Error Handling: From Theory to Practice*

Ivan Lanese[1] and Fabrizio Montesi[2]

[1] Focus Team, Università di Bologna/INRIA, Bologna, Italy
lanese@cs.unibo.it
[2] Focus Team, Università di Bologna/INRIA,
Bologna and italianaSoftware s.r.l., Italy
fmontesi@italianasoftware.com

Abstract. We describe the different issues that a language designer has to tackle when defining error handling mechanisms for service-oriented computing. We first discuss the issues that have to be considered when developing error handling mechanisms inside a process calculus, i.e. an abstract model. We then analyze how these issues change when moving from a process calculus to a full-fledged language based on it. We consider as an example the language Jolie, and the calculus SOCK it is based upon.

1 Introduction

Nowadays computing systems are made of different interacting components, frequently heterogeneous and distributed. Components exploit each other functionalities to reach their goals, communicating through some network middleware. Components may belong to different companies, and they are not always reliable. Also, the underlying network infrastructure may be unreliable too, thus connections may break and components may disconnect. Nevertheless, applications should provide reliable services to their users. For these reasons, it becomes more and more important to deal with unexpected events, so to be able to manage them and get correct results anyway. That is, error (or fault) handling is today a major concern.

Service-oriented computing is a programming paradigm for developing complex distributed applications by composing simpler, loosely coupled services. This is implemented as a set of standards allowing to describe service interface and behavior, to look for services to perform the task at hand, to invoke them and to compose them so to produce the desired result. What said above about unexpected events holds, in particular, in the case of services. Thus different techniques and primitives for fault handling have been proposed in this field. For instance, WS-BPEL [21], the de-facto standard for web service composition, provides scopes, fault handlers, compensation handlers and termination handlers to deal with unexpected events.

However, the problem of finding good programming abstractions and primitives for programming reliable applications out of unreliable services is far from being solved. Take WS-BPEL for instance. Its specification is informal and unclear, and the interactions between the different primitives not clarified. This is

* Research supported by Project FP7-231620 HATS.

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 66–81, 2010.

witnessed by the fact that different implementations of WS-BPEL behave in different ways on many programs [14]. To avoid ambiguities, to clarify the expected behavior of programs, and to prove properties of the available mechanisms, formal methods are needed. Thus, there have been many proposals trying to specify WS-BPEL semantics in a formal way [16,15,22], and more in general proposing primitives for modeling web services and their fault handling mechanisms (see the related work paragraph below).

However, most of these proposals are at a very abstract level, and quite far from real programming languages. Thus the problem of exporting primitives and techniques from high-level theoretical models to full-fledged programming languages usable to program service-oriented applications in an industrial context has rarely been tackled, even less solved. This paper aims at describing Jolie [20,9], a language for programming service-oriented applications built on top of the calculus SOCK [4], which has been developed and exploited in practice by company italianaSoftware s.r.l. In particular, we will concentrate on its mechanisms for error handling, detailing the reasoning that drove their development, from the theoretical calculus SOCK to the full language Jolie.

SOCK and Jolie are a good choice to exemplify our ideas. First, they propose a novel approach to error handling with original features such as dynamic handler update and automatic fault notification. Also, Jolie has been developed closely following the semantics of SOCK, in particular as far as its error handling mechanisms are concerned.

Related works. There are many works in the literature on error handling for concurrent systems, such as service-oriented computing ones. Many of them are based on process calculi. The mechanisms they propose range from basic constructs such as the interrupt of CSP [8] and the try-catch found in most programming languages, to complex proposals such as the ones of Webπ [13], StAC [5], SAGAs calculi [3], dcπ [24], SOCK [6], ... Those models differ on many respects, ranging from flow composition models, where basic activities are composed and compensated (e.g., StAC or SAGAs calculi), to calculi taking into account communication and distribution, aiming at modeling distributed error handling (e.g., Webπ or SOCK). Another thread of research [2,11,12] tries to compare the expressive power of different models.

Our work has however a different perspective: we are not proposing new mechanisms nor comparing existing ones, but analyzing the requirements that all those mechanisms have to satisfy. We are not aware of other papers on this topic.

Structure of the paper. Section 2 introduces the basics of SOCK, without considering error handling. Section 3 discusses the main aims that should be reached when developing error handling mechanisms, considering both the cases of a calculus and of a full-fledged language. Section 4 presents the main features of SOCK for error handling and illustrates how it has tried to fulfill the requirements in Section 3. Section 5 does the same analysis for the peculiar mechanisms of Jolie. Finally, Section 6 concludes the paper.

