Fundamenta Informaticae 95 (2009) 73–102
DOI 10.3233/FI-2009-143
IOS Press

Dynamic Error Handling in Service Oriented Applications*

Claudio Guidi, Ivan Lanese†, Fabrizio Montesi, Gianluigi Zavattaro
Computer Science Department, University of Bologna
Mura A. Zamboni, 7, 40127 Bologna, Italy
{*cguidi, lanese, fmontesi, zavattar*}*@cs.unibo.it*

Abstract. Service Oriented Computing (SOC) allows for the composition of services which communicate using unidirectional *one-way* or bidirectional *request-response* communication patterns. Most service orchestration languages proposed so far provide also primitives for error handling based on fault, termination, and compensation handlers. Our work is motivated by the difficulties encountered in programming some error handling strategies using current error handling primitives. We propose as a solution an orchestration programming style in which handlers are *dynamically installed*. We assess our proposal by formalizing our approach as an extension of the process calculus SOCK and by proving that our formalization satisfies some expected high-level properties.

1. Introduction

Service Oriented Computing (SOC) is a distributed computing paradigm based on the notion of service, intended as a publicly available software component that provides some precise functionality. Service oriented applications are developed via composition of services: services are first selected and then appropriately "orchestrated" in order to accomplish a predefined task. By service orchestration we mean a precisely defined flow of service invocations and collection of responses. In order to support this approach to the development of applications, SOC provides languages for the description of services, mechanisms for service advertisement and discovery, protocols for service invocation, and languages for service orchestration. The most credited example of a SOC model is given by the Web Services technology: services are described using WSDL (Web Services Description Language) [39], advertised

*Research partially funded by EU Integrated Project Sensoria, contract n. 016004. Some preliminary results reported in this paper appeared in [22] and [30].
†Address for correspondence: Computer Science Department, University of Bologna, Mura A. Zamboni, 7, 40127 Bologna, Italy

and retrieved by means of UDDI (Universal Description Discovery and Integration) [32], invoked using SOAP (Simple Object Access Protocol) [38], and orchestrated exploiting WS-BPEL (Web Services Business Process Execution Language) [33].

Since both services and the network infrastructure are unreliable, orchestration languages have to provide mechanisms to deal with unexpected situations. WS-BPEL (BPEL for short), 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. Our work is motivated by the difficulties encountered in programming some specific error handling strategies using current orchestration languages. In particular, it is not easy to adjust the behavior of the handlers according to information available only at run time. The traditional approach used to solve this problem is based on bookkeeping variables that store the information required by the handlers in order to appropriately adjust their behavior. As we will discuss in the next section, this approach is not always satisfactory because there are atomicity problems in the update of the bookkeeping variables and, when the complexity of the code increases, the bookkeeping could become complex and error-prone.

Instead of using bookkeeping variables, we investigate a novel solution based on the *dynamic* installation/update of the handlers. According to this dynamic approach, when an event that could influence the behavior of the handlers occurs, instead of modifying a bookkeeping variable, we directly modify the handlers. In order to appropriately evaluate the approach, we formalize a possible semantics for dynamic error handling presenting an extension of the process calculus SOCK [23] that includes, besides the standard primitives of orchestration languages already formalized in [23], also the new primitives for dynamic error handling. We assess the proposed semantics by means of the formal proof that some expected high-level properties related to error handling are actually satisfied by the proposed extension of SOCK. Notably, we have also investigated the practical impact of our approach by extending the orchestration language JOLIE, the implementation of SOCK, according to the new semantics proposed in this paper. We have used the new version of JOLIE to experiment with dynamic error handling in the implementation of a prototype of the automotive car repair scenario described in [37].

Structure of the paper. Section 2 discusses informally our dynamic approach to error handling, comparing it with the static one. Section 3 recalls the syntax and the semantics of SOCK as presented in [23], while in Section 4 we present its extension with dynamic error handling. Section 5 assesses the proposed semantics for error handling via formal statements about properties that are satisfied by the proposed extension of SOCK. A detailed comparison with the related literature can be found in Section 6, together with some conclusive remarks.

A preliminary version of some of the results in this paper has appeared in [22].

2. Key concepts

Error handling in Service Oriented Computing generally 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 recovery. Recovery mechanisms are implemented by exploiting *handlers*, which are processes to be executed when faults occur. Handlers are defined within a scope which represents the boundary of their execution. 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. One may have a different fault handler for each possible fault. Termination handlers are automatically executed when the scope defining it must be smoothly stopped because it is reached by a fault raised by an external process. Each scope may define at most one termination handler at each time. Finally, compensation handlers are used to undo the activities of a scope after its execution has successfully finished. Compensation handlers have to be explicitly invoked during the execution of another handler. Each scope can only ask for the compensation of a child scope. Asking for the compensation of a scope that has not successfully terminated its execution has no effect.

Let us now analyze how different handlers are executed. Assume that, at runtime, a fault f is raised within a scope q. First, all the running activities enclosed in q are terminated. If any of such activities is a scope, it is terminated (thus recursively terminating the subactivities), and its termination handler is automatically executed. After the termination of subactivities has been completed, if a fault handler for f is defined in q, then it is executed. Otherwise, the fault is propagated upwards to the parent scope. Again activities inside the parent scope are terminated, and if available in the parent scope a fault handler for f is executed, otherwise the fault is propagated up along the scope hierarchy. We highlight the following concept:

> *Termination is always propagated to siblings and children scopes, and it is always completed before the enclosing scope executes the fault handler.*

Both fault handlers and termination handlers can be programmed to compensate any child scope that has successfully completed its activity before f was raised by executing the related compensation handler. Compensation handlers can recursively invoke compensation handlers of subactivities. We highlight the following concept:

> *A compensation handler is activated by means of a specific primitive, which can only be used inside a handler.*

Even if there are three different kinds of handlers there is no possibility of confusion among them, since they are responsible for different error handling activities, and they are invoked in different ways. Fault and termination handlers are both activated automatically, the first one to manage a fault coming from inside, the second one to manage a fault from outside. Compensation handlers instead have to be explicitly invoked, and they undo the activities of scopes that have already terminated (with success) their execution. Fault handlers are identified by the name of the corresponding fault, termination/compensation handlers by the name of the corresponding scope.

Fig. 1 shows an example of error handling activities. In the figure, numbers represent ordered events and *stm 1, stm 2, ...,stm n* are statements. A scope q_1 encloses a generic process P and two scopes q_2 and q_3. At 1, scope q_3 finishes successfully its execution. Thus its compensation handler becomes available for invocation. It is stored in the parent scope q_1. At 2, process P raises a fault f inside scope q_1. All activities inside q_1, i.e. process P and scope q_2, have to be terminated. We assume that scope q_2 is still executing when the fault is raised. At 3, scope q_2 is terminated and its termination handler is executed. At 4 the fault handler for f in scope q_1 is executed and, at 5, it compensates scope q_3 (assuming that the handler specifies so).

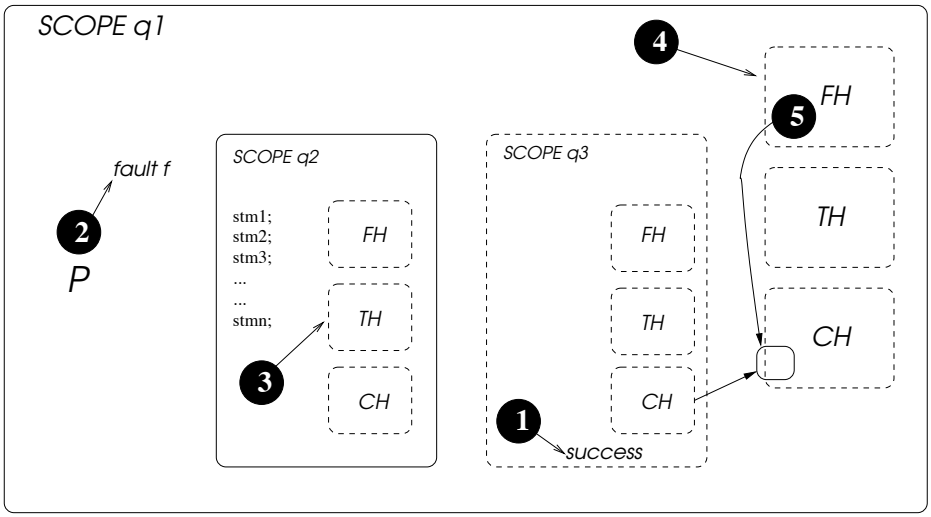

Figure 1. Handler mechanisms

2.1. Shortcomings of the static approach

Most of the approaches to error recovery in the literature, most notably the one of BPEL [33], statically associate handlers to scopes, i.e. they provide a primitive like $\texttt{scope}_q\{P, \mathcal{FH}, \mathcal{TH}, \mathcal{CH}\}$, which defines a scope with name q, executing process P, and with 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. Consider, e.g., the following pseudo-code:

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

where R is a process (running in 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 fault f, which triggers the termination 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. Atomicity problems must be taken into account too: let us introduce in the previous example an array named a which stores a 0 if the process P is executed and 1 otherwise.

