On the Interplay Between Fault Handling and Request-Response Service Invocations*

Claudio Guidi Ivan Lanese Fabrizio Montesi
Gianluigi Zavattaro
Department of Computer Science, University of Bologna, Italy
{cguidi,lanese,fmontesi,zavattar}@cs.unibo.it

Abstract

Service Oriented Computing (SOC) allows for the composition of services which communicate using unidirectional notification or bidirectional request-response primitives. Most service orchestration languages proposed so far provide also primitives to handle faults and compensations. The interplay between fault handling and request-response invocations is nontrivial since, for instance, faults should be notified to the request-response communication partners in order to compensate also the remote activities. Our work is motivated by the difficulties encountered in programming, using current orchestration languages, some fault handling strategies. We propose as a solution an orchestration programming style in which fault and compensation handlers are dynamically installed. *We show the adequacy of our proposal defining its semantics, and proving that it satisfies some expected high-level properties. Finally, we also show how to apply dynamic handler installation in a nontrivial automotive scenario.*

1 Introduction

Service Oriented Computing (SOC) provides languages and mechanisms for describing, publishing, retrieving and combining autonomous services. We are particularly interested in service composition, which is usually dealt with using orchestration languages such as the de-facto standard WS-BPEL (BPEL for short) [17]. Since both services and the network infrastructure are unreliable, orchestration languages have to provide mechanisms to deal with unexpected situations. BPEL, for instance, permits to specify *fault handlers* to manage faults, *termination handlers* to smoothly terminate an ongoing activity when an external fault occurs and *compensation handlers* to undo the effect of a completed activity during error recovery.

Besides traditional one-way communication, SOC usually supports a bidirectional communication pattern composed by the *solicit-response* operation on the client-side, which sends a request and waits for the reply, and the symmetric *request-response* operation on the server-side.

In this paper we investigate the interplay between fault handling and the request-response pattern. For instance, if a fault occurs on the client-side during the execution of a solicit-response operation, the answer from the partner, that could be either successful or unsuccessful, should be taken into account inside the fault handling activity. In fact, it should be possible to program on the client-side a fault handler that compensates the activity executed by the server if and only if the remote activity has been successfully completed. Interestingly, this rather intuitive behavior is not easily programmable in current service orchestration languages such as BPEL. In fact, quoting the BPEL specifications, "when a synchronous invoke activity (corresponding to a request/reply operation) is interrupted and terminated prematurely, the response (if received) for such a terminated activity is silently discarded". In this way, since the (either successful or unsuccessful) response is discarded, it is not possible to take it into account inside the fault handling activity.

A rigorous analysis of the interplay between fault handling and request-response invocations requires the definition of a formal model. The model that we exploit is achieved by adding mechanisms for fault, termination and compensation handling to SOCK [12], the unique (to the best of our knowledge) process calculus for SOC providing the request-response communication pattern. In order to prove general results about the interplay between fault handling and the request-response pattern, we consider a very general and flex-

*Research partially funded by EU Integrated Project SENSORIA, contract n. 016004.

ible framework for error recovery. In particular, we assume that fault, termination, and compensation handlers are not statically defined, but that they can be updated at runtime. More precisely, we consider a primitive for *dynamic handler installation* $\text{inst}(\mathcal{H})$ which updates the handlers according to a handlers specification $\mathcal{H}$. We show in Section 2 that the dynamic approach is more general than the static one.

Besides being more general, dynamic handler installation provides an elegant way to program the dependency between fault handling and the request-response pattern described above: it is sufficient to permit the update of the fault handlers upon successful completion of the solicit-response operation.

Another important interplay between fault handling and request-response communication is the notification to the clients of a server failure occurring in between the execution of the receive and the reply actions. BPEL, for instance, does not specify a precise policy to manage this case. In Active-BPEL [1], one of the most well-known engines for BPEL, when an engine terminates, a `missing-reply` exception is sent to all those clients waiting to complete a request-response interaction with that engine. Again, we consider a more general approach: we ensure that the response is always sent to the client and, in case of fault, the specific fault occurred during the computation is notified so that the client can adapt its error-recovery procedure.

Technical contribution. As stated above, we propose a formal model to investigate error-recovery mechanisms in the presence of request-response communication. This is achieved extending SOCK [12] with primitives for the installation of fault, termination and compensation handlers, for throwing faults, and for compensating successfully completed activities. The main novelties of the proposed framework are *dynamic handler installation* and *automatic failure notification*. We first describe the key concepts of error handling in Section 2, then we formalize our approach in Section 3.

In the design of the framework we have been driven by five correctness properties that formalize the expected behaviour of fault and compensation handling and the request-response pattern. These properties are defined in Section 4 and proved in [11].

As additional contribution, we formalize in Section 5 a nontrivial service based scenario: the *automotive case study* of the EU Project Sensoria. Section 6 contains some conclusions and a comparison with related work.

2 Error handling key concepts

Fault handling in SOC involves four basic concepts: *scope*, *fault*, *termination* and *compensation*. A scope is a process container denoted by a unique name and able to manage faults. A fault is a signal raised by a process towards the enclosing scope when an error state is reached, in order to allow for its recovering. Termination and compensation are mechanisms exploited to recover from errors. Termination is automatically triggered when a running scope must be smoothly stopped because of a fault thrown by a parallel process. Compensation, instead, is explicitly invoked by the programmer to undo the effect of a scope whose execution has already successfully completed. Recovering mechanisms are implemented by exploiting *handlers* which contain processes to be executed when faults, terminations or compensations occur. Handlers are defined within a scope which represents the execution boundaries for their application. There are three kinds of handlers: *fault handlers*, *termination handlers* and *compensation handlers*. Fault handlers are executed when a fault is thrown by the internal process of the scope, termination handlers are executed when a scope is reached by a fault raised by an external process and, finally, compensation handlers have to be explicitly invoked by another handler for recovering the activities of a child scope whose computation has already successfully finished. At runtime, when a fault f is raised within a scope q, all its enclosed running activities are terminated. Note that any of the enclosed activities could be a scope, and in this case its termination handler is automatically executed. After that, if q has a fault handler for f, it executes it. Otherwise, the fault is propagated upwards to the parent scope. It is worth noting that handlers can be programmed to compensate any child scope that has successfully completed its activity before f was raised. Compensation is achieved by executing the related compensation handler.