Table 1. Service behavior syntax

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

$P, Q, \ldots ::= \mathbf{0}$		null process		
	$\overline{\epsilon}$	output	ϵ	input
	$x := e$	assignment	if χ then P else Q	conditional
	$P; Q$	sequential comp.	$P \mid Q$	parallel comp.
	$\sum_{i \in W} \epsilon_i; P_i$	non-det. choice	while χ do (P)	iteration

2 SOCK

In this section we introduce SOCK [4], the calculus underlying Jolie [20,9]. We leave to next sections the description of its approach to fault handling, concentrating here on its standard behavior.

SOCK is a three-layers calculus. The behavior layer describes how single services act and communicate. The engine layer describes how the state of services is stored and how their sessions are instantiated and managed. The network layer allows to compose different engines in a network.

Error handling is mostly dealt with at the behavior layer, thus we will concentrate on it. We refer to [4] for a description of the other layers.

The language for defining behaviors in SOCK is inspired both from concurrent calculi, featuring for instance a built-in parallel composition operator, and imperative languages, providing for instance assignment (SOCK is stateful) and sequential composition.

A main point in SOCK behaviors concerns communication. SOCK behaviors communicate with each other using two modalities (inspired by WSDL [25]): one-way, where one message is sent, and request-response, where a message is sent and a response is computed and sent back.

To define the syntax of SOCK behaviors we consider 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. Also, we use z to range over locations. The syntax for processes is defined in Table 1. There, $\mathbf{0}$ is the null process. Outputs can be notifications (for one-way communication) $\overline{o}@z(\boldsymbol{y})$ or solicit-responses (for request-response communication) $\overline{o_r}@z(\boldsymbol{y}, \boldsymbol{x})$ where $o \in \mathcal{O}$, $o_r \in \mathcal{O}_R$ and z is a location. Notification $\overline{o}@z(\boldsymbol{y})$ sends a one-way communication to operation o located at location z (locations of behaviors are defined at the network layer), and variables $\boldsymbol{y}$ contain the data to be sent. Similarly, solicit-response $\overline{o_r}@z(\boldsymbol{y}, \boldsymbol{x})$ sends a request-response communication to operation o_r located at location z, communicating values in variables $\boldsymbol{y}$, and then waits for an answer. When the answer is received, received values are assigned to variables in $\boldsymbol{x}$. Dually, inputs can be one-ways $o(\boldsymbol{x})$ or request-responses $o_r(\boldsymbol{x}, \boldsymbol{y}, P)$ where the notations are as above. Additionally, P is the process to be executed upon request to produce the response. Assignment $x := e$ assigns the result of the expression e to

the variable x. We do not present the syntax of expressions: we just assume that they include the arithmetic and boolean operators, values, variables and arrays. Conditional is written as $\mathsf{if}\ \chi\ \mathsf{then}\ P\ \mathsf{else}\ Q$. $P;Q$ and $P|Q$ are sequential and parallel composition respectively, whereas $\sum_{i\in W}\epsilon_i;P_i$ is input-guarded non-deterministic choice. Finally, $\mathsf{while}\ \chi\ \mathsf{do}\ (P)$ models iteration.

Example 1. Let us consider a (very simplified) service for performing money transfers between two accounts. Such a service can be invoked by:

$$\overline{pay_r}@bank(\langle src, dest, amount\rangle, \langle transId\rangle)$$

where *src* is the source account, *dest* the destination account, *amount* the amount of money to be moved and $transId$ a transaction Id to be used for later referring to the transaction.

A possible implementation for the service (again, very simplified) could be:

$$\begin{array}{l} pay_r(\langle src, dest, amount\rangle, \langle transId\rangle,\\ \quad acc[src] := acc[src] - amount;\\ \quad \underline{acc[dest]} := acc[dest] + amount;\\ \quad \overline{gen - id_r}@bank(\langle src, dest, amount\rangle, \langle transId\rangle)) \end{array}$$

We assume that this behavior is located at location *bank* and that there is another service at the same location, $gen - id$, which takes care of generating the transaction Id. Also, *acc* is an array containing accounts' credit.

