Advanced Mechanisms for Service Combination and Transactions*

Carla Ferreira[2], Ivan Lanese[1], Antonio Ravara[2], Hugo Torres Vieira[2], and Gianluigi Zavattaro[1]

[1] Focus Team, Università di Bologna/INRIA, Italy
{lanese,zavattar}@cs.unibo.it
[2] CITI and Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal
{carla.ferreira,aravara,htv}@fct.unl.pt

Abstract. Languages and models for service-oriented applications usually include primitives and constructs for *exception* and *compensation* handling. Exception handling is used to react to unexpected events while compensation handling is used to undo previously completed activities. In this chapter we investigate the impact of exception and compensation handling in message-based process calculi and the related theories developed within SENSORIA.

1 Introduction

Long-running transactions (henceforth LRTs) are computer activities that may last long periods of time. These kinds of activities are particularly common in systems composed by loosely coupled components communicating by message passing, like most distributed systems and, in particular, service-oriented systems.

Due to the nature of these systems and to the time duration of the activities, it is not feasible to lock (non-local) resources, and thus, LRTs do not enjoy some of the usual ACID properties of database transactions (namely isolation, since the execution of a single LRT is not intended to block the whole system). Therefore, to recover from partial executions of LRTs (due to their abortion because of system failures like unreachability of a partner or time-out of communication, or to some other unexpected event), it is necessary to foresee special activities to regain system consistency, i.e., to *compensate* the fact that the transaction has been aborted. These activities should be triggered in case of transaction failure, and need to be programmed *a priori*. Note that, in general, the execution of a compensation does not exactly "undo" the activities already performed by the LRT (what is, in general, impossible).

1.1 Content of the Chapter

Programming or specification languages provide these days two kinds of recovery mechanisms: *exception handling* and *compensation handling*. The former uses

* This work has been partially sponsored by the project SENSORIA, IST-2005-016004.

M. Wirsing and M. Hölzl (Eds.): SENSORIA Project, LNCS 6582, pp. 302–325, 2011.

primitives like throw to raise failure signals and try-catch to manage them. The latter uses primitives to install and activate dedicated compensation activities. This chapter presents linguistic primitives and associated semantic models for dealing with transaction failure. The main features under inspection are the mechanisms to deal with:

1. failures: *exceptions* or *compensations*;
2. non-interruptable units of process execution: *protection operator*;
3. nested computations: *nested transactions* and *nested failures*.

The models are either based on (mobile) process calculi, or on the service-oriented core calculi developed within SENSORIA. We address three questions:

1. What is the relative expressive power of the mechanisms proposed?
   Section 2 is dedicated to basic linguistic primitives for exception and compensation handling. We present: (1) a study of the expressive power of two well-known exception handling mechanisms, in the context of the Calculus of Communicating Systems, CCS [28]; and (2) compensation handling mechanisms and their relative expressiveness, in the context of mobile process calculi.
2. How can these recovery mechanisms be used in the context of Service-Oriented Computing (SOC)?
   Section 3 presents the application of the mechanisms in some of the SENSORIA calculi.
3. How can one ensure that the compensation activities implement a particular recovery policy?
   Section 4 presents three different models to reason about the compensation activities: two of them use abstract descriptions of the desired behavior, in BPEL and SAGAs respectively, and the last one defines a state-based compensation model to reason about the correctness of the activities.

1.2 Overview of Process Calculi Approaches

In process calculi there are several approaches toward the formalization of LRTs, whose proposals differ with respect to the mechanisms to recover from transaction failure. Table 1 presents a summary of the use of compensation handling mechanisms in different message-based calculi, which we group in three families: π-calculus [29,34] based, session-based and correlation-based. Compensation handling has been investigated also in the context of event-based communication: this is the subject of Chapter 3-4 where compensation handling is investigated in the context of the Signal Calculus SC [14].

π-calculus Based Calculi. The πt-calculus is an extension of asynchronous polyadic π-calculus [34] with the notion of transaction [2]. The compensation mechanism is static, and transaction abort triggers the execution of the compensations of all terminated subtransactions. The cJoin calculus [7] extends the Join calculus [15] with primitives for representing transactions with static compensations.

Table 1. Features of message-based calculi with compensation handling

	communication mechanism	compensation definition	nested vs non-nested	protection operator
πt [2]	π-based	static	nested	no
c-join [7]	π-based	static	nested	no
webπ [24]	π-based	static	non-nested	implementable
webπ_∞ [27]	π-based	static	non-nested	implementable
dcπ [36]	π-based	parallel	nested	yes
CaSPiS [3]	sessions	static	nested	no
CC [37]	sessions	static	nested	no
COWS [26]	correlation	static	nested	yes
SOCK [18]	correlation	dynamic	nested	implementable

Transactions can however dynamically merge, thus merging their compensations. Laneve and Zavattaro defined webπ [24], which is an extension of asynchronous polyadic π-calculus with a timed transaction construct. An untimed version of webπ, called webπ_∞, was proposed by Mazzara and Lanese [27]. Both webπ and webπ_∞ support a non-nested static compensation mechanism. The dcπ calculus [36] is also based on the asynchronous polyadic π-calculus, extended with primitives for representing nested transactions and dynamic compensations. This is obtained by adding information about compensation update to input prefix. Compensation items are composed in parallel. Section 2 presents in more detail the compensation handling mechanisms of webπ_∞ and dcπ.

Session-Based Calculi. The coordinated handling of exceptions of several parties involved in a service conversation is of particular importance, since an exception local to a party must be somehow propagated to all other parties involved in the service task.

CaSPiS [3] includes primitives for compensating aborted sessions (see Section 3 for more details). The Conversation Calculus, CC [37], supports error recovery with two exception primitives: try-catch and throw. Section 4.3 presents in more detail the exception handling mechanism and the soundness model of CC. In the approach of Carbone et al. [13], as in SCC [4] and CaSPiS [3], such error propagation is modeled internally to the semantics of the exception handling primitives. CC considers a different approach, by providing the exception handling primitives with a standard "local" semantics, leaving to the programmer the task of coordinating the exception handling activities. The approach of Carbone et al. [13] already aims at a typed exception handling model, allowing to prove safety and liveness results.

Correlation-Based Calculi. COWS [26] provides a primitive to kill processes within a scope. We show in Section 4.1 how these primitives can be used to encode a BPEL-style scope construct (BPEL [32] is a language for service orchestration which provides static nested compensations). SOCK [18] includes also

explicit primitives for dynamic handler update and automatic failure notification to remote partners. Section 3 presents in more detail the compensation handling mechanisms of SOCK. An implementation of SOCK, the language JOLIE [21], inherits its fault handling capabilities.

2 Basic Mechanisms

In this section we focus on different basic linguistic primitives that have been proposed for programming long-running transactions (LRTs), inter-relating them. We leave to the next section their application to service-oriented systems.

2.1 Exception Handling

Here we present and compare with respect to expressiveness two well established mechanisms, the first taken from the tradition of process calculi—the interrupt operator of CSP [20]; the second from popular programming languages—the try-catch operator of languages such as C++ or Java.

Interrupt Versus Try-catch. The interrupt operator $P \triangle Q$ executes P until Q executes its first action; when Q starts executing, the process P is interrupted. The $\texttt{try}\, P\, \texttt{catch}\, Q$ operator executes P, but if P performs a throw action it is interrupted and Q is executed instead. We have found these operators particularly useful because, even if very simple, they are the basic building blocks to model the typical operators for programming LRTs.

These two operators are apparently very similar as they both allow for the combination of two processes P and Q, where the first one executes until the second one performs its first action. Nevertheless, there is an interesting distinguishing feature, as shown by the following example.

Consider for instance a bank payment activity PAY, which may set a variable res to false in case of failure. Failure management can be performed quite simply using try-catch:

$$\texttt{try PAY; if } res = \texttt{F then throw else 0; ... catch manageFault}$$

The interrupt operator, instead, needs some help from an external process.

$$\texttt{PAY; if } res = \texttt{F then } throw \texttt{ else 0; ...} \triangle (f.\ \texttt{manageFault}) \ \mid\ \overline{throw}.\overline{f}$$

where we assume that $throw$ synchronizes with $\overline{throw}$ and f with $\overline{f}$. Here in case of failure the external process is called, and then it enables the compensation. When the compensation starts, the main activity is interrupted. Note however that the interruption is not atomic as in previous case.

