Dynamic fault handling mechanisms for service-oriented applications [*]

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

Abstract

Dynamic fault handling is a new approach for dealing with fault management in service-oriented applications. Fault handlers, termination handlers and compensation handlers are installed at execution time instead of being statically defined. In this paper we present this programming style and our implementation of dynamic fault handling in JOLIE, providing finally a nontrivial example of its usage.

1. Introduction

Service-oriented applications are usually distributed, and they are generally composed of a large number of services aggregated by means of so-called orchestrators. These services usually rely upon a workflow programming approach, enhanced with specific communication primitives for exchanging messages with other peers. This is the case of WS-BPEL (BPEL for short) [9], which is an XML based language for building orchestrators. In this context we usually refer to the term service as a loosely coupled application with a standardized interface (à la Web Services); moreover we use the term service composition as the ability to design orchestrators in order to manage business activities which involve different services. In the last years we have investigated composition by proposing a formal calculus, named SOCK [5], which formalizes its basic mechanisms and an implementation of it called JOLIE [8]. Due to the distributed nature of service-oriented applications, one of the most interesting issues raised by service composition is fault management. Indeed, given a system of services, a fault raised by a single service could be propagated by triggering a fault message chain that can involve several services. Mechanisms such as *fault handlers*, *termination handlers* and *compensation handlers* have been proposed for orchestration languages, as for example in BPEL, for providing programming constructs which allow for the management of recovery activities within each orchestrator. Fault handlers are used for directly managing faults, termination handlers are used for smoothly terminating an ongoing activity when an external fault occurs and compensation handlers are used for undoing the effect of a completed activity during error recovery. These constructs are usually programmed statically [9]. Recently, we have studied the fault management issues in the service-oriented context from a formal view point and in [4] we have proposed a new approach for dealing with error recovery: *dynamic handling*. According to this new approach handlers are not statically programmed but they are dynamically installed at execution time. In this way fault handlers and termination handlers can be tuned depending on the part of code which has been already executed. From this perspective, a handler installation sets a point in the executing code after which recovery procedures must be changed. Therefore, in order to avoid that a fault can break the code before installing a handler, it is fundamental that a handler which is willing to be installed must be served with priority w.r.t. fault handling procedures. Our formal framework guarantees that faults cannot be managed before the installation of a handler which is willing to be installed. Furthermore, since besides traditional one-way communication, service-oriented computing usually supports a bidirectional communication pattern composed by the *solicit-response* and the symmetric *request-response* operation, we also considered how faults interact with these two communication patterns.

In this paper, we present the implementation of our formal framework by enhancing the JOLIE language with all the primitives for dealing with fault management: *scope(q)*$\{P\}$ allows for the definition of a scope named q whose inner process is P, *throw(f)* allows for raising a fault f by possibly specifying some extra data (*throw(f,data)*), *install(*$\mathcal{H}$*)* allows for the installation of a handler where $\mathcal{H}$ is a function from fault and scope names to processes, *comp(q)* allows for the execution of a compensation handler where q is the name of the scope to be compensated, *cH* al-

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

978-0-7695-3399-5/08 $25.00 © 2008 IEEE
DOI 10.1109/ECOWS.2008.20

lows for the retrieving of the previous installed handler for a given scope or fault, *op@loc(i)(o)[$\mathcal{H}$]* is an enhanced version for the primitive SolicitResponse which allows for handler installation at the response message reception. Finally, *dynamic code generation* allows us to freeze data within an installed process. It is worth noting that our implementation exactly follows the formal specification defined in our previous works and it satisfies some basic properties proved there.

In Sections 2 and 3 we provide a survey on fault management key concepts and JOLIE implementation respectively. In Section 4 we present a nontrivial example and, finally, in Section 5 conclusions and related works are presented.

2. Key concepts

Fault 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 recovering. Termination and compensation are mechanisms exploited to recover from errors. Termination is automatically triggered when a running scope must be smoothly stopped because of a fault thrown by a parallel process. Compensation, instead, is explicitly invoked by the programmer to undo the effect of a scope whose execution has already successfully completed. Recovering mechanisms are implemented by exploiting *handlers* which are processes to be executed when faults, terminations or compensations occur. Handlers are defined within a scope which represents the execution 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, termination handlers are executed when a scope is reached by a fault raised by an external process and, finally, compensation handlers have to be explicitly invoked by another handler for recovering the activities of a child scope whose computation has already successfully finished. At runtime, when a fault f is raised within a scope q, all its enclosed running activities are terminated. Note that any of the enclosed activities could be a scope, and in this case its termination handler is automatically executed. After that, if q has a fault handler for f, it executes it. Otherwise, the fault is propagated upwards to the parent scope. It is worth noting that handlers can be programmed to compensate any child scope that has successfully completed its activity before f was raised. Compensation is achieved by executing the related compensation handler. Fig. 1 provides an intuitive representation of handler mechanisms where numbers represent ordered events and *stm1,stm2,...,stmn* represent a list of generic statements. A scope A encloses a generic process P and two scopes B and C. At 1 scope C terminates successfully by promoting its compensation handler to be executable by the enclosing scope A. At 2, process P raises a fault which is propagated to scope B. We suppose that B is still executing when reached by the fault so, at 3, it executes its termination handler and terminates. At 4 the fault handler of scope A is executed and, at 5, it compensates scope C (supposing that the handler specifies so).

