Recovery within Long Running Transactions

CHRISTIAN COLOMBO and GORDON J. PACE
University of Malta

As computer systems continue to grow in complexity, the possibilities of failure increase. At the same time, the increase in computer system pervasiveness in day-to-day activities brought along increased expectations on their reliability. This has led to the need for effective and automatic error recovery techniques to resolve failures. Transactions enable the handling of failure propagation over concurrent systems due to dependencies, restoring the system to the point before the failure occurred. However, in various settings, especially when interacting with the real world, reversal is not possible. The notion of compensations has been long advocated as a way of addressing this issue, through the specification of activities which can be executed to undo partial transactions. Still, there is no accepted standard theory; the literature offers a plethora of distinct formalisms and approaches.

In this survey, we review the compensations from a theoretical point of view by: (i) giving a historic account of the evolution of compensating transactions; (ii) delineating and describing a number of design options involved; (iii) presenting a number of formalisms found in the literature, exposing similarities and differences; (iv) comparing formal notions of compensation correctness; (v) giving insights regarding the application of compensations in practice; and (vi) discussing current and future research trends in the area.

Categories and Subject Descriptors: H.2.4 [**Systems**]: Transaction processing

General Terms: Reliability, Theory

Additional Key Words and Phrases: Compensations

1. INTRODUCTION

Over the past decades we have witnessed a dramatic increase in the pervasiveness of computer systems in day-to-day activities, at the same time came an increase in their size and complexity. This, together with the tightly knit interaction with other activities made it virtually impossible to have systems which never fail. However, their unavoidable role in sensitive human activities also put higher expectations of reliability. To handle such scenarios, fault tolerance techniques started playing a crucial role in software development. The philosophy behind fault tolerance techniques is that, given that, even after rigorous testing, failures in a complex system are considered a possibility or in some settings inevitable, mechanisms are introduced to handle them.

This research has been partially funded by the Malta National Research and Innovation (R&I) Programme 2008 project number 052.

Authors' addresses: Department of Computer Science, Faculty of ICT, University of Malta, Malta; email: {christian.colombo, gordon.pace}@um.edu.mt.

A commonly used mechanism for fault tolerance is *error recovery,* consisting of attempting to fix the system state upon the discovery of an error to enable normal execution to proceed unhindered. Error recovery techniques can be categorised into *backward recovery* and *forward recovery* approaches. Backward recovery backtracks to an earlier and correct state of the system before proceeding, while forward recovery attempts to correct the error before proceeding.

Although error recovery in small and simple systems tends to be manageable, an increase in complexity brings about a disproportionate increase of complexity when handling recovery. For instance, with a system supporting concurrency, dependencies between concurrent processes introduce issues with the ordering of recovery actions which need not be considered in linear systems. Forward recovery techniques are limited in such scenarios, since they they are typically not aware of the history which led up to the failure which they are trying to fix.

Various approaches to backward recovery have been developed, but the main issue with this approach is that some processes with which a system interacts may not support backward error recovery. Typical examples would be real-life processes such as bank account transfers, shipping, etc. Such processes cannot be simply undone and forgotten. In such cases, instead of undoing some actions, one might actually need to execute further "counter"-actions, better known as *compensations.* For example in the case of bank account transfers, one might have to add a processing fee over and above the return of funds to the original account, while in the case of shipping one might need to ship some items back.

Compensations have become even more relevant with the advent of the Internet which enabled widespread interaction and collaboration particularly through the use of web services. This phenomenon facilitated interactions across entities which might not even have been aware of each other before the interaction. In such a scenario, with usually long-running interactions, backward recovery is not an option. To this end, compensations are heavily used to support such interactions with the current de facto standard being the Business Process Execution Language (BPEL).

The proliferation in the use of compensations motivated extensive research of the area, particularly by suggesting different formal models of compensation and defining formal semantics for BPEL. While from a practical perspective, BPEL is a *de facto* standard supporting compensations, from a theoretical perspective, various approaches have been proposed — varying in the way compensations are modelled but, more fundamentally, also in their exploration of compensation design space. The comparison of these approaches is crucial to identify what the core theoretical underpinnings of compensation are.

In this survey we start by giving a historic account of the rationale behind the conception of compensating transactions (Section 2). We explore the design aspects of compensations (Section 3), comparing and contrasting different options and solutions as proposed and adopted in the literature (Section 4). Furthermore, we review compensation correctness concepts (Section 5) and ways of verifying compensating transactions (Section 6) which are crucial in setting a sound foundation for the specification of compensating transactions. Although the main focus of the survey is the theory of compensations, Section 7 deals with various practical issues, useful for anyone wanting to apply the theory into practice. We conclude this survey by discussing open philosophical issues and comment about current and future trends in the area of compensations.