As seen in the examples, the main difference is that in the try-catch operator, the decision to interrupt the execution of P is taken inside P itself (by means of the execution of the throw action), while in the interrupt operator $P \triangle Q$ such decision is taken from Q (by executing any initial action). Another difference

Table 2. Interrupt vs try-catch

	interrupt	try-catch
replication	$\mathsf{CCS}_{!}^{\triangle}$ existential termination undecidable universal termination decidable	$\mathsf{CCS}_{!}^{\mathsf{tc}}$ existential termination undecidable universal termination decidable
recursion	$\mathsf{CCS}_{rec}^{\triangle}$ existential termination undecidable universal termination decidable	$\mathsf{CCS}_{rec}^{\mathsf{tc}}$ existential termination undecidable universal termination undecidable

between the try-catch and the interrupt operators is that the former includes an implicit scoping mechanism which has no counterpart in the interrupt operator. More precisely, the try-catch operator defines a new scope for the special throw action which is bound to a specific instance of exception handler.

Starting from these intuitive and informal evaluations of the differences between such operators, a more rigorous and formal investigation has been performed [5]. To this aim, two restriction-free fragments of CCS [28] have been considered, one with replication and one with restriction, and they have been both extended with either the interrupt or the try-catch operator thus obtaining four different calculi: $\mathsf{CCS}_{!}^{\triangle}$, $\mathsf{CCS}_{!}^{\mathsf{tc}}$, $\mathsf{CCS}_{rec}^{\triangle}$, and $\mathsf{CCS}_{rec}^{\mathsf{tc}}$ as depicted in Table 2. Calculi without restriction, the standard explicit binder operator of CCS, have been considered in order to be able to observe the impact of the implicit binder of try-catch. Moreover, replication and recursion have been considered separately because in CCS there is an interesting interplay between these operators and binders [9]: in the case of replication it is possible to compute, given a process P, an upper bound to the nesting depth of binders for all derivatives of P (i.e. those processes that can be reached from P after a sequence of transitions). In CCS with recursion, on the contrary, this upper bound cannot be computed in general.

For these four calculi, the decidability of the following termination problems has been investigated: *existential termination* (i.e., there exists a terminating computation) and *universal termination* (i.e., all computations terminate). The obtained results are depicted in Table 2.

These results about the decidability of existential/universal termination in the considered calculi establish two interesting discrimination results:

- **Basic mechanisms for interruption cannot be in general encoded using only communication primitives.** In CCS without restriction, existential termination is decidable [5], while it turns out to be undecidable when either the interrupt or the try-catch operators are also considered.
- **The try-catch mechanism cannot be in general encoded using communication primitives and the interrupt operator.** In the considered calculus with recursion, universal termination is decidable in the presence of the interrupt operator, while this is not the case for try-catch.

2.2 Compensation Handling

The operators above offer a local approach to error handling and compensations: the trigger of the fault, the executing process and the compensation are all defined inside P and Q. However, in a concurrent and distributed system, fault triggers may also arise from other processes running concurrently.

Static Compensations. Such an aspect has been tackled first by webπ [24] and its untimed version webπ_∞ [27]. There, web transactions, i.e., long-running transactions involving web applications, have been considered, and modeled by adding a workunit construct to the asynchronous π-calculus. We concentrate here on the untimed version proposed by [27], since time introduces a degree of expressive power which is orthogonal to the one represented by compensation primitives we investigate here.

A workunit $\langle\!|P\ ;\ Q|\!\rangle_t$ executes process P until a message $\overline{t}$ (without parameters) is received on channel t. After that, process P is killed and compensation Q is executed. Thus message $\overline{t}$ acts as throw in the case of the try-catch operator. However, here message $\overline{t}$ may come both from inside P, as for throw in try-catch, or from parallel processes. Also, the message may be directed to a specific workunit, instead of being forced to kill the nearest enclosing workunit.

Thus the above example of bank payment can be written in webπ_∞ as:

$$\langle\!|\ \texttt{PAY.if}\ res = \texttt{F then}\ \overline{t}\ \texttt{else 0}....\ ;\ \texttt{manageFault}|\!\rangle_t$$

where we used prefixing instead of sequential composition (simply because this is the control flow mechanism provided by webπ_∞). However, the example can be simply modified to allow for an external activity to interrupt the transaction. Assume that in parallel some checks on the payment are done. If the checks do not succeed, the transaction can also be interrupted by the parallel process:

$$\langle\!|\ \texttt{PAY.if}\ res = \texttt{F then}\ \overline{t}\ ;\ \texttt{manageFault}|\!\rangle_t\ |\ ....\texttt{if}\ checkRes = \texttt{F then}\ \overline{t}...$$

Since now the failure signal may also come from outside, it is necessary to define when a workunit has terminated, thus to avoid interrupting and compensating terminated transactions. In webπ_∞ the transaction is considered terminated, and thus discarded, when its body P becomes $\mathbf{0}$. A few other aspects have to be considered for killing and termination. First, since webπ_∞ is asynchronous, messages are considered sent as soon as they become enabled. Thus they can freely float out of workunits. Moreover, they are not deleted when the workunit is interrupted. Another important aspect is transaction nesting. Two approaches exist in the literature: *nested failure* and *non-nested failure*. In the nested failure approach, when a transaction is killed all its subtransactions are killed too. In the non-nested failure approach instead, subtransactions are preserved and continue their regular execution. Suppose for instance that the payment workunit described above is part of a more complex transaction with body Q:

$$\langle\!|\langle\!|\ \texttt{PAY}....\ ;\ \texttt{manageFault}|\!\rangle_t\,|\,Q\ ;\ \texttt{manageLargerFault}|\!\rangle_{\texttt{s}}$$

With the nested failure approach in case of failure of s also t is killed. However this is not the behavior of webπ_∞, which follows the non-nested failure approach. In webπ_∞ this behavior can be obtained by adding an explicit kill of t as part of the management of the larger fault, e.g., by replacing the compensation with the process $\overline{t} \,|\, \texttt{manageLargerFault}$.

The webπ_∞ calculus is equipped with a reduction semantics formally describing the behavior of systems based on web transactions, and with a labeled transition system supporting the standard observational equivalence—weak barbed congruence [30]. Furthermore, weak asynchronous bisimilarity [1], adapted to the webπ_∞ setting, where transaction kill has to be explicitly considered, characterizes weak barbed congruence. Therefore, transformations of webπ_∞ processes can be coinductively proved correct (with respect to weak asynchronous bisimilarity, and thus, with respect to weak barbed congruence).

While referring to the paper [27] for the technical details, we present here a sample law, illustrating handlers reducibility:

$$\langle\!|P \;;\; Q|\!\rangle_x = (x'x'')(\langle\!|P \;;\; \overline{x'}\,|\!\rangle_x \,|\, \langle\!|x'.Q \;;\; \mathbf{0}|\!\rangle_{x''})$$

for each $x', x'' \notin \mathrm{fn}(P) \cup \mathrm{fn}(Q), x' \neq x'' \neq x$. In other words, it is not necessary to have a generic process Q as compensation of a workunit, but it is enough to have a simple output message $\overline{x'}$. In fact, it is enough to put the compensation in another workunit, guarded by an input on the name x'. Note that both the name x' and the name x'' of the auxiliary workunit need to be private (this is done by the restriction operator $(x'x'')$) to avoid interferences.

Dynamic Compensations. In webπ_∞, the compensation of each workunit is static, i.e., in a workunit $\langle\!|P \;;\; Q|\!\rangle_x$, P is not allowed to update Q. Assume that P is a complex activity, e.g., executing a sequence of bank payments. If a failure occurs before any bank payment, then no particular error recovery is needed (possibly just some garbage collection or error notification). Instead, if a few bank payments have been completed and an error requires to abort the transaction, then the already completed bank payments have to be annulled. In webπ_∞ this can be done for instance by keeping track of the performed bank payments, and by having the compensation Q checking which of them have been completed to annul them. Another solution, suggested by the law above, is to put the compensation in a different workunit to be replaced with an updated one each time a new payment is completed. However, both the solutions are complex and error-prone [19]. In general, one may want to adapt the compensation of a complex transaction to the evolving state of its process P. This kind of problems has been tackled by dcπ [36], by compensable processes [22], and by SOCK [18]. The three approaches differ in a few technical decisions, but they all share the idea that a compensation can be dynamically updated. We present here the general approach in the framework of π-calculus [36,22], leaving to next section the discussion of the interplay with service-oriented features.