3 The Quest for Error Handling Primitives

As described in the Introduction, the problem of finding good programming primitives for error handling is hard, as witnessed by the huge number and variety of proposals that have been put forward in the literature. Even considering a unique kind of model, many variants exist. SAGAs calculi [3,2,10] for instance may differ on whether parallel flows of computation are interrupted when an error occurs, on whether the compensation is centralized or distributed, and on whether the order of compensations depends on the static structure of the term or on the dynamic execution.

The difficulty in finding the best model for error handling in service-oriented computing is due to the many concerns such a model has to answer:

full specification: the model should define the behavior of error handling primitives in all the possible cases, including when the management of different errors interfere;

expressiveness: the available primitives should allow to specify all the error handling policies that may be necessary to program complex applications;

intuitiveness: the behavior of the provided primitives should match the intuition of programmers, allowing to understand the behavior of the applications;

minimality: we look for the simplest possible set of primitives able to model the required behavior, and, in particular, the different proposed mechanisms should be as much orthogonal as possible.

We describe in the next section how the error handling mechanisms proposed for SOCK have tried to satisfy the requirements above.

However, when moving from theoretical models to full-fledged languages, while most of the concerns above remain (actually, intuitiveness becomes even more important, while minimality is less critical), new ones emerge. The main ones are the following:

usability: the proposed primitives should be easy to use for the programmers when developing complex applications: while this is connected to intuitiveness and expressiveness, this goes beyond. For instance, this includes the development of suitable macros or additional primitives to simplify the writing of common patterns. Note that this is in contrast with the concern for minimality, which is more important in theoretical models than in real languages;

robustness: most theoretical models assume perfect communications, and do not consider low level failures, however these failures may happen in practice, and should be taken into account;

compatibility: in practice applications do not live alone, but they are immersed in a world including the network middleware and other applications, possibly developed using different languages and technologies: thus, in real languages, mechanisms should be provided to interact with other entities, which may follow different policies for error handling.

These additional concerns force the error handling approaches used in practice to be different, and in general more complex, w.r.t. the ones considered in theoretical models. This makes difficult to export the results obtained working on theoretical models (expressiveness results, property guarantees) to full-fledged languages. We will describe in Section 5 how those practical concerns have influenced the development of Jolie, discussing whether properties of SOCK are preserved in Jolie.

4 Error Handling in SOCK

Error handling in SOCK has been inspired by error handling in WS-BPEL, but explored some new directions, in particular concerning dynamic handler update and automatic fault notification.

As in WS-BPEL, error handling in SOCK is based on the concepts of fault, scope, fault handler, termination handler and compensation handler. A fault is an unexpected event that causes the interruption of the normal flow of computation. A scope defines the boundaries for error handling activities. In particular, each scope defines handlers specifying how to manage internal and external faults. A handler is a piece of code specifying how to react to particular faults. We consider three kinds of handlers: fault handlers specify how to deal with

Table 2. Service behavior syntax

$P, Q, \ldots ::= \ldots$		standard processes		
	$\{P\}_q$	scope	$\mathsf{inst}(\mathcal{H})$	install handler
	$\mathsf{throw}(f)$	throw	$\mathsf{comp}(q)$	compensate
	cH	current handler	$\overline{o_r}@z(\boldsymbol{y}, \boldsymbol{x}, \mathcal{H})$	solicit-response

internal faults, termination handlers specify how to smoothly terminate a scope when an external fault reaches it, compensation handlers specify how to undo the activities performed by a scope that already terminated with success, if this is needed for error recovery. All these concepts are realized by extending the syntax of SOCK with the primitives in Table 2. There f denotes a fault name and q a scope name. Furthermore, $\mathcal{H}$ denotes a function from fault and scope names to processes. We refer to [7] for a detailed description of the behavior of the primitives, including the formal semantics.

As already said, a scope $\{P\}_q$ is the main mechanism for structuring error handling. It has name q, and executes process P taking care of its faults. At the beginning, it defines no policy for error handling: policies are specified dynamically by installing handlers using operation $\mathsf{inst}(\mathcal{H})$. The intended semantics is that assigning a process P to a fault name f defines P as fault handler for f, while assigning P to the name q of the scope defines P as its termination handler. Handlers may replace or update previously defined handlers with the same name. This is done using the placeholder cH (for current handler), that during installation is replaced by the code of the old handler. Thus for instance $\mathsf{inst}([f \mapsto cH|Q])$ adds Q in parallel to the old handler for fault f.