Usually, error recovery is managed by statically associating handlers to scopes, i.e. providing a primitive like $\mathtt{scope}_q(P, \mathcal{FH}, \mathcal{TH}, \mathcal{CH})$, which defines a scope with name q, executing process P and fault, termination and compensation handlers $\mathcal{FH}$[1], $\mathcal{TH}$ and $\mathcal{CH}$, respectively.

In some cases static declaration of handlers is not enough to easily model a given scenario. For instance, consider the following pseudo-code:

$R \mid \mathtt{scope}_q(\mathtt{while}(i < 100)$
$\quad(i = i{+}1; \mathtt{if}\ i\%2 = 0\ \mathtt{then}\ P\ \mathtt{else}\ Q), \mathcal{F}H, \mathcal{T}H, \mathcal{C}H)$

where R is a generic process (parallel to scope q) which can raise a fault f. Scope q contains a loop which executes 100 cycles. Odd cycles execute process P, even cycles process Q. Suppose that, at some point of execution, R raises a fault f, which triggers the termina-

[1] $\mathcal{FH}$ may define more than one fault handler, for treating different kinds of faults.

tion of the scope q. Suppose also that the termination handling policy for scope q requires to compensate the activities executed so far in the reverse order of completion. Thus, one has to remember how many P and Q activities have been executed, and in which order, for compensating them accordingly. Without any specific support from the language the programmer has to use some bookkeeping variables, but as the complexity of the code increases the bookkeeping becomes more complex and error-prone. In order to address this problem we propose *dynamic handling*, which allows for the updating of the handlers while the computation progresses.

Technically, we consider a scope construct of the form $\mathtt{scope}_q(P, \mathcal{H})$ where q is the name of the scope, P is the process to be executed, and $\mathcal{H}$ is a function associating fault handlers to fault names and termination and compensation handlers to scope names. Dynamic handling is addressed by an *installing* primitive, $\mathrm{inst}(\mathcal{H}')$, which updates the current handler function with $\mathcal{H}'$. Thus, the handler code can be updated at runtime, depending on the current state of the scope. The example above could be rewritten by exploiting the dynamic handler mechanism as:

$$
\begin{array}{l}
R \mid \mathtt{scope}_q(\mathtt{while}(i < 100) \\
\quad (i = i + 1; \mathtt{if}\ i\%2 = 0\ \mathtt{then}\ P; \mathrm{inst}([q \mapsto P'; cH]) \\
\qquad\qquad \mathtt{else}\ Q; \mathrm{inst}([q \mapsto Q'; cH])), \mathcal{H})
\end{array}
$$

where cH represents the previously installed termination handler. In this case, when P completes its execution, the statement $\mathrm{inst}([q \mapsto P'; cH])$ updates the current termination handler for q, pointed by cH, by adding process P' (which specifically compensates process P) to it, whereas if Q is executed the termination handler is updated by adding Q'. When reached by a fault f, scope q executes the last installed termination handler, compensating the whole sequence of activities. Different compensation strategies can easily be programmed.

Note that in the example above it should never be the case that an execution of P has been completed and its compensation has not been installed, since otherwise the termination handler of the scope would not be up-to-date. This can be obtained in the dynamic scenario by disallowing the interruption of the execution flow whenever an inst statement is to be executed. The same behaviour cannot be obtained in the static approach, where we simulate handlers updating by using bookkeeping variables, as we cannot distinguish whether a variable assignment is related to fault management or not.

Fault handlers are installed by means of the same inst statement, by associating a handler to the fault name (e.g. in the example above $\mathrm{inst}([f \mapsto R])$ would install a fault handler with process R for fault f). Moreover, when a scope successfully ends, its last defined termination handler becomes its compensation handler. Note that there is no ambiguity between the two handlers: a termination handler is executed by the scope itself when interrupted by a fault generated by a parallel activity, whereas a compensation handler can only be executed by the parent scope. This allows also to trivially simulate the static approach with the dynamic one: the construct $\mathtt{scope}_q(P, \mathcal{FH}, \mathcal{TH}, \mathcal{CH})$ can be simply rephrased as $\mathtt{scope}_q(\mathrm{inst}([f \mapsto \mathcal{FH}]); \mathrm{inst}([q \mapsto \mathcal{TH}]); P; \mathrm{inst}([q \mapsto \mathcal{CH}]), \mathcal{H}_0)$ in which the fault and termination handlers are installed before the execution of the activity, the compensation handler at the end, and $\mathcal{H}_0$ defines no handlers.

3 SOCK: the service behavior layer

In order to present a formalization of our approach we extend the syntax and semantics of SOCK [12] with the primitives described in the previous section. SOCK is a calculus for modeling service oriented systems, inspired by WSDL and BPEL [17]. Its primitives include both uni-directional and bi-directional WSDL communication patterns, control primitives from imperative languages, and parallel composition from concurrent languages. SOCK is structured in three different layers: (i) the service behavior layer to specify the actions performed by a service, (ii) the service engine layer dealing with state, service instances and correlation sets, and (iii) the services system layer allowing different engines to interact. Since faults and compensations are managed in the service behavior layer we refer to [11] for the detailed description of the other layers. Here instead we present the service behavior layer, starting from the standard part and adding then the fault and compensation primitives.

Syntax. We consider the following (disjoint) sets: Var, ranged over by x, y, for variables, Val, ranged over by v, for values, $Faults$, ranged over by f, for faults, $Scopes$, ranged over by q, for scopes, $\mathcal{O}$, ranged over by o, for one-way operations, and $\mathcal{O}_R$, ranged over by o_r for request-response operations. Also, Loc is a subset of Val containing locations, ranged over by l. We consider a corresponding subset of Var, $VarLoc$ containing location variables and ranged over by z. We use $q_\perp$ to range over $Scopes \cup \{\perp\}$, whereas u ranges over $Faults \cup Scopes \cup \{\perp\}$. Finally, we use the notation $\vec{k} = \langle k_0, k_1, ..., k_i \rangle$ for vectors.