Parallel Recovery. The simplest proposal is the one of dcπ. There, scopes (similar to webπ_∞ workunits) have the form $t[P]$ where P is the executing process and

t the scope name. Inputs in P may install compensations. For instance, assume that a message $\overline{payConf}\langle \boldsymbol{v} \rangle$ confirms that a payment has been completed, and that $\boldsymbol{v}$ contains the data of the payment. In $\mathsf{dc}\pi$ such a message can be received by an input $payConf(\boldsymbol{x})\%\overline{Annul}\langle \boldsymbol{x} \rangle.Q$ that after receiving the message $\overline{payConf}\langle \boldsymbol{v} \rangle$ installs in the nearest enclosing scope a new compensation item $\overline{Annul}\langle \boldsymbol{v} \rangle$ and continues as $Q\{^{\boldsymbol{v}}/_{\boldsymbol{x}}\}$. When a scope is killed, all the installed compensation items are executed in parallel. This form of recovery is called *parallel recovery*. Note that input and compensation update form a unique atomic primitive. This is important since it should never be the case that the state of the transaction is changed (because of the received input), and the compensation has not been changed accordingly. In our example this would cause a performed payment not to be annulled. It would be difficult to ensure this atomicity property if compensation update has to be mimicked as described above.

General Recovery Policies. As shown, $\mathsf{dc}\pi$ allows to dynamically add new compensation items in parallel. However, it may be handy to have more control on the order of execution of compensation items, and to be able to remove compensation items when they are no more useful. A more general approach has been proposed in the framework of SOCK[18], and analyzed in the framework of π-calculus [22]. We describe here the latter, where compensable processes are defined. Compensable processes define a scope construct $t[P, Q]$ similar to the workunit $\langle\!|P \ ; \ Q|\!\rangle_t$ of $\mathsf{web}\pi_\infty$. However compensable processes provide in addition a compensation update primitive $\mathsf{inst}\lfloor \lambda X.Q' \rfloor.R$ that replaces the current compensation Q in the nearest enclosing scope with the new compensation $Q'\{^{Q}/_{X}\}$. This allows for instance to add a new compensation item in parallel, by choosing $Q' = Q'' \mid X$ where X does not occur in Q'', mimicking $\mathsf{dc}\pi$ parallel recovery. However, many other options are available. For instance one may execute compensations of different activities in reverse order of completion (this policy is called *backward* recovery [16]). In compensable processes such behavior is obtained by using compensations of the form $\lambda X.(finished)(Q' \mid finished.X)$ where the actual compensation Q' signals its termination with an output on the private channel $finished$. Moreover, the compensation can be deleted by installing $\lambda X.\mathbf{0}$, or replaced with a new compensation by installing $\lambda X.NewComp$ where $NewComp$ does not contain X.

Consider the following scenario: a few bank payments are executed by sending messages to the banks in charge of them. If something goes wrong in one of the payments, all of the performed payments have to be annulled. At the end a final check is performed, and if it succeeds then annul is no more possible. This can be implemented in compensable processes as follows:

$$t[PAY_1.\,\mathsf{inst}\lfloor \lambda X.ANNUL_1.X \rfloor. \ldots . PAY_n.\,\mathsf{inst}\lfloor \lambda X.ANNUL_n.X \rfloor. \\ CHECK.\ \mathtt{if}\ check = \mathtt{ok}\ \mathtt{then}\ \mathsf{inst}\lfloor \lambda X.\mathbf{0} \rfloor\ \mathtt{else}\ \bar{t}, \mathbf{0}]$$

where $PAY_1, \ldots, PAY_n$ are activities executing the payments, $ANNUL_1, \ldots, ANNUL_n$ the corresponding annul activities and $CHECK$ performs the final verification putting the result in *check*.

Differently from $\mathsf{web}\pi_\infty$, compensable processes have been given both a nested failure semantics and a non-nested failure one, while $\mathsf{dc}\pi$ follows the nested failure approach. However both compensable processes and $\mathsf{dc}\pi$ provide a protection operator $\langle P \rangle$ that executes P in a protected way and that can be used to avoid undesired external kills. The non-nested failure approach can thus be mimicked by enclosing each transaction in a protected block.

Another difference between $\mathsf{web}\pi_\infty$ and compensable processes is that compensable processes scopes never commit. However, $\mathsf{web}\pi_\infty$ commit behavior can be easily recovered since a scope $(t)t[\mathbf{0}, \mathbf{0}]$ with a restricted name, no body and no compensation is equivalent to $\mathbf{0}$. Note that "no compensation" can be forced in compensable processes with a suitable compensation update, while the same is not possible for $\mathsf{web}\pi_\infty$.

The definition of the semantics of compensation update requires a bit of care. As said above, in fact, it should never be the case that a state change requiring a compensation update has been performed, and the corresponding compensation update has not been executed. For instance in:

$$t[PAY_1.\,\mathsf{inst}\lfloor \lambda X.ANNUL_1.X \rfloor \ldots]$$

if the transaction is killed after PAY_1 has been completed but before the compensation has been updated, no annul is performed. For this reason, compensation update has priority w.r.t. other actions. Thus a compensation update is executed as soon as it becomes enabled. This feature comes for free in $\mathsf{dc}\pi$, since the input and the compensation update are composed in a unique primitive.

The expressive powers of static recovery, parallel recovery, backward recovery and dynamic recovery have been compared in [22]. There the existence/non existence of suitable encodings (compositional [17], preserving testing equivalence [33], and not introducing divergency) has been discussed. Two main results were achieved:

- **An encoding of parallel recovery into static recovery which satisfies the conditions above and preserves also weak bisimilarity**. The existence of such an encoding proves that parallel recovery and static recovery have the same expressive power. The encoding stores the dynamically created compensation items in the running process protected by protected blocks, and exploits suitable messages to enable them only when needed.
- **A separation result proving that no encoding satisfying the properties above exists from backward recovery to static recovery neither from compensable processes to static recovery**.

The results above, together with the ones presented at the beginning of the section, prove that primitives for interruption and compensation are an important feature of languages, since they can not be encoded in an easy way, and that also the choice of the exact kind of primitives may change the expressive power of the language. Thus a careful choice is needed to decide which of these primitives have to be included in a language. Next section shows how these primitives can be applied to service-oriented systems.

3 Exploiting the Mechanisms in SOC

In this section we show how the mechanisms introduced in the previous sections to deal with failures and compensations can be exploited in service-oriented computing models and languages, where an application is composed by orchestrating different services. Service instances interact giving rise to sessions involving possibly many partners. Thus errors may be both internal to a single session, and in this case the techniques described in the previous section may be applied directly, or may involve different services. The first case has been considered for instance in the Conversation Calculus [37] (see also Chapter 2-1), where a conversation is a set of related interactions that take place in a dedicated medium—a conversation context—which may be accessed from several distributed conversation access pieces (cf. endpoints), each one held by a different party. The Conversation Calculus manages errors by using the try-catch operator discussed in the previous section. As we show in Section 4.3, this is enough to model cCSP [11].

Different approaches were chosen in various other SENSORIA calculi. We explain these approaches below.

3.1 Static Compensation Policies

We present the compensation mechanisms of COWS and of CaSPiS.

Killing Activities. COWS [26] includes primitives used to force immediate termination of concurrent threads. The syntax of COWS and an informal explanation of its semantics are presented in Chapter 2-1. Besides allowing generation of 'fresh' private names (as 'restriction' in π-calculus [29]), the delimitation operator of COWS provides a means for modeling a named scope for grouping certain activities. A named scope $[k]\,s$ can be then equipped with suitable termination activities, as well as ad hoc fault and compensation handlers, thus laying the foundation for guaranteeing transactional properties in spite of services' loose coupling. This can be conveniently done by relying on the *kill* activity $\mathbf{kill}(k)$, that causes immediate termination of all concurrent activities inside the enclosing $[k]$ (which stops the killing effect), and the *protection* operator $\{\!|s|\!\}$, that preserves intact a critical activity s also when one of its enclosing scopes is abruptly terminated.