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

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

If a fault f is raised between the execution of P and the updating of the array a ($a[i] = 0$), the termination handler cannot recover correctly scope q because array a does not record the last execution of P. BPEL [33] allows to solve this problem by providing a default termination handler that compensates subactivities in reverse order of completion. Thus the problem above can be solved in BPEL by enclosing P and Q into subscopes and using the default termination handler. However this kind of behavior is hard-wired into BPEL definition, and it is difficult to adapt it to different, albeit simple, behaviors. In the following section we present the dynamic approach to error handling, and we show how it can be used to execute the compensations of the different activities, e.g., in reverse order, in forward order, or in parallel. A detailed comparison with BPEL is presented in Section 6.

2.2. Dynamic approach

In order to address the problems raised by the static approach, we propose *dynamic error handling*, which allows for the updating of the handlers while the computation progresses. Dynamic handler update is addressed by a specific install primitive, $\mathrm{inst}(\mathcal{H}')$, which updates the handlers of the current scope. For simplicity, instead of having separated fault, termination and compensation handlers, we group them into the handler function $\mathcal{H}$. The handler function $\mathcal{H}$ associates fault handlers to fault names and termination and compensation handlers to scope names. Thus the scope construct becomes $\mathtt{scope}_q\{P, \mathcal{H}\}$, where q is the name of the scope, P is the process to be executed, and $\mathcal{H}$ the handler function. At the beginning, the handler function defines no handlers: they are installed dynamically during the execution of P. Assume that statement $\mathrm{inst}([f \mapsto P])$ is executed, requiring to associate handler P to fault f (in the current scope). If no handler for f was previously defined, then P becomes the new handler, otherwise the old handler is deleted and P is installed in its place. However, one may want to update the old handler without deleting it. This can be done thanks to placeholder cH, which stands for "current handler". Placeholder cH can be used only inside a handler update, and it is dynamically replaced by the old handler (with the same name). For instance, $\mathrm{inst}([f \mapsto P'; cH])$ requires to update the fault handler for f by executing P' before executing the old handler. In fact, if the old handler is P then the new handler is $P'; P$. Let us consider, as an example, the following process S defined as a scope q where a sequence of processes Q, Q', Q'' and Q''' is interleaved with four inst primitives:

0) $S ::= \mathtt{scope}_q\{$
1) $\quad Q\ ;\ \mathrm{inst}([f \mapsto P]);$
2) $\quad Q'\ ;\ \mathrm{inst}([q \mapsto F, f' \mapsto T, f \mapsto P'; cH]);$
3) $\quad Q''\ ;\ \mathrm{inst}([f \mapsto P'']);$
4) $\quad Q'''\ ;\ \mathrm{inst}([q \mapsto F'])\}$

In the following table the different columns represent the state of the handler function for the scope q after a given line has been executed.

0)	1)	2)	3)	4)
	$f \mapsto P$	$f \mapsto P'; P$	$f \mapsto P''$	$f \mapsto P''$
		$q \mapsto F$	$q \mapsto F$	$q \mapsto F'$
		$f' \mapsto T$	$f' \mapsto T$	$f' \mapsto T$

At the beginning the handler function is empty. The inst primitive defines new handlers for a specific fault or scope name if no handler for that name has been already specified (e.g., fault f at line 1 and fault

f' at line 2), whereas it replaces the current handler otherwise (fault f at lines 2 and 3). Note that in line 2 the new handler for f includes the old one, recovered using the placeholder cH. We highlight the following concept:

The install primitive updates the current handler for the specified fault or scope names.

Since our framework is concurrent, dynamically updating the handler corresponding to some activity, as allowed by our approach, requires some care to avoid race conditions. Consider, e.g., the process S above in parallel with a process R within a parent scope r. Assume also that a fault handler G for fault f is installed inside r:

$$\mathtt{scope}_r\{\mathrm{inst}([f \mapsto G]); (R \mid S)\}$$

Assume that R throws a fault f. The fault f reaches scope q in S, and since it is an external fault, the termination handler for q is executed. The termination handler for q is the process associated with the scope name q when the termination is triggered. In S the programmer has specified that at line 2 the termination handler for scope q must be updated because process Q' has been executed, and it has to be recovered in case of termination. However, if the fault f from R is raised between the execution of Q' and of $\mathrm{inst}([q \mapsto F, f' \mapsto T, f \mapsto P'; cH])$, then the termination handler is not consistent with the programmer expectations, since Q' has been executed, but the corresponding termination code will not. Such a case should be avoided, and in order to do that we execute the primitive inst with priority w.r.t. the fault processing mechanism. Thus, if Q' has been executed, the semantics of the inst primitive guarantees that handler F is installed before any fault is processed. We highlight the following concept:

The install primitive is executed with priority w.r.t. fault processing.

The ability to dynamically install handlers allows one to deal in a uniform way with termination and compensation handlers. Intuitively, the behavior of a termination handler for a scope that is about to finish its computation and of a compensation handler for the same scope when it has just finished the computation are strongly related. Thus, when a scope is terminating successfully, it is natural to define its compensation handler as the last defined termination handler. Being able to update the handlers allows the programmer to have different behaviors before and after termination, if needed (it is enough to have a handler update as last primitive in the scope). The compensation handler is stored in the parent scope. The main concept we highlight is:

A compensation handler is a termination handler promoted to the parent scope when the child scope finishes successfully.

One can 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:

$$\mathtt{scope}_q\{\mathrm{inst}([f \mapsto \mathcal{FH}]); \mathrm{inst}([q \mapsto \mathcal{TH}]); P; \mathrm{inst}([q \mapsto \mathcal{CH}])\}$$

where fault and termination handlers are installed before the execution of the activity, and the compensation handler at the end. Note that because of priority of the install primitive it is never the case that scope q is killed and its termination handler has not been installed. Thus we can rewrite the *while* example of the previous section as:

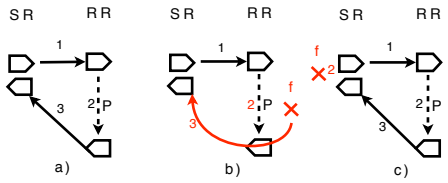

Figure 2. Solicit-response and request-response behaviors

$$R \mid \texttt{scope}_q\{i = 0\,;\, \texttt{while}(i < 100)\{ \\ i = i + 1\,;\ \texttt{if}\ i\%2 = 0\ \texttt{then}\ \{P; \mathrm{inst}([q \mapsto P'; cH])\} \\ \texttt{else}\ \{Q; \mathrm{inst}([q \mapsto Q'; cH])\}\}$$

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. For instance by changing the compensation update for P to $\mathrm{inst}([q \mapsto cH; P'])$ (and similarly for Q) one has forward order recovery, while $\mathrm{inst}([q \mapsto cH|P'])$ performs parallel recovery. Note that, thanks to the execution priority of the inst primitive, in the example above it is never the case that an execution of P has been completed and its compensation has not been installed. The same behavior 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.

2.3. Error handling inside request-response communication pattern

The request-response communication pattern is a main communication pattern in SOC, as witnessed by the fact that both WSDL [39] and BPEL [33] have dedicated operators for it. A request-response pattern allows a client to send a request message to some server, the invoked server performs some computation and sends back an answer, which is received by the client. Due to the strong relationship between the request and the response messages (stronger than the one existing when a bidirectional interaction is implemented by two one-way communications), such a kind of pattern raises some interesting issues from the point of view of error handling. In fact, errors should not spoil the pattern. We show that dynamic handler installation allows one to solve the problem in a simple way, by updating handlers upon receipt of a correct answer from the request-response.

The request-response pattern is composed by two communication primitives: the *solicit-response* (SR) at client side and the *request-response* (RR) at server side (note that the server side primitive has the same name of the whole pattern). The RR includes the code to be executed in order to produce the request answer. Note that a SR is blocked until the response message is received. Fig. 2.a represents the interplay between a SR and a RR: in step 1 the request message is sent from the SR to the RR, in step 2 the RR process P is executed and then, at the end, the response message is sent back in step 3.

