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

Claudio Guidi, Ivan Lanese, Fabrizio Montesi, and 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 of the service orchestration languages proposed so far provide also primitives to handle faults and manage the subsequent compensation activities. The interplay between these two aspects is non trivial since, for instance, faults should be notified to the request-response communication partners in order to compensate also the remote activities. We first present a simple orchestration scenario requiring a precise distributed fault handling strategy. We show that this strategy cannot be programmed using current orchestration languages; then, we propose a new style for orchestration programming able to specify the required fault management strategy. Finally, we show the generality of our approach by analyzing its properties and applying it to a nontrivial scenario.

1 Introduction

Service Oriented Computing (SOC) intends to provide 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) [OAS]. 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 (possibly partially) undo the effect of a completed activity during error recovery.

Besides traditional one-way communication, SOC usually supports also 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,

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

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 the request-response service invocation pattern requires the definition of a formal model. The formal model that we exploit is achieved by adding mechanisms for fault, termination and compensation handling to SOCK [GLG+06], the unique (to the best of our knowledge) process calculus for service oriented computing 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 flexible framework for error recovery. In particular, we assume that fault, termination, and compensation handlers are not statically defined, but that they can be also updated at runtime. More precisely, we consider a primitive for *dynamic handler installation* inst($\mathcal{H}$) which updates the handlers according to a handlers specification $\mathcal{H}$. We will show in Section 2 that the dynamic approach is more general than the static one.

Besides being more general, dynamic handler installation provides also 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. We have checked the behavior of Active-BPEL [act], one of the most well-known engines for BPEL, and we have seen that when an engine terminates, a `missing-reply` exception is sent to all those clients which are still waiting to complete a request-response interaction with that engine. Also in this case, 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.

1.1 Technical contribution

As stated above, we propose a formal model to investigate error-recovery mechanisms in the presence of bidirectional request-response communication. This is achieved extending SOCK [GLG+06] 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 error recovery framework are *dynamic handler installation* and *automatic failure notification*. We first informally 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 not only driven by the intuitive interplay between fault handling and request-response communication described above, but also by five correctness properties that we formalize in Section 4 (the proofs instead can be found in Appendix B). The first one formalizes the expected behavior of a scope whose computation can be completed either successfully returning the compensation handler to be used to undo it, or unsuccessfully returning a fault. The second property deals with scopes that are terminated due to the failure of a sibling scope: the termination activity neither returns a compensation handler (they are not expected to be undone) nor another fault. The third and fourth properties formalize the expected behavior of request-response communications from the client and the server perspective, respectively. Finally, the fifth property concerns dynamic installation: we ensure that the execution of fault installation cannot be delayed in favor of fault actions. This last property guarantees that when a fault occurs, it will be managed by the most updated handlers.

As additional contribution, we present in Section 5 the formalization of a nontrivial service based scenario, namely the *automotive case study* of the EU Project SENSORIA. Furthermore, the semantics for the basic mechanisms of SOCK that we present is equivalent but simpler w.r.t. the one in [GLG+06]. We complete the paper with some conclusive remarks and a detailed comparison with related work in Section 6.

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 triggered when a scope must be smoothly stopped, whereas compensation is triggered to undo the effect of a scope whose execution has already successfully terminated. Recovering mechanisms are implemented by exploiting *handlers* which contain processes to be executed when faults, terminations or compensations are triggered. 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 triggered 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 can be explicitly invoked by another handler for recovering the activities of a child scope whose

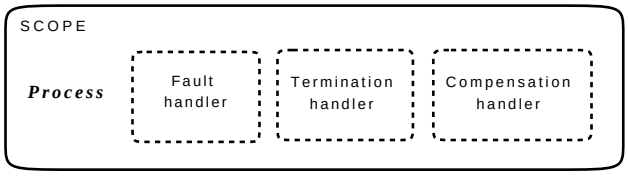

Fig. 1. Handlers in a scope

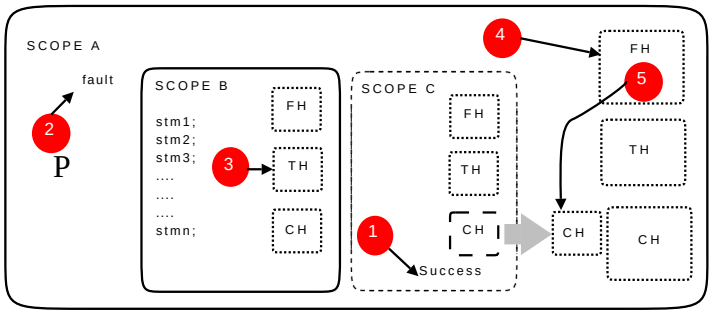

Fig. 2. Handler mechanisms

computation has already successfully finished. Fig. 1 shows all the elements composing a scope. A language managing error recovery via statically defined scopes (such as BPEL) should provide a primitive like $\texttt{scope}_q(P, \mathcal{FH}, \mathcal{TH}, \mathcal{CH})$ where q is the scope name, P the executing process and $\mathcal{FH}$, $\mathcal{TH}$ and $\mathcal{CH}$ are, respectively, the fault, termination and compensation handlers. When a fault is raised, it is propagated and it causes the termination of all the other activities inside the same scope. After that, if the fault handler for that fault is defined, the scope executes it, otherwise it forwards the fault to the outer scope. It is worth noting that a terminating activity could be a scope, and in this case its termination handler should be executed. Also, some linguistic primitive, such as `comp(q)`, can be used to require the execution of the compensation handler of the scope named q. Fig. 2 provides an intuitive representation of handler mechanisms where numbers represent ordered events and *stm1,stm2,...,stmn* represent a list of generic statements. A scope A encloses a generic process P and two scopes B and C. At 1 scope C finishes successfully by promoting its compensation handler to be executable by the enclosing scope A. At 2, process P raises a fault which is propagated to scope B. We suppose that B is still executing when reached by the fault so, at 3, it executes its termination handler and terminates. At 4 the fault handler of scope A is executed and, at 5, it compensates scope C (supposing that the handler specifies so).