Primitive $\mathsf{throw}(f)$ throws fault f: the fault propagates by killing the ongoing activities around itself until it reaches a scope. Then the handler for the fault is looked for: if it is available then it is executed, and fault propagation is stopped. Otherwise the scope is killed, and the fault propagates to the outside. During propagation, the fault may kill sibling scopes: in this case their termination handler is executed to ensure smooth termination. All error handling activities are executed in a protected way, thus ensuring that they are completed before taking care of successive errors. During error handling, it may be necessary to undo previously completed activities. To this end, compensation handlers are used. Compensation handlers are defined when a scope successfully terminates, and they correspond to the last defined termination handler. Thus, they are available and they can be executed using primitive $\mathsf{comp}(q)$.

The last main point concerning error handling is related to the request-response communication pattern. This communication pattern enforces a strong relationship between the caller and the callee: for instance, if the callee gives back no answer then the caller remains stuck. For this reason we want that errors on the callee are notified to the caller. In particular, if there is a local fault f in the callee, the same fault is sent back to the caller, where it is re-raised as a local fault, triggering management of the remote fault. In particular, the caller is guaranteed to receive either a successful answer or a fault notification and thus do not get stuck (unless the callee diverges).

Example 2. Consider a slightly more refined version of the bank service in Example 1, which checks first whether there is enough money in the source account, and throws fault f otherwise.

$$\begin{array}{l} pay_r(\langle src, dest, amount\rangle, \langle transId\rangle, \\ \quad \mathsf{if}\ acc[src] \geq amount\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathsf{throw}(f); \\ \quad acc[src] := acc[src] - amount; \\ \quad \underline{acc[dest]} := acc[dest] + amount; \\ \quad \overline{gen - id_r}@bank(\langle src, dest, amount\rangle, \langle transId\rangle)) \end{array}$$

Thus, in case there is not enough money in the source account, the operation will fail and fault f is thrown both at the callee and at the caller sides.

Faults in the caller may influence the communication pattern too: if there is a failure which is concurrent to the solicit-response, different cases may occur. If the fault happens before the solicit-response is started, the solicit-response is not executed at all, and the remote partner is unaffected. If it is after instead, the answer for the partner is waited for. If this is successful, meaning that the remote partner has performed its task, then the local handler is updated according to the handler update defined in the solicit-response primitive. Thus, this handler update can take care of undoing the remote computation. If instead the remote computation has failed, an error notification is received, and the local handler update is not performed, since the remote computation had no effect. Also, the remote fault is not propagated locally, since the local computation has already failed.

Example 3. Consider again the service in Example 2. The client for such a service has to manage fault f. It can be written for instance as:

$$\begin{array}{l} \{\mathsf{inst}([f \mapsto ErrorMsg := \text{"Not enough money for the transfer"}; \\ \quad \overline{print}@user(\langle ErrorMsg\rangle)]); \\ \overline{pay_r}@bank(\langle src, dest, amount\rangle, \langle transId\rangle, [q \mapsto \overline{undo}@bank(\langle transId\rangle)])\}_q \end{array}$$

Now, if everything goes fine, upon receipt of the answer, the handler for q is installed, thus if later this scope has to be compensated, the undo of the payment operation is requested. If instead a fault occurs on the remote side, the handler is not updated and the undo will never be required. Instead, an error message is sent to the user. Even if there is a local fault, the answer will be waited for, and the handler update will be performed only if the answer is successful, thus the termination handler for q will undo the payment iff it has actually been performed.

4.1 Full Specification

The definition of the semantics of error handling (as well as of normal processing), should cover all the possible cases. Questions such as:

1. What happens if, while a fault is being managed, an external fault occurs?
2. What happens if both the caller and the callee of a request-response fail?
3. What happens if a fault handler causes a fault?

should not be left unanswered. Notice however that for informal specifications such as WS-BPEL one [21], it is very difficult to check whether all the cases have been specified. Instead, this is not normally a problem for formal specifications: the only possible transitions are the ones defined by the model, and the model fully describes what happens in each case (at worst, it specifies that no transition is possible). This is for instance the case for SOCK semantics [6]. Considering the questions above, it is easy to deduce the following answers:

1. The internal fault handler is executed in a protected way, thus the management of the external fault has to wait for the completion of local recovery.
2. A fault notification is sent back from the callee to the caller, the handler update specified by the solicit-response is not applied, but the remote fault is not propagated to the caller.
3. The fault is propagated as usual, and dealt with by existing handlers. Note that when the handler for fault f is executed, its definition is removed, thus further faults with the same name should be dealt with by external scopes.