Between the request message and the response one, faults may be raised, both at the SR side (from parallel processes) and at the RR one (both from parallel processes and from process P). These faults should not break the communication pattern. In particular, a response message should always be sent

back to the client, even in case of server fault; in this last case a fault message should be sent so that the fault is notified to the client. On the other side, the client should wait for the message even if interrupted because of a local fault. Figures 2.b and 2.c represent the two scenarios. In Fig. 2.b a fault f is raised on the server side during the execution of process P in the RR. In this case a fault response is immediately sent to the SR, which was waiting for the reply message. In Fig. 2.c instead a fault f is raised on the client side before the reception of the response message from the RR. In this case the response message is waited for by the SR. Only after the message has been received the fault f is managed. The two last concepts we highlight are:

> *In a request-response primitive, if a fault is raised on the server side after the request message and before the response one, a fault response is sent to the client.*
>
> *If a fault is raised inside a scope containing a solicit-response primitive that has already sent the request message, but is waiting for the response one, the response message is waited for before handling the fault.*

As we have seen, server faults are notified to the client side, where they are treated as local faults. Thus local compensations for remote activities can be executed, allowing for distributed error recovery. In this situation it is very important that the message from the server is waited for even in case of client failure, and that error recovery can be based on whether the received message is a normal answer or a faulty one. Consider, e.g., the case of a client invoking a bank service for a payment. Assume that then the client fails, and thus wants to undo the payment. If the bank fails too, then no payment has been performed and no compensation is required. If the bank operation succeeds, instead the client has to require an undo of the bank operation during its own recovery procedure. If the answer from the bank is not waited for, then the client has no information about whether recovery is needed or not, and it needs to start a new interaction with the bank asking whether the payment has been done or not. BPEL uses this last approach, which requires more effort from the programmer. See Section 6 for a detailed comparison with BPEL.

3. SOCK

To give a formal presentation of our approach, we extend the syntax and semantics of SOCK [23] with the primitives described in the previous section. SOCK is a calculus for modeling service oriented systems, inspired by WSDL [39] and BPEL [33]. Its primitives include both uni-directional (one-way) and bi-directional (request-response) WSDL communication patterns, control primitives from imperative languages, and parallel composition from concurrent languages. SOCK is the only calculus we are aware of featuring request-response as a native operator, and this allows us to directly model the interplay between error handling and request-response described in the previous section. SOCK is structured in three 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, here we give a minimal description of the other layers, not including in particular correlation sets and sessions generation, which are orthogonal features. A full description of those layers can be found in [21].

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

$P, Q, \ldots ::= \epsilon$	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
$if\ \chi\ then\ P\ else\ Q$	det. choice	$while\ \chi\ do\ (P)$	iteration
$\mathbf{0}$	null process		
$o_r(\vec{x})$	response in solicit	$Exec(P, o_r, \vec{y}, l)$	Req.-Resp. execution

Table 1. Service behavior syntax

3.1. Service behavior layer

The service behavior layer describes the actions performed by services. Actions can be operations on the state (SOCK has a state like imperative languages), or communications according to the one-way and request-response communication patterns. Basic actions can be composed using composition operators. In SOCK services are identified by the name of their operations, and by their location. Locations are managed at the services system layer. In order to model those aspects we need the following (disjoint) sets: Var, ranged over by x, y, for variables, Val, ranged over by v, for values, $\mathcal{O}$, ranged over by o, for one-way operations, and $\mathcal{O}_R$, ranged over by o_r for request-response operations. 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. Finally, we use the notation $\vec{k} = \langle k_0, k_1, ..., k_i \rangle$ for vectors.

The syntax for service behavior processes, ranged over by $P, Q, \ldots$, is defined in Table 1. We denote as SC the set of service behavior processes. $\mathbf{0}$ is the inactive process. Outputs can be notifications $\overline{o}@z(\vec{y})$ or solicit-responses $\overline{o_r}@z(\vec{y}, \vec{x})$, corresponding to the client side of one-way and request-response communication patterns respectively. A notification operation $\overline{o}@z(\vec{y})$ invokes the operation named o (with $o \in \mathcal{O}$) of a service located at the location stored in location variable z. Vector $\vec{y}$ contains the variables storing the values to be communicated during the invocation. Similarly, a solicit-response operation $\overline{o_r}@z(\vec{y}, \vec{x})$ invokes using a request-response communication pattern operation o_r (now $o_r \in \mathcal{O}_R$, since names of request-response operations are different from names of one-way operations) of a service located at the location stored in location variable z. Again $\vec{y}$ is a vector of variables storing the values to be communicated during the invocation. In addition, now $\vec{x}$ is the vector of variables that will be assigned the values received as answer of the invocation. 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, with $\vec{y}$ containing values to be sent and $\vec{x}$ containing variables that will receive the communicated values. Additionally, P is the process to be executed between the request and the response. Essentially, a notification $\overline{o}@z(\vec{y})$ will interact with a one-way $o(\vec{x})$ located at the location stored in z, and values in variables $\vec{y}$ will be sent and copied inside variables in $\vec{x}$. Consider instead a solicit-response $\overline{o_r}@z(\vec{y}, \vec{x})$ and a corresponding request-response $o_r(\vec{x}_1, \vec{y}_1, P)$. After the invocation values from $\vec{y}$ are copied into $\vec{x}_1$. Then process P is executed on the server side. Finally the answer in $\vec{y}_1$ is sent back to the client and copied into variables $\vec{x}$. Only at this point the execution at the client side can continue.

Assignment $x := e$ assigns the result of the expression e to the variable $x \in Var$ (state is local to each behavior). We do not present the syntax of expressions: we just assume that they include the arithmetic and boolean operators, values in Val and variables. We assume a function Var that given an expression e computes the set of variables in e, and we denote as $[\![e]\!]$ the evaluation of ground expression e. We use χ to range over boolean expressions. $P;Q$ and $P|Q$ are sequential and parallel composition respectively. $\sum_{i\in I} \epsilon_i; P_i$ is input-guarded non-deterministic choice: whenever one of the input operations ϵ_i (either a one-way or a request-response) is invoked, continuation P_i is executed. Iteration is modeled by $while\ \chi\ do\ (P)$.

The two last operators in Table 1 are not part of the static syntax, but they are used to give semantics to the calculus. The first one, $o_r(\vec{x})$ is used to wait for the response in a solicit-response interaction. Note that it is not a one-way, since the operation o_r is a request-response operation. The second operator, $Exec(P, o_r, \vec{y}, l)$ is a running request-response: P is the running process, o_r the name of the operation, $\vec{y}$ the vector of variables to be used for the answer, and l the location of the client. The name of the operation and the location of the client are needed to send back the answer.

Semantics. The service behavior layer does not deal with state, leaving this issue to the service engine layer. Instead, it generates all the transitions allowed by the process behavior, specifying the constraints on the state that have to be satisfied for them to be performed. The state, and the conditions on it, are substitutions of values for variables. We use σ to range over substitutions, and write $[\vec{v}/\vec{x}]$ for the substitution assigning values in $\vec{v}$ to variables in $\vec{x}$. Given a substitution σ, $\mathrm{Dom}(\sigma)$ is its domain.

The semantics follows the idea above: 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, we use structured labels of the form $\iota(\sigma : \theta)$ where ι is the kind of action while σ and θ are substitutions containing respectively the assumptions on the state that should be satisfied for the action to be performed and the effect on the state.

Definition 3.1. (Service behavior layer semantics)
We define $\rightarrow \subseteq SC \times Act \times SC$ as the least relation which satisfies the rules of Tables 2, and closed w.r.t. structural congruence $\equiv$, the least congruence relation satisfying the axioms in Table 3.

Rule ONE-WAYOUT defines the solicit operation: the first part of the label, $\overline{o}(\vec{v})@l$ is the actual action. Here l is the location of the invoked service, taken from variable z. The other two arguments define the effect and the requirements on the state respectively. Substitution $[l/z, \vec{v}/\vec{x}]$ specifies that this transition can be performed only if the state assigns value l to variable z and values in $\vec{v}$ to variables in $\vec{x}$. This assumption will be checked by the service engine layer. The empty substitution $\emptyset$ specifies that the operation does not affect the state. Rule ONE-WAYIN corresponds to the one-way operation. Here there are no conditions on the state, but a state update is required by the label: values in $\vec{v}$ should be assigned to variables in $\vec{x}$. State updates are performed by the service engine layer. The solicit and the one-way operations are synchronized in the services system layer.