Let SC be the set of service behavior processes, ranged over by $P, Q, \ldots$. $\mathcal{H}$ denotes a function from

$\epsilon ::= o(\vec{x}) \mid o_r(\vec{x}, \vec{y}, P)$
$\overline{\epsilon} ::= \overline{o}@z(\vec{y}) \mid \overline{o_r}@z(\vec{y}, \vec{x}, \mathcal{H})$

$P, Q, \ldots ::=$	ϵ	input
	$\overline{\epsilon}$	output
	$x := e$	assignment
	$P; Q$	sequential composition
	$P \mid Q$	parallel composition
	$\sum_{i \in W} \epsilon_i; P_i$	non-det. choice
	$while\ \chi\ do\ (P)$	iteration
	$\mathbf{0}$	null process
	$\{P\}_q$	scope (for $\{P : \mathcal{H}_0 : \bot\}_q$)
	$\mathrm{inst}(\mathcal{H})$	install handler
	$\mathrm{throw}(f)$	throw
	$\mathrm{comp}(q)$	compensate
	cH	current handler

Table 1. Service behavior syntax

$P, Q, \ldots ::=$	$Exec(P, o_r, \vec{y}, l)$	Req.-Resp. execution
	$\{P : \mathcal{H} : u\}_{q_\bot}$	active scope
	$o_r(\vec{x}, \mathcal{H})$	response in Solicit
	$o_r\langle\vec{x}, \mathcal{H}\rangle$	dead response in Solicit
	$\langle P \rangle$	protection
	$\overline{o_r}!f@l$	sending fault

Table 2. Extended service behavior syntax

$Faults$ and $Scopes$ to processes extended with $\bot$, i.e. $\mathcal{H} : Faults \cup Scopes \rightarrow SC \cup \{\bot\}$. In particular, we write the function associating P_i to u_i for $i \in \{1, \ldots, n\}$ as $[u_1 \mapsto P_1, \ldots, u_n \mapsto P_n]$. Also, σ ranges over substitutions, written as $[\vec{v}/\vec{x}]$. The syntax for processes is defined in Table 1. The first part contains the standard constructs. Here $\mathbf{0}$ is the null process. Outputs can be notifications $\overline{o}@z(\vec{y})$ or solicit-responses $\overline{o_r}@z(\vec{y}, \vec{x}, \mathcal{H})$ where $o \in \mathcal{O}$, $o_r \in \mathcal{O}_R$ and $z \in VarLoc$. The two tuples $\vec{y}$ and $\vec{x}$ are respectively the parameters to be sent in the invocation and the variables where the received values will be stored (only for solicit-response). Also, $\mathcal{H}$ contains the handlers to be installed to compensate the remote activity. Dually, inputs can be one-ways $o(\vec{x})$ or request-responses $o_r(\vec{x}, \vec{y}, P)$ where the notations are as above. Additionally, P is the process to be executed between the request and the response. Assignment $x := e$ assigns the result of the expression e to the variable $x \in Var$ (function $[\![e]\!]$ evaluates a closed expression e). We do not present the syntax of expressions: we just assume that they include the arithmetic and boolean operators, values in Val and variables. $P; Q$ and $P \mid Q$ are sequential and parallel composition respectively, whereas $\sum_{i \in W} \epsilon_i; P_i$ is input-guarded non-deterministic choice. Also, $while\ \chi\ do\ (P)$ models iteration.

The last five cases concern faults and compensations and are the main novelty. We consider two kinds of handlers: *fault handlers* and *termination/compensation handlers*. Handlers are installed by $\mathrm{inst}(\mathcal{H})$ where $\mathcal{H}$ is a partial function from fault and scope names to processes. $\{P\}_q$ defines a scope named q to execute process P. This is a shortcut for $\{P : \mathcal{H}_0 : \bot\}_q$ where $\mathcal{H}_0$ is the function that evaluates to $\bot$ for all fault names and to $\mathbf{0}$ for all scope names. Commands $\mathrm{throw}(f)$ and $\mathrm{comp}(q)$ respectively raises fault f and asks to compensate scope q. Finally, cH refers to the previously installed handler during an handler update.

Well-formedness rules: informally, $\mathrm{comp}(q)$ and cH occur only within handlers, and q can only be a child of the enclosing scope. Also, for each $\mathrm{inst}(\mathcal{H})$, $\mathcal{H}$ is undefined on all scope names q but the one of the nearest enclosing scope, i.e. a process can define the termination/compensation handler only for its own scope. Finally, we assume that scope names are unique.

Semantics. In order to define the semantics we exploit the extended syntax in Table 2. There $\{P : \mathcal{H} : u\}_{q_\bot}$ is an active scope, i.e. a scope where handlers may have been installed into $\mathcal{H}$ and an handler named u is waiting to be executed (if $u \neq \bot$). Also, the scope name may be $\bot$, denoting that the scope has been killed and is now in a zombie state (i.e., it can no more end with success nor throw faults but it may e.g. wait for incoming messages, to be used during the recovery activity). Also, $o_r(\vec{x}, \mathcal{H})$ is used to wait for the response in a solicit-response interaction. $\mathcal{H}$ is installed iff a non faulty response is received, allowing to program the compensation for the remote activity. $o_r\langle\vec{x}, \mathcal{H}\rangle$ is the corresponding zombie version, which can not throw faults. $Exec(P, o_r, \vec{y}, l)$ is a running request-response interaction. $\langle P \rangle$ executes P in a protected way, i.e. not influenced by external faults. This is needed to ensure that recovery from a fault is completed even if another fault happens. Finally, $\overline{o_r}!f@l$ notifies fault f to a partner.

The service behavior layer does not deal with state, leaving this issue to the service engine layer, but it models all the possible execution paths for all the possible values of variables. The semantics follows this idea: the labels contain all the possible actions, together with the necessary requirements on the state. Formally, let Act be the set of actions, ranged over by a. To simplify the interaction with upper layers (see [11]) we use mainly structured labels of the form $\iota(\sigma : \theta)$ where ι is

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

Table 4. Structural congruence

the kind of action while σ and θ are substitutions containing respectively the assumptions on the state that should be satisfied for the action to be performed and the effect on the state. If the second argument is not meaningful for the action at hand we write _ instead. Furthermore we use the following unstructured actions: $\{th(f), cm(q, P), inst(\mathcal{H})\}$.