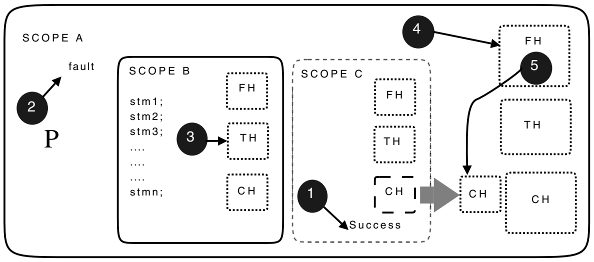

Figure 1. Handler mechanisms

2.1. Static approach

Usually, error recovery is managed by statically associating handlers to scopes, i.e. providing a primitive like $\mathtt{scope}_q(P, \mathcal{FH}, \mathcal{TH}, \mathcal{CH})$, which defines a scope with name q, executing process P and fault, termination and compensation handlers $\mathcal{FH}$[1], $\mathcal{TH}$ and $\mathcal{CH}$, respectively. In some cases static declaration of handlers is not enough to easily model a given scenario. For instance, consider the following pseudo-code:

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

where R is a generic process (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 a 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. For instance, let us introduce in the previous example an array named a which stores a 0 if the process P is executed and 1 otherwise.

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

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

In this case, 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.

2.2. Dynamic approach

In order to address the problem raised by the static approach, in [4] we have proposed *dynamic handling*, which allows for the updating of the handlers while the computation progresses. Technically, we consider a scope construct of the form $\texttt{scope}_q\{P, \mathcal{H}\}$ where q is the name of the scope, P is the process to be executed, and $\mathcal{H}$ is a function associating fault handlers to fault names and termination and compensation handlers to scope names. No handlers are specified when the scope is defined but they are installed dynamically in $\mathcal{H}$ during the execution of P. Dynamic handling installation is addressed by two specific primitives: $\mathrm{inst}(\mathcal{H}')$, which updates the current handler function with $\mathcal{H}'$ and cH which introduces the possibility to refer to the previous installed handlers. It is worth noting that the primitive *inst* can associate handlers both to scope names and fault names. As an example, let us consider the following process Θ defined as a scope q where a sequence of processes Q, Q', Q'', Q''' and Q'''' is interleaved with four different install primitives at lines 1,3,5,7:

```
0) $\Theta ::= \texttt{scope}_q\{Q;$
1)              $\mathrm{inst}([f \mapsto P]);$
2)              $Q';$
3)              $\mathrm{inst}([q \mapsto F, f' \mapsto T, f \mapsto P'; cH]);$
4)              $Q'';$
5)              $\mathrm{inst}([f'' \mapsto U, f \mapsto P'']);$
6)              $Q''';$
7)              $\mathrm{inst}([q \mapsto F'])\}$
```

In the following table, where the different columns represent the state of the handler function at a particular code line, we report how the handler function for the scope q is updated when the install instructions are executed.

0)	2)	4)	6)	8)
	$f \mapsto P$	$f \mapsto P'; P$	$f \mapsto P''$	$f \mapsto P''$
		$q \mapsto F$	$q \mapsto F$	$q \mapsto F'$
		$f' \mapsto T$	$f' \mapsto T$	$f' \mapsto T$
			$f'' \mapsto U$	$f'' \mapsto U$

It is worth noting that at the beginning the handler function is empty. The install primitive defines new handlers for a specific fault or scope name if no handlers for that name have already been specified (e.g. fault f at line 1 and fault f' at line 3) whereas it updates the current handler otherwise (fault f at line 3 and 5). In particular, at line 3, the handler for the fault f is not replaced by a new process, but the current one (represented by cH) is enriched with the process P'. The first concept we highlight is:

> *The install primitive updates the current handler for the specified fault or scope name*

The semantics of fault handlers is different from that of termination and compensation ones. The fault handler is executed when the scope is reached by an inner fault. In the example above if we replace the process Q''' with the fault raising primitive $\texttt{throw}(f)$, at line 6 the code is stopped and the fault handler P'' is executed. On the contrary, the termination handler is executed only when a scope is terminated because an outer scope has raised a fault. In order to clarify this aspect we consider the process Θ in parallel with a process R within a parent scope r where a fault handler G is installed for fault f before starting the parallel:

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

Now, assume that R throws a fault f when Θ is executing line 4. Since the fault is raised by an outer scope, the fault handler of Θ for f is not executed, the termination is executed instead. The termination handler is the process associated with the scope name q when the termination is triggered. At line 4, q is joined with process F, which is executed when scope q is terminated. It is worth noting that in Θ, at line 3, the programmer has specified that the termination handler for scope q must be updated because process Q' has been executed and it must be recovered by process F but, if the fault f from R is raised between the execution of Q' and $\mathrm{inst}([q \mapsto F, f' \mapsto T])$, the termination handler is not consistent with the programmer expectations. Such a case is avoided by the semantics of the primitive *inst* which is executed with priority w.r.t. the fault processing mechanism. Thus, if line 2 is executed, the semantics of the install primitive guarantees that also line 3 is executed before any fault is processed. The second concept we highlight is:

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

After Θ has been terminated, the parent scope, which contains both process R and process Θ, can execute its fault handler for f. It is worth noting that the termination mechanism is always propagated to all the children of a given scope which have to be terminated before their father is. In our example let us consider that Q'' at line 4 is a scope k which executes a process W:

```
$\Theta$ ::= scope$_q\{Q$;
              inst$([f \mapsto P])$;
              $Q'$;
              inst$([q \mapsto F, f' \mapsto T, f \mapsto P'; cH])$;
              scope$_k\{W\}; ...\}$
```

If a fault is raised by R when Θ is executing W, first the termination handler for scope k is executed, then the termination for scope q (i.e. F) is executed and, finally, the fault handler for the parent scope r is executed. The third concept we highlight is:

> *Termination is always propagated to sibling and child scopes, and it is always completed before the enclosing scope processes the fault handler*

If scope q finishes successfully without having been interrupted by any faults, the current termination handler ($q \mapsto F'$) is promoted to compensation handler for q inside the parent scope r. A compensation handler can be executed only if it is explicitly required by means of the primitive comp(q), where q is the name of the scope to compensate. The compensation primitive is used within the fault handler of the parent scope and it can take effect only if the compensation handler has been promoted by the child scope, it is skipped otherwise. In the following we modify the fault handler of scope r in order to invoke the compensation for scope q.

$$\texttt{scope}_r\{\text{inst}([f \mapsto G; \texttt{comp}(q)]); (R \mid \Theta)\}$$

The next two concepts we highlight are:

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

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

Note that there is no ambiguity between termination and compensation handlers because a termination handler is executed by the scope itself when interrupted by a fault generated by a parallel activity, whereas a compensation handler can only be executed by the parent scope. This allows also to trivially simulate the static approach with the dynamic one: the construct scope$_q(P, \mathcal{FH}, \mathcal{TH}, \mathcal{CH})$ can be simply rephrased as scope$_q\{\text{inst}([f \mapsto \mathcal{FH}]); \text{inst}([q \mapsto \mathcal{TH}]); P; \text{inst}([q \mapsto \mathcal{CH}]), \mathcal{H}_0\}$ in which the fault and termination handlers are installed before the execution of the activity, the compensation handler at the end, and $\mathcal{H}_0$ defines no handlers. In light of these observations we can rewrite the *while* example of the previous section in the following way:

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

In this case, when P completes its execution, the statement inst$([q \mapsto P'; cH])$ updates the current termination handler for q, pointed by cH, by adding process P' (which specifically compensates process P) to it, whereas if Q is executed the termination handler is updated by adding Q'. When reached by a fault f, scope q executes the last installed termination handler, compensating the whole sequence of activities. Different compensation strategies can easily be programmed. Note that, thanks to the execution priority of the installing primitive, in the example above it should never be the case that an execution of P has been completed and its compensation has not been installed. The same behaviour cannot be obtained in the static approach, where we simulate handlers updating by using bookkeeping variables, as we cannot distinguish whether a variable assignment is related to fault management or not.

2.3. Request-Response communication pattern

The Request-Response communication pattern deals with the sending of a request message and the reception of its own response. Due to the strong relationship between the request and the response messages, such a kind of pattern raises some interesting issues from the point of view of the fault handling mechanism. In SOCK and in JOLIE the Request-Response pattern is modelled by means of two atomic communication primitives: the *RequestResponse* and the *SolicitResponse*. The RequestResponse ($op(i_m, o_m, P)$) primitive specifies the operation op on which the message exchange is performed, the variable i_m on which the request message is stored, the variable o_m where the reply message is stored, and the process P to execute between the reception and the response sending. The SolicitResponse $\overline{op}@loc(o_m, i_m, \mathcal{H})$ specifies the operation op and the location loc to use for performing the message exchange, the variable o_m where the request message is stored, the variable i_m where the reply message will be stored and the handlers $\mathcal{H}$ to install when the response message is received. In particular, the handlers will be installed only if the response message does not contain a fault. It is worth noting that a SolicitResponse is blocked until the response message is received. In Fig. 2.a) we have represented the interplay between a SolicitResponse and a RequestResponse denoted by SR and RR respectively: in step 1 the request message is sent from the SR to the RR, in step

2 the RR process is executed and then, at the end, the response message is sent in step 3.