In some cases static declaration of handlers is not enough to easily model a given scenario. Beyond the case of solicit-response that will be illustrated in

Section 4, consider the following pseudo-code:

$$\texttt{scope}_q(\texttt{while}(i < 100)(\texttt{if } i\%2 = 0 \texttt{ then } P \texttt{ else } Q), \mathcal{F}H, \mathcal{T}H, \mathcal{C}H)$$

Scope q contains a loop which executes 100 cycles. Even cycles execute process P whereas odd cycles execute process Q. If scope q is reached by a fault, in order to correctly recover its activities, it has to remember their exact sequence and recover them in the desired order. One can use some bookkeeping variables, but as far 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 to update handlers as far as the computation progresses. We associate to each scope a function $\mathcal{H}$ associating fault handlers to fault names and termination and compensation handlers to scope names.

Technically, dynamic handling is addressed by an *installing* primitive, $\text{inst}(\mathcal{H})$, which updates the current handler function with $\mathcal{H}$. Thus, the handler code can be updated depending on the current state of the scope. The example above could be rewritten by exploiting the dynamic handler mechanism as:

$\texttt{scope}_q(\texttt{while}(i < 100)$
$\quad \texttt{if } i\%2 = 0 \texttt{ then } P; \text{inst}([q \mapsto P'; cH])$
$\quad \texttt{else } Q; \text{inst}([q \mapsto Q'; cH])$
$, \mathcal{H})$

where cH allows to recover the previously installed handler with the same name. In this case when P is executed the termination handler is updated with process P', which specifically recovers process P ($\text{inst}([q \mapsto P'; cH])$), whereas if Q is executed the termination handler is updated with Q'. When reached by a fault, scope q executes the last installed termination handler, thus recovering the whole sequence of activities. Different strategies can easily be programmed. Notice 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 compensation would not be up-to-date. This can be obtained in the dynamic scenario by giving precedence to the inst primitive, while the same can not be done for the bookkeeping code needed in the static framework.

In this scenario, when a scope successfully terminates, the last defined termination handler becomes its compensation handler. It is worth noting that there is no ambiguity between the two handlers since they are triggered in different ways. Termination handler is executed by the scope itself which stops its normal code (in Fig. 2 scope B stops and executes its termination handler), whereas the compensation handler is always executed by the enclosing scope (in Fig. 2 the fault handler of the scope A executes the compensation of scope C) . This allows also to trivially simulate the static approach with the dynamic one: the construct $\texttt{scope}_q(P, \mathcal{FH}, \mathcal{TH}, \mathcal{CH})$ can be simply rephrased as $\texttt{scope}_q(\text{inst}(\mathcal{FH}); \text{inst}(\mathcal{TH}); P; \text{inst}(\mathcal{CH}))$ in which the fault and termination handlers are installed before the execution of the activity, and the compensation handler at the end.

3 SOCK: the service behavior layer

In order to present a formalization of our approach we extend the syntax and semantics of SOCK [GLG+06] with the primitives described in the previous section. SOCK is a calculus for modeling service oriented systems, inspired by WSDL [Wor] and BPEL [OAS]. In fact, its primitives include both uni-directional and bi-directional WSDL communication patterns, control primitives from imperative languages, and parallel composition and signals from concurrent languages. A distinctive trait of SOCK is that its definition is structured in three different layers: (i) the service behaviour 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 behaviour layer we refer to Appendix A for the detailed description of the other layers. Here instead we present the service behaviour layer, starting from the standard part and adding then the fault and compensation primitives.

3.1 Syntax

We consider the following (disjoint) sets: Sig, ranged over by s, for signal names, 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 $Scopes \cup \{\perp\}$. Finally, we use u to range over $Faults$, $Scopes$ and $\{\perp\}$.

We use the notation $\vec{k} = \langle k_0, k_1, ..., k_i \rangle$ for vectors (of variables, values, ...).

Let SC be the set of service behavior processes, ranged over by $P, Q, \ldots$. We use $\mathcal{H}$ to denote a function from $Faults$ and $Scopes$ to processes extended with $\perp$, i.e. $\mathcal{H} : Faults \cup Scopes \rightarrow SC \cup \{\perp\}$. 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, we use σ to range over substitutions and use standard notation $[\vec{v}/\vec{x}]$ for them.

The syntax of processes is defined in Table 1.

The first part contains the standard constructs. Here $\mathbf{0}$ is the null process. Outputs can be signals $\overline{s}$, notifications $\overline{o}@z(\vec{y})$ or solicit-responses $\overline{o_r}@z(\vec{y}, \vec{x}, \mathcal{H})$ where $s \in Sig$, $o \in \mathcal{O}$, $o_r \in \mathcal{O}_R$ 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 manage error recovery during the solicit-response. Dually, inputs can be input signals s, 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 $[\![e]\!]$ is used to evaluate a closed expression e). For the sake of brevity, we do not present the syntax of expressions, we just assume that they include at least the arithmetic and boolean operators,

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

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

Table 1. Service behavior syntax

values in Val and variables. If-then-else is denoted by $\chi?P : Q$, where χ is a boolean expression. $P; Q$ and $P|Q$ represent sequential and parallel composition respectively, whereas $\sum_{i \in W} \epsilon_i; P_i$ is the non-deterministic choice restricted to be input-guarded. Such a restriction is due to the fact that we are not interested in modeling internal non-determinism in service behavior. Our calculus indeed aims at supplying a language for designing service behaviours where designers have full control of the internal machinery and the only non-predictable choices are those driven by the reception of external messages. Also, *while* χ *do* (P) is the construct to model guarded iteration.

The last five cases concern faults and compensations and are the main novelty. We consider two kinds of handlers: *fault handlers* to deal with an internal fault and *termination/compensation handlers* to deal with compensations of an activity in case of an external fault. Handlers are installed by 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 throw(f) and comp(q) respectively raises fault f and asks to compensate scope q. Finally, cH allows to refer to the previously installed handler during an handler update.