Definition 1 (Service behavior layer semantics) *We define $\rightarrow \subseteq SC \times Act \times SC$ as the least relation which satisfies the rules of Tables 3 and 5, and closed w.r.t. structural congruence $\equiv$, the least congruence relation satisfying the axioms in Table 4.*

The rules in Table 3 describe the standard executions of the system. For this reason, label $th(f)$ is never considered in this table. Rules One-WayOut and One-WayIn define the one-way operation. Notice that the output specifies the location of the invoked service. The two operations are synchronized at the services system level. Similarly rules Solicit and Request start a solicit-response operation. Notice that the input stores the location of the invoker, since it is necessary for the response. Notice also that the solicit-response primitive allows for the specification of some handlers. These are installed just after the answer has been received, and only in case of success. The dedicated syntax is needed to ensure that the handlers are installed only in case of success of the remote activity. The response is dealt with by rules Request-Response and Solicit-Response. The response is computed by rule Request-Exec. This is necessary since in case of failure the ongoing computation should be treated in a particular way: the failure should be notified to the partner. Rule Scope allows for standard execution of a process inside a scope. The other rules in Table 3 are standard.

The rules in Table 5 define the semantics of scopes, faults and compensations. We use operator $\oplus$, defined as follows, for updating the handlers function:

$$(\mathcal{H} \oplus \mathcal{H}')(f) = \begin{cases} (\mathcal{H}'(f))[\mathcal{H}(f)/cH] \\ \quad \text{if } f \in \mathrm{Dom}(\mathcal{H}') \cap \mathrm{Dom}(\mathcal{H}) \\ (\mathcal{H}'(f))[\mathbf{0}/cH] \\ \quad \text{if } f \in \mathrm{Dom}(\mathcal{H}'), f \notin \mathrm{Dom}(\mathcal{H}) \\ \mathcal{H}(f) \text{ otherwise} \end{cases}$$

where we assume that inst is a binder for cH, i.e. substitutions are not applied inside the inst primitive.

Intuitively the above definition means that handlers in $\mathcal{H}'$ replace the corresponding handlers in $\mathcal{H}$, and occurrences of cH in the new handlers are replaced by the old handlers. For instance, $\mathrm{inst}([q \mapsto P|cH])$ adds P in parallel to the old handler for q. We also use $\mathrm{cmp}(\mathcal{H})$ to denote the part of $\mathcal{H}$ dealing with compensations, i.e. $\mathrm{cmp}(\mathcal{H}) = \mathcal{H}|_{Scopes}$.

$$\begin{array}{l}
killable(\{P : \mathcal{H} : u\}_q, f) = \langle \{killable(P, f) : \mathcal{H} : q\}_q \rangle \\
\qquad \text{if } P \not\equiv \mathbf{0} \\
killable(P \mid Q, f) = killable(P, f) \mid killable(Q, f) \\
killable(P; Q, f) = killable(P, f) \text{ if } P \not\equiv \mathbf{0} \\
killable(Exec(P, o_r, \vec{y}, l), f) = killable(P, f) | \langle \overline{o_r}!f@l \rangle \\
killable(\langle P \rangle, f) = \langle P \rangle \text{ if } killable(P, f) \\
killable(o_r(\vec{y}, \mathcal{H}), f) = \langle o_r \langle \vec{y}, \mathcal{H} \rangle \rangle \\
killable(P, f) = \mathbf{0} \\
\qquad \text{if } P \in \{\mathbf{0}, \epsilon, \overline{\epsilon}, x := e, \chi?P : Q, \\
\qquad\qquad while\ \chi\ do\ (P), \overline{o_r}!f'@l, o_r\langle \vec{y}, \mathcal{H} \rangle, \\
\qquad\qquad \sum_{i \in W} \epsilon_i; P_i, \mathrm{throw}(f), \mathrm{comp}(q)\}
\end{array}$$

Table 6. Function $killable$

Fault and compensation 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. This allows to compensate a terminated activity. A compensation execution is asked by rule Compensate. Notice that the actual compensation code is guessed, and the guess is checked by rule Compensation. Faults are raised by rule Throw. A fault is catched by rule Catch-Fault when a scope defining the corresponding handler is met. The name of the handler is stored in the third component of the scope construct: the handler is executed only after a few activities have been performed. These are individuated by the rules for fault propagation (Throw-Sync, Throw-Seq, ReThrow, Throw-RExec) and by the partial function *killable*. This function has a double aim. On one side it guarantees that handlers are installed before any fault is thrown, i.e. handlers are always up-to-date (see Proposition 5). Technically this is obtained by making $killable(P, f)$ undefined (and thus rule Throw-Sync not applicable) if some handler installation is pending in P. On the other side $killable(P, f)$ computes the activities that have to be completed before the handler is executed. In particular, when a sub-scope is terminated, its termination handler is marked as next handler to be executed. Notice that it may substitute a previously marked fault handler, following the intuition that a request of termination has priority w.r.t. an internal activity such as a fault processing. Also, if an *Exec* (i.e., an ongoing request-response computation) is terminated the fault is notified to the partner (this is where the parameter f is needed). Finally, a receive waiting for the answer of

(One-WayIn) $o(\vec{x}) \xrightarrow{o(\vec{v})(\emptyset:\vec{v}/\vec{x})} \mathbf{0}$

(Request) $o_r(\vec{x},\vec{y},P) \xrightarrow{\uparrow o_r(\vec{v})@l(\emptyset:\vec{v}/\vec{x})} Exec(P,o_r,\vec{y},l)$

(Request-Response) $Exec(\mathbf{0},o_r,\vec{y},l) \xrightarrow{\downarrow\overline{o_r}(\vec{v})@l(\vec{v}/\vec{y}:\_)} \mathbf{0}$

(One-WayOut) $\overline{o}@z(\vec{x}) \xrightarrow{\overline{o}(\vec{v})@l(l/z,\vec{v}/\vec{x}:\_)} \mathbf{0}$