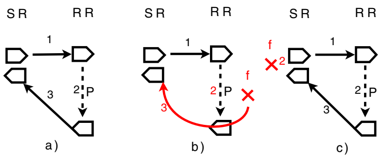

Figure 2. SolicitResponse and RequestResponse behaviours

Between the request and the response messages faults can be raised both on the SR and in the RR side, depending on the contexts they are inserted in. The behaviour we have modelled in SOCK for SR and RR always guarantees that the response message is sent before handling a fault. In Fig. 2.b) a non catched fault f is raised during the execution of process P in the RR, in this case a fault response is immediately sent to the SR which is waiting for the reply message. In Fig. 2.c) a non catched fault f is raised on the SR side before the reception of the response message from the RR, in this case the SR behaviour forces the reception of the response message from RR before handling the fault f. It is worth noting that in the case depicted in Fig. 2.b) a fault f is propagated between the two services that run the SR and the RR, whereas in the case depicted in Fig. 2.c) the fault f is not propagated but it is managed within the service that runs the SR. The main concepts we highlight are:

> *In a RequestResponse primitive, if a non catched fault is raised between the request message and the response one, a fault response is always sent to the invoker.*
>
> *In a SolicitResponse primitive, if a non catched fault is raised between the request message and the response one, the SolicitResponse always waits for the incoming response before the fault is handled by the service.*

In case of fault between the request and the response message, in order to take into account that the request message exchange has been performed, it is important to guarantee the possibility to install a termination handler for recovering from the request both at the level of SR and RR. Since the RequestResponse allows for the definition of an inner process between the request and the response no particular installation mechanisms have been provided. It is sufficient to exploit the install primitive like in the following example:

$$\mathtt{scope}_q\{login(username, result, \mathrm{inst}([q \rightarrow Q]); ...)\}$$

After the reception of the *username*, the RequestResponse *login* installs the termination handler Q for the scope q which encloses the RequestResponse primitive. In the case of the SR, the semantics always guarantees that the response message is received but it is not possible to specify an inner process to be executed between the request and the response. Thus, it may seem sufficient to install the termination handler after the completion of the SR as in the following example:

$$\begin{aligned}\mathtt{scope}_{q'}\{&\mathrm{inst}([q' \rightarrow Q']);\\ &\overline{login}@loc(username', result');\\ &\mathrm{inst}([q' \rightarrow Q''])\}\end{aligned}$$

Such a solution is not correct because if a fault is raised between the request and the response the code execution is stopped and the termination handler previously installed (i.e. Q') is executed after the reception of the response. The following installation of termination Q'' is discarded because the SR is not completed. In order to avoid such a scenario we have enhanced the SR primitive by introducing a handler installation which is atomically performed when the SR receives the response message as in the following example:

$$\mathtt{scope}_{q'}\{\overline{login}@loc(username', result', [q' \rightarrow Q])\}$$

When $result'$ is received the termination for the scope q' is automatically installed[2].

3. Dynamic handling in JOLIE

In this section we present JOLIE, a language for service orchestration, and we analyze how JOLIE can exploit the key concepts of dynamic handling. First, we offer a brief introduction to JOLIE and its features; then, we show how the language implements dynamic handling and the key features of this implementation.

3.1. JOLIE: Java Orchestration Language Interpreter Engine

JOLIE [8], Java Orchestration Language Interpreter Engine, is an open-source project [6] released under the LGPL license [10]. JOLIE is based upon our formal calculus for service orchestration, SOCK [5]. In SOCK we have formalized the basic features of the service-oriented computing paradigm and we have provided a language syntax with a few basic constructs which allows for the composition of services. JOLIE implements the semantics of SOCK, thus allowing for the formal reasoning on JOLIE programs. Language extensions and refinements have been made in order to offer to the programmer a powerful and intuitive

[2] The handler is installed only if the response message does not contain a fault.

environment, suitable to build both orchestrators and single services.

In JOLIE one can use the classic `while` loop instruction and `if-then-else` conditional statement. Also, one can compose statements in a workflow by making sequences, parallel compositions and non-deterministic choices, with the possibility to synchronize parallel processes and to model timeouts. JOLIE can interact with other services by means of communication primitives inspired by the four WSDL operation types (One-Way, Request-Response, Notification and Solicit-Response). Moreover, using its communication primitives and its composition operators, JOLIE can compose other services by exploiting their input operations.