Well-formedness rules: informally, the terms comp(q) and cH can occur only within a fault handler or a termination handler, and q can only be a child of the enclosing scope. Also, for each inst($\mathcal{H}$), $\mathcal{H}$ is undefined on all scope names

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

Table 2. Extended service behavior syntax

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.

3.2 Semantics

In order to define the semantics we exploit the extended syntax in Table 2. There $\{P : \mathcal{H} : u\}_{q_\perp}$ 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 $u \neq \perp$). Also, the scope name may be $\perp$, denoting that the scope has been killed and is now in a zombie state (i.e., it can no more terminate with success nor throw faults but it may e.g. wait for some incoming messages). Also, $o_r(\vec{x}, \mathcal{H})$ is used to wait for the response in a solicit-response interaction. $\mathcal{H}$ is installed if and only if 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, $\overline{o_r}!f@l$ is used to notify a partner when a fault has happened.

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 Appendix A) we use mainly 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 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. The action kinds can be divided as $Kind = In \cup Out \cup \{\tau\}$ where $In = \{o(\vec{v}), \uparrow o_r(\vec{v})@l, \downarrow o_r(\vec{v}), o_r(f)\}$ and $Out = \{\overline{o}(\vec{v})@l, \uparrow \overline{o_r}(\vec{v})@l, \downarrow \overline{o_r}(\vec{v})@l, \overline{o_r}(f)@l\}$. Furthermore we use the following unstructured actions: $\{s, \bar{s}, th(f), cm(q, P), inst(\mathcal{H})\}$.

Definition 1 (Service behaviour layer semantics). *We define* $\rightarrow \subseteq SC \times Act \times SC$ *as the least relation which satisfies the axioms and rules of Table 3*

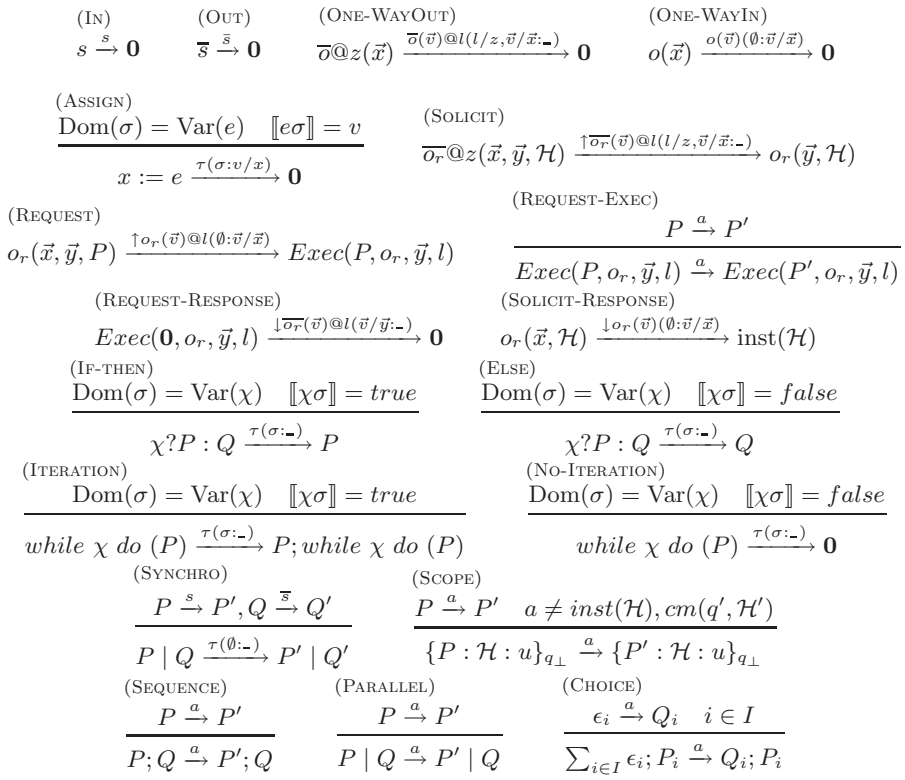

Table 3. Rules for service behavior layer, $a \neq th(f)$

and 5 and closed w.r.t. structural congruence $\equiv$, which is 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 In, Out and Synchro allow CCS-like synchronization between parallel processes of the same service behavior. 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 handlers are installed only in case of success of the remote activity, and in particular that they are not deleted by other faults. The response is dealt with by rules Request-Response and Solicit-Response. The computation of the response is executed by rule Request-

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

Table 4. Structural congruence

Exec. This is necessary since in case of failure the ongoing computation should be treated in a particular way: the partner should be notified of the failure. 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. Notice that we use operator $\oplus$, defined as follows, for updating the scope function:

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

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. $\mathrm{cmp}(\mathcal{H}) = \mathcal{H}|_{Scopes}$.

Fault and compensation handlers are installed in the nearest enclosing scope using rules AskInst and Install. According to rule Scope-Success, when a scope successfully terminates, all its compensation handlers are propagated. This allows to compensate a terminated activity. A compensation execution is asked by rule Compensate. Notice that here 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 to be executed is stored in the third component of the scope construct, to be executed only after a few activities have been performed. These are individuated by the rules for fault propagation (rules Throw-Sync, Throw-Seq, ReThrow, Throw-RExec) and by the partial function *killable*. This function has a double aim. On one side it guarentees 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 his 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

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

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

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

(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

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

Table 6. Function *killable*

partner (this is where the f parameter is needed). Finally, a receive waiting for the answer of a solicit-response is preserved, thus preserving the pattern of communication. The $\langle P \rangle$ operator (described by rule PROTECTION) is used in order to guarentee 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 (as in BPEL *rethrow*). Notice that a scope that has handled an internal fault can still terminate 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 terminate with success, nor throw faults. This is denoted by the $\perp$ that substitutes the scope name and obtained by rules SCOPE-FAIL and IGNORE-FAULT. Note that 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 give some more insights on the features of the proposed semantics, both via examples and by formal statements (whose proofs are collected in Appendix B).