Failure management operators can be programmed and assembled in COWS by simply exploiting these basic operators. For example, the *try-catch* block used for the bank payment activity can be written as follows:

$$PAY \mid [if, then, k]\,(\, if \bullet then?\langle \text{false}\rangle.(\,\mathbf{kill}(k) \mid \{\!|\, manageFault \,|\!\}\,) \mid if \bullet then?\langle \text{true}\rangle.s\,)$$

Suppose that the result of the payment transaction is provided by the process PAY through the invoke activity $if \bullet then!\langle x_{res}\rangle$ and by setting the variable x_{res} to communicate the success ($x_{res} = \text{true}$) or failure ($x_{res} = \text{false}$) of the transaction. The delimitation of the killer label k confines the transaction, otherwise

uncontrolled faults can jeopardize service composition. Suppose that the failure is risen by the activity $\mathit{if} \bullet \mathit{then}!\langle \text{false} \rangle$. The management of the corresponding fault can be activated while the activity $\mathit{if} \bullet \mathit{then}?\langle \text{true} \rangle.s$ is abruptly terminated by means of the activity $\mathbf{kill}(k)$. To ensure a proper execution order in the above transaction, i.e. the management of the fault should not be performed before the termination of the killing effect of $\mathbf{kill}(k)$, kill activities in COWS have higher priority than other activities.

Finally, restriction and protection operators implicitly provide embedded mechanisms for handling nested failures. The following simple example illustrates the effect of executing a kill activity within a nested protection block:

$$[k]\,(\{\!|s_1 \mid [k']\,\{\!|s_2 \mid \mathbf{kill}(k')|\!\} \mid \mathbf{kill}(k)|\!\} \mid s_3) \mid s_4$$

evolves to

$$[k, k']\,\{\!|s_2 \mid \mathbf{kill}(k')|\!\} \mid s_4$$

For simplicity, we assume that s_1 and s_3 do not contain protected activities. In essence, $\mathbf{kill}(k)$ terminates all parallel services inside delimitation $[k]$ (i.e. s_1 and s_3), except those that are protected at the same nesting level of the kill activity (i.e. $s_2 \mid \mathbf{kill}(k')$).

Closing Sessions. The Service Centered Calculus (SCC) [4] and its evolution CaSPiS [3] propose another approach. As shown in Chapter 2-1, conversations in CaSPiS are structured as binary sessions involving a client and a service instance, dynamically created during service invocation. CaSPiS features primitives for session closure. Recall that, using close, a partner can leave a session at any time; the semantics will then guarantee that the other party is informed and that nested sessions are closed as well. In terms of transactions, completing and abandoning a session may be understood respectively as commit and failure. CaSPiS compensation handling can be classified as *static*, indeed compensations are programmed once and for all by means of listeners, $k \cdot P$, at design time.

Below, we briefly illustrate the use of session-closing primitives for programming compensations by means of a simple example. Further details can be found in [3], while the (similar) approach proposed by SCC is described in [4].

Consider another version of the bank example, where the bank (process B below) offers a service `pay` that, after receiving the amount to be paid and the user's credentials, invokes an auxiliary service, `checkAmt`, in order to check the client's available funds. A client (process C below) invokes this service and requires the payment of an amount a. The example contains three listeners. In B, upon invocation, the listener of the service definition $\mathtt{pay}_k$, $k \cdot \mathsf{close}$, closes the current session and notifies the closure to the invoker, while the listener of service invocation, $\overline{\mathtt{checkAmt}}_{k'}$, also closes the enclosing session by spawning $\dagger(k)$. The listener of service invocation $\overline{\mathtt{pay}}_{k''}$, $k'' \cdot \mathit{payNotAllowed}$, is activated in case of failure in the payment process on the service side: *payNotAllowed* encodes the execution of appropriate recovery actions

$$
\begin{aligned}
B \triangleq\ & (\nu\, k)\mathtt{pay}_k.\big(k \cdot \mathsf{close} \\
& \quad \mid (?amt, ?id)\mathbf{select}\ ?go \\
& \qquad \mathbf{from}\ (\nu\, k')\overline{\mathtt{checkAmt}}_{k'}.(k' \cdot (\mathsf{close} \mid \dagger(k)) \mid \\
& \qquad\quad \langle amt, id\rangle(?rep)\langle rep\rangle^{\uparrow}) \\
& \qquad \mathbf{inif}\ go\ \ \mathbf{then}\ \cdots\ \ \mathbf{else}\ \mathsf{close}\big) \\
C \triangleq\ & (\nu\, k'', a, id)\overline{\mathtt{pay}}_{k''}.\big(k'' \cdot payNotAllowed \mid \langle a, id\rangle \cdots\big)\ .
\end{aligned}
$$

Consider the system $S \triangleq B \mid C$. The session installed between the client and the bank can be terminated unexpectedly in two cases: when the auxiliary service closes the interaction unexpectedly or when the `checkAmt` service answers negatively. After the synchronization of service definition and invocation and the first intra-session communication for sending the amount and the id, the system becomes S' below.

$$
\begin{aligned}
S' \triangleq\ & (\nu\, r, k, k'', a, id) \\
& \Big(r \rhd_{k''} \big(k \cdot \mathsf{close} \mid \mathbf{select}\ ?go \\
& \qquad \mathbf{from}\ (\nu\, k')\overline{\mathtt{checkAmt}}_{k'}.(k' \cdot (\mathsf{close} \mid \dagger(k)) \mid \\
& \qquad\quad \langle a, id\rangle(?rep)\langle rep\rangle^{\uparrow}) \\
& \qquad \mathbf{inif}\ go\ \ \mathbf{then}\ \cdots\ \ \mathbf{else}\ \mathsf{close}\big) \\
& \mid r \rhd_k \big(k'' \cdot payNotAllowed \mid \cdots\big)\Big)
\end{aligned}
$$

If the service call $\overline{\mathtt{checkAmt}}$ returns false, S' reduces to S'' and (omitting the terminated session originated by the service invocation $\overline{\mathtt{checkAmt}}_{k'}$.) we have

$$
\begin{aligned}
S'' \triangleq\ & (\nu\, r, k, k'', a, id)\Big(r \rhd_{k''} \big(k \cdot \mathsf{close} \mid \mathsf{close}\big) \mid r \rhd_k \big(k'' \cdot payNotAllowed \mid \cdots\big)\Big) \\
& \qquad \longrightarrow \\
& (\nu\, r, k, k'', a, id)\Big(\blacktriangleright \big(k \cdot \mathsf{close} \mid \dagger(k'')\big) \mid r \rhd_k \big(k'' \cdot payNotAllowed \mid \cdots\big)\Big) \\
& \qquad \longrightarrow \\
& (\nu\, r, k, k'', a, id)\Big(\blacktriangleright \big(k \cdot \mathsf{close}\big) \mid r \rhd_k \big(payNotAllowed \mid \cdots\big)\Big) \ \triangleq\ S'''\ .
\end{aligned}
$$

In S''', the client proceeds by taking appropriate recovery actions (defined in $payNotAllowed$).

In case the closure is originated by service `checkAmt`, the signal $\dagger(k)$ will be captured by the listener $k \cdot \mathsf{close}$ and the session closure protocol will proceed similarly.

3.2 Dynamic Compensation Policies

The last approach we consider is the one of the Service Oriented Computing Kernel (SOCK) [10]. As described in Chapter 2-1, SOCK is a calculus for service-oriented computing that has been inspired by the main technologies in the field,

in particular WSDL [38], the standard for defining web service interfaces, and WS-BPEL [32], the de-facto standard for web services composition. SOCK allows the definition of services exploiting the one-way and request-response patterns provided by WSDL.

From a compensation point of view, SOCK has been extended in [18] with mechanisms that integrate the WS-BPEL concepts of scope, termination and compensation with the dynamic approach to error recovery described in the previous section.