Similarly rules SOLICIT and REQUEST start a solicit-response operation. The main difference between the solicit-response and the notification is that the solicit-response leaves an operation waiting for the response. The request-response instead, after invocation, becomes an active construct executing process P, and storing all the information needed to send back the answer. The execution of P is managed by rule REQUEST-EXEC. When the execution of P is terminated, rule REQUEST-RESPONSE sends back the answer, exploiting the stored information about the name of the operation and the loca-

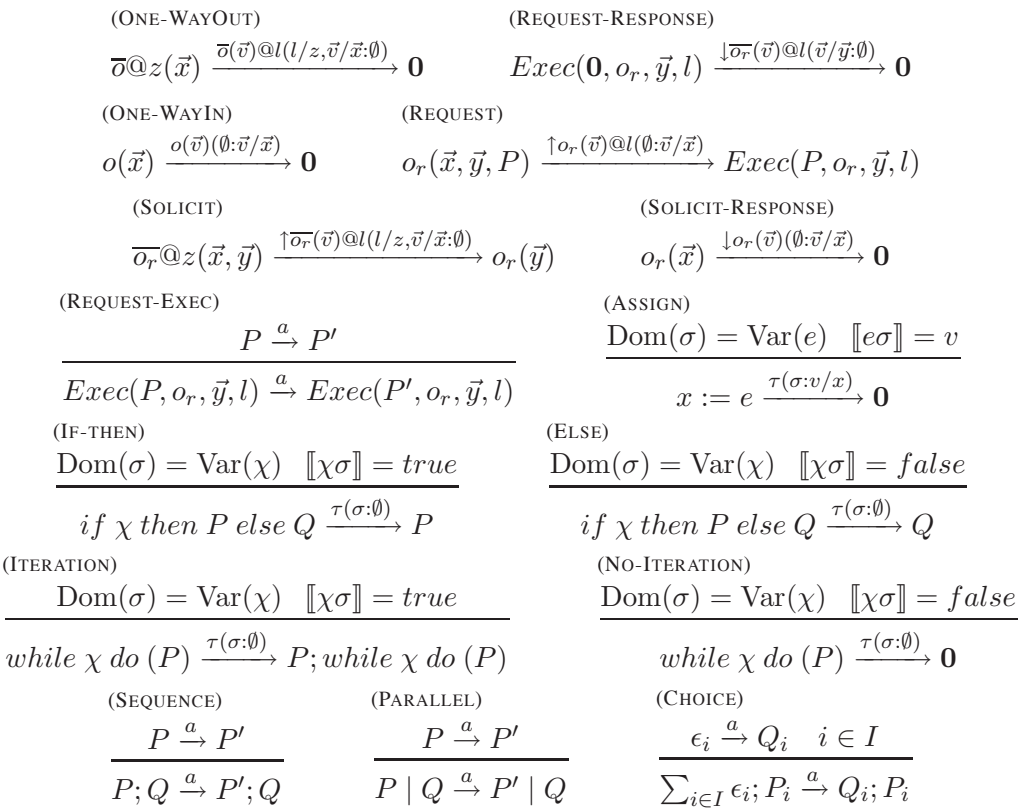

Table 2. Rules for service behavior layer ($a \neq th(f)$)

tion of the invoker. This synchronizes with rule SOLICIT-RESPONSE on the client side, concluding the communication pattern.

The other rules in Table 2 are standard, apart from the fact that the label stores the conditions on the state. For instance assignment $x := e$ produces an internal step, and requires to update the state by assigning value v to variable x, provided that the state provides a substitution σ for variables in e such that the evaluation of $e\sigma$ is v.

Car repair scenario. We discuss now as an example a SOCK implementation of the car repair scenario of the automotive case study [37], a case study proposed inside the EU Project Sensoria [18] to assess techniques and tools for the development of service-oriented applications.

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. The system is composed by five main services: the garage booking service, the truck booking service, the car rental service, the bank service and the car service. Both the garage booking and the truck service exhibit two operations: `book` and `revbook`. The former allows for booking the garage (resp. truck) and the latter allows for the revoking of the reservation in case of

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

Table 3. Structural congruence

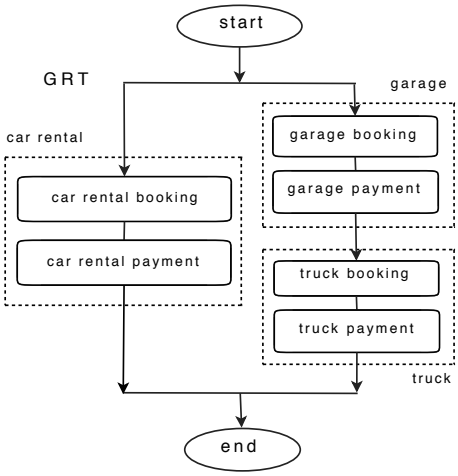

Figure 3. Car service workflow

$$\begin{aligned}
CAR_P ::= \quad & (G_P; T_P) \mid R_P \\
G_P ::= \quad & \overline{book}@G(failure, \langle G_{acc}, G_{id} \rangle); \\
& \overline{pay}@B(\langle DRV_{acc}, G_{acc}, G_{id} \rangle, G_{payid}) \\
T_P ::= \quad & \overline{book}@T(\langle CAR_{coords}, G \rangle, \langle T_{acc}, T_{id} \rangle); \\
& \overline{pay}@B(\langle DRV_{acc}, T_{acc}, T_{id} \rangle, T_{payid}) \\
R_P ::= \quad & \overline{book}@R(G_{coords}, \langle R_{acc}, R_{id} \rangle); \\
& \overline{pay}@B(\langle DRV_{acc}, R_{acc}, R_{id} \rangle, R_{payid});
\end{aligned}$$

Figure 4. The car service workflow in SOCK, without error handling

failure. Besides these operations the car rental service provides also a `redirect` operation which allows for the redirection of the rented car. The bank service offers the following operations: `pay` for asking to perform a payment and `revpay` for revoking it.

For the sake of brevity, in the following we only discuss the car service workflow, which is enough to exemplify our approach. The car service workflow is represented in Fig. 3.

Fig. 4 shows a first version of the car service workflow in SOCK, without error handling. In Section 4 we will extend the example to manage possible errors. The workflow is composed by three different activities, corresponding to the garage rental (G), tow truck rental (T), and car rental (R). The garage rental activity and the tow truck rental activity are performed sequentially (it is not useful to book the tow truck if no garage is available), and this sequential composition and the car rental activity are executed concurrently. Each activity is composed by two invocations, the first one to the booking service, and the second one to the bank service. The car system uses variables for containing the information about other services: variables X_{acc} (where X can be G, T, or R) contain their account information, X_{id} the identifier of the corresponding interaction, X_{payid} the identifier of the corresponding bank interaction. In addition, $failure$ contains the description of the car failure, DRV_{acc} the driver's bank account, CAR_{coords} the car location and G_{coords} the garage location.

3.2. Service engine layer

In general the service engine layer deals with state, correlation sets and session spawning. However since correlation sets and sessions play no role in error recovery, we will consider only state.

$$\text{(LIFT)}\ \frac{Y \xrightarrow{\iota} Y'}{Y@l \xrightarrow{\iota} Y'@l} \qquad \text{(NORMALSYNC)}\ \frac{Y@l' \xrightarrow{\lambda@l} Y'@l' \quad Z@l \xrightarrow{\lambda'} Z'@l \quad compl(\lambda,\lambda')}{Y@l' \parallel Z@l \xrightarrow{\tau} Y'@l' \parallel Z'@l}$$

$$\text{(PAR-EXT)}\ \frac{E_1 \xrightarrow{\iota} E_1'}{E_1 \parallel E_2 \xrightarrow{\iota} E_1' \parallel E_2} \qquad \text{(SOLICIT-REQUESTSYNC)}\ \frac{Y@l' \xrightarrow{\uparrow\overline{o_r}(\vec{v})@l} Y'@l' \quad Z@l \xrightarrow{\uparrow o_r(\vec{v})@l'} Z'@l}{Y@l' \parallel Z@l \xrightarrow{\tau} Y'@l' \parallel Z'@l}$$

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 4. Rules for services system layer

The service engine layer syntax is:

$$Y ::= (P, \mathcal{S})$$

where P is a service behavior process and $\mathcal{S}$ is a state.

A state is 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 denote with $\oplus$ the update operation on a state.

The service engine layer is described by the following rule, which verifies that the condition σ on the state is satisfied, and updates the state itself with ρ.

$$\text{(ENGINE-STATE)}\ \frac{P \xrightarrow{\iota(\sigma:\rho)} P' \quad \mathcal{S} \vdash \sigma}{(P, \mathcal{S}) \xrightarrow{\iota} (P', \mathcal{S} \oplus \rho)}$$

3.3. Services system layer

The services system layer allows one to compose different engines into a system. The service engines are composed in parallel and equipped with a location that allows one 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 4 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. The location is needed only for request-response, since it is used by the server to send back the answer. Finally rule PAR-EXT deals with parallel composition.

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

$P, Q, \ldots ::= \epsilon$	input	$\overline{\epsilon}$	output
$\ldots$	other standard ops.		
$\{P\}_q$	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		
$Exec(P, o_r, \vec{y}, l)$	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	$\overline{o_r}!f@l$	fault output

Table 5. Service behavior syntax with faults

4. Error handling in SOCK

In this section we extend SOCK by adding to its service behavior layer the primitives for fault and compensation handling described in Section 2.

