial(⟨res_num⟩) );
           P
```

The $book_r$ operation transmits the credit card number CC and the dates of the reservation and waits for the reservation number. In case the user wants to cancel the reservation before receiving an answer from the hotel a fault can be used to kill this operation. In such a case the annul operation is invoked when the answer is received to compensate the $book_r$ operation. The annul operation will be executed in a new session by using our mechanism based on bubbles.

As a more concrete instance, we could consider the case where the user is willing to wait a limited amount of time for the answer from the hotel, after which (s)he will cancel the reservation. This case could be programmed by assuming a service $timeout$ that offers a request-response operation that sends back an answer after n seconds[1]:

```
CLIENT:==
    res_num:= 0;
    { inst(f ↦ if res_num==0 then throw(tm));
      (timeout_r@timeout(⟨60⟩,⟨ ⟩,0); throw(f)
      | book_r@hotel_Imperial(⟨CC,dates⟩,⟨res_num⟩,
          annul@hotel_Imperial(⟨res_num⟩) ); throw(f)   )
    }_q ; P
```

In this scenario the timeout operation is in parallel with the booking. The first operation that finishes raises the fault f that is caught by the handler of the scope q. The fault will kill the remaining operation and if the hotel response has not arrived yet (i.e. the value of res_num is still 0) then the fault tm is raised. P is executed otherwise.

A similar solution is not viable in BPEL: in case of timeout, the booking invocation is killed, and if an answer arrives, it is discarded. Thus one does not know whether the invocation succeeded or not, neither the reservation number in case of success.

In Jolie, the answer is used for error recovery. However, in case no answer is received from the booking service, the whole service engine gets stuck. In our approach instead the main session can continue its execution without delays.

It is difficult to apply the proposed solution when two or more solicits install handlers or require compensation. One may try to exploit the handler update primitive, but in this way compensations are executed inside the scope, thus they have to be terminated before execution can proceed. This problem, and other technical difficulties, justify the multiple solicit response primitive introduced in the next section.

4 Multiple Request-Response Communication Pattern

The request-response pattern allows one invocation to be sent and one answer to be received. For optimization reasons, it may be important to invoke many services in parallel, and only consider the first received answer (speculative parallelism).

[1] Clearly, because of network delay the answer may be received later than expected.

Table 5. Multiple request-response pattern rules

(MSR-SOLICIT)
$$\frac{z_1 = \overline{o_r}@l(\boldsymbol{y}, \boldsymbol{x}, P) \mapsto Q \quad w_{m+1} = \mathit{Wait}(o_r, \boldsymbol{y}, P) \mapsto Q}{\mathit{Wait}^+(z_1, \ldots, z_n \triangleright w_1, \ldots, w_m) \xrightarrow{\overline{o_r}(\boldsymbol{v})@l(\emptyset:\boldsymbol{v}/\boldsymbol{x})} \mathit{Wait}^+(z_2, \ldots, z_n \triangleright w_1, \ldots, w_m, w_{m+1})}$$

(MSR-RESPONSE)
$$\frac{\forall k \in \{1, \ldots, n\} : w_k = \mathit{Wait}(o_{r_k}, \boldsymbol{y}_k, P_k) \mapsto Q_k \;\; i \in \{1, \ldots, n\} \;\; J = \{1, \ldots, n\} \setminus \{i\}}{\mathit{Wait}^+(\triangleright w_1, \ldots, w_n) \xrightarrow{o_{r_i}(\boldsymbol{v})(\emptyset:\boldsymbol{v}/\boldsymbol{y_i})} Q_i | \prod_{j \in J} \mathit{Bubble}(\mathit{Wait}(o_{r_j}, \boldsymbol{y_j}, \boldsymbol{0}); P_j)}$$

(MSR-IGNORE FAULT)
$$\frac{n > 1 \;\; w_i = \mathit{Wait}(o_{r_i}, \boldsymbol{y_i}, P_i) \mapsto Q_i \;\; i \in \{1, \ldots, n\}}{\mathit{Wait}^+(\triangleright w_1, \ldots, w_n) \xrightarrow{o_{r_i}(f)(\emptyset:\emptyset)} \mathit{Wait}^+(\triangleright w_1, \ldots, w_{i-1}, w_{i+1}, \ldots, w_n)}$$

$$\mathit{Wait}^+(\triangleright \mathit{Wait}(o_r, \boldsymbol{y}, P) \mapsto Q) \equiv \mathit{Wait}(o_r, \boldsymbol{y}, P); Q$$

We model this communication pattern using a dedicated primitive that we call multiple solicit-response (MSR for short). A MSR consists of a list of solicit-responses, each one equipped with its own continuation. Formally, we define the syntax of the MSR primitive as $\mathit{MSR}\{z_1, \ldots, z_n\}$ where each z_i is a *solicit-response with continuation* written $z_i = \overline{o_{r_i}}@l_i(\boldsymbol{y}_i, \boldsymbol{x}_i, P_i) \mapsto Q_i$. Intuitively, the continuation Q_i is executed only when $\overline{o_{r_i}}@l_i(\boldsymbol{y}_i, \boldsymbol{x}_i, P_i)$ is the first to receive a successful answer.