A scope in SOCK is a process container denoted by a name and able to manage faults. Faults are thrown by the primitive $\texttt{throw}(f)$ where f is the name of the fault. Inside a scope, three different kinds of handler can be defined. A fault handler f specifies the recovery code to be executed when fault f is thrown inside the scope. A termination handler, which has the name of the scope containing it, specifies how to smoothly terminate the scope when it is reached by an external fault. Finally, compensation handler q specifies how to undo the activities of the finished scope q if required during error recovery inside an outer scope. For instance

$$\{\texttt{PAY} : [f \rightarrow \texttt{manageFault}, q \rightarrow \texttt{manageExternalFault}]\}_q$$

is a scope that executes activity PAY, executes code `manageFault` in case PAY throws fault f and executes code `manageExternalFault` in case of external failure.[1]

Assume that activity PAY throws fault f, e.g. since $PAY = PAY' \mid \texttt{throw}(f)$. First, all the activities inside PAY' are terminated, including subscopes. Termination handlers of those subscopes are executed. Then the fault handler for f is looked for inside q. Since it is available then it is executed, handling fault f. If no fault handler was found, the fault would be rethrown to the enclosing scope, let us call it q', while q terminates with a failure. Error handling would continue in q', and the fault would be recursively thrown to the nearest enclosing scope until a handler is found. Both termination handler `manageExternalFault` and fault handler `manageFault` may use the primitive $\mathsf{comp}(q_1)$ to execute the compensation handler of some subscope q_1 of q to undo its activity. This is available only if q_1 has terminated with success.

Up to here, this is the error recovery policy used also by WS-BPEL. However, in WS-BPEL handlers are defined statically inside the scope. SOCK allows to update them at runtime, thus following the dynamic approach. Consider the scope q above. SOCK provides a compensation update primitive, $\mathsf{inst}([f \rightarrow \texttt{newHandler}])$, similar to the one of compensable processes, to replace the old handler `manageFault` with the new handler `newHandler`. Differently from compensable processes, now the name of the handler(s) to be updated has to be specified. Like in compensable processes, the old handler may not be discarded. In fact, one can use the placeholder cH inside the handler update primitive to recover the old handler. For instance, $\mathsf{inst}([f \rightarrow \texttt{newHandler}; cH])$, adds the new handler `newHandler` before the old handler `manageFault` (here ; is sequential

[1] The actual syntax is slightly more complex, cfr. [18].

composition), producing the new handler `newHandler`;`manageFault`. Both fault and termination handlers can be updated in this way. A compensation handler instead is just the last defined termination handler when the scope terminates. This is justified by the fact that intuitively the behavior of a service should be the same if the fault occurs just before or just after its termination. Anyway, the ability of dynamically updating the handlers allows to redefine termination handler just before termination if a different behavior is desired for compensation handler. Notice that the update primitive is executed with priority w.r.t. other instructions, so to ensure that the state of the error handlers always matches the state of the computation.

Until now we have managed errors involving just one service instance. As already said, services in SOCK may interact using two modalities: one-way $\overline{o}@z(\boldsymbol{y})$ and request-response $\overline{o}_r@z(\boldsymbol{y},\boldsymbol{x})$. With the one-way, a service invokes another service o located at z and does not care about the result. This is a loosely coupled interaction pattern, thus does not poses particular problems from an error handling point of view. With the request-response instead a client invokes a service o_r located at z and waits for an answer. This interaction pattern may be spoiled by errors both on client side and on service side. Assume for instance that the invoked service fails because of some fault f, either from the service code or from the service environment. In WS-BPEL such a service will not send back any answer, and the client would wait undefinitely. Vice versa, if the client fails (because of some fault in a parallel process), the answer from the service may be lost. Consider our example of bank payment. Now the payment may be required by a process

$$\{\overline{pay}_r@bank(\boldsymbol{y},\boldsymbol{x}) \mid Q : [f \rightarrow \texttt{manageFault}, g \rightarrow \texttt{manageRemoteFault}]\}_q$$

Suppose that after the pay_r service has been invoked Q throws fault f. Thus the client will not know whether the operation has been successful (and money has been taken from the account) or not. Clearly the two scenarios require different compensation policies on the client side.

To answer these problems SOCK proposes an approach based on automatic error notification and allows to exploit those notifications during error recovery. In particular, if the server pay_r above fails because of fault g, a faulty answer is automatically sent to the waiting client, where it is considered as a local fault. This has a double aim: on the one side the client will not be stuck waiting for a response that will not arrive, on the other side the client may specify a suitable handler for g allowing to recover locally from the remote error. For instance the handler `manageRemoteFault` may notify the user or look for other payment methods.

Furthermore, if the client fails while waiting for the answer of the request-response operation (the fault comes from Q), the answer from pay_r is waited for before error recovery is started. Also, a non-faulty answer may update the error-handler on the client side requiring for instance to undo the remote activity. This can be obtained by modifying the request-response above into

$$\overline{pay}_r@bank(\boldsymbol{y},\boldsymbol{x},[f \rightarrow \texttt{annulPay}; cH])$$

Now, upon successful answer from pay_r, the fault handler for f is updated specifying that in case of such a fault (that, we assume, makes the whole transaction fail) the pay operation should be undone. The compensation update is performed only if the remote operation has been successful, and even if there has been a local fault in the meanwhile.

The proposed approach has been validated (see [18] for details) in different ways. First, by formally proving that the formalism satisfies some expected high-level properties such as "each request-response receives an answer, either a normal one or a faulty one" or "it is never the case that a fault is managed by an handler that has not been updated". Second, SOCK error handling primitives have been used to program error handling for the automotive case study (see Chapter 0-3). Third, SOCK primitives have been introduced in the language JOLIE [31,21], a full-fledged language to program service-oriented applications inspired by SOCK, and used to program real applications.

4 Models of Compensations

In the previous sections we have presented different mechanisms for defining long-running transactions and compensations. However those mechanisms are not all at the same abstraction level. They range from some low level mechanisms, such as the ones of COWS [26] providing basic operators such as kill and protection, to more complex mechanisms such as the ones of SOCK [19] and WS-BPEL [32]. In the literature there are also abstract descriptions of the desired behavior that compensated activities should have, such as the one provided by SAGAs calculi [8]. Also, some approaches aiming at proving the correctness of compensations are emerging [12,35].

In this section we present three comparisons between approaches at different levels of abstraction. These can be exploited with different aims. On one side they provide a way to assess the expressive power of languages, showing that they are able to implement some abstract behavior. On the other side they help the programmer of the application, who can specify the desired recovery strategy at the high level of abstraction and exploit an automatic translation to derive an implementation which is correct by construction. Finally, techniques and strategies developed at one abstraction level can be exported to other levels.

4.1 Encoding BPEL Scopes in COWS

The first encoding that we present shows how COWS basic mechanisms are powerful enough to implement WS-BPEL [32] scope construct.

Consider the following version of the WS-BPEL scope activity:

$$[s : \mathbf{catch}(\phi_1)\{s_1\} : \ldots : \mathbf{catch}(\phi_n)\{s_n\} : s_c]_i$$

This construct permits explicitly grouping activities together[2]. The declaration of a scope activity contains a unique scope identifier i, a service s representing the normal behavior, an optional list of fault handlers $s_1, \ldots, s_n$, and a compensation handler s_c. The fault generator activity $\mathbf{throw}(\phi)$ can be used by a service to rise a fault signal ϕ. This signal will trigger execution of activity s', if a construct of the form $\mathbf{catch}(\phi)\{s'\}$ exists within the same scope. The compensate activity $\mathbf{compensate}(i)$ can be used to invoke a compensation handler of an inner scope named i that has already completed with success. Compensation can only be invoked from within a fault or a compensation handler. Here, we fix two syntactic constraints: handlers do not contain scope activities and, as in WS-BPEL (see [32]), for each $\mathbf{compensate}(i)$ occurring in a service there exists at least an inner scope i. Notably, an activity $[s : \mathbf{catch}(\phi_1)\{s_1\} : \ldots : \mathbf{catch}(\phi_n)\{s_n\} : s_c]_i$ acts as a binder for $\phi_1, \ldots, \phi_n$; in this way, a scope can only catch and handle faults coming from its enclosed activities.