(Solicit) $\overline{o_r}@z(\vec{x},\vec{y},\mathcal{H}) \xrightarrow{\uparrow\overline{o_r}(\vec{v})@l(l/z,\vec{v}/\vec{x}:\_)} o_r(\vec{y},\mathcal{H})$

(Solicit-Response) $o_r(\vec{x},\mathcal{H}) \xrightarrow{\downarrow o_r(\vec{v})(\emptyset:\vec{v}/\vec{x})} \mathrm{inst}(\mathcal{H})$

(Assign)
$$\frac{\mathrm{Dom}(\sigma)=\mathrm{Var}(e) \quad [\![e\sigma]\!]=v}{x := e \xrightarrow{\tau(\sigma:v/x)} \mathbf{0}}$$

(Request-Exec)
$$\frac{P \xrightarrow{a} P'}{Exec(P,o_r,\vec{y},l) \xrightarrow{a} Exec(P',o_r,\vec{y},l)}$$

(Iteration)
$$\frac{\mathrm{Dom}(\sigma)=\mathrm{Var}(\chi) \quad [\![\chi\sigma]\!]=true}{while\ \chi\ do\ (P) \xrightarrow{\tau(\sigma:\_)} P; while\ \chi\ do\ (P)}$$

(No-Iteration)
$$\frac{\mathrm{Dom}(\sigma)=\mathrm{Var}(\chi) \quad [\![\chi\sigma]\!]=false}{while\ \chi\ do\ (P) \xrightarrow{\tau(\sigma:\_)} \mathbf{0}}$$

(Scope)
$$\frac{P \xrightarrow{a} P' \quad a \neq inst(\mathcal{H}), cm(q',\mathcal{H}')}{\{P:\mathcal{H}:u\}_{q_\perp} \xrightarrow{a} \{P':\mathcal{H}:u\}_{q_\perp}}$$

(Sequence)
$$\frac{P \xrightarrow{a} P'}{P;Q \xrightarrow{a} P';Q}$$

(Parallel)
$$\frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q}$$

(Choice)
$$\frac{\epsilon_i \xrightarrow{a} Q_i \quad i \in I}{\sum_{i\in I} \epsilon_i; P_i \xrightarrow{a} Q_i; P_i}$$

Table 3. Rules for service behavior layer, $a \neq th(f)$

(Throw) $\mathrm{throw}(f) \xrightarrow{th(f)} \mathbf{0}$

(Compensate) $\mathrm{comp}(q) \xrightarrow{cm(q,P)} P$

(AskInst) $\mathrm{inst}(\mathcal{H}) \xrightarrow{inst(\mathcal{H})} \mathbf{0}$

(Scope-Handle-Fault) $\{\mathbf{0}:\mathcal{H}:f\}_{q_\perp} \xrightarrow{\tau(\emptyset:\_)} \{\mathcal{H}(f):\mathcal{H}\oplus[f\mapsto\perp]:\perp\}_{q_\perp}$

(Scope-Success) $\{\mathbf{0}:\mathcal{H}:\perp\}_q \xrightarrow{inst(\mathrm{cmp}(\mathcal{H}))} \mathbf{0}$

(Scope-Handle-Term) $\{\mathbf{0}:\mathcal{H}:q\}_q \xrightarrow{\tau(\emptyset:\_)} \{\mathcal{H}(q):\mathcal{H}\oplus[q\mapsto\mathbf{0}]:\perp\}_\perp$

(Scope-Fail) $\{\mathbf{0}:\mathcal{H}:\perp\}_\perp \xrightarrow{\tau(\emptyset:\_)} \mathbf{0}$

(Install)
$$\frac{P \xrightarrow{inst(\mathcal{H})} P'}{\{P:\mathcal{H}':u\}_{q_\perp} \xrightarrow{\tau(\emptyset:\_)} \{P':\mathcal{H}'\oplus\mathcal{H}:u\}_{q_\perp}}$$

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

(Protection)
$$\frac{P \xrightarrow{a} P'}{\langle P\rangle \xrightarrow{a} \langle P'\rangle}$$

(Throw-Sync)
$$\frac{P \xrightarrow{th(f)} P', killable(Q,f)=Q'}{P|Q \xrightarrow{th(f)} P'|Q'}$$

(Throw-Seq)
$$\frac{P \xrightarrow{th(f)} P'}{P;Q \xrightarrow{th(f)} P'}$$

(Catch-fault)
$$\frac{P \xrightarrow{th(f)} P', \mathcal{H}(f)\neq\perp}{\{P:\mathcal{H}:u\}_{q_\perp} \xrightarrow{\tau(\emptyset:\_)} \{P':\mathcal{H}:f\}_{q_\perp}}$$

(Ignore-fault)
$$\frac{P \xrightarrow{th(f)} P', \mathcal{H}(f)=\perp}{\{P:\mathcal{H}:u\}_\perp \xrightarrow{\tau(\emptyset:\_)} \{P':\mathcal{H}:u\}_\perp}$$

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

(Throw-RExec)
$$\frac{P \xrightarrow{th(f)} P'}{Exec(P,o_r,\vec{y},l) \xrightarrow{th(f)} P' | \langle\overline{o_r}!f@l\rangle}$$

(Send-Fault) $\overline{o_r}!f@l \xrightarrow{\overline{o_r}(f)@l(\emptyset:\_)} \mathbf{0}$

(Receive-Fault) $o_r(\vec{x},\mathcal{H}) \xrightarrow{o_r(f)(\emptyset:\_)} \mathrm{throw}(f)$

(Dead-Solicit-Response) $o_r\langle\vec{x},\mathcal{H}\rangle \xrightarrow{\downarrow o_r(\vec{v})(\emptyset:\vec{v}/\vec{x})} \mathrm{inst}(\mathcal{H})$

(Dead-Receive-Fault) $o_r\langle\vec{x},\mathcal{H}\rangle \xrightarrow{o_r(f)(\emptyset:\_)} \mathbf{0}$