One of the most prominent advantages of JOLIE is the elegant separation between the program behaviour (or workflow) and the underlying communication technologies. The same behaviour can be used with different communication mediums (such as bluetooth, local memory, sockets, etc.) and protocols (such as HTTP, REST, SOAP, SODEP[3], etc.) without being changed. Moreover, the communication infrastructure of JOLIE offers the possibility to extend the JOLIE range of supported communication mediums and protocols by means of simple Java libraries, called JOLIE extensions.

The ability to extend its communication capabilities and the ability to exploit the Java language in its workflows have proven to be key factors in integrating JOLIE with a wide range of existing technologies. Thanks to this advantage, JOLIE can create a service oriented application even by orchestrating legacy applications that do not support the Web Services specifications.

JOLIE provides an intuitive syntax, resembling that of the C and Java languages. This is in contrast with the most credited Web Services orchestration languages, such as XLANG [11] and BPEL, which are based upon XML. Nevertheless, JOLIE can interoperate with XML-based applications by using its XML and Web Services extensions. Moreover, the JOLIE syntax allows the programmer to manipulate XML data structures easily. Consider the following XML element:

```
<person>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
</person>
```

In JOLIE, such an XML element would be constructed with the following code:

```
person.firstname = "John";
person.lastname = "Smith"
```

and the variable `person` would be automatically converted to XML in case of an outgoing SOAP transmission (and vice versa in case of an incoming transmission).

For the sake of clarity, in the following we outline the internal architecture of the interpreter.

Code analysis – JOLIE offers a library for code analysis, which is the same used by the interpreter itself for parsing its input files and obtain an optimized abstract syntax tree. The library offers the possibility to exploit the `Visitor` object oriented design pattern [12], in order to analyze and/or manipulate the parsing result. For example, the JOLIE internal code optimizer and program well-formedness checker (which takes its rules by the SOCK specifications) are implemented by means of this approach.

Object Oriented Interpretation Tree – Responsible for the execution of the behavioural code, the Object Oriented Interpretation Tree (OOIT) is a tree composed by small execution units. Each semantic rule specified by the behavioural layer of SOCK is implemented by an OOIT execution unit. This approach based on encapsulation makes very simple to update the interpreter semantics to follow new developments of SOCK. The OOIT is produced by JOLIE starting from the optimized abstract syntax tree.

Runtime environment – The runtime environment handles the creation of new sessions, the synchronization of processes and the service state. It interacts with all the other components of JOLIE and abstracts the OOIT from session state handling.

Communication core – The communication core permits to keep the OOIT separated from communication related problematics. This component handles incoming connections and internal message routing to the various sessions, along with the service input interface deployment. Moreover, its modularized design permits to easily provide support for new protocols and communication mediums (such as files and local memory).

Summarizing, the main advantages of JOLIE are: (1) it follows rigorously the operational semantics of SOCK, thus allowing for the formal reasoning on JOLIE programs; (2) it can be easily extended by introducing external libraries; (3) it offers a programmer-friendly syntax, that allows for fast orchestration prototyping and the subsequent step-by-step incremental refinement; (4) it is able to interoperate with different technologies and it can be integrated in heterogeneous environments; (5) the program workflow is independent from the underlying communication technologies.

3.2. Dynamic handling in JOLIE

JOLIE offers the aforementioned fault handling mechanisms by means of a new set of instructions, each one

[3] SODEP (Simple Operation Data Exchange Protocol) is a new, efficient protocol purposely developed for JOLIE. The protocol has been implemented also as C++, Google Web Toolkit and J2ME libraries.

resembling the primitives that have been introduced in SOCK:
1. throw(f), throw(f,$data$): raise a fault signal f which can be equipped with extra $data$;
2. install($\mathcal{H}$): installs $\mathcal{H}$ in the current scope;
3. comp(q): compensates a successfully finished scope q;
4. cH: refers to the previously installed handler;
5. scope(q)$\{P\}$: executes P inside the scope q;
6. $op@loc$(i)(o)[$\mathcal{H}$]: calls the service located at loc on operation op sending the variable i, installs $\mathcal{H}$ and then waits for a response;
where q is a scope name, f is a fault name, and $\mathcal{H}$ is a function describing the handlers to install. Note that JOLIE supports all the concepts that have been highlighted throughout the paper. In the following we analyze the main features offered by the JOLIE implementation.

Automatic fault transmission – As in SOCK, faults that are thrown from inside a Request-Response operation are automatically transmitted to the caller.

Dynamic code generation – The cH element implies that the language must be able to generate behavioural code dynamically. Consider the following example:

```
1) scope(s) {
2)   install(f => i = i + 2);
3)   install(f => i++; cH)}
```

In (3) the install instruction contains a reference to the current handler. In order to execute the instruction correctly, JOLIE must first replace cH; so, the install instruction that gets executed at (3) is:

```
install(f => i++; i = i + 2)
```

We say that the code (in this case `i = i + 2`) has been *dynamically generated* by the interpreter at the time of installation.