2. THE RATIONALE BEHIND COMPENSATIONS

The higher the reliability expected of a system, the more precautions have to be taken, possibly introducing additional complexity and sources of unreliability. The situation is further aggravated by the many possible sources of faults in a computer system: hardware faults, operating system faults, and faults within the computer system itself. Potentially, faults cause the system to reach an erroneous state which may then lead to failure[1], i.e. making the effects of having reached an erroneous state visible from outside of the system.

Given that failure in large systems has to be accepted as part of reality, the solution is typically not to try to engineer systems which never fail but rather to create safety nets within and around the system so that the system can tolerate faults in such a way that they do not lead to failure. Furthermore, when a system depends on other external systems, the possibility of failure has to be handled internally since one has no control over external failure. A commonly used fault-tolerance technique [Randell et al. 1978] is error recovery — the task of dealing with an error before it has time to cause a failure.

2.1 Forward and Backward Error Recovery

The various strategies of error recovery [Randell et al. 1978; Verhofstad 1978] are typically classified as *backward* or *forward recovery*. Backward recovery refers to strategies which first revert the current (erroneous) state to a previous (correct) state before attempting to continue execution, while forward recovery attempts to correct the current (erroneous) state and then continues normal execution. The main difference is that backward recovery uses the execution context to fix the problem, which is not necessary in forward recovery. On one hand, one can see backward error recovery as a particular form of forward recovery, in which additional data structures are used to store the history of execution. On the other hand, forward error recovery can also be considered as an optimisation of backward error recovery [Randell et al. 1978], in that the recovery path is deduced without keeping a log of the past actions.

Forward error recovery is particularly useful when the failure — the symptom of the error — is sufficient to determine which solution to apply. A simple mechanism to encode forward recovery in most modern programming languages is the use of exception and error event handlers. The code used to recover acts as a *reparation*, intended to fix the problem encountered. Note that, unless additional mechanisms are used to keep a log of what actions the system performed, exception handlers are only aware of failure happening in a particular block of code.

To make backward recovery possible, one has to keep track of the system's previous states or transitions, which involves recording past data or actions which have been carried out. One approach to backward recovery is that of backing up the system's data (also called *checkpointing*) so that if an erroneous state is reached,

[1] The terminology is used in line with that introduced in [Melliar-Smith and Randell 1977].

the data can be restored to a past but sane copy. A dual approach is that of keeping an audit trail, recording sequences of actions which had been carried out, so that the system state can be restored by reversing the actions previously performed. While the first approach incurs a high overhead in data storage, the latter requires knowledge on how to reverse a system's actions. One way of undoing a system's recent actions is through *rollbacks,* which are perfect reversals of previous actions, i.e. leaving no evidence of either the original action or its reversal. However, perfect action reversal is not always possible, particularly when a system interacts with other external systems (e.g. a bank system or a shipping agent) which do not support perfect reversal of actions. In this case, rather than a perfect reversal of an action, one would need to execute a "counter"-action, better known as a *compensation* which semantically undoes the original action as much as possible. So, for example, to compensate for a bank transfer one might need to reverse the involved sum and charge an extra fee. Observe that, even if no fee is incurred, the two transfers back and forth distinguish the resulting account from one never involved in a transfer. Similarly, to compensate for a wrong shipment, one might need to book a reverse shipment followed by shipment to the correct destination. Note the fundamental difference between rollback and compensation in that the latter does not remove evidence of the erroneous action but simply executes a correction.

The main gain with a compensation-based approach over checkpointing and rollback comes when one handles long running transactions due to the increased effort involved in maintaining error recovery information and transaction isolation. Transactions in a loosely-coupled concurrent setting are usually long running. Moreover, makes the task of maintaining overall correctness even more challenging due to the high degree of independence allowed to the concurrent elements. Historically, this has been one of the main drives behind the study of compensation-based error handling.

2.2 Failure Recovery in a Concurrent World

Concurrency is highly desirable in many computer systems, enabling sharing of resources, and distributing computation. However, concurrency introduces a further level of complexity to the task of error recovery since concurrent systems usually share resources or at least use each others' results. This causes what is termed as the "domino effect" [Randell 1975] where a failure of a process potentially causes the failure of other concurrent tasks. For example, consider two concurrent money transfers: one transfers 100 euros into an account A while the other transfers 90 euros from account A to another account B. Without the first transfer, account A might not hold sufficient funds to supply 90 euros for the second transfer. If this is the case, the failure of the first transfer would result in the failure of the second.