As already said, the main building blocks for error handling are scopes. A scope can either terminate successfully (possibly after some internal error recovery) or not. For instance $\{throw(f)\}_q$ will terminate unsuccessfully (the transition is derived using rule RETHROW). However,

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

manages its internal fault by executing $HANDLE$, and then terminates with success. In this way compensation $COMP$ will be available if outer scopes need to invoke it.

In general, an isolated scope either terminates with success or with failure.

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 terminates successfully*[1]*: $P_i \xrightarrow{inst(\mathcal{H})} \mathbf{0}$ 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 terminate 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: $P_n = \{P' : \mathcal{H} : u\}_q$.*

In a complex application, a scope can also be terminated because of an external fault. In this case termination will not be successful.

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 terminate 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'.*

Notice 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.

Notice that, even in scopes that do not terminate with success, solicit-response patterns are always preserved: in case of faults in the caller the answer is waited for and kept into account during error recovery, while in case of fault in the callee the fault is notified to the caller.

Let us considerthe scenario discussed in the introduction: a remote activity pay_r to be compensated if and only if it has been successfully executed. We consider the most difficult case, namely when a fault f occurs at the client-side:

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

Let us consider the different cases in which fault f may occur:

- before the solicit-response is started: in this case the solicit-response will never start and the executed termination handler will be empty as required;
- after the solicit but before the response: handler q will be scheduled for execution, but the response is waited for, thus when q is executed its value is empty if a fault has been received by the payment service (i.e. pay_r has not succeeded), $UNDO$ otherwise;

[1] 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.

– after scope q has terminated its activity: in this case the compensation handler for q has been propagated upstream (to a scope not represented) and it is available, 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.

More in general, the following proposition guarentees that the answer of a solicit-response is always waited for.

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: $P_n \xrightarrow{\downarrow o_r(\vec{v'})(\emptyset:\vec{v'}/\vec{x'})} P'$.*

Symmetrically, a request-response will never terminate its execution without notifying the caller, either sending him a result or notifying him a fault. Before stating the proposition we need an auxiliary definition, to define when a part of a process is being executed.

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

$$
\begin{array}{lll}
C[\bullet] = \bullet & & (1)\\
\quad C[\bullet]; Q & & (2)\\
\quad C[\bullet] | Q & & (3)\\
\quad Q | C[\bullet] & & (4)\\
\quad \{C[\bullet] : \mathcal{H} : u\}_{q_\perp} & & (5)\\
\quad Exec(C[\bullet], o_r, \vec{y}, l) & & (6)\\
\quad \langle C[\bullet] \rangle & & (7)\\
\quad Q; C[\bullet] \text{ if } Q \equiv \mathbf{0} & & (8)
\end{array}
$$

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, since our handlers are installed dynamically, it is important that handlers are installed as soon as they are available, and in particular before any fault is triggered. This is guarenteed by the following proposition.

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

5 Automotive scenario

This section is devoted to the discussion of the automotive scenario [WCG+06], which has been chosen as case study inside the EU Project SENSORIA [Eur].

In the scenario, a severe 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 directly to the location of the broken car.

While the whole system can be modeled in SOCK, we present here the most significant part of the behavior of the car service system, as it suffices to show the suitability of the new compensation and fault handling mechanisms.

The car orchestrator CAR_P contains three modules: R_P interacts with the car rental service, G_P interacts with the garage service and T_P interacts with the towing truck service. 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 follows:

$$
\begin{array}{l}
CAR_P ::= \{ \\
\quad \text{inst}([fG \mapsto \text{comp}(r), fT \mapsto \text{comp}(g) \mid \text{comp}(r)]); \\
\quad (\,(G_P; T_P) \mid R_P\,) \\
\}_{main}
\end{array}
$$

$$
\begin{array}{l}
G_P ::= \{ \\
\quad \overline{book}@G(failure, \langle G_{acc}, G_{id}\rangle, [fB \mapsto \overline{revbook}@G(G_{id}); \text{throw}(fG)]); \\
\quad \overline{pay}@B(\langle CAR_{acc}, G_{acc}, G_{id}\rangle, G_{payid}, [g \mapsto \overline{revbook}@G(G_{id}) \mid \overline{revpay}@B(G_{payid})]) \\
\}_g
\end{array}
$$

$$
\begin{array}{l}
T_P ::= \{ \\
\quad \overline{book}@T(\langle CAR_{coords}, G\rangle, \langle T_{acc}, T_{id}\rangle, [fB \mapsto \overline{revbook}@T(T_{id}); \text{throw}(fT)]); \\
\quad \overline{pay}@B(\langle CAR_{acc}, T_{acc}, T_{id}\rangle, T_{payid}, \emptyset) \\
\}_t
\end{array}
$$

$$
\begin{array}{l}
RHandler_P ::= \overline{book}@R(CAR_{coords}, \langle R_{acc}, R_{id}\rangle, \emptyset); Rpay_P \\
Rredirect_P ::= \overline{redirect}@R(\langle R_{id}, CAR_{coords}\rangle, R_{id}, \emptyset) \\
Rpay_P ::= \overline{pay}@B(\langle CAR_{acc}, R_{acc}, R_{id}\rangle, R_{payid}, \emptyset) \\
Rrevbook_P ::= \overline{revbook}@R(R_{id}) \\
Rrevpay_P ::= \overline{revpay}@B(R_{payid})
\end{array}
$$

$$
\begin{aligned}
&R_P ::= \{ \\
&\quad \underline{\text{inst}}([fR \mapsto \mathbf{0}, fB \mapsto Rrevbook_P; \text{inst}([r \mapsto \mathbf{0}]), r \mapsto RHandler_P]); \\
&\quad \overline{book}@R(G_{coords}, \langle R_{acc}, R_{id} \rangle, [r \mapsto Rredirect_P; Rpay_P]); \\
&\quad \overline{pay}@B(\langle CAR_{acc}, R_{acc}, R_{id} \rangle, R_{payid}, [r \mapsto Rredirect_P, fR \mapsto Rrevpay_P]); \\
&\quad \text{inst}([r \mapsto \{\text{inst}([fR \mapsto Rrevpay_P]); cH]\}_{rc}) \\
&\}_r
\end{aligned}
$$