Actual programming experience showed that dynamic code generation must take particular care to the evaluation of expressions. JOLIE offers this feature by means of the ^ operator, which can be used to prefix a variable and *freeze* its state in a handler that is going to be installed. In order to understand this concept, consider the following JOLIE code:

```
scope(s) {
  for(i = 0, i < 3, i++) {
    install(f => println@Console(^i); cH)
  };throw(f)}
```

where `println@Console(` x `)` is the standard JOLIE operation to print x on the console. The program cycles over the `i` variable, and once it completes the `for` block it throws fault `f`, thus causing the installed fault handler to be executed. At each iteration, the `for` body updates the fault handler for `f` by prefixing a console output of `^i` to the currently installed handler; at each installation, the ^ operator replaces the value of `i` with its current value. The final handler for `f`, just before the throw instruction is reached, results then as:

```
println@Console(2);
println@Console(1);
println@Console(0)
```

Structured fault data – JOLIE supports the association of structured data to a fault signal. This ability can be used to attach additional information to a fault, that can be retrieved and used later in the fault handler execution. In order to do this, JOLIE extends the throw instruction to support an optional parameter: throw(f, x). This new primitive attaches the data contained in x to the fault signal f and then raises the latter. In the following we provide a usage example.

```
scope( s ) {
  install(f =>
    // This will print "Hello, world!"
    println@Console(s.f.message));
  data.message = "Hello, world!";
  throw(f, data)}
```

Note that in order to refer to the fault data of `f` we exploit the scope name: **`s`**`.f.message`. This is due to the fact that in SOCK and JOLIE variables are shared, so if two scopes in parallel receive the same fault by their internal activities we need to store their respective fault data in two different variables to avoid a memory race condition. In order to do so, we exploit the name of the scope receiving the fault as a prefix. Note also that fault data is transparently transmitted over the network in case of faults thrown inside a request-response operation execution.

Basic safety properties – In [4] we point out some basic properties that SOCK always assures about fault handling. We describe again those properties here, as JOLIE respects them and the programmer can make use of them in his or her reasoning about an orchestrator behaviour: (1) a scope ends successfully *if* it does not throw any fault upstream, i.e. its internal process does not throw any fault or the scope handles all of the faults thrown by its internal process; (2) a scope installs its compensations in the parent scope *iff* it ends successfully; (3) a scope that is terminated by a sibling parallel process (i) does not end successfully and (ii) does not throw any fault upstream anymore; (4) if a Solicit-Response process starts (i.e. it sends a request message) it always waits for the response; (5) if a Request-Response process starts (i.e. it receives a request message) it always supplies a response to the caller, be it a normal message or a fault. Properties (1), (2) and (3) offer to the programmer the means to safely predict the behaviour of a scope. Properties (4) and (5) ensure that the Request-Response pattern is always respected, even when the program has to deal with fault handling.

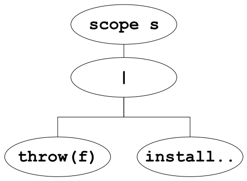

Figure 3. OOIT scope representation

Install statement priority – One of the most important aspects of our dynamic handling approach is that the install primitive has priority w.r.t. fault processing. This introduces the necessary determinism to assure that fault handling behaviour is predictable by the programmer. JOLIE implements this mechanism exploiting its internal execution architecture, the Object-Oriented Interpretation Tree (OOIT). In the following, we hint to how this is obtained. Consider the following code

```
scope( s ) {
  throw( f )
  | install(f => println@Console("Hello, world!"))
}
```

where the behaviour is composed by two processes in parallel: the former throws a fault f, whereas the latter installs a fault handler for f. This workflow is internally represented by the OOIT in Figure 3. Basically, every OOIT node is responsible for implementing a specific SOCK semantic rule. Fault signals are propagated upwards in the tree. When the fault signal f reaches the | node (i.e. the node representing the parallel composition), the latter informs every other child node that the parallel composition is now in a fault handling situation and waits for their confirmation. Normally, a node aborts its execution and returns immediately, but this is not the case for a node that has to perform a handler installation: an *install* node returns its confirmation to the | node only after actually performing the installation. Thus, the parallel composition is forced to wait for the handler installation and it propagates the fault signal to the scope(s) node only afterwards.

4. Example

In this section we discuss the implementation of the automotive scenario [13], which has been chosen as case study inside the EU Project SENSORIA[4]. 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.