Table 5. Rules for service behavior layer: faults and compensation rules

a solicit-response is preserved, thus preserving the pattern of communication. The $\langle P\rangle$ operator (described by rule Protection) guarantees that the enclosed activity will not be disturbed by external faults. Rule Scope-Handle-Fault executes an handler for a fault. The fault is removed from the function $\mathcal{H}$ in order to allow *throw* primitives for the same fault in the handler to propagate the fault to the outer scope. Notice that a scope that has handled an internal fault can still end with success. Instead a scope that has been terminated (rule SCOPE-HANDLE-TERM) or has been unable to handle an internal fault (rule RETHROW) reaches a zombie state: it can no more end with success, nor throw faults. This is denoted by the $\bot$ that substitutes the scope name and obtained by rules SCOPE-FAIL and IGNORE-FAULT. This last rule is necessary only for faults thrown by handlers, since no other fault can be generated by a zombie scope. Rules SEND-FAULT and RECEIVE-FAULT allow to send a fault notification to a partner, where it is treated as a fault. Rules DEAD-SOLICIT-RESPONSE and DEAD-RECEIVE-FAULT define the behavior of operator $o_r\langle\vec{x},\mathcal{H}\rangle$, which behaves like $o_r(\vec{x},\mathcal{H})$ but can not raise any fault.

4 Properties and examples

In this section we present five properties which show that the semantics presented in the previous section precisely models the most important error recovery mechanisms. Each of them states that a specific linguisitic primitive of SOCK always has a determined behaviour. The first two properties deal with scope behaviours, the third and fourth properties deal with solicit-response and request-response communication patterns whereas the fifth property deals with handler installation.

The first property states that an isolated scope can end only with success, i.e. by successfully completing its inner activities, or with failure, i.e. by raising a fault. For instance $\{throw(f)\}_q$ will end unsuccessfully. Note that a scope able to internally recover a fault will end with success. As an example, let us consider the following case

$$\{\mathrm{inst}([q \mapsto COMP, f \mapsto HANDLE]); \mathrm{throw}(f)\}_q$$

where scope q will manage its internal fault by executing $HANDLE$, and then end with success. In this way compensation $COMP$ will be available for outer scopes.

Proposition 1 *Let* $P \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2 \ldots \xrightarrow{a_n} P_n$ *be a computation. Suppose that* $P = \{Q\}_q$. *Then there are three possible cases:*

1. *the scope ends successfully*[2]*:* $P_i \xrightarrow{inst(\mathcal{H})} \mathbf{0}$ *for some* i*, furthermore no fault is raised before:* $a_j \neq th(f')$ *for each* $j < i$ *and each fault* f'*;*
2. *the scope raises a fault:* $P_i \xrightarrow{th(f)} P_{i+1}$*, furthermore:*
   - *(a)* P_{i+1} *will never end with success:* $a_j \neq inst(\mathcal{H})$ *for each* $j > i$ *and each* $\mathcal{H}$*;*
   - *(b) no other fault will be raised, i.e.* $a_j \neq th(f')$ *for each* $j > i$ *and each* f'*.*
3. *the scope is still executing:* $P_n = \{P' : \mathcal{H} : u\}_q$*.*

[2] We identify successful termination for P from the label $inst(\mathcal{H})$: the same label is also generated by the primitive inst, but this case never occurs if P is a scope.

The second property takes into account the fact that, in general, in a complex application, a scope can also be terminated because of an external fault. This is the case for example of the following process

$$\{\overline{pay_r}@z(\vec{x},\vec{y},[q \mapsto UNDO])\}_q | \,\mathrm{throw}(f)$$

where scope q could be terminated by the external fault f. In this case, the scope will never end successfully.

Proposition 2 *Let* $P \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2 \ldots \xrightarrow{a_n} P_n$ *be a computation. Suppose that* P *is a scope that has been terminated, i.e.* $P = killable(\{P' : \mathcal{H} : u\}_q, f)$*. Then:*

1. P *will never end with success:* $a_i \neq inst(\mathcal{H})$ *for each* i *and each* $\mathcal{H}$*;*
2. *no other fault will be raised, i.e.* $a_i \neq th(f')$ *for each* i *and each* f'*.*

Note that the two propositions above cover all the possible cases, since the request of termination is the only effect that a context can have on a process which is not acknowledged by transitions of the process itself.

The third and fourth properties guarantee that the request-response pattern is always preserved, even if a fault occurs in the middle of the message exchange (i.e., the request has been sent, but the response has not been received). In particular, the third property ensures that the requester always waits for the response of the callee. The response can then be used during error recovery. Dually, the fourth property ensures that the callee always sends an answer, either a normal message or a fault notification.

As an example, let us consider the scenario above where a remote activity pay_r, within scope q, must be compensated iff it has been successfully executed. The external fault f may occur at three different points: (i) before the solicit-response is started: the solicit-response will never start and the executed termination handler will be empty as required; (ii) between the solicit and the response: the termination handler for q will be scheduled for execution, but the response is waited for, thus when the handler for q is executed its value is empty if a fault has been received by the payment service (i.e. pay_r has failed), $UNDO$ otherwise; (iii) after scope q has completed: the compensation handler for q has been propagated upstream (to a scope not represented), so that the handler for f can use it to compensate the remote activity; instead, if the response was a fault notification then no compensation handler has been propagated, as no compensation is needed.

Proposition 3 *Let* $P \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2 \ldots \xrightarrow{a_n} P_n$ *be a computation. Let* a_1 *be* $\uparrow \overline{o_r}(\vec{v})@l(l/z, \vec{v}/\vec{x} : \_)$, *i.e. the start action in a solicit-response. Then there are two possible cases:*

1. *the response has been received:* $a_i = \downarrow o_r(\vec{v'})(\emptyset : \vec{v'}/\vec{x'})$ *or* $a_i = o_r(f)(\emptyset : \_)$ *for some* i;
2. *the process is waiting for the response:* $P_n \xrightarrow{\downarrow o_r(\vec{v'})(\emptyset:\vec{v'}/\vec{x'})} P'$.

To show the fourth property, we need to define when a part of a process is being executed.