Module G_P is a scope containing the solicit-response invocations needed, respectively, for the booking of the garage and its related payment. 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, except for the fact that 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 interruptions caused by a fault raised by 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 $RHandler_P$). Instead, if the rented car has already been booked, 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 invocation of the booking and the invocation 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 in order 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 service invocation pattern and the mechanisms for fault and compensation handling that are usually provided by service orchestration languages. The most relevant language which combines both aspects is BPEL, the de-facto standard for Web Services orchestration. 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. Another relevant difference is that dynamic handler installation permits to avoid a syntactic distinction between the termination and the 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 [**?**], webπ [**?**] and cJoin [**?**], all featuring statically defined 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 be used to 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 [**?**], cCSP [**?**] and Sagas [**?**], 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 also the request-response pattern, but we think that it can be applied to many other different frameworks, from basic theoretical calculi such as π-calculus to applied frameworks such as BPEL. In particular, it would be interesting to apply these ideas to other calculi for service oriented computing such as COWS [LPT07], SCC [BBC$^+$06] and SSCC [LVMR07]. We plan also to further validate our primitives by adding them to JOLIE [**?**,Ope], 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 [BGG$^+$05] or [CHY07].

References

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

[BBC+06] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos, and G. Zavattaro. SCC: A Service Centered Calculus. In *Proc. of WS-FM'06*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–56. Springer-Verlag, 2006.

[BGG+05] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration: A synergic approach for system design. In *Proc. of ICSOC'05*, volume 3826 of *Lecture Notes in Computer Science*, pages 228–240, 2005.

[CHY07] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *Proc. of ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer-Verlag, 2007.

[Eur] European Integrated Project Sensoria. Public web site. http://www.sensoria-ist.eu/.

[GLG+06] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A calculus for service oriented computing. In *Proc. of ICSOC'06*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer-Verlag, 2006.

[LPT07] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *Proc. of ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer-Verlag, 2007.

[LVMR07] I. Lanese, V. Vasconcelos, F. Martins, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *Proc. of SEFM'07*, 2007. To appear.

[MGLZ07] F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. JOLIE: A Java Orchestration Language Interpreter Engine. In *Proc. of CoOrg'06*, volume 181 of *Electronic Notes in Theoretical Computer Science*, pages 19–33. Elsevier, 2007.

[OAS] OASIS. *Web Services Business Process Execution Language Version 2.0, Working Draft.* http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf.

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

[WCG+06] M. Wirsing, A. Clark, S. Gilmore, M. Holzl, A. Knapp, N. Koch, and A. Schroeder. Semantic-based development of service-oriented systems. In *Proc. of FORTE'06*, volume 4229 of *Lecture Notes in Computer Science*, pages 24–45. Springer-Verlag, 2006.

[Wor] World Wide Web Consortium. *Web Services Description Language (WSDL) 1.1.* http://www.w3.org/TR/wsdl.

A Higher layers of SOCK

We present here the syntax and semantics of the higher layers of SOCK. While these are necessary to complete the definition of the semantics, they are not fundamental to understand the new fault and compensation mechanisms.

A.1 Service engine calculus

In a service engine, all the executed sessions of a service behavior are joined by a state and a correlation set. Furthermore, a service engine always executes sessions by following the specifications defined within the service declaration.

The service engine calculus syntax is:

$$I ::= (P, \mathcal{S}) \mid \quad I \,|\, I \qquad Y ::= c \triangleright P[I]$$

where P is a service behavior process, $\mathcal{S}$ is a state and c is a correlation set, i.e. a subset of Var.

A state is for us a substitution of values for variables. Given a state $\mathcal{S}$ and a substitution σ we say that $\mathcal{S}$ satisfies σ, written $\mathcal{S} \vdash \sigma$, if σ is a subset of $\mathcal{S}$. We also write $\mathcal{S}(x) = \bot$ when x is undefined in state $\mathcal{S}$.

Semantics. The lts rules for service engine state layer are as following.

$$\text{(Engine-State 1)}\quad \frac{P \xrightarrow{\iota(\sigma:\vec{v}/\vec{x})} P', \mathcal{S} \vdash \sigma, \iota \neq \tau}{(P, \mathcal{S}) \xrightarrow{\iota(\vec{v}/\vec{x}:\mathcal{S}(\vec{x}))} (P', \mathcal{S} \oplus [\vec{v}/\vec{x}])} \qquad \text{(Engine-State 2)}\quad \frac{P \xrightarrow{\iota(\sigma:\_)} P', \mathcal{S} \vdash \sigma}{(P, \mathcal{S}) \xrightarrow{\iota} (P', \mathcal{S})}$$

$$\text{(Engine-State 3)}\quad \frac{P \xrightarrow{\tau(\sigma:\vec{v}/\vec{x})} P', \mathcal{S} \vdash \sigma}{(P, \mathcal{S}) \xrightarrow{\tau} (P', \mathcal{S} \oplus [\vec{v}/\vec{x}])} \qquad \text{(Engine-State 4)}\quad \frac{P \xrightarrow{\iota} P', \iota \in \{th(f), inst(\mathcal{H})\}}{(P, \mathcal{S}) \xrightarrow{\iota} (P', \mathcal{S})}$$

Rule Engine-State 1 verifies that the condition σ on the state is satisfied and updates it with $[\vec{v}/\vec{x}]$. The old values are tracked in the label since they are needed to check correlation of messages. The second rule is simpler, since it deals with actions that do not update the state and do not require correlation. The third one deals with assignments. Rule Engine-State 4 treats faults or compensation installations that reach service engine.

When an input is received by a service engine, it is possible that several sessions are waiting on the same operation. The session chosen for message delivery depends on the values of the correlated variables. Given two values v and w, a variable x and a correlation set c, v is correlated to x coherently with c, written $v/x \vdash_c w$, if any of the following conditions hold:

- the variable x belongs to c and its actual value is $w = v$,
- the variable x belongs to c and $w = \bot$,
- the variable x does not belong to c.