Now we show that this version of fault and compensation handling can be easily encoded in COWS. The most interesting cases of the encoding are the following:

$$\begin{aligned}
\langle\!\langle [s : \mathbf{catch}(\phi_1)\{s_1\} : \ldots : \mathbf{catch}(\phi_n)\{s_n\} : s_c]_i \rangle\!\rangle_k \;=\; & \\
[\phi_1, \ldots, \phi_n]\ (& \langle\!\langle \mathbf{catch}(\phi_1)\{s_1\} \rangle\!\rangle_k \mid \ \ldots \ \mid \langle\!\langle \mathbf{catch}(\phi_n)\{s_n\} \rangle\!\rangle_k \\
& \mid [k_i]\, \langle\!\langle s \rangle\!\rangle_{k_i} \,;\, (\, x_{done} \bullet o_{done}!\langle\rangle \mid [k']\, \{\!|\, make \bullet undo?\langle i \rangle.\langle\!\langle s_c \rangle\!\rangle_{k'} \,|\!\}\,)\,) \\
\langle\!\langle \mathbf{catch}(\phi)\{s\} \rangle\!\rangle_k \;=\; & raise \bullet throw?\langle \phi \rangle.[k']\, \langle\!\langle s \rangle\!\rangle_{k'} \\
\langle\!\langle \mathbf{compensate}(i) \rangle\!\rangle_k \;=\; & make \bullet undo!\langle i \rangle \mid x_{done} \bullet o_{done}!\langle\rangle \\
\langle\!\langle \mathbf{throw}(\phi) \rangle\!\rangle_k \;=\; & \{\!|\, raise \bullet throw!\langle \phi \rangle \,|\!\} \mid \mathbf{kill}(k)
\end{aligned}$$

The two distinguished endpoints $raise \bullet throw$ and $make \bullet undo$ are used for exchanging fault and compensation signals, respectively. Each scope identifier i or fault signal ϕ can be used to activate scope compensation or fault handling, respectively.

The encoding $\langle\!\langle \cdot \rangle\!\rangle_k$ is parametrized by the label k that identifies the closest enclosing scope, if any. The parameter is used when encoding a fault generator, to launch a kill activity that forces termination of all the remaining activities of the enclosing scope, and when encoding a scope, to delimit the field of action of inner kill activities. The compensation handler s_c of scope i is installed when the normal behavior s successfully completes, but it is activated only when signal $make \bullet undo!\langle i \rangle$ occurs. Similarly, if during normal execution a fault ϕ occurs, a signal $raise \bullet throw!\langle \phi \rangle$ triggers execution of the corresponding fault handler (if any). Installed compensation handlers are protected from killing by means of $\{\!|\_|\!\}$. Notably, the compensate activity can immediately terminate (thus enabling possible sequential compositions by signaling its completion through the endpoint $x_{done} \bullet o_{done}$); this, of course, does not mean that the corresponding handler is terminated.

[2] This version only permits to compensate specified inner scopes and does not provide an automatic compensation mechanism à la SAGAs. This latter mechanism, however, can be implemented in COWS by relying on 'queues' (we refer the interested reader to [25] for further details).

4.2 SAGAs in SOCK

The next encoding that we present is from the SAGAs calculi [8] to SOCK [18]. SAGAs calculi are based on the composition of basic activities. An activity A may either terminate with success, or with failure. An activity A may have an associated compensation activity B whose aim is to compensate the activity A in case of failure of the transaction. Activities can be composed using sequential and parallel composition, and grouped into subtransactions. For instance a SAGA executing two payment requests and annulling them in case of failure can be written as:

$$\{\!|PAY_1 \% ANNUL_1; PAY_2 \% ANNUL_2|\!\}$$

Different recovery policies are defined, specifying how to compose compensations and when to execute them. The general idea is that sequential activities are compensated in backward order while parallel activities are compensated in parallel. SAGAs calculi provide different policies, depending on whether parallel activities are stopped in case of fault, and on whether compensations are executed in a centralized or distributed way. We concentrate here on "coordinated interruption", where parallel branches are stopped when a flow aborts, and compensations are handled in a centralized way.

This policy has been implemented using SOCK mechanisms in [23]. SAGA activities have been implemented by SOCK services, invoked using the request-response interaction pattern. For instance, the activity PAY_1 above is implemented by a service PAY_1 located at location l_{PAY_1} and invoked by a request-response $PAY_1@l_{PAY_1}$ (parameters are not considered since they are not important from a failure point of view).

If the activity PAY_1 succeeds, then it sends back an answer (values sent in the answer are not important too). If it fails, then it generates a specific fault c. Through the automatic fault notification mechanism of SOCK, this fault is notified to the caller, where it is raised signaling that the current SAGA is aborting and has to be compensated.

Abortion of a SAGA is managed by using SOCK fault and compensation handlers. Each activity invocation is inside a dedicated scope. If the activity successfully finishes, then its compensation is installed as compensation handler for the scope. At the SAGA level, a fault handler for c is installed, invoking the compensations of the different inner activities in the required order, which is extracted from the structure of the term. For instance, the SAGA of the example above is modeled by a scope of the form

$$\{\mathsf{inst}([c \rightarrow \mathsf{comp}(pay_2); \mathsf{comp}(pay_1)]; \ldots); \{\ldots\}_{pay_1}; \{\ldots\}_{pay_2}\}_u$$

Since compensation handlers are available only after the corresponding activity successfully ends, then only those activities are compensated, as required.

Assume now that the compensating activity $ANNUL_1$ is executed as part of the recovery. As specified by the compensation handler for PAY_1, this is executed with a different handler w.r.t. normal activities. In particular, in case of failure, the fault c is caught, and a fault f (for fail) is raised instead. Fault f is never

caught and makes the whole SAGA fail, according to the SAGA idea that failure is a catastrophic event.

The translation outlined above has been described in detail and proved correct in [23]. We outline here also the correctness result. SAGA behavior is defined in terms of a big-step LTS semantics, with rules of the form $\Gamma \vdash S \xrightarrow{\alpha} \square$ where S is a SAGA, Γ a function that specifies for each activity in S whether it succeeds or it fails, α an observation of the computation, specifying the composition of successful activities executed (the composition contains sequential and parallel operators) and $\square$ may be either success, abort (i.e., success of the compensation) or failure. For instance the SAGA above has a big-step transition of the form

$$PAY_1 \mapsto \boxdot, PAY_2 \mapsto \boxtimes, ANNUL_1 \mapsto \boxdot \vdash \{[PAY_1 \% ANNUL_1; PAY_2 \% ANNUL_2]\} \xrightarrow{PAY_1; ANNUL_1} \boxtimes$$

specifying that the SAGA aborts if activities PAY_1 and $ANNUL_1$ succeed and PAY_2 aborts.

SOCK instead has a small step semantics, including different observations such as service invocations and replies, uncaught faults and others. Thus the correctness is expressed in terms of an abstraction of the possible SOCK computation containing only the events corresponding to successful answers from request responses.

The correctness result can be stated as follows (see [23] for a more formal statement).

Theorem 1. *Let S be a SAGA. $\Gamma \vdash S \xrightarrow{\alpha} \square$ iff for each observation o which is a linearization[3] of α one of the following happens:*

- *$\square$ is success and there is a computation starting from the translation of S that does not contain uncaught faults whose abstracted observation is o;*
- *$\square$ is abort and there is a computation starting from the translation of S whose abstracted observation is o and which terminates with an uncaught fault c which is the only uncaught fault;*
- *$\square$ is failure and there is a computation starting from the translation of S whose abstracted observation is o and which terminates with an uncaught fault f which is the only uncaught fault.*

For the SAGA above the theorem guarantees that the translation of the SAGA has a computation whose abstracted observation is $PAY_1; ANNUL_1$ and which has a unique uncaught fault, c.

4.3 Analysis of Compensations in the Conversation Calculus

In this section we show how the Conversation Calculus (CC) [37] (see also Chapter 2-1) may be used to model and reason about structured compensating transactions, following the techniques detailed in [12]. To reason about compensations

[3] A linearization is obtained by taking an actual interleaving for parallel activities.

in an abstract way, independently from a particular language implementation, we introduce a general model of stateful compensating transactions. We then take the core language for structured compensations introduced in [11], the compensating CSP calculus (cCSP), but reinterpret its semantics in our generic compensating model framework and prove the fundamental property expected in any compensation model, namely atomicity of transactions (Theorem 2). Lastly, we present an embedding of cCSP transactions in the Conversation Calculus, which is proven correct since it induces a stateful model of compensating transactions (Theorem 3). In the remainder of this section we describe the main ideas that are at the basis of our development.