Definition 2 (Enabling contexts) *We define enabling contexts by structural induction as follows:*

$$C[\bullet] = \bullet \mid C[\bullet]|Q \mid \{C[\bullet] : \mathcal{H} : u\}_{q_\perp} \mid \langle C[\bullet]\rangle \mid C[\bullet]; Q \mid Q|C[\bullet] \mid Exec(C[\bullet], o_r, \vec{y}, l) \mid Q; C[\bullet] \text{ if } Q \equiv \mathbf{0}$$

Proposition 4 *Let* $P \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2 \ldots \xrightarrow{a_n} P_n$ *be a computation. Let* a_1 *be* $\uparrow o_r(\vec{v})@l(\emptyset : \vec{v}/\vec{x})$, *i.e. the start action in a request-response. Then there are three possible cases:*

1. *the response is sent:* $a_i = \downarrow \overline{o_r}(\vec{v'})@l(\vec{v'}/\vec{y} : \_)$ *or* $a_i = \overline{o_r}(f)@l(\emptyset : \_)$ *for some* i;
2. *the request-response is still executing:* $P_n = C[Exec(P, o_r, \vec{y}, l)]$ *for some enabling context* $C[\bullet]$;
3. *a fault is ready to be notified to the partner:* $P_n = C[\overline{o_r}!f@l]$ *for some enabling context* $C[\bullet]$.

Finally, the fifth property guarantees that handlers are installed as soon as they are available, i.e. available handlers are always installed before any fault is triggered. This is an important feature in dynamic handler installation, since one has to ensure that error recovery is always done according to the most recent handlers.

Proposition 5 *If* $P \xrightarrow{th(f)} P'$ *then it never occurs that* $P \equiv C[\mathrm{inst}(\mathcal{H})]$ *for any enabling context* $C[\bullet]$, *i.e. no handler is waiting to be installed.*

5 Automotive scenario

This section discusses the automotive scenario [19], which has been chosen as case study inside the EU Project SENSORIA.

In the scenario, a car engine failure occurs so that the car is no longer drivable. The car service system must take care of bookings and payments for the necessary assistance, calling in particular a car rental, a garage and a towing truck service. If both garage and tow truck are available, the rented car has to go to the garage (the client will be brought there by the tow truck), otherwise the rented car must go to the location of the broken car. While the whole system can be modeled in SOCK, we present here only the most significant part of the car service system behavior. The car orchestrator CAR_P contains three modules: R_P, G_P and T_P, interacting with the car rental service, the garage service and the towing truck service respectively. G_P and T_P are sequentially composed, as the tow truck requires the garage to be available, whereas R_P is executed in parallel. Each module handles both the booking of the corresponding service and the invocation of the bank service for the payment. We assume that CAR_P knows the following pieces of information: the locations of the garage service (G), of the towing truck service (T), of the car rental service (R) and of the bank service (B), their prices and bank accounts (represented, respectively, by the variables subscripted by *price* and *acc*), the faults throwable by them (fG, fT, fR and fB respectively), and the garage and car coordinates (G_{coords} and CAR_{coords}). The SOCK implementation is in Table 7. Module G_P is a scope containing the solicit-response invocations needed for the booking and payment of the garage. The booking invocation ($\overline{book}$) may receive and raise fault fG, which is handled at the higher level by CAR_P. The fault from the bank service (fB), instead, is managed by the local fault handler, which compensates the garage booking and re-throws the fault upstream as a garage fault fG. Finally, when the last solicit-response invocation receives a successful response, the specified compensation handler for the scope is installed. Module T_P is analogous, but it does not install a compensation handler as its compensation is never required. Module R_P is more complex since it has to deal with possible faults from G_P or T_P. In this case the rented car, if not already booked, has to be requested directly at the broken car location (this is achieved by executing $RHandler_P$); otherwise it has to be redirected to the broken car location (by executing $Rredirect_P$). The termination handler for R_P should behave differently according to when it is invoked: before the invocation of the booking, between the invo-

$$
\begin{array}{rl}
CAR_P ::= \{ & \mathrm{inst}([fG \mapsto \mathrm{comp}(r), fT \mapsto \mathrm{comp}(g) \mid \mathrm{comp}(r)]);\ ((G_P; T_P) \mid R_P\ )\ \}_{main} \\
G_P ::= \{ & \overline{book}@G(failure, \langle G_{acc}, G_{id}\rangle, [fB \mapsto \overline{revbook}@G(G_{id}); \mathrm{throw}(fG)]); \\
& \overline{pay}@B(\langle CAR_{acc}, G_{acc}, G_{id}\rangle, G_{payid}, [g \mapsto \overline{revbook}@G(G_{id}) \mid \overline{revpay}@B(G_{payid})])\ \}_g \\
T_P ::= \{ & \overline{book}@T(\langle CAR_{coords}, G\rangle, \langle T_{acc}, T_{id}\rangle, [fB \mapsto \overline{revbook}@T(T_{id}); \mathrm{throw}(fT)]); \\
& \overline{pay}@B(\langle CAR_{acc}, T_{acc}, T_{id}\rangle, T_{payid}, \emptyset)\ \}_t \\
RHandler_P ::= & \overline{book}@R(CAR_{coords}, \langle R_{acc}, R_{id}\rangle, \emptyset); Rpay_P \\
Rredirect_P ::= & \overline{redirect}@R(\langle R_{id}, CAR_{coords}\rangle, R_{id}, \emptyset) \\
Rpay_P ::= & \overline{pay}@B(\langle CAR_{acc}, R_{acc}, R_{id}\rangle, R_{payid}, \emptyset) \\
Rrevbook_P ::= & \overline{revbook}@R(R_{id}) \\
Rrevpay_P ::= & \overline{revpay}@B(R_{payid}) \\
R_P ::= \{ & \mathrm{inst}([fR \mapsto \mathbf{0}, fB \mapsto Rrevbook_P; \mathrm{inst}([r \mapsto \mathbf{0}]), r \mapsto RHandler_P]); \\
& \overline{book}@R(G_{coords}, \langle R_{acc}, R_{id}\rangle, [r \mapsto Rredirect_P; Rpay_P]); \\
& \overline{pay}@B(\langle CAR_{acc}, R_{acc}, R_{id}\rangle, R_{payid}, [r \mapsto Rredirect_P, fR \mapsto Rrevpay_P]); \\
& \mathrm{inst}([r \mapsto \{\mathrm{inst}([fR \mapsto Rrevpay_P]); cH]\}_{rc})\ \ \}_r
\end{array}
$$