Rules for service engine correlation lts layer are as follows.

$$\text{(Correlated)}\quad \frac{I \xrightarrow{\iota(\vec{v}/\vec{x}:\vec{w})} I', \vec{v}/\vec{x} \vdash_c \vec{w}}{I \xrightarrow{\iota,c} I'} \qquad \text{(NotCorrelated)}\quad \frac{I \xrightarrow{\iota} I'}{I \xrightarrow{\iota,c} I'} \qquad \text{(Par)}\quad \frac{I \xrightarrow{\iota,c} I'}{I|I'' \xrightarrow{\iota,c} I'|I''}$$

The first rule ensures that an input is received by a correlated session, while the second one deals with actions that need no correlation. In this case any correlation set is fine. The last rule deals with parallel composition.

Service declaration contains all the necessary information for executing sessions. Since this part is not central for this work we consider just the case of sessions executed in parallel with non persistent state and refer to [GLG+06] for a description of the other possibilities.

$$\text{(Exec)}\ \frac{I \xrightarrow{\iota,c} I'}{c \triangleright P[I] \xrightarrow{\iota} c \triangleright P[I']}$$

$$\text{(Spawn)}\ \frac{(P, \mathcal{S}_\perp) \xrightarrow{\iota,c} (P', \mathcal{S}),\ \nexists \mathcal{S}_i \in \text{extr}(I).(P, \mathcal{S}_i) \xrightarrow{\iota,c} (P', \mathcal{S}'_i), \iota \in In}{c \triangleright P[I] \xrightarrow{\iota} c \triangleright P[I|(P', \mathcal{S})]}$$

In rule Spawn $\mathcal{S}_\perp$ is the state undefined on all variables and $\text{extr}(I)$ is a function extracting all states occurring in I.

The first rule allows to execute an existing session, while the second spawns a new session provided that an input that cannot be handled by the available sessions is received.

A.2 Services system calculus

The services system calculus allows to compose different engines into a system. The service engines are composed in parallel and equipped with a location that allows us to univocally distinguish them. The calculus syntax is:

$$E ::= Y@l \ \mid \ E \parallel E$$

A service engine system E can be a located service engine $Y@l$ or a parallel composition of them. The semantics is defined by the rules in Table 7 and closed w.r.t. the structural congruence $\equiv$ therein.

Rule Lift propagates an action to a located engine. Rule NormalSync allows to synchronize an output with the corresponding input (according to the predicate *compl*), checking that the location of the receiving process is the desired one. Rule Solicit-RequestSync additionally checks the correctness of the guess in the input label about the location of the invoking process. Finally rule Par-Ext deals with parallel composition.

B Proofs

In this section we show the proofs of the properties stated in Section 4. Some auxiliary lemmas are presented too.

$$\frac{Y \xrightarrow{\iota} Y'}{Y@l \xrightarrow{\iota} Y'@l}\ (\textsc{Lift}) \qquad \frac{Y@l' \xrightarrow{\lambda@l} Y'@l' \ ,\ Z@l \xrightarrow{\lambda'} Z'@l \ ,\ compl(\lambda,\lambda')}{Y@l' \parallel Z@l \xrightarrow{\tau} Y'@l' \parallel Z'@l}\ (\textsc{NormalSync})$$

$$\frac{E_1 \xrightarrow{\iota} E_1'}{E_1 \parallel E_2 \xrightarrow{\iota} E_1' \parallel E_2}\ (\textsc{Par-Ext}) \qquad \frac{Y@l' \xrightarrow{\uparrow\overline{o_r}(\vec{v})@l} Y'@l' \ ,\ Z@l \xrightarrow{\uparrow o_r(\vec{v})@l'} Z'@l}{Y@l' \parallel Z@l \xrightarrow{\tau} Y'@l' \parallel Z'@l}\ (\textsc{Solicit-RequestSync})$$

where $compl(\overline{o}(v), o(v)), compl(\downarrow \overline{o_r}(v), \downarrow o_r(v)), compl(\overline{o_r}(f), o_r(f))$.

$$E_1 \parallel E_2 \equiv E_2 \parallel E_1 \qquad E_1 \parallel (E_2 \parallel E_3) \equiv (E_1 \parallel E_2) \parallel E_3$$

Table 7. Rules for services system lts layer

Proof (of Proposition 1). The proof is by induction on n. The basic case is trivially true. To prove the inductive case we show that if $P' = \{P : \mathcal{H} : u\}_q$ with $u \notin Scopes$ and $q \neq \bot$ and $P' \xrightarrow{a} P''$ then either $a = inst(\mathcal{H})$ or $a = th(f)$ or P'' has the same structure. There are different cases. If the transition has been derived using rules Scope, Install, Scope-Handle-Fault, Catch-Fault or Compensation then the thesis follows by inductive hypothesis. If the transition has been derived using rules Scope-Success or ReThrow then the thesis follows directly. Note that rules Scope-Fail and Ignore-Fault can not be applied because by hypothesis $q \neq \bot$. Rule Scope-Handle-Term can not be applied either since by hypothesis $u \notin Scopes$.

We still have to prove that if $P_i \xrightarrow{th(f)} P_{i+1}$ then no other $th(f')$ action will be generated in the computation. Notice that the only rule that causes $P_i \xrightarrow{th(f)} P_{i+1}$ is ReThrow. Then $P_{i+1} = \langle\{P'' : \mathcal{H} : u\}_\bot\rangle$. We can prove the thesis by induction on the length of the remaining computation. Each transition is derived using one of the rules for scopes followed by rule Protection. Only rules Scope, Install, Scope-Handle-Fault, Scope-Fail, Scope-Handle-Term, Catch-Fault, Ignore-Fault and Compensation can be applied (in particular, rule ReThrow is not applicable). Since all of them but Scope-Fail preserve the structure of the process and Scope-Fail terminates the computation, and no rule generates the label $th(f')$ or $inst(\text{cmp}(\mathcal{H}))$ then the thesis follows.