In our model, the most elementary program is a *primitive action*, similar to a SAGA activity. A primitive action enjoys the following atomicity property: it either executes successfully to completion, or it aborts. In case of abortion, a primitive action is required not to perform any relevant observable behavior, except signaling abortion by throwing an exception. A transaction, which may involve several primitive actions, must also enjoy the atomicity property: it either executes successfully to completion, or it aborts leaving the system in a state equivalent to the one right before the transaction started executing. An aborted transaction must not have any visible effect on the state of the system, so any actions that were executed up to the point of the abortion must be in some way reverted. Compensations provide a means to achieve this reversibility: if we attach to every action a compensation that reverts the effect of the action, then by executing all compensations of the previously executed actions (in the reverse order) we end up in a state that should be in some sense equivalent to the state right before the transaction started executing.

We define an abstract notion of compensating model, leaving open the intended notion of "similarity" ($\bowtie$) between states, up to to which reversibility is to be measured. The definition is also independent of the concrete underlying operational model.

Definition 1 (Compensation Model). *A compensation model is a pair* $(\mathcal{S}, \mathcal{D})$ *where* $\mathcal{S}$ *gives its static structure and* $\mathcal{D}$ *gives its dynamic structure. The static structure* $\mathcal{S} = (S, \mid, \#, \bowtie)$ *is defined such that:*

- S *is a set of (abstract) states*
- $\mid$ *is a partial composition operation on states*
- $\#$ *is an apartness relation on states*
- $\bowtie$ *is an equivalence relation on* S

The dynamic structure $\mathcal{D} = (\Sigma, \xrightarrow{a})$ *is defined such that:*

- Σ *is a set of primitive actions*
- $\xrightarrow{a}$ *is a labeled (by elements of* Σ*) transition system between states.*

On the one hand, the compensation model describes the static structure which consists in a set of states S, a composition operation over states (defined only when such states are independent/apart $\#$) and an equivalence relation that introduces flexibility at the level of measuring the cancellation effect of compensations: since compensations, in general, may not be able to leave the system in

exactly the same state, we must consider a flexible notion of equivalence that allows us to capture that the compensations produce an "equivalent enough" state. On the other hand, the dynamic structure of the compensation model is described by a labeled transition system between the states.

Using this abstract notion of compensation model we proceed by equipping the cCSP with a semantics defined in terms of interpretations of a compensation model. The semantics captures the effects and final status of cCSP programs—the states in which the system is left after executing the program, and a signal that indicates that the program successfully completed or aborted. cCSP programs are split in two categories: basic programs and compensable programs. The simplest compensable program is a pair $P \div Q$ where P and Q are atomic actions and action Q is the compensation of action P. Thus action Q intended to undo the effect of the P action, leading to a $\bowtie$-equivalent state to the state right before P was executed. Complex structured compensable programs may then be defined by composition under various control operators: sequential composition $T; R$, parallel composition $T \mid R$, and others. An arbitrary compensable program T may then be encapsulated as a basic program, by means of the operator $\langle T \rangle$.

The compensation model already allows us to state conditions on basic actions precise enough to derive general properties, namely the following atomicity result, that may then be reused in each particular application of the model.

Theorem 2 (Atomicity). *Let R be a $\bowtie$-consistent compensable program. Then $\langle R \rangle \sqsubseteq R^+ \oplus \mathit{throw}$.*

Theorem 2 guarantees that the behavior of transactions implemented over $\bowtie$-consistent compensable programs approximate atomicity: a transaction either aborts (*throw*) doing "nothing", or ($\oplus$) terminates successfully after executing all of its forward actions (R^+) (P is the forward action in $P \div Q$). The $\bowtie$-consistent condition ensures that for each compensation pair $P \div Q$ in the program, action Q reverts the effect of P up to $\bowtie$.

We now present our provably correct embedding of the cCSP language for structured compensating transactions in the Conversation Calculus. We consider that primitive actions are implemented by CC processes conforming to the following behavior: after some interactions with the environment it either sends (only once) the message $ok^{\downarrow}!$ in the current conversation context without any further action, or aborts, by throwing an exception. If the outcome is abortion, the system should be left in the "same" state (up to $\bowtie$) as it was before the primitive action started executing.

We show a selection of our encoding in Fig. 1. We use $[\,P\,] \triangleq (\boldsymbol{\nu} n)(n \blacktriangleleft [\,P\,])$ as an abbreviation to represent an anonymous (restricted) context (useful to frame local computations). We denote by $[\![P]\!]_{ok}$ the encoding of a basic program P (namely structured compensating transactions) into a conversation calculus process. The *ok* index represents the message label that signals the successful completion of the basic program, while abortion is signaled by throwing an exception. The encoding of compensable transaction T is denoted by $[\![T]\!]_{ok,ab,cm,cb}$. The encoding of T will either issue a single message $ok^{\downarrow}$ to signal successful completion (and the implicit installation of compensation handlers)

$$
\begin{aligned}
[\![\langle T \rangle]\!]_{ok} &\triangleq [\ [\![T]\!]_{ok,ab,cm,cb} \mid ab?.\texttt{throw } \mathbf{0} \mid ok?.ok^{\uparrow}!\ ] \\
[\![P \div Q]\!]_{ok,ab,cm,cb} &\triangleq [\ \texttt{try } [\![P]\!]_{ok} \texttt{ catch } ab^{\uparrow}! \mid \\
&\qquad ok?.ok^{\uparrow}!.(cm^{\uparrow}?.[\![Q]\!]_{cb} \mid cb?.cb^{\uparrow}!)\ ] \\
[\![T_1; T_2]\!]_{ok,ab,cm,cb} &\triangleq [\ [\![T_1]\!]_{ok_1,ab_1,cm_1,cb} \mid ab_1?.ab^{\uparrow}! \mid \\
&\qquad ok_1?.[\![T_2]\!]_{ok,ab,cm,cm_1} \mid ab?.cm_1!.cb?.ab^{\uparrow}! \mid \\
&\qquad ok?.ok^{\uparrow}!.cm^{\uparrow}?.cm!.cb?.cb^{\uparrow}!\ ] \\
[\![T_1 \mid T_2]\!]_{ok,ab,cm,cb} &\triangleq [\ [\![T_1]\!]_{ok_1,ab,cm_1,cb_1} \mid [\![T_2]\!]_{ok_2,ab,cm_2,cb_2} \mid \\
&\qquad ok_1?.ok_2?.ok^{\uparrow}!.cm^{\uparrow}?.(cm_1! \mid cm_2! \mid cb_1?.cb_2?.cb^{\uparrow}!) \mid \\
&\qquad ab?.(ok_1?.cm_1!.cb_1?.ab^{\uparrow}! \mid ok_2?.cm_2!.cb_1?.ab^{\uparrow}! \mid ab?.ab^{\uparrow}!)\ ]
\end{aligned}
$$

Fig. 1. Encoding of structured compensating transactions in the CC (selected cases)

or (in exclusive alternative) a single message $ab^{\downarrow}$ to signal abortion. After successful completion, reception of a single message $cm^{\downarrow}$ ("compensate me") by the residual will trigger the compensation process. When compensation terminates, a single message $cb^{\downarrow}$ ("compensate back") will be issued, to trigger compensation of previous successfully terminated activities.

We prove that our encoding is correct, by showing that it induces a compensation model in the sense of Definition 1 and with respect to the cCSP semantics (see [12]).

Theorem 3 (Correctness). *Let* $\mathcal{S} = (S, \mid, \#, \bowtie)$ *and* $\mathcal{D} = (\Sigma, \xrightarrow{a})$ *define a* CC *compensating model* $\mathcal{M} = (\mathcal{S}, \mathcal{D})$*. If* $\langle T \rangle$ *is a* $\bowtie$*-consistent* CC *program over* Σ*, then* $[\![\langle T \rangle]\!]_{ok}$ *is a* CC *atomic activity, that either behaves as* T^{+}*, or aborts without any observable behavior modulo* $\bowtie$*.*

Theorem 3 states that the mapping $[\![-]\!]_{ok}$ yields a sound embedding of arbitrary ($\bowtie$-consistent) structured compensating transactions in any CC compensating model. By showing that our encoding is an instance of the cCSP semantics, we directly recover the property stated in Theorem 2 to any CC compensation model.