Syntax. We need the following additional sets: $Faults$, ranged over by f, for faults, and $Scopes$, ranged over by q, for scope names. We use $q_\bot$ to range over $Scopes \cup \{\bot\}$, whereas u ranges over $Faults \cup Scopes \cup \{\bot\}$. Here $\bot$ is used to specify that an handler is undefined. $\mathcal{H}$ denotes a function from $Faults$ and $Scopes$ to processes extended with $\bot$, i.e. $\mathcal{H} : Faults \cup Scopes \to SC \cup \{\bot\}$. In particular, we write the function associating P_i to u_i for $i \in \{1, \ldots, n\}$ as $[u_1 \mapsto P_1, \ldots, u_n \mapsto P_n]$. The extended syntax for processes is defined in Table 5. Note that the syntax for outputs $\overline{\epsilon}$ has changed, since now the solicit-response $\overline{o_r}@z(\vec{y}, \vec{x}, \mathcal{H})$ includes a handler update $\mathcal{H}$, whose purpose will be clarified later.

In addition to the new solicit-response primitive we have five new static constructs, and other auxiliary constructs to help the definition of the semantics. Handlers are installed by $\mathrm{inst}(\mathcal{H})$, where $\mathcal{H}$ is a partial function from fault and scope names to processes: we write $\mathcal{H}(f) = \bot$ to specify that $\mathcal{H}$ is undefined on fault name f. $\{P\}_q$ defines a scope named q and executing 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 (i.e., at the beginning no fault handler is defined) and to $\mathbf{0}$ for all scope names (i.e., the default termination or compensation handler has no effect). The third parameter is the name of a handler waiting to be executed: at the beginning no handler has to be executed, thus $\bot$ is used. Primitives $\mathrm{throw}(f)$ and $\mathrm{comp}(q)$ respectively raises fault f and asks to compensate scope q. Finally, cH is a placeholder for the previously installed handler with the same name, to be used inside a 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. 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. As said, in order to define the semantics we exploit an extended syntax. There $\{P : \mathcal{H} : u\}_{q_\bot}$ is an active scope. An active scope includes a handler function $\mathcal{H}$ specifying the installed handlers. The third argument, u, is the name of an handler waiting to be executed, or $\bot$ if no handler is waiting to be

executed. This is needed, for instance, when scope q is killed while waiting for the answer of a request-response interaction: the termination handler for q has to be executed, thus q is written as third parameter. However, the termination handler is not executed immediately, but the answer from the request-response is waited for. This answer may update the termination handler. When the termination handler has to be executed, the updated version is used. When a scope has failed its execution, either because it has been killed from a parent scope, or because it has not been able to catch and manage an internal fault, it reaches a zombie state. Zombie scopes have $\perp$ as scope names. Scopes in a zombie state are no more able to throw faults. This ensures that each scope may throw at most one fault. The receive operation $o_r(\vec{x}, \mathcal{H})$ 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. If the remote activity has failed, no compensation for it is required. $o_r\langle\vec{x}, \mathcal{H}\rangle$ is the corresponding zombie version, which cannot throw faults. This is created when the normal version is killed because of an external fault, again to ensure that no fault is raised by dead activities. $\langle P\rangle$ 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. $Exec(P, o_r, \vec{y}, l)$ is a running request-response interaction (as for the calculus without faults). Finally, $\overline{o_r}!f@l$ notifies fault f to the client (located at l) of a request-response pattern. This is created when an executing request-response is killed because of a local fault.

In the semantics, in addition to the structured actions introduced in Section 3, we use the following unstructured actions: $\{th(f), cm(q, P), inst(\mathcal{H})\}$. They represent respectively the propagation of fault f, the check that the compensation code for scope q is P, and the request to apply handler update $\mathcal{H}$.

Definition 4.1. (Service behavior layer semantics)
We define $\rightarrow \subseteq SC \times Act \times SC$ as the least relation which satisfies the rules of Tables 2 (where we assume $th(f)$ never occurs as label) and 6, and closed w.r.t. structural congruence $\equiv$, the least congruence relation satisfying the axioms in Table 3.

The rules in Table 6 define the semantics of scopes, faults and compensations. We use operator $\boxplus$, defined as follows, for updating the handler function:

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

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 with the same name. 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 terminations/compensations, i.e. $\mathrm{cmp}(\mathcal{H}) = \mathcal{H}|_{Scopes}$.

Rules SOLICIT and SOLICIT-RESPONSE replace the corresponding rules in Table 2, ensuring that when the answer is received, handler update $\mathcal{H}$ is installed. The handler update is not performed if a fault answer is received (see rule RECEIVE-FAULT). The internal process P of a scope can execute thanks to rule SCOPE. 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 to the parent scope. This is needed to allow the parent scope to compensate the finished child. Compensation handlers of subscopes are propagated too, to allow to recursively compensate them. Compensation execution is required by rule COMPENSATE. The actual

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

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

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

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

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

(COMPENSATE)
$$\mathrm{comp}(q) \xrightarrow{cm(q,Q)} Q$$

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

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

(SCOPE-HANDLE-FAULT)
$$\{\mathbf{0}:\mathcal{H}:f\}_{q_\perp} \xrightarrow{\tau(\emptyset:\emptyset)} \{\mathcal{H}(f):\mathcal{H}\boxplus[f\mapsto\perp]:\perp\}_{q_\perp}$$

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

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

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

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

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

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

(RECEIVE-FAULT)
$$o_r(\vec{x},\mathcal{H}) \xrightarrow{o_r(f)(\emptyset:\emptyset)} \mathrm{throw}(f)$$

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

Table 6. Rules for service behavior layer: faults and compensation rules

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

Table 7. Function $killable$

compensation code Q is guessed, and the guess is checked by rule COMPENSATION. When the compensation is executed, the corresponding handler is removed, so to ensure that the same activity is not compensated twice. Faults are raised by rule THROW. A fault is caught 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 the activities in P' have been completed. These include, for instance, waiting for response messages in request-response interactions, terminating subscopes, and terminating internal error recovery. This is managed by the rules for fault propagation THROW-SYNC, THROW-SEQ, RETHROW and THROW-REXEC, and by the partial function $killable$ (see Table 7). The function $killable$ is applied to parallel components by rule THROW-SYNC. This has a double aim. On the one hand it guarantees that when a fault is thrown there is no pending handler update, i.e. it realizes priority of handler update w.r.t. fault processing. This ensures that handlers are always up-to-date, and solves the race condition issues discussed in Section 2.2. 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 hand $killable(P, f)$ 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 (see the definition of function $killable$ for scopes). 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. If an $Exec$ (i.e., an ongoing request-response computation) is terminated then the fault is notified to the partner (this explains why function $killable$ needs as parameter the name f of the fault). Finally, a receive waiting for the answer of a solicit-response is preserved, thus preserving the pattern of communication, but changed to its zombie version, ensuring that no other faults will be thrown. The $\langle P \rangle$ operator (described by rule PROTECTION) guarantees that the enclosed activity will not be killed by external faults (because of the fifth rule in the definition of function $killable$). Rule SCOPE-HANDLE-FAULT executes a handler for a fault. This is done only after the activities discussed above have terminated. The fault handler 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. Note that a scope that has handled an internal fault can still end with success. Instead, a scope that has been terminated from the outside is in zombie state. It can execute its termination handler thanks to rule SCOPE-HANDLE-TERM, and then terminate with failure (thus discarding its compensation handlers) using rules SCOPE-FAIL. Similarly, a scope enters the zombie state when reached by a fault it cannot handle, as specified by rule RETHROW. The fault is propagated up along the scope hierarchy. Zombie scopes cannot throw faults any more, since rule IGNORE-FAULT has to be applied instead of

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

Figure 5. The car service workflow in SOCK, with error handling

RETHROW. Rule IGNORE-FAULT is necessary only for faults thrown by handlers, since no other fault can be generated by a zombie scope. When an executing request-response is reached by a fault, it is transformed into a fault notification (see the definition of function $killable$). Fault notification is executed by rule SEND-FAULT, and it will interact with the waiting receive thanks to rule RECEIVE-FAULT. When received, the fault is ready to be rethrown at the client side, where it is treated as a local fault. If the receive is in zombie state instead, the fault is discarded (rules DEAD-SOLICIT-RESPONSE and DEAD-RECEIVE-FAULT are used instead of rules SOLICIT-RESPONSE and RECEIVE-FAULT).

Car repair scenario. We now extend the SOCK implementation of the car repair scenario presented in Section 3 with error handling, to show how the primitives presented in the previous section can be applied in practice. The example also allows us to assess the expressive power of the proposed primitives from an empirical point of view. Fig. 5 shows a possible implementation of the car repair scenario able to manage all faults raised by the invoked services and communicated through automatic notification (we assume that there are no faults from the car service system itself). W.r.t. Fig. 4, now each activity is executing inside a scope.