Proof (of Proposition 2). We have $P = \langle\{killable(P', f) : \mathcal{H} : q\}_q\rangle$ for $q \neq \bot$. We will prove the thesis by induction on the nesting of scopes in P'. We will prove both the basic case and the inductive case by induction on the length of the computation. More precisely we will prove that each process of the form (i) $\langle\{Q : \mathcal{H} : q\}_q\rangle$ where Q is a derivative of $killable(P', f)$ for some P' and $q \neq \bot$ or (ii) of the form $\langle\{Q : \mathcal{H} : u\}_\bot\rangle$ will never terminate with success nor throw faults and will always move to a process of one of the two forms above. The transition is derived by applying rule Protection to a rule for scope. For case (i) the applied rule is Scope, Install, Compensation, Scope-Handle-

Term, Catch-Fault or ReThrow. In the first three cases the thesis follows by inductive hypothesis. If the applied rule is Scope-Handle-Fault instead the thesis follows directly. Actually the last two cases never apply since a derivative of $killable(P', f)$ never throw faults. This can be proved by induction on the structure of P'. The only difficult case is when P' is a scope, but here we can exploit the induction on the nesting of scopes. For case (ii) the applied rule is Scope, Install, Scope-Handle-Fault, Scope-Fail, Scope-Handle-Term, Catch-Fault, Ignore-Fault, Compensation. In all cases but Scope-Fail the thesis follows by inductive hypothesis. For Scope-Fail it follows directly.

Lemma 1. *$P \equiv C[\bullet]$ iff there exists an enabling context $C'[\bullet]$ such that $P = C'[\bullet]$.*

Proof. By hypothesis we know that $P =_{\text{ax}} P_1 =_{\text{ax}} \ldots =_{\text{ax}} P_n = C'[\bullet]$ where each $=_{\text{ax}}$ is an application of an axiom of structural congruence. If we prove the thesis for $n = 1$ then the thesis in the general case follows by induction. All the axioms can be applied either to the subterms of the productions denoted by Q in Definition 2 or to the operators. In the first case the thesis follows trivially since it is enough to choose an enabling context with a suitable Q' instead of Q (also for the last production since if $Q \equiv \mathbf{0}$ then also $Q' \equiv \mathbf{0}$). For the second case let us do a case analysis according to the used axiom.

Commutativity of $|$: this can be applied only to the cases 3 and 4, and this corresponds to use the other case;

$P|\mathbf{0} \equiv P$**:** this corresponds either to skip one step in the derivation which uses case 3, or to introduce such a step;

Associativity of $|$: there are a few cases. Either one step uses cases 3 or 4 with $Q = Q_1|Q_2$, and in this case we have to use two separate steps for Q_1 and Q_2. The opposite case is also possible. Finally, this may correspond to exchange the order of two subsequent steps that use the cases 3 and 4;

$\mathbf{0}; P \equiv P$**:** this corresponds either to skip one step in the derivation which uses the last case, or to introduce such a step;

$\langle \mathbf{0} \rangle \equiv \mathbf{0}$**:** this can not be applied outside the Q subterms.

Lemma 2. *If $P = C[o_r(\vec{x}, \mathcal{H})]$ or $P = C[\langle o_r\langle \vec{x}, \mathcal{H}\rangle\rangle]$ for some enabling context $C[\bullet]$ then $killable(P, f)$, if defined, has either the form $C'[o_r(\vec{x}, \mathcal{H})]$ or $C'[\langle o_r\langle \vec{x}, \mathcal{H}\rangle\rangle]$.*

Proof. The proof is an easy induction on the structure of $C[\bullet]$.

Proof (of Proposition 3). The proof is by induction on n. One can prove by induction on the derivation of $P \xrightarrow{a_1} P_1$ that $P = C[\overline{o_r}@z(\vec{x}, \vec{y}, \mathcal{H})]$ for some enabling context $C[\bullet]$. Also, $P_1 = C[o_r(\vec{x}, \mathcal{H})]$. One can prove by structural induction on enabling contexts $C[\bullet]$ that if $P' = C[o_r(\vec{x}, \mathcal{H})]$ or $P' = C[\langle o_r\langle \vec{x}, \mathcal{H}\rangle\rangle]$ then $P' \xrightarrow{\downarrow o_r(\vec{v'})(\emptyset:\vec{v'}/\vec{x'})} P''$ for some P''. Thus the basic case is satisfied. We prove now that each P' with the structure above will either do the receive action or move to a process with the same structure. The thesis will follow. The proof is by structural induction on $C[\bullet]$. Notice that because of Lemma 1 we have no need to consider structural congruence.

$C[\bullet] = \bullet$**:** the thesis holds since both $o_r(\vec{x}, \mathcal{H})$ and $\langle o_r\langle \vec{x}, \mathcal{H}\rangle\rangle$ have as only possible actions the desired receive transition.

$C[\bullet] = C'[\bullet]; Q$**:** the transition can be derived either using rule SEQUENCE or using rule THROW-SEQ: in both the cases the thesis follows by inductive hypothesis.

$C[\bullet] = C'[\bullet]|Q$ **and symmetric:** there are a few cases to consider. If the transition is derived using rules SYNCHRO or PARALLEL then the thesis follows by inductive hypothesis. If the transition has been derived using rule THROW-SYNC there are two cases: either the throw action is from $C'[\bullet]$ and inductive hypothesis can be applied, or it is from Q. In this case the thesis follows from Lemma 2.

$C[\bullet] = \{C'[\bullet] : \mathcal{H} : u\}_{q_\perp}$**:** again different cases have to be considered. If the transition has been derived using rules SCOPE, INSTALL, CATCH-FAULT, IGNORE-FAULT, RETHROW and COMPENSATION then the thesis follows by inductive hypothesis. Note that rules SCOPE-SUCCESS, SCOPE-FAIL, SCOPE-HANDLE-FAULT and SCOPE-HANDLE-TERM instead can not be applied since the left hand side does not satisfy the hypothesis.