Our framework naturally supports distributed transactions since primitive actions may be realized by calls to remote services. For example, let us consider a cCSP specification of a compensable transaction, which captures a credit request operation between a client and a bank, where the financial ranking of the client is updated according to the credit request operation, e.g., so as to indicate his financial status is less reliable.

$$\langle \mathit{StartCreditRequest} \div \mathit{AbandonData}; \mathit{UpdateRate} \div \mathit{RestoreRate}; \mathit{ClientAccept} \div \mathit{skip} \rangle$$

Whenever a credit request operation starts, some data is created and the client's financial rate is updated. Then either the client accepts or otherwise the transaction is aborted. In the latter case, the client rate is restored and the data of the operation is cleared. Primitive actions such as *UpdateRate* and *RestoreRate* may be implemented via calls to services that realize the expected tasks, for instance:

$UpdateRate \triangleq \texttt{new}\ Bank \cdot \texttt{UpdateRate} \Leftarrow \texttt{lowerClientRate!.(ok?.ok}^{\uparrow}\texttt{! + ko?.throw)}$

$RestoreRate \triangleq \texttt{new}\ Bank \cdot \texttt{RestoreRate} \Leftarrow \texttt{raiseClientRate!.ok?.ok}^{\uparrow}\texttt{!}$

Notice that these CC programs either send a single *ok* message or abort by throwing an exception, and hence fit in our previous description of primitive actions. We may then directly obtain a CC implementation of the cCSP transaction specified above, via such implementations of the primitive actions and via the developed embedding of the cCSP compensation operators in CC.

5 Conclusion

In this chapter we have summarized the main results of the SENSORIA project concerning fault and compensation handling in message-based calculi. They concern different aspects. On one side, we have studied different primitives for modeling long-running transactions and compensations, adapting also them to the particular needs of service-oriented computing. In particular, the idea of dynamic handlers is new, and has been studied in details. On the other side, we have analyzed the expressive power of the different primitives, proving some interesting separation results. Finally, we have exploited these mechanisms by inserting them into calculi and languages for service-oriented computing, such as CaSPiS [3], COWS [26], the Conversation Calculus [37], SOCK [10] and Jolie [31]. Similar results for event-based calculi are presented in Chapter 3-4. For instance, a mapping of SAGAs into the Signal Calculus [14] has been presented in [6].

While we have today a huge toolbox of primitives able to deal with the challenges of service-oriented computing, the understanding of the relationships among them is still far. A few works [22,23,12] have appeared analyzing encodings and separation results, but many pieces are missing, and the whole picture is still quite obscure. Keep also in mind that the problem is made hard since the expressive power depends not only on the chosen primitives for fault and compensation handling, but also on the underlying language. Another important stream for future work is the proof of correctness of compensation strategies. For long-running transactions one cannot require, as can be done for ACID transactions instead, that in case of failure the system goes back to the starting state, since recovery is not perfect. A few approaches are emerging here too. The previous section presented a framework for reasoning about the correct recovery, measured up to some particular behavioral equivalence parametrically defined in the framework. An alternative approach is to examine observations: a relation between performed activities and executed compensations is required, based again on some user-defined pattern [35].

Acknowledgments. The work reported herein is the result of a collaborative effort of many researchers, not just of the authors. Special thanks to Lucia Acciai for the contribution on CaSPiS in Section 3, and to Rosario Pugliese and Francesco Tiezzi for the contribution on COWS in the same section.

António Ravara was partially supported by the Security and Quantum Information Group, Instituto de Telecomunicações, Portugal.

References

1. Amadio, R.M., Castellani, I., Sangiorgi, D.: On bisimulations for the asynchronous pi-calculus. Theoretical Computer Science 195(2), 291–324 (1998)
2. Bocchi, L., Laneve, C., Zavattaro, G.: A calculus for long-running transactions. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 124–138. Springer, Heidelberg (2003)
3. Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and pipelines for structured service programming. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 19–38. Springer, Heidelberg (2008)
4. Boreale, M., et al.: SCC: a Service Centered Calculus. In: Bravetti, M., Núñez, M., Tennenholtz, M. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 38–57. Springer, Heidelberg (2006)
5. Bravetti, M., Zavattaro, G.: On the expressive power of process interruption and compensation. Mathematical Structures in Computer Science 19(3) (2009)
6. Bruni, R., Ferrari, G.L., Melgratti, H.C., Montanari, U., Strollo, D., Tuosto, E.: From theory to practice in transactional composition of web services. In: Bravetti, M., Kloul, L., Tennenholtz, M. (eds.) EPEW/WS-FM 2005. LNCS, vol. 3670, pp. 272–286. Springer, Heidelberg (2005)
7. Bruni, R., Melgratti, H., Montanari, U.: Nested commits for mobile calculi: Extending join. In: Proc. of IFIP TCS 2004, pp. 563–576. Kluwer, Dordrecht (2004)
8. 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)
9. Busi, N., Gabbrielli, M., Zavattaro, G.: Replication vs. recursive definitions in channel based calculi. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 133–144. Springer, Heidelberg (2003)
10. 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)
11. Butler, M.J., Hoare, C.A.R., Ferreira, C.: A trace semantics for long-running transactions. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. The First 25 Years. LNCS, vol. 3525, pp. 133–150. Springer, Heidelberg (2005)
12. Caires, L., Ferreira, C., Vieira, H.T.: A process calculus analysis of compensations. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 87–103. Springer, Heidelberg (2009)
13. Carbone, M., Honda, K., Yoshida, N.: Structured interactional exceptions for session types. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 402–417. Springer, Heidelberg (2008)
14. Ferrari, G.L., Guanciale, R., Strollo, D.: JSCL: A middleware for service coordination. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 46–60. Springer, Heidelberg (2006)
15. Fournet, C., Gonthier, G.: The join calculus: A language for distributed mobile programming. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 268–332. Springer, Heidelberg (2002)
16. Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., Salem, K.: Coordinating multi-transaction activities. Technical Report Report No. UMIACS-TR-90-24, Univ. of Maryland Institute for Advanced Computer Studies (1990)

17. Gorla, D.: Towards a unified approach to encodability and separation results for process calculi. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 492–507. Springer, Heidelberg (2008)
18. 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)
19. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: Dynamic error handling in service oriented applications. Fundamenta Informaticae 95(1), 73–102 (2009)
20. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
21. Jolie website, http://www.jolie-lang.org/
22. 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)
23. Lanese, I., Zavattaro, G.: Programming sagas in SOCK. In: Proc. of SEFM 2009, pp. 189–198. IEEE Computer Society Press, Los Alamitos (2009)
24. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
25. Lapadula, A.: A Formal Account of Web Services Orchestration. PhD thesis, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze (2008), http://rap.dsi.unifi.it/cows
26. Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007)
27. Mazzara, M., Lanese, I.: Towards a unifying theory for web services composition. In: Bravetti, M., Núñez, M., Tennenholtz, M. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 257–272. Springer, Heidelberg (2006)
28. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
29. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, part I/II. Information and Computation 100, 1–77 (1992)
30. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 685–695. Springer, Heidelberg (1992)
31. 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)
32. Oasis. Web Services Business Process Execution Language Version 2.0 (2007), http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
33. Rensink, A., Vogler, W.: Fair testing. Information and Computation 205(2), 125–198 (2007)
34. Sangiorgi, D., Walker, D.: Pi-Calculus: A Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
35. Vaz, C., Ferreira, C.: Towards compensation correctness in interactive systems. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 161–177. Springer, Heidelberg (2010)
36. 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)
37. Vieira, H.T., Caires, L., Seco, J.C.: The conversation calculus: A model of service-oriented computation. In: Gairing, M. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 269–283. Springer, Heidelberg (2008)
38. World Wide Web Consortium. Web Services Description Language (WSDL) 1.1 (2001), http://www.w3.org/TR/wsdl