4.2 Expressiveness

The available primitives should be able to express all the policies that may be necessary for programming applications. As stated, this is a very vague goal, since it is quite difficult to guess which kinds of policies may be necessary. For formal models, such a constraint is usually checked by relying on case studies and on encodings. As for SOCK, it has been applied to the specification of the Automotive [26] and Financial [1] case studies of the European project Sensoria [23], and the derived language Jolie is applied every day for programming service-oriented applications such as, for instance, a web portal for managing employer time sheets, VOIP service monitoring, and others. The results of these tests may trigger refinements of the language for improving its expressive power.

Another way of assessing the expressive power of SOCK is via encodings. By showing that another calculus can be encoded into SOCK, one shows that SOCK is at least as expressive as the other calculus. This has been done [12] for instance in the case of SAGAs. The results therein show that both static SAGAs with interruption and centralized compensations [2] and dynamic SAGAs [12] can be modeled into SOCK preserving some notion of behavior. This guarantees that each policy that can be expressed in these flavors of SAGAs can also be expressed in SOCK.

Another result concerning expressiveness is related to dynamic handler update: it is easy to show that SOCK dynamic handler update can easily model WS-BPEL static scopes. In WS-BPEL, each scope has statically associated a fault handler F_i for each fault f_i, a termination handler T and a compensation handler C. Using dynamic handler installation, this can be simulated as follows:

$$\{\mathsf{inst}([f_1 \mapsto F_1, \ldots, f_n \mapsto F_n, q \mapsto T]); P; \mathsf{inst}([q \mapsto C])\}_q$$

In [11] it is shown that dynamic handler update is strictly more expressive than both static recovery (as in WS-BPEL), and parallel recovery (where additional pieces of handlers can be added only in parallel). Albeit this result can not directly be applied to SOCK, since it is proved on a stateless calculus, this is another hint of the expressive power of dynamic handler update.

4.3 Intuitiveness

This is one of the most important, yet difficult to reach, goals for a programming language, and, in particular, for error handling mechanisms. Intuitiveness means that the behavior of the primitives follows the intuition of the programmer (or, better, of a programmer that has understood the basics of the approach). While the formal specification of the calculus is normally quite complex to understand for a programmer without specific background on formal methods, it is required that such a programmer can learn how to program in the language by reading some informal description (one has to resort anyway to the formal specification to work out the behavior in the most complex cases). This becomes much easier if the specification of the language is built on top of a few clear and orthogonal concepts. Having a formal specification, one may guarantee that those intuitive properties are actually valid in all the cases.

Let us consider as an example the case of SOCK scopes. Their behaviors can be characterized by Property 1 below.

Property 1. *A scope may either succeed and thus install its compensation handler, or fail by raising a (unique) fault. Furthermore, if it succeeds, it will never throw faults, and if it raises a fault it will never install its compensation.*

Such a property clearly describes the intuition about scope outcomes, and in [6] it is proved to hold for each SOCK process.

Other sample interesting properties of this kind valid for SOCK follows.

Property 2. *A request-response that terminates its execution always sends back an answer, either a successful one or an error notification.*

Property 3. *When a fault is triggered, there is no handler update that is ready to be installed but has not been installed yet.*

4.4 Minimality

When developing a calculus, one has to look for simplicity and minimality, avoiding for instance redundant or overlapping primitives. This makes the calculus more understandable and simplifies and shortens the proofs. In fact, some of the most successful calculi in the literature such as CCS [17] and π-calculus [18], are the most simple and compact way for modeling the desired features, interaction for CCS and mobility for π-calculus.

SOCK is different w.r.t. those calculi, since it is nearer to current technologies (e.g., it is the only calculus featuring request-response), and thus more complex than other calculi. However, each of its error handling primitives has a well-defined and non-overlapping role. Take for instance the three kinds of handlers. They take care of orthogonal features: internal faults for fault handlers, external faults for termination handlers and undoing of complete activities for compensation handlers.

Also, SOCK provides a unified way to deal with installation of fault and termination handlers (and, indirectly, of compensation handlers), and this dynamic installation is (probably, since this has not been proved for SOCK yet) needed to ensure the expressive power of the language. If SOCK would only allow to add pieces of code in parallel to existing handlers, as happens in dcπ [24], then it would not be minimal, since it has been shown in [11] that such a mechanism can be defined as a macro by exploiting the other constructs.