$C[\bullet] = Exec(C'[\bullet], o_r, \vec{y}, l)$**:** the transition has been derived using either rule REQUEST-EXEC or rule THROW-REXEC. In both the cases the thesis follows by inductive hypothesis.

$C[\bullet] = \langle C'[\bullet]\rangle$**:** the transition has been derived using rule PROTECTION, and the thesis follows by inductive hypothesis.

$C[\bullet] = Q; C'[\bullet]$ **with** $Q \equiv \mathbf{0}$**:** this has no derivable transitions.

Lemma 3. *If* $P' = C[Exec(P, o_r, \vec{y}, l)]$ *or* $P' = C[\langle \overline{o_r}!f@l\rangle]$ *for some enabling context* $C[\bullet]$ *then* $killable(P, f)$*, if defined, has one of the two forms above too.*

Proof. The proof is an easy induction on the structure of $C[\bullet]$.

Proof (of Proposition 4). The proof is by induction on n. One can prove by induction on the derivation of $P \xrightarrow{a_1} P_1$ that $P = C[o_r(\vec{x}, \vec{y}, P)]$ for some enabling context $C[\bullet]$. Also, $P_1 = C[Exec(P, o_r, \vec{y}, l)]$. One can prove by structural induction on enabling contexts $C[\bullet]$ that if (i) $P' = C[Exec(P, o_r, \vec{y}, l)]$ or (ii) $P' = C[\langle \overline{o_r}!f@l\rangle]$ then the thesis is satisfied. Thus the basic case is satisfied. We prove now that each P' with the structure above will either do the receive action or move to a process with the same structure. The thesis will follow. The proof is by structural induction on $C[\bullet]$. Notice that because of Lemma 1 we have no need to consider structural congruence.

$C[\bullet] = \bullet$**:** for case (i) there are three possible subcases. If the transition has been derived using rule REQUEST-EXEC then the thesis follows by inductive hypothesis. If it has been derived using rule REQUEST-RESPONSE then the thesis follows directly since the response is sent. If it has been derived using rule THROW-REXEC then the thesis follows since $P' | \langle \overline{o_r}!f@l\rangle$ has the structure in (ii). For case (ii) the thesis follows since the only possible transition has label $o_r(f)(\emptyset : \_)$.

$C[\bullet] = C'[\bullet]; Q$**:** the transition can be derived either using rule SEQUENCE or using rule THROW-SEQ: in both the cases the thesis follows by inductive hypothesis.

$C[\bullet] = C'[\bullet]|Q$ **and symmetric:** there are a few cases to consider. If the transition is derived using rules SYNCHRO or PARALLEL then the thesis follows by inductive hypothesis. If the transition has been derived using rule THROW-SYNC there are two cases: either the throw action is from $C'[\bullet]$ and inductive hypothesis can be applied, or it is from Q. In this case the thesis follows from Lemma 3.

$C[\bullet] = \{C'[\bullet] : \mathcal{H} : u\}_{q_\perp}$**:** again different cases have to be considered. If the transition has been derived using rules SCOPE, INSTALL, CATCH-FAULT, IGNORE-FAULT, RETHROW and COMPENSATION then the thesis follows by inductive hypothesis. Note that rules SCOPE-SUCCESS, SCOPE-FAIL, SCOPE-HANDLE-FAULT and SCOPE-HANDLE-TERM instead can not be applied since the left hand side does not satisfy the hypothesis.

$C[\bullet] = Exec(C'[\bullet], o_r, \vec{y}, l)$**:** the transition has been derived using either rule REQUEST-EXEC or rule THROW-REXEC. In both the cases the thesis follows by inductive hypothesis.

$C[\bullet] = \langle C'[\bullet]\rangle$**:** the transition has been derived using rule PROTECTION, and the thesis follows by inductive hypothesis.

$C[\bullet] = Q; C'[\bullet]$ **with** $Q \equiv \mathbf{0}$**:** this has no derivable transitions.

Lemma 4. *If* $killable(P, f)$ *is defined then* $P \neq C[\text{inst}(\mathcal{H})]$ *for each enabling context* $C[\bullet]$.

Proof. By case analysis according to the definition of $killable(P, f)$.

Proof (of Proposition 5). The proof is by rule induction. Thanks to Lemma 1 we need only to prove that if $P \xrightarrow{th(f)} P'$ then $P \neq C'[\text{inst}(\mathcal{H})]$ for each enabling context $C'[\bullet]$.

The thesis holds trivially for all rules with labels different from $th(f)$. Let us consider the different cases.

Throw: the thesis holds trivially.

Protection: the case is non trivial only for $a = th(f)$. We can have $\langle P\rangle = C'[\text{inst}(\mathcal{H})]$ only if $P = C''[\text{inst}(\mathcal{H})]$, but this is impossible by inductive hypothesis.

Throw-Sync: we can have $P|Q = C'[\text{inst}(\mathcal{H})]$ only if either $P = C''[\text{inst}(\mathcal{H})]$ or $Q = C''[\text{inst}(\mathcal{H})]$. The first case is impossible by inductive hypothesis. The second case is impossible because of Lemma 4.

Throw-Seq: we can have $P; Q = C'[\text{inst}(\mathcal{H})]$ only if either $P = C''[\text{inst}(\mathcal{H})]$ or if $P \equiv \mathbf{0}$ and $Q = C''[\text{inst}(\mathcal{H})]$. The first case is impossible by inductive hypothesis. The second case is impossible since a process structural congruent to $\mathbf{0}$ has no transitions, thus the premise of the rule can not hold.

ReThrow: we can have $\{P : \mathcal{H} : u\}_q = C'[\text{inst}(\mathcal{H})]$ only if $P = C''[\text{inst}(\mathcal{H})]$, but this is impossible by inductive hypothesis.

ThrowRExec: to have $Exec(P, o_r, \vec{y}, l) = C'[\text{inst}(\mathcal{H})]$ we need $P = C''[\text{inst}(\mathcal{H})]$, but this is impossible by inductive hypothesis.