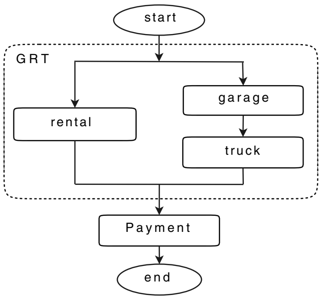

Figure 4. Car service workflow

Both the garage booking service and the truck one exhibit two operations: book and revbook. The former allows for booking the garage (resp. truck) and the latter allows for revoking the reservation in case of failure. Besides these operations the car rental service provides also the redirect operation which allows for the redirection of the rented car from a destination to another one. The bank service[5] offers the following operations: openTransaction, payTransaction and reverseTransaction where openTransaction allows for the opening of a transaction to pay, payTransaction allows for the payment of an opened transaction and, finally, reverseTransaction allows for the revocation of a payment. The workflow of the car service, which is in charge to rent a car and book both the garage and the truck, is represented in Fig. 4. It is composed by two main parts: a first parallel composition of booking activities where the rental booking activity is executed in parallel with the sequential composition of the garage booking and the truck booking activities and a second payment part where all the expected payments are performed. For the sake of brevity, in the following we only discuss the car service workflow which includes all the basic JOLIE mechanisms discussed so far. We will exploit variable flags rFlag, gFlag and tFlag for denoting if the reservation for the car rental service, the garage service and the truck service respectively, are available (flag value set to true). Let us now discuss the code of the garage activity:

```
1)  define bookGarage {
2)    scope(garage) {
3)      install(BookFault => throw(GarageFault));
4)      book@Garage(failure)(g_payData, g_id);
5)      install(this => revbook@Garage(g_id))}}
```

At line 1 the procedure bookGarage is defined whereas at line 2 we define the scope garage. At line 3 we install the fault handler which manages a booking fault that can

[4] IST-FET Integrated Project Sensoria, contract no. 016004

[5] For the sake of this example we suppose there is only one bank service.

be raised from the garage service: such a fault is re-thrown to the parent scope as a GarageFault fault. At line 4 we perform the garage booking by invoking the operation book of the garage service where failure is the car failure description, g_payData contains the payment data for the garage service and g_id is the reservation id. Finally, at line 5, we install the compensation handler for the scope garage which revokes the reservation. The truck booking activity bookTruck is equal to bookGarage with the exceptions that variables are prefixed by t_ instead of g_ and that the scope name is truck instead of garage.

Differently from *bookGarage* and *bookTruck* the car rental activity *rentCar* installs a compensation handler where a redirection is programmed instead of a booking revoking. Indeed, the car rental reservation will be revoked only if the payment will not succeed and it will be managed by the parent scope that we will show in the following. The code of *rentCar* follows:

```
1)  define rentCar {
2)    scope(car_rental) {
3)      install(BookFault => rFlag = false);
4)      coords = g_coords;
5)      book@CarRental(coords)(r_payData, r_id);
6)      install( this =>  coords = car_coords;
7)          redirect@CarRental(r_id, coords)())}}
```

At line 1 we define the procedure for the car rental activity and at line 2 we open the related scope. At line 3 we install the fault handler related to the booking fault which can be raised by the car rental service. In case of fault the variable flag for the rental service is set to false. At line 4 we set the destination coordinates where the car must be sent and at line 5 we perform the car rental booking by invoking operation book. At line 6-7 we install the compensation handler for the car rental activity where the destination coordinates are set to the car ones and the redirection operation is called. Let us now consider the code of the scope enclosing the three (garage, truck and car rental) activities:

```
1)  define grtActivity {
2)    scope(grt) {
3)      gFlag=true; tFlag=true; rFlag=true;
4)      install(GarageFault => comp(car_rental);
5)                tFlag=false;gFlag=false);
7)      install(TruckFault =>
8)                tFlag=false;gFlag = false;
9)                comp(garage);comp(car_rental));
10)     {rentCar  | {bookGarage ; bookTruck }};
11)     install(this => revbook@RentalPT(r_id);
12)                  comp(garage);comp(truck))}
```

At lines 1-2 we define the procedure and the scope for the composed activity grtActivity. At line 3 we initialize flag variables values, at lines 4-5 we install the fault handler for the fault raised by the garage activity which compensates the rental one (which performs a redirection) and then sets the flag variables for garage and truck activities to false. At lines 7-9 the fault handler for the truck fault is installed: both garage and rental activities are compensated and flag variables for garage and truck are set to false. Line 10 defines the parallel composition of the rental activity and the sequence between the garage and the truck activities. Finally, at lines 11-12 the compensation handler for the scope grt is installed. In this handler the garage and the truck activities are compensated whereas for the rental one we programmed the reservation revoking, instead of redirecting with a compensation, because something has gone wrong in the following payment scope. Finally, we present the code of the main activity, invoking the payment services and completing:

```
1)  main {
2)    install(BankFault => comp(grt));
3)    grtActivity;
4)    scope(payment) {
5)      install(BankFault => throw(BankFault))
6)      if (rFlag != false) {
7)        payTransanction@Bank(r_payData)(payId)[
8)          BankFault =>
9)            reverseTransaction@Bank(^payId);
10)           cH]};
11)     if (gFlag != false) {
12)       payTransanction@Bank(g_payData)(payId)[
13)         BankFault =>
14)             reverseTransaction@Bank(^payId);
15)             cH]};
16)     if (tFlag != false) {
17)       payTransanction@Bank(t_payData)(payId)}
18) }}
```