Let us consider G_P, the module for garage booking and payment. The booking invocation may receive and raise fault fG, which is handled at the higher level by CAR_P, asking to compensate activity R_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 solicit-response $\overline{pay}$ for the bank payment receives a successful response, the compensation handler $GHandler_P$ for the scope is installed. This requires to revoke both the booking of the garage and the corresponding bank payment.

Module T_P is analogous, but it does not install a compensation handler as its compensation is never required.

Note that both module G_P and module T_P do not install termination handlers, since they never receive external faults. In fact, R_P always manages its own faults, and we assume that the rest of the car service system (not represented in this example) never throws faults.

Module R_P is more complex, since it has to deal with possible faults from G_P or T_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 of the payment service, or after the invocation of the payment service. This corresponds to the different termination handlers installed during the execution. For instance, before booking, the rented car 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 $Rredirect_P$).

As in the case of G_P both the booking and the payment may fail, but R_P provides local fault handlers for each possible failure, guaranteeing that the faults are not propagated upstream to the main scope or to the sibling activities. 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 $Rredirect_P$), and makes it executable inside an auxiliary scope. This is necessary since $Rredirect_P$ may raise fault fG, and the old handler specified in scope r will not be available anymore. 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, thus guaranteeing a safe termination.

The formalization of the scenario highlights the main features of our approach: the automatic notification of remote faults so that they can be safely managed by the invoker and the dynamic updating of handlers according to the evolution of the state of the computation (e.g., the termination handler for activity R_P).

5. Properties

In this section we prove five properties that the error handling mechanisms defined in the previous section satisfy. On one side, this shows that our semantics correctly captures the intuition underlying the proposed error handling mechanisms. On the other side, these are examples showing how the formalization above can be used to prove interesting properties of the modeled systems. The first two properties capture the meaning of scope failure and scope success, respectively in the case of isolated scope (Proposition 5.1) and scope killed by an external failure (Proposition 5.2). The third and the fourth properties show that in a request-response interaction on the client side the response is always waited for (Proposition 5.3), and that a response is always sent back from the server (Proposition 5.4). Finally, Proposition 5.5 shows priority of handler update w.r.t. error recovery. The last three properties formalize some of the concepts we highlighted in Section 2.

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. Furthermore, if the scope raises a fault, it will never terminate with success and it will be no longer able to raise another fault. For instance

$\{\mathrm{throw}(f)\}_q$ will end unsuccessfully. Note that a scope able to internally recover a fault will end with success. As an example, let us consider

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

where scope q manages its internal fault by executing $HANDLE$, and then ends with success. In this way compensation $COMP$ (assuming that $HANDLE$ does not install an handler different from $COMP$) will be available for outer scopes.

Proposition 5.1. Let $P \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2 \ldots \xrightarrow{a_n} P_n$ be a computation. Suppose that P is a scope $\{Q\}_q$. 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: $P_n = \{P' : \mathcal{H} : u\}_q$.

Proof:
The proof is by induction on n. The base 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, IGNORE-FAULT and SCOPE-HANDLE-TERM cannot be applied because by hypothesis $q \neq \bot$.

We still have to prove that if $P_i \xrightarrow{th(f)} P_{i+1}$ then no other $th(f')$ action nor $inst(\mathcal{H})$ action will be generated in the computation. 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 $inst(\mathcal{H})$ then the thesis follows. □

In general, a scope is not isolated, but part of a complex system. However, interactions with the rest of the system are acknowledged by the scope itself, thus they correspond to computations considered by the proposition above. The only exception is the killing of a scope because of an external fault, obtained by using function $killable$. The second proposition shows that also in this case the behavior of the scope

[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.

follows the intuition. In particular, since the scope has been terminated the only possibility corresponds to case 2 in Proposition 5.1, i.e. to the case of a failed scope. This is the case for example of the process:

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

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

Proposition 5.2. 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'.

Proof:
We have $P = \langle \{killable(P', f) : \mathcal{H} : q\}_\perp \rangle$. We prove the thesis by induction on the length of the computation. More precisely, we prove that each process of the form $\langle \{Q : \mathcal{H} : u\}_\perp \rangle$ will never terminate with success nor throw faults and will always move to a process of the same form or terminate. The transition is derived by applying rule PROTECTION to a rule for scope. The applicable rules are SCOPE, INSTALL, SCOPE-HANDLE-FAULT, COMPENSATION, SCOPE-HANDLE-TERM, CATCH-FAULT, IGNORE-FAULT and SCOPE-FAIL. The thesis follows by induction hypothesis for all the rules but SCOPE-FAIL, directly for this last. □

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 client always waits for the response of the server. The response can then be used during error recovery. Dually, the fourth property ensures that the server always sends an answer, either a normal message or a fault notification.

In order to prove those properties, we need to define when a part of a process is being executed.

Definition 5.1. (Enabling contexts)
We define enabling contexts by structural induction as follows:

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

We also need the following lemmas.

Lemma 5.1. $P \equiv C[\bullet]$ for some enabling context $C[\bullet]$ iff there exists an enabling context $C'[\bullet]$ such that $P = C'[\bullet]$.

Proof:
The proof is by induction on the length of the proof of congruence of $P \equiv C[\bullet]$, with a case analysis on the axiom applied. □

Lemma 5.2. 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 , if defined, $killable(P, f)$ 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 $C[\bullet]$. □

Lemma 5.3. 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 either the form $C'[Exec(P', o_r, \vec{y}, l)]$ or $C'[\langle \overline{o_r}! f@l \rangle]$ too.

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

We can now prove the two propositions.

Proposition 5.3. 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 $\uparrow \overline{o_r}(\vec{v})@l(l/z, \vec{v}/\vec{x} : \emptyset)$, 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 $a_i = o_r(f)(\emptyset : \emptyset)$ for some i;
2. the process is waiting for the response: $P_n \xrightarrow{\downarrow o_r(\vec{v'})(\emptyset:\vec{v'}/\vec{x'})} P'$.

Proof:
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 base 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 5.1 we have no need to consider structural congruence. We show the most interesting cases, the others follow by inductive hypothesis.

$C[\bullet] = \bullet$**:** 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.

$C[\bullet] = C'[\bullet]|Q$ **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 5.2.

$C[\bullet] = \{C'[\bullet] : \mathcal{H} : u\}_{q_\perp}$**:** 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 cannot be applied since the left hand side does not satisfy the hypothesis.

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

□

Proposition 5.4. 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: $a_i = \downarrow \overline{o_r}(\vec{v'})@l(\vec{v'}/\vec{y} : \emptyset)$ or $a_i = \overline{o_r}(f)@l(\emptyset : \emptyset)$ 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 $C[\bullet]$.

Proof:
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 base 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 $C[\bullet]$. Because of Lemma 5.1 we have no need to consider structural congruence. We show the most interesting cases: the others are similar.

$C[\bullet] = \bullet$**:** 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 $o_r(f)(\emptyset : \emptyset)$.

$C[\bullet] = C'[\bullet]|Q$ **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 and the thesis follows from Lemma 5.3.

□

Finally, the fifth property guarantees that handler update has priority w.r.t. fault management, thus handlers are always up-to-date and no race conditions between handlers update and fault management occur. This property is fundamental for the correct behavior of the dynamic approach to error recovery. Before stating the property, we present a lemma characterizing the shape of processes P such that $killable(P, f)$ is defined.

Lemma 5.4. If $killable(P, f)$ is defined then $P \neq C[\mathrm{inst}(\mathcal{H})]$ for each enabling context $C[\bullet]$.

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

Proposition 5.5. 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.

Proof:
The proof is by rule induction. Thanks to Lemma 5.1 we need only to prove that if $P \xrightarrow{th(f)} P'$ then $P \neq C'[\mathrm{inst}(\mathcal{H})]$ 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 $\langle P \rangle = C'[\mathrm{inst}(\mathcal{H})]$ only if $P = C''[\mathrm{inst}(\mathcal{H})]$, but this is impossible by inductive hypothesis.

THROW-SYNC: we can have $P|Q = C'[\mathrm{inst}(\mathcal{H})]$ only if either $P = C''[\mathrm{inst}(\mathcal{H})]$ or $Q = C''[\mathrm{inst}(\mathcal{H})]$. The first case is impossible by inductive hypothesis. The second case is impossible because of Lemma 5.4.

THROW-SEQ: we can have $P; Q = C'[\mathrm{inst}(\mathcal{H})]$ only if either $P = C''[\mathrm{inst}(\mathcal{H})]$ or if $P \equiv \mathbf{0}$ and $Q = C''[\mathrm{inst}(\mathcal{H})]$. The first case is impossible by inductive hypothesis. The second case is impossible since if $P \equiv \mathbf{0}$ then P has no transitions, thus the premise of the rule cannot hold.