Service behavior calculus - extension. We extend the service behavior calculus with the MSR primitive and with some auxiliary operators:

$$\begin{array}{rll} P, Q ::= & \ldots & \\ & \mathit{MSR}\{z_1, \ldots, z_n\} & \text{multiple solicit-response} \\ & \mathit{Wait}^+(z_1, \ldots, z_n \triangleright w_1, \ldots, w_m) & \text{multiple wait} \\ z ::= & \overline{o_r}@l(\boldsymbol{y}, \boldsymbol{x}, P) \mapsto Q & \text{solicit with continuation} \\ w ::= & \mathit{Wait}(o_r, \boldsymbol{y}, P) \mapsto Q & \text{wait with continuation} \end{array}$$

In a MSR the solicits are sent one after the other, and only when all the requests have been sent the MSR can receive a response. For this reason we introduce the multiple wait $\mathit{Wait}^+(z_1, \ldots, z_n \triangleright w_1, \ldots, w_m)$ that specifies the solicits that still have to be sent $z_1, \ldots, z_n$, and the ones that will wait for an answer $w_1, \ldots, w_n$. Thus, the MSR primitive $\mathit{MSR}\{z_1, \ldots, z_n\}$ above is a shortcut for $\mathit{Wait}^+(z_1, \ldots, z_n \triangleright)$. Moreover, we have that a multiple wait with only one waiting process is structurally equivalent to a standard wait. We formally define the behavior of the MSR primitive by extending the service behavior semantics with the rules presented in Table 5.

The multiple wait executes all the solicit-responses through rule MSR-SOLICIT. Once all the solicits have been sent, the multiple wait receives a successful answer through rule MSR-RESPONSE. It continues the execution with the corresponding continuation code, and kills all the other solicits by creating a bubble for each remaining waiting process. If a fault notification arrives as an answer, it is discarded by rule MSR-IGNORE FAULT if there is at least another available wait. If instead there is no other solicit waiting for an answer, the last fault received is raised (rule RECEIVE FAULT

described in Table 4). When an external fault arrives a bubble containing a dead solicit response is created for every solicit that has been sent, as specified by the function $killable$ that is extended in the following way:

$$killable(Wait^{+}(z_1,\ldots,z_n \triangleright w_1,\ldots,w_m), f) = \prod_{Wait(o_{r_j},\boldsymbol{y_j},P_j)\mapsto Q_j \in \{w_1,\ldots,w_m\}} Bubble(Wait(o_{r_j},\boldsymbol{y_j},\mathbf{0});P_j)$$

The MSR primitive is perfectly suited to capture speculative parallelism scenarios. Consider for instance the hotel reservation problem defined in the introduction. Suppose to use two booking services for making the hotel reservation, and that you would like to get the acknowledgment in 1 minute. If the booking services are located at A and B and if we use the $timeout$ service introduced before, this service could be defined as:

```
CLIENT :== msr {
    timeout_r@timeout(⟨60⟩,⟨ ⟩,0) ↦ throw(tm)
    book_r@H_1(⟨CC,dates⟩,⟨res_num⟩, annul@H_1(⟨res_num⟩) ) ↦ 0
    book_r@H_2(⟨CC,dates⟩,⟨res_num⟩, annul@H_2(⟨res_num⟩) ) ↦ 0 }
```

5 Related and Future Work

Among the most related approaches, Web-π [6] has no request-response pattern and its treatment of faults is rather different from ours. Orc [5] has a pruning primitive similar to our MSR. However, since Orc has no notion of fault, all the difficulties coming from error management do not emerge. Finally, the service oriented calculi CaSPiS [1] and COWS [7] include low-level mechanisms allowing the programmer to close ongoing conversations. However, our approach is different, since we aim at providing primitives which free the programmer from this burden.

As a future work, we plan to incorporate the primitives we propose in Jolie. Also we would like to study their expressive power: the implementation of MSR in terms of the existing primitives is not easy and we believe that a separation result could be proved.

References

1. Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and Pipelines for Structured Service Programming. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 19–38. Springer, Heidelberg (2008)
2. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: On the interplay between fault handling and request-response service invocations. In: ACSD 2008, pp. 190–198. IEEE Press (2008)
3. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: Dynamic error handling in service oriented applications. Fundamentae Informaticae 95(1), 73–102 (2009)
4. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: SOCK: A Calculus for Service Oriented Computing. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 327–338. Springer, Heidelberg (2006)
5. Kitchin, D., Quark, A., Cook, W., Misra, J.: The Orc Programming Language. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS 2009. LNCS, vol. 5522, pp. 1–25. Springer, Heidelberg (2009)

6. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
7. Lapadula, A., Pugliese, R., Tiezzi, F.: A Calculus for Orchestration of Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007)
8. OASIS. Web Services Business Process Execution Language Version 2.0, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
9. World Wide Web Consortium. Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/wsdl