5 From SOCK to Jolie

As said before, when moving from a theoretical calculus like SOCK to a full-fledged language such as Jolie, new concerns have to be taken into account. Before analyzing those new concerns in detail, we give a general description of Jolie.

Jolie, Java Orchestration Language Interpreter Engine, is an open-source project released under the LGPL license. Its reference implementation is an interpreter written in Java. We refer to [9] for a detailed description of its features, concentrating here on the ones more useful for our discussion (some of them are also outlined in [19]). Jolie refines and extends SOCK so to offer to the programmer a powerful and intuitive environment, suitable to build both complex applications and single services.

One of the most prominent advantages of Jolie is the elegant separation between the program behavior and the underlying communication technologies. The same behavior can be used with different communication mediums (such as bluetooth, local memory, sockets, etc.) and protocols (such as HTTP, REST, SOAP, etc.) without being changed. This can be obtained since Jolie basic data structures are XML-like trees, which are automatically translated from and to XML files (or other suitable formats) for communication. Thus a Jolie variable is a tree, with java-style field access used to denote subtrees, and the array notation used to distinguish different subtrees with the same name. Thus, for instance, *var.subtree*[1] denotes the first subtree of variable *var* named *subtree*.

Jolie may also perform type checking on communicated data: each operation may specify types constraining the kind of data that may be sent or received, and checks are made at runtime to verify that those constraints are satisfied. Constraints are published in the service interface, so that remote partners may know the typing constraints to be satisfied for interacting with a service. We refer to [9] for more details on the type system.

We can now move to the description of how Jolie tries to satisfy the requirements in Section 3.

5.1 Usability

While features such as intuitiveness and expressiveness are fundamental for usability, other needs emerge. In particular, SOCK and its error handling mechanisms have been developed concentrating on issues such as synchronization of different entities and interaction between different error handling activities, but there has been scarce emphasis on data management. However, this aspect becomes fundamental in a real language, where applications managing possibly complex data structures are common. The major importance of data handling in Jolie w.r.t. SOCK has influenced also its mechanisms for error handling, as detailed below.

First, faults in Jolie include also a datum, which is normally used to carry information about the error itself (for instance, an error message, or a stack trace). Thus the throw primitive in Jolie has the syntax `throw(f,v)` where `f` is the fault and `v` a value. The handler can access the data with the special syntax $scopename.faultname$. The prefix *scopename* is needed to avoid interferences in case different scopes manage the same kind of fault concurrently (the scope of variables is the whole behavior). Note that such a modification in the throw primitive does not change the possible error handling policies (e.g., the properties described in Section 4.3 are unaffected), but makes the generation of error messages much easier.

Example 4. Consider the client in Example 3. In Jolie, one can exploit data attached to fault to simplify error handling. Now the server can specify the desired error message together with the fault, including for instance how much money is missing to perform the transfer[1]:

```
pay(varIn)(transId){
    if (acc[varIn.src] >= amount) {nullProcess} else
         {msg = "Missing "+string(amount-acc[varIn.src])+" euros";
         throw(f,msg)}
...
```

The client may use this information to present a more detailed error message to the user.

```
scope(q) {
install(f => print@user(q.f));
...
```

Another important point concerns data management inside handlers. Handlers in SOCK contain variables whose value is looked for when the handler is executed. However, sometimes one wants to use the values that variables had when the handler has been installed, to keep track of information concerning the computation that caused handler installation. This concern has been tackled in Jolie by adding a freeze prefix ^ to variables: if a variable `x` in a handler occurs

[1] The Jolie syntax should be rather intuitive, but we refer to [9] for details.

freezed, i.e. as `^x`, then its value is looked for and fetched at handler update time. Consider for instance Example 3. Assume that many invocations are performed inside a while loop. In case of later error one wants all the transactions to be canceled. Thus the correct handler update would be:

```
this => cH;undo@bank(^transId)
```

Without the freeze operator for `transId`, the value of `transId` in all the calls would be the last one.

As before, this is a mechanism that does not change the error handling properties, but that comes in handy when writing actual programs.