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

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

□

6. Related work and conclusion

We have investigated the impact of dynamic error handling in the context of service orchestration languages. Besides presenting and assessing a novel approach to error handling, this paper includes also the formal investigation of the interplay between error handling and the request-response communication pattern. In fact, excluding BPEL [33], we are not aware of models in which there is interplay between fault and compensation handling and bidirectional communication. However SOCK differs from BPEL in many respects. First of all, 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 of some experimentations done with the ActiveBPEL [3] engine, one of the most well-known engines for BPEL. Notice however that BPEL specifications are frequently ambiguous, and the different implementations follow different interpretations, as proved by the comparison in [29]. BPEL follows the static approach to error handling described in Section 2.1. Tags `<scope>`, `<faultHandlers>`, `<compensationHandler>` and `<terminationHandler>` are used to represent scopes, fault handlers, compensation handlers and termination handlers, respectively. Faults can be generated either explicitly by means of construct `<throw>`, or received as response of a request-response invocation, if the invoked service sends them. If a fault is raised within a scope, the related fault handler is executed whereas, if raised by an external scope, the termination handler is started. When an activity successfully finishes, its compensation handler is promoted to the parent scope.

The first difference between BPEL and SOCK is that in SOCK dynamic handling 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, BPEL and SOCK differ in how they define completion of a scope that manages an internal fault: in BPEL such a scope is never considered successful, on the contrary in SOCK it is considered successful provided that it does not rethrow any fault. Thus, in SOCK, its last defined termination handler is promoted to the parent

scope as compensation handler. This feature allows for the updating of the compensation handler within a fault handler. In order to clarify this aspect, consider the following example, where we want to book a hotel and a public transport service, which can be a train or, if the train is not available, a bus.

$\{\, inst([fT \mapsto Bus; inst([q \mapsto cH; revBus])]);$
$\quad Hotel; inst([q \mapsto revHotel]);$
$\quad Train; inst([q \mapsto cH; revTrain])\}_q$

$Hotel$, $Train$ and Bus are the processes in charge of performing the booking of the hotel, the train and the bus, respectively. Process Bus is executed only if process $Train$ raises the fault fT. In this case, the booking procedure for the bus is executed, and the compensation handler for scope q is updated. Thus, if compensation is required, processes $revHotel$ and $revBus$, which are in charge of reversing the booking of the hotel and of the bus, are executed. On the contrary, if a train is available, the compensation handler will execute processes $revHotel$ and $revTrain$, where the last one is in charge of reversing the booking of the train. In BPEL this scope could be represented as follows[3], where we assume that each scope (`Hotel`, `Train` and `Bus`) has its own compensation handler, reversing the corresponding activity:

```
<scope name="q">
  <compensationHandler> Compensate Hotel, Train and Bus </compensationHandler>
  <sequence>
    <scope name="Hotel">...</scope>
    <scope name="Train">...</scope>
    if Train failed then <scope name="Bus">...</scope>
  </sequence>
</scope>
```

In BPEL it is necessary to test if the scope `Train` has finished successfully or not, in order to execute the scope `Bus` only in the second case. To implement this test, one needs to use an auxiliary variable, to be set within the fault handler of the scope `Train`. In fact, if scope `Train` fails, we cannot rethrow the fault at the level of scope `q`, since we want `q` to complete with success if either a train or a bus has been booked.

BPEL and SOCK differ also in how the order of compensation of different activities is specified. In BPEL it is possible either to compensate single scopes or to compensate groups of activities in a default order. In the latter case, the construct `<compensate>` asks to execute the compensations available in an order that depends on the static structure of activities, e.g. in reverse order for activities in sequence and in parallel for activities in parallel. In SOCK instead the compensation is built according to the order of execution of the install primitives, which is only known at runtime. However the programmer may specify how to build the handler, thus implementing different policies, as outlined in the example at the end of Section 2.2. The BPEL approach is enough to implement the backward recovery case of that example. To do that, processes P and Q have to be enclosed into subscopes with compensation handlers P' and Q'. If triggered, the default compensation at the level of the parent scope q will execute all the installed compensation handlers in reverse order. In SOCK, no default compensation order is needed since the desired order of compensation can be defined using the install primitive in a much more flexible way.

[3] For the sake of brevity we use a simplified version of the BPEL syntax.

Another important difference concerns the policy for fault propagation inside the request-response communication pattern. In BPEL request-response activities are split into two different constructs: `<receive>` and `<reply>`. The former is in charge of receiving an incoming message on a request-response operation, whereas the latter is in charge of sending the related response. The `<reply>` construct is also used for sending a fault message, which must be explicitly specified as an attribute. In other words, fault communication in BPEL is not automatic, but it must be specified by the programmer. If no `<reply>` is specified inside a fault handler triggered from within a request-response, in case of fault the related `<receive>` has no matching `<reply>`. These receive activities are called *orphaned inbound message activities* and when they are detected a `missing-reply` exception is both triggered locally and sent to the invoker. On the contrary, in SOCK the request-response operation is atomic and fault propagation is automatically encoded in the semantics of the primitive. In SOCK indeed, it is not necessary to specify reply activities within fault handlers, since when an error occurs the fault is automatically propagated to the sender. Furthermore, as far as the invocation of request-response operation is concerned, quoting the BPEL specifications, "when a synchronous invoke activity (corresponding to a request-response pattern) 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. This is not the case of SOCK, which waits for the response message before processing the error handler.

Finally, SOCK has no `rethrow` primitive, used in BPEL to pass a fault to the enclosing scope, since $\mathrm{throw}(f)$, 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.

As far as the car repair scenario is concerned, actually most of it can be implemented using BPEL, while this is not the case for the forward and parallel recovery variants of the example at the end of Section 2.2. In the car repair scenario, the most difficult part to be dealt with in BPEL is remote faults management, since in BPEL remote faults are discarded in case of concurrent local faults. Consider, e.g., any bank payment in the car repair scenario. In our approach, the handler is updated so to allow to undo the payment only if it has been successfully completed. In BPEL this information is not available at the client location, thus, during error recovery, the client has to invoke the server asking whether the payment has successfully completed. This requires the addition of a new operation on the bank service, an additional request-response interaction between the client and the bank, and some non-trivial correlation mechanism to avoid mixing information concerning different payments. Clearly, the additional communication may also add new points of failure.

In the area of workflow systems, the rigidity of classical mechanisms for exception handling was already pointed out in [34] where a systematic overview of the workflow management systems available at the time of the paper showed that all of them adopted a static approach for the management of exceptions. The adoption of a dynamic approach for workflow systems was more recently proposed by Adams et al [4] for YAWL (Yet Another Workflow Language) [1], a workflow language designed as an extension of Petri nets to allow for a more direct and intuitive support of the so-called workflow patterns identified by van der Aalst and Hofstede in [2].

The main ingredient of the proposal in [4] is the availability of a Worklet Service that, orthogonally to the workflow engine, maintains an extensible repertoire of exception handling processes and an associated set of contextual rules governing the activation of the appropriate process in case an exception is thrown at run time. More precisely, when an exception occurs, the corresponding exception handler is selected from the repertoire according to the type of the exception, the current state of the running pro-

cess (and data associated to previously executed processes) and a set of user-defined rules. Following the traditional approach of workflow engines, intended as tools interacting with human users in order to drive the flow of their activities, the repertoire of exception handling processes as well as the set of rules can be dynamically updated and modified (by means of a worklet editor and of a rule editor, respectively).

Differently from the approach adopted in [4] we do not consider an external service responsible for managing the exceptions, but these are thrown and caught "internally" following an approach which is closer to the typical exception handling mechanisms of traditional programming languages. In fact, instead of considering a dynamically-modifiable global catalog of exception handling processes, we dynamically gather them in the error handler functions of our scopes. Moreover, instead of using user-defined rules in order to exploit the execution context in the selection of the most appropriate exception handler, we directly adapt our handlers to the execution context using the inst primitive. Additionally, the cH (current handler) placeholder introduces a form of higher-order programming that allows for the dynamic creation of new exception processes, even without the human intervention.

The practical impact of the results exposed in this paper have been investigated by applying them to JOLIE [31, 24], a service-oriented programming language that implements the semantics of SOCK, released under an open source license [20]. JOLIE joins the formal semantics inherited by SOCK with an intuitive syntax, thus allowing for formal reasoning on JOLIE programs and easy prototyping and refining of orchestrators. One of the most prominent features of JOLIE is the elegant separation between the program behavior and the underlying communication technologies. The same behavior can be used with different communication media (such as bluetooth, local memory, sockets, etc.) and protocols (such as HTTP, SOAP, SODEP[4], etc.). Moreover, the communication infrastructure of JOLIE can be extended with the support for additional communication media and protocols by means of simple Java libraries, called *JOLIE extensions*. JOLIE implements the semantics of our dynamic error handling approach, preserving its properties. The language offers all the error handling constructs that we introduced in SOCK, plus a few more functionalities and syntax shortcuts that are aimed to ease the program writing process and make the resulting text more elegant. Major enhancements are the possibilities to attach user-defined data to a fault signal and the ability to access during error recovery the state of variables at the time the handler has been installed. These mechanisms introduce additional context information to the fault handling procedure of JOLIE. To better understand the practical applicability of our proposal, the interested reader may find at [15] a JOLIE implementation of the car repair scenario discussed in Section 4.