At line 2 we install the fault handler for a fault that can be raised from a bank payment and we programmed the compensation of the grt scope. At line 3) we execute the grt scope. At line 4 we open the scope payment and at line 5 we install the fault handler for the bank fault by programming a re-throwing towards the enclosing scope. At lines 6-7 we perform the payment for the car rental service which must be done only if the related flag is true. At lines 8-10 we update the fault handler for the bank fault which must take into account the fact that the payment for the car rental service has been done. It is worth noting that current handler, previously installed at line 5, is here updated thanks to the operator cH, furthermore we exploit operator ^ for freezing the transaction identifier payId. At lines 11-15 we perform the payment for the garage service and at lines 16-17 we perform the truck service one. In this last payment the fault handler for the bank fault is not updated because the payment can only finish with success or failure. In the former case no fault is raised whereas in the latter a bank fault is raised but only the previous installed payments must be revoked.

5. Conclusions

In this paper we have presented our implementation of the fault mechanisms we have formally introduced in our previous work [4]. The JOLIE language has been enriched with fault and dynamic handling primitives such as *throw(f)*, *throw(f,data)*, *install*, *scope*, *cH*, *op@loc(i)(o)[$\mathcal{H}$]*, and *dynamic code generation* without altering the formal properties proved in our framework. Thanks to the preservation of the properties, JOLIE can be considered as a good candidate for programming service-oriented applications by following the new dynamic handling approach. In the future we intend to analyze the same mechanisms in choreography languages which are particularly suitable for representing service-oriented systems from a global viewpoint. Moreover, this work could be at the basis for reasoning about an enhancement of WS-BPEL by introducing dynamic handling. In particular a specific $\langle$*install*$\rangle$ primitive, which follows the same semantics presented in [4], could be added within primitive $\langle$*scope*$\rangle$.

Related Works. At the best of our knowledge JOLIE provides the first implementation of dynamic fault handling mechanisms. Other languages deal with fault handling in a service-oriented environment but they follow the static approach. The language which can be mainly related with JOLIE is WS-BPEL [9], whose most credited implementation is activeBPEL [1]. In BPEL fault handlers, termination handlers and compensation handlers are statically defined and the RequestResponse communication instruction is splitted into a *receive* instruction and a *reply* one. COWS [7] is a formal calculus similar to SOCK, even if it does not provide Request-Response communication primitives. In COWS handlers are statically defined and the faults are managed by exploiting three basic mechanisms: *kill*, *protection* and *delimitation*. An on the fly model checker and interpreter for COWS exists and can be found at [3]. Another formal language for service-oriented applications which deals with fault management is CaSPiS whose implementation is JCaSPiS [2]. In CaSPiS handlers are statically defined, and error recovery is approached by means of a basic termination mechanism which also includes an automatic partner notification of the fault as in the JOLIE RequestResponse primitive.

References

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

[2] L. Bettini, R. De Nicola, and M. Loreti. Implementing session centered calculi. In Doug Lea and Gianluigi Zavattaro, editors, *In Proc. of Coordination Models and Languages, 10th International Conference, COORDINATION 2008*, volume 5052 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 2008.

[3] F.Mazzanti. On the fly COWS model checker and interpreter. [http://fmt.isti.cnr.it/cmc/].

[4] C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro. On the interplay between fault handling and request-response. In *Proc. of 8th International Conference on Application of Concurrency to System Design (ACSD 2008)*, IEEE Computer Society Press, pages 190–199, 2008.

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

[6] JOLIE. JOLIE: a Java Orchestration Language Interpreter Engine. [http://jolie.sourceforge.net/], 2006.

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

[8] F. Montesi, C. Guidi, and G. Zavattaro. Composing services with JOLIE. In *In Proc. of 5th IEEE European Conference on Web Services (ECOWS 2007)*, pages 13–22.

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

[10] GNU Project. GNU Lesser General Public License. [http://www.gnu.org/copyleft/gpl.html], 2006.

[11] S. Thatte. XLANG: Web Services for Business Process Design. Microsoft Corporation, 2001.

[12] Wikipedia. Visitor pattern. [http://en.wikipedia.org/wiki/Visitor_pattern], 2007.

[13] M. Wirsing et al. Semantic-based development of service-oriented systems. In *Proc. of FORTE'06*, volume 4229 of *Lecture Notes in Computer Science*, pages 24–45. Springer-Verlag, 2006.