5.2 Robustness

Many calculi, and SOCK in particular, do not model network or node failures, while, in practice, these events may occur. Jolie has faced this problem by adding system faults. A system fault is a fault that is not generated by the throw primitive, but it is generated by the Jolie runtime system to notify the behavior of some problem. In particular, Jolie defines the system fault `IOException`, which is generated when an error occurs during communication. Such a fault can be managed in the same way of other faults, by defining and installing suitable handlers. For instance the Jolie code:

```
scope(q) {install( IOException => ... );
    pay@bank(...)(...)
```

allows to manage network failures in our payment example.

5.3 Compatibility

SOCK mechanisms have been devised to work in a close world, i.e. a world composed only by SOCK processes. However, Jolie applications are aimed at being executed over the net, interacting with other applications developed using different technologies and adhering to different standards.

In Jolie, this is mainly taken care by the communication module, that allows for specifying the protocol to be associated with each communication, and automatically translates messages to and from the desired format. However, a few aspects influence also error handling.

First, while Jolie guarantees remote error notifications inside the request-response pattern, most of the other technologies do not. However, even when interacting with other technologies, communication in Jolie is implemented by connection-oriented technologies such as tcp/ip, unix sockets or bluetooth connections. Thus the Jolie engine is notified when the connection is broken, and can react by generating system fault `IOException`. This is less informative w.r.t. the usual Jolie error notification, which describes exactly the kind of fault that happened on the remote client, but it is however enough to preserve Property 2 (or better its dual).

Another compatibility issue concerns typing. Assuming that each service correctly exposes its typing information, it would be enough to check types when messages are sent. However, when interacting with non Jolie applications there is no guarantee that they check types of communicated messages, thus Jolie services may receive ill-typed messages. For this reason, type checking is also performed on incoming messages. Type errors are managed in different ways according to where they happen. In one-way operations, a type mismatch of an outgoing message generates locally a system fault `TypeMismatch`. Instead, incoming messages that do not respect typing are discarded. The management is similar for request-responses, but, in case of type mismatch in receptions, the sender is also notified with a `TypeMismatch` fault, thus ensuring the preservation of the properties of the request-response pattern.

5.4 Property Preservation

As we have seen in the previous sections, Jolie is an extension and a refinement of SOCK. Also, some of the assumptions that are used to prove SOCK properties do not always hold for Jolie programs in a real environment. Thus, proving that a property of SOCK programs, such as one of those in Section 4.3, holds also for Jolie applications is non trivial.

We discuss now a few of the reasons that make this happen, analyzing their effect on a few sample properties.

Low level errors: SOCK, and theoretical models in general, rely on some basic assumptions ensuring the correct behavior of the system itself. Thus global failures due for instance to end of memory, to system crashes or to programming errors in the Jolie implementation are not considered. It is clear that these kinds of errors break most of the interesting properties, thus one has to assume that these events do not occur. One can exploit formal methods to ensure that these assumptions are satisfied, but this requires dedicated techniques whose description goes far beyond the aim of this paper. For instance, end of memory can be checked and avoided by a suitable resource analysis, system crashes superseded via techniques for reliability such as the use of redundant engines, and errors in the Jolie implementation avoided by using certified compilers and correctness proofs.

Jolie added features: as discussed above, Jolie includes features that are not available in SOCK, such as data in faults. Other additional features not related to error handling are described in [9]. Those features are normally related to aspects which are abstracted away in models, thus they do not affect global properties such as the ones in Section 4.3 (this has however to be checked for each property and each extension). However, because of this, not all Jolie programs are correct SOCK processes, thus it becomes much more difficult to prove properties of specific programs. To this end one has to find a SOCK process which is equivalent to the Jolie one, trying to implement Jolie additional features as macros. When this is not possible, one has to extend the theory to match the practice. For instance, a typed theory of

SOCK is not yet available, but it is on our research agenda. This will allow to prove properties of Jolie type system.

Assumptions on the environment: we refer here to the fact that network failures are not modeled in SOCK, and that interaction with non Jolie programs may raise new issues, as described in Section 5.3. In these cases, one may think to extended models taking care of this, but, mainly for interaction with non Jolie programs, it becomes quite difficult because of the huge variety in their behaviors. Thus, the simplest approach is to analyze their impact on each property, as outlined in Section 5.3, and introduce in Jolie mechanisms to deal with these problems in a uniform way w.r.t. similar issues in SOCK programs. An example of this is the introduction of system faults, which can be managed similarly to normal faults, and can enjoy (most of) their properties. Clearly, local properties such as Property 1 are largely unaffected by these issues, while properties concerning communication such as Property 2 are less robust.