Dynamic handler installation distinguishes our calculus w.r.t. other calculi in the literature, such as πt-calculus [6], webπ [27] and cJoin [9], all featuring static compensations. Furthermore, 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 [13], cCSP [12] and SAGAs calculi [10], 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, but we think that it can be applied to

[4] SODEP (Simple Operation Data Exchange Protocol) is a new, efficient protocol developed in the JOLIE project. The protocol has been implemented also as C++, Java and J2ME libraries.

many other frameworks, from basic theoretical calculi such as π-calculus to industrial frameworks such as BPEL. A recent work [35] supports this claim, since it presents an application of dynamic handling to the asynchronous π-calculus. In this work dynamic installation of handlers is obtained by attaching them to input prefixes, and this ensures also that it takes priority w.r.t. fault processing. Since π-calculus does not include complex interaction patterns like request-response, the problem of fault propagation does not occur at the language level: if faults should be propagated between interacting processes, the programmer has to specify how this should be done.

It would be interesting to apply the dynamic approach also to other calculi for SOC such as COWS [28], SCC [8], SSCC [26], CaSPiS [7], SC [19] and CC [36], in order to understand whether there are subtle interplays between dynamic error recovery and specific features of those calculi, such as for request-response in the case of SOCK. At the moment COWS, CaSPiS and CC provide some mechanisms for error recovery. COWS has basic mechanisms based on a kill primitive with static scope and protection of parts of code, while CC uses a standard try-catch construct. Only CaSPiS has mechanisms for session termination, supporting the notification of termination to the remote partner. However CaSPiS has no mechanisms for handling local errors, and remote messages are managed in a static way. Concerning implementations, CaSPiS has been implemented in JCaSPiS [5] ans SC has inspired JSCL [19]. Both the implementations are Java libraries, thus different from JOLIE which is a full-fledged language.

SOCK provides the mechanisms for error handling, but it would be interesting to study how general error recovery strategies such as the ones proposed by SAGAs calculi [10] can be programmed inside SOCK. This could allow to apply to our language the results in [14], which allow to guarantee the correctness of compensations.

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 dynamic fault and compensation handling on the notion of conformance (as formalized for instance in [11], [16] or [25]) as done for static error handling in [17].

Acknowledgements

We acknowledge the anonymous reviewers for their insightful comments useful to improve the presentation of the paper contribution.

References

[1] van der Aalst, W., ter Hofstede, A.: YAWL: Yet Another Workflow Language, *Inf. Syst.*, **30**(4), 2005, 245–275.

[2] van der Aalst, W. M. P., ter Hofstede, A. H. M.: Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages, *Proc. of CPN'02*, 560, 2002.

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

[4] Adams, M., ter Hofstede, A., van der Aalst, W., Edmond, D.: Dynamic, Extensible and Context-Aware Exception Handling for Workflow, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, 4803, Springer, 2007.

[5] Bettini, L., Nicola, R. D., Loreti, M.: Implementing Session Centered Calculi, *Proc. of COORDINATION'08*, 5052, Springer, 2008.

[6] Bocchi, L., Laneve, C., Zavattaro, G.: A Calculus for Long-Running Transactions, *Proc. of FMOODS'03*, 2884, Springer, 2003.

[7] Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and Pipelines for Structured Service Programming, *Proc. of FMOODS'08*, 5051, Springer, 2008.

[8] Boreale, M. et al: SCC: A Service Centered Calculus, *Proc. of WS-FM'06*, 4184, Springer, 2006.

[9] Bruni, R., Melgratti, H., Montanari, U.: Nested Commits for Mobile Calculi: Extending Join., *Proc. of IFIP TCS'04*, Kluwer, 2004.

[10] Bruni, R., Melgratti, H., Montanari, U.: Theoretical Foundations for Compensations in Flow Composition Languages., *Proc. of POPL'05*, ACM Press, 2005.

[11] Busi, N. et al.: Choreography and Orchestration: A Synergic Approach for System Design, *Proc. of ICSOC'05*, 3826, 2005.

[12] Butler, M., Hoare, C., Ferreira, C.: A Trace Semantics for Long-Running Transactions., *25 Years Communicating Sequential Processes*, 3525, Springer, 2004.

[13] Butler, M. J., Ferreira, C.: An Operational Semantics for StAC, a Language for Modelling Long-Running Business Transactions, *Proc. of COORDINATION'04*, 2949, Springer, 2004.

[14] Caires, L., Ferreira, C., Vieira, H.: A Process Calculus Analysis of Compensations., *Proc. of TGC'08*, 5474, Springer, 2008.

[15] Car Repair Scenario in JOLIE, `http://www.jolie-lang.org/examples/fi09/automotive_fi09.zip`.

[16] Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centred Programming for Web Services, *Proc. of ESOP'07*, 4421, Springer, 2007.

[17] Carbone, M., Honda, K., Yoshida, N.: Structured Interactional Exceptions in Session Types, *Proc. of CONCUR'08*, 5201, Springer, 2008.

[18] European Integrated Project Sensoria: Web Site, `http://www.sensoria-ist.eu/`.

[19] Ferrari, G. L., Guanciale, R., Strollo, D.: JSCL: A Middleware for Service Coordination, *Proc. of FORTE'06*, 4229, Springer, 2006.

[20] Free Software Foundation: GNU Lesser General Public Licence v 2.1, `http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html`.

[21] Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: On the Interplay between Fault Handling and Request-Response Service Invocations (TR), `http://www.cs.unibo.it/~lanese/publications/fulltext/TR-COMP.pdf`.

[22] Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: On the Interplay between Fault Handling and Request-Response Service Invocations, *Proc. of ACSD'08*, IEEE Computer Society Press, 2008.

[23] Guidi, C. et al.: SOCK: A Calculus for Service Oriented Computing, *Proc. of ICSOC'06*, 4294, Springer, 2006.

[24] *JOLIE: Java Orchestration Language Intepreter Engine*, `http://www.jolie-lang.org`.

[25] Lanese, I., Guidi, C., Montesi, F., Zavattaro, G.: Bridging the gap between Interaction- and Process-Oriented Choreographies, *Proc. of SEFM'08*, IEEE Computer Society Press, 2008.

[26] Lanese, I., Martins, F., Vasconcelos, V. T., Ravara, A.: Disciplining Orchestration and Conversation in Service-Oriented Computing, *Proc. of SEFM'07*, IEEE Computer Society Press, 2007.

[27] Laneve, C., Zavattaro, G.: Foundations of Web Transactions, *Proc. of FOSSACS'05*, LNCS, 2005.

[28] Lapadula, A., Pugliese, R., Tiezzi, F.: A Calculus for Orchestration of Web Services, *Proc. of ESOP'07*, 4421, Springer, 2007.

[29] Lapadula, A., Pugliese, R., Tiezzi, F.: A Formal Account of WS-BPEL, *Proc. of COORDINATION'08*, 5052, Springer, 2008.

[30] Montesi, F., Guidi, C., Lanese, I., Zavattaro, G.: Dynamic Fault Handling for Service Oriented Applications, *Proc. of ECOWS'08*, IEEE Computer Society Press, 2008.

[31] Montesi, F., Guidi, C., Zavattaro, G.: Composing Services with JOLIE, *Proc. of ECOWS'07*, IEEE Computer Society Press, 2007.

[32] Oasis: *UDDI - Universal Description, Discovery and Integration of Web Services*, `http://www.uddi.org/specification.html`.

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

[34] Russell, N., van der Aalst, W., ter Hofstede, A.: Workflow Exception Patterns, *Proc. of CAiSE'06*, 4001, Springer, 2006.

[35] Vaz, C., Ferreira, C., Ravara, A.: Dynamic Recovering of Long Running Transactions., *Proc. of TGG'08*, LNCS, Springer, 2008.

[36] Vieira, H. T., Caires, L., Seco, J. C.: The Conversation Calculus: A Model of Service-Oriented Computation, *Proc. of ESOP'08*, 4960, Springer, 2008.

[37] Wirsing et al., M.: Semantic-based Development of Service-Oriented Systems, *Proc. of FORTE'06*, 4229, Springer, 2006.

[38] World Wide Web Consortium: *SOAP Version 1.2 Part 1: Messaging Framework*, `http://www.w3.org/TR/soap12-part1/`.

[39] World Wide Web Consortium: *Web Services Description Language 1.1*, `http://www.w3.org/TR/wsdl`.