Table 7. The automotive case study

cation of the booking and of the payment service, or after the invocation of the payment service. This corresponds to the different termination handlers installed during the execution. We exploit the solicit-response primitive semantics to ensure that, in case the solicit has already been sent, the response from the partner is waited for and the corresponding handlers are installed if and only if the response is not a fault, i.e. the operation has been performed successfully. As in the case of G_P both the booking and the payment can fail, but R_P provides local fault handlers for each possible failure, guaranteeing that the faults are not propagated to the environment. Notice that the handler for fR (the fault thrown by a booking or redirection failure) is updated in order to reverse the payment only if it has already been performed (third line of R_P). When the activity terminates successfully, its compensation handler is defined. It retrieves via cH the last defined termination handler (which is $Rredirect$), and makes it executable inside an auxiliary scope. This is necessary since $Rredirect$ may raise fault fG, and the old handler specified in scope r will not be available any more. Finally, CAR_P handles faults fG and fT, compensating the other successfully terminated sub-scopes. Notice in fact that if a sub-scope has not terminated its execution, calling the *comp* primitive for its compensation does nothing; however, its termination handler has been executed during fault propagation.

6 Conclusions

We have investigated the interplay between the request-response pattern and the mechanisms for fault and compensation handling usually provided by service orchestration languages. The most relevant language which combines both aspects is BPEL. BPEL is not equipped with a formal semantics, thus the comparison with our formally defined language is done on the basis of the informal specifications and some experimentations done with the ActiveBPEL engine. Some basic differences between BPEL and our calculus have been already discussed in the Introduction, where we have also justified the choice to adopt dynamic handler installations. Note that this permits to avoid a syntactic distinction between termination and compensation handlers: the compensation handler is the last termination handler installed before successful completion of a scope. Moreover, we do not need any `rethrow` primitive, used in BPEL to pass a fault to the enclosing scope, since throw(f), when used inside an handler for f, has the same behavior. In BPEL this is not possible, as the language allows the activities enclosed in a scope to throw more than one fault. Dynamic handler installation distinguishes our calculus w.r.t. other calculi in the literature, such as πt-calculus [2], webπ [14] and cJoin [5], all featuring static compensations. Also, since the underlying languages, π-calculus and Join, do not provide bidirectional interactions, the problem of failure notification never occurs. Actually, cJoin transactions can model bidirectional interactions, and the fact that two interacting transactions are merged can be seen as a strong form of failure propagation. We decided to just notify the failure to the partner to model loosely coupled services. Other approaches for compensation handling are StAC [9], cCSP [8] and Sagas [6], but they are built on top of models not supporting interprocess communication. Among these only StAC features a small amount of dynamism, by allowing for the removal of the currently installed compensation handlers. In this paper we have formalized our proposal for error handling extending SOCK, as it comprises natively the request-response pattern, but we think that it can be applied to many other frameworks, from basic theoretical calculi such as π-calculus to industrial frameworks such as BPEL. In particular, it would be interesting to apply these ideas to other calculi for SOC such as COWS [15], SCC [4], SSCC [13] and CaSPiS [3]. We plan also to further validate our primitives by adding them to JOLIE [16, 18], the implementation of SOCK. Another interesting line for future research is the investigation of the relationship between choreography and orchestration languages. For instance, it could be interesting to study the impact of fault and compensation primitives on the notion of conformance as formalized in [7] or [10].

References

[1] ActiveBPEL Open Source Engine. `http://www.active-endpoints.com/active-bpel-engine-overview.htm`.

[2] L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In *Proc. of FMOODS'03*, volume 2884 of *LNCS*, pages 124–138. Springer, 2003.

[3] M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In *Proc. of FMOODS'08*, LNCS. Springer, 2008. To appear.

[4] M. Boreale et al. SCC: A Service Centered Calculus. In *Proc. of WS-FM'06*, volume 4184 of *LNCS*, pages 38–56. Springer, 2006.

[5] R. Bruni, H. Melgratti, and U. Montanari. Nested commits for mobile calculi: Extending join. In *Proc. of IFIP TCS'04*, pages 563–576. Kluwer, 2004.

[6] R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *Proc. of POPL'05*, pages 209–220. ACM Press, 2005.

[7] N Busi et al. Choreography and orchestration: A synergic approach for system design. In *Proc. of ICSOC'05*, volume 3826 of *LNCS*, pages 228–240, 2005.

[8] M. Butler, C.A.R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*, volume 3525 of *LNCS*, pages 133–150. Springer, 2004.

[9] M. J. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *Proc. of COORDINATION'04*, volume 2949 of *LNCS*, pages 87–104. Springer, 2004.

[10] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[11] C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro. On the interplay between fault handling and request-response service invocations (TR). Available at `http://www.cs.unibo.it/~lanese/publications/fulltext/TR-COMP.pdf`.

[12] C. Guidi et al. SOCK: A calculus for service oriented computing. In *Proc. of ICSOC'06*, volume 4294 of *LNCS*, pages 327–338. Springer, 2006.

[13] I. Lanese et al. Disciplining orchestration and conversation in service-oriented computing. In *Proc. of SEFM'07*, pages 305–314. IEEE Computer Society Press, 2007.

[14] C. Laneve and G. Zavattaro. Foundations of web transactions. In *Proc. of FOSSACS'05*, LNCS, pages 282–298, 2005.

[15] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.

[16] F. Montesi, C. Guidi, and G. Zavattaro. Composing services with JOLIE. In *Proc. of ECOWS'07*, pages 13–22. IEEE Computer Society Press, 2007.

[17] OASIS. *Web Services Business Process Execution Language Version 2.0*. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`.

[18] Open source project. *JOLIE: Java Orchestration Language Intepreter Engine*. `http://sourceforge.net/projects/jolie`.

[19] M. Wirsing et al. Semantic-based development of service-oriented systems. In *Proc. of FORTE'06*, volume 4229 of *LNCS*, pages 24–45. Springer, 2006.