6 Conclusion and Future Works

In this paper we have discussed the main concerns that should be kept into account when designing error handling mechanisms for service-oriented computing. We have considered both the design of a theoretical calculus and of a full-fledged language. We have considered the language Jolie and the underlying calculus SOCK as an example.

Concerning future work, the relations between formal models and practically relevant languages for service-oriented computing are still largely unexplored. Even in the case of SOCK/Jolie, which have been developed in a strongly connected way, many mismatches exist. Theory should be developed so to match interesting aspects of Jolie applications such as the type system, or network failures. For other differences instead, analysis should be carried out so to better understand the effect that they have on formal properties. However, Jolie is continuously evolving to face new programming challenges, thus making it a moving target. For instance, timeouts are an important aspect in practice, to break deadlocks, and work for introducing them in Jolie is ongoing.

Acknowledgments. We thank Gianluigi Zavattaro for his useful comments.

References

1. Banti, F., Lapadula, A., Pugliese, R., Tiezzi, F.: Specification and analysis of SOC systems using COWS: A finance case study. In: Proc. of WWV 2008. ENTCS, vol. 235, pp. 71–105. Elsevier, Amsterdam (2009)
2. Bruni, R., et al.: Comparing two approaches to compensable flow composition. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 383–397. Springer, Heidelberg (2005)

3. Bruni, R., Melgratti, H., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: Proc. of POPL 2005, pp. 209–220. ACM Press, New York (2005)
4. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: SOCK: a calculus for service oriented computing. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 327–338. Springer, Heidelberg (2006)
5. Butler, M.J., Ferreira, C.: An operational semantics for StAC, a language for modelling long-running business transactions. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 87–104. Springer, Heidelberg (2004)
6. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: On the interplay between fault handling and request-response service invocations. In: Proc. of ACSD 2008, pp. 190–199. IEEE Computer Society Press, Los Alamitos (2008)
7. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: Dynamic error handling in service oriented applications. Fundamenta Informaticae 95(1), 73–102 (2009)
8. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
9. Jolie website, `http://www.jolie-lang.org/`
10. Lanese, I.: Static vs dynamic sagas. In: Proc. of ICE 2010 (to appear, 2010)
11. Lanese, I., Vaz, C., Ferreira, C.: On the expressive power of primitives for compensation handling. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 366–386. Springer, Heidelberg (2010)
12. Lanese, I., Zavattaro, G.: Programming Sagas in SOCK. In: Proc. of SEFM 2009, pp. 189–198. IEEE Computer Society Press, Los Alamitos (2009)
13. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
14. Lapadula, A., Pugliese, R., Tiezzi, F.: A formal account of WS-BPEL. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 199–215. Springer, Heidelberg (2008)
15. Lohmann, N.: A feature-complete petri net semantics for WS-BPEL 2.0. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 77–91. Springer, Heidelberg (2008)
16. Lucchi, R., Mazzara, M.: A pi-calculus based semantics for WS-BPEL. J. Log. Algebr. Program. 70(1), 96–118 (2007)
17. Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
18. Milner, R., Parrow, J., Walker, J.: A calculus of mobile processes, I and II. Information and Computation 100(1), 1–40, 41–77 (1992)
19. Montesi, F., Guidi, C., Lanese, I., Zavattaro, G.: Dynamic fault handling mechanisms for service-oriented applications. In: Proc. of ECOWS 2008, pp. 225–234. IEEE Computer Society Press, Los Alamitos (2008)
20. Montesi, F., Guidi, C., Zavattaro, G.: Composing services with JOLIE. In: Proc. of ECOWS 2007, pp. 13–22. IEEE Computer Society Press, Los Alamitos (2007)
21. Oasis: Web Services Business Process Execution Language Version 2.0 (2007), `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`
22. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL. Sci. Comput. Program. 67(2-3), 162–198 (2007)

23. Sensoria Project. Public web site, http://sensoria.fast.de/
24. Vaz, C., Ferreira, C., Ravara, A.: Dynamic recovering of long running transactions. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 201–215. Springer, Heidelberg (2009)
25. W3C: Web services description language (wsdl) version 2.0 part 0: Primer (2007), http://www.w3.org/TR/wsdl20-primer/
26. Wirsing, M., et al.: Semantic-based development of service-oriented systems. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 24–45. Springer, Heidelberg (2006)