To effectively deal with this problem, it has long been suggested [Davies 1973] that processes which are dependent should be clearly bounded by a *sphere of control* — a logical container which explicitly demarcates depedency and is thus a boundary for propagation of errors. As a result, a generalised approach can be applied to recovery — the container implicitly defines which processes need to be recovered. Along the years, related (and almost synonymous) notions have emerged including *conversations* [Randell 1975], *transactions* [Eswaran et al. 1976], and *recovery lines* [Randell et al. 1978]. The following is a brief overview of these approaches:

— *Sphere of control:* A sphere of control incorporates a number of participant processes which are allowed to share resources amongst themselves. A process may request a resource in three ways: (i) *reference only* — the process making the request does not care if the resource is changed by other processes during the use of the resource (e.g. a process monitoring other processes is aware that the monitored values are continually changing and is not affected by this fact); (ii) *dependent* — the requester wants to know of any changes of the resource but is willing to give up the resource at any time (e.g. a background process accessing employees' information but which is willing to pause if some data is being modified); and (iii) *committed* — the requester is not willing to give up the resource (e.g. a shipping order which has been committed by a process cannot be "uncommitted"). In the case of dependent resources, backward recovery is possible due the requester's readiness to give up the allotted resource. However, in the case of the committed mode of resource sharing, execution cannot be undone as by definition the resource is committed, thus making backward recovery impossible, hence requiring forward recovery to correct the state of the system. Taking the example of the shipping order, if the order needs to be changed or cancelled, it is impossible to reverse the physical ship executing the order. A possible option would be to set up another transaction which (for example) ships back part of the order.

— *Conversations:* The idea of conversations is very similar to a sphere of control in that they both use backward recovery followed by forward recovery. However, whereas the latter supports customised forward recovery based on the cause of the failure, in conversations, forward recovery is preplanned in terms of a *recovery block*. Thus, the recovery block provides an alternative execution algorithm to the one which failed without taking into consideration what caused the failure. The aim of conversations is to avoid the domino effect and provide a "firewall" around interacting processes so that a failure in one of the processes does not have undesirable effects on the rest of the system (but only on other processes in the conversation). Upon completion of the recovery block execution, the system state is checked through the application of an *acceptance test*. If the test succeeds, then the state is considered committed and any backward recovery information is discarded. Hence from this point onwards, backward recovery of the committed execution is no longer possible. Note that this might be a problem in particular cases such as the case of the shipping order above.

— *Transaction:* A transaction is a means of isolating an action such that it appears to processes outside the transaction that an uninterruptable (atomic) action has taken place. The main motivation behind transactions is that consistency rules cannot be maintained on a per-operation basis. For example, if a bank system has the following consistency constraint: "unless a deposit or withdrawal takes place, the total sum of money in the bank cannot change", then in a money transfer between accounts, the constraint will be violated after the update of the source account (before the update of the destination account). Thus certain operations have to be grouped into a transaction and appear to other processes as a single action which either succeeds or fails. These principles together with that of *durability* (see below) form the bases of transactions and are referred to as the ACID principles (due to which the transactions we are describing are something referred to as ACID transactions): *atomicity* — ensuring that either the transaction fully succeeds or else it leaves no effect on the system state; *consistency* — ensuring that the system state progresses from one correct state to another; *isolation* — ensuring that intermediate results of a transaction are not visible to other transactions, giving the impression that each transaction works in isolation; and *durability* — ensuring that the outcome of a transaction is persisted and never undone. For many years these have proven to be an adequate way of handling database operations. This arrangement frequently relies on a resource-locking policy whereby transactions operate under the illusion of complete isolation from the rest of the world. Thanks to this strict approach, recovering from an error during a transaction simply requires a save point for all processes (participating in the transaction) exactly before the start of the transaction. In case of failure, the involved processes can be reverted to such a save point. Once the transaction commits, then the save points are discarded, i.e. reversing the effects of the transaction is no longer possible beyond the commit operation. This is also in line with the principle of durability mentioned above. However, once more we are faced with a problem when a situation similar to the shipping order example arises — there is no way of reversing a committed transaction.

— *Recovery line:* Although processes participating in a transaction would all have a recovery point at the start of the transaction, other recovery points belonging to individual processes (based on information flow techniques[2] rather than simply the start of a transaction) might also be used. The motivation is that when possible, recovery need not backtrack to the start of a whole execution sequence but rather to the most recent recovery point. A recovery line refers to the most recent set of consistent[3] recovery points across all the processes participating in an operation. The problem with information flow techniques is that due to a commitment by one of the processes, a recovery line might cease to exist. Furthermore, if two processes with no common recovery line interact, then there is no way the two processes can be reverted back to a common past state (using the individual recovery points of both might not produce a globally consistent state). Therefore this is known as an *interaction commitment*. At other times commitment is deliberate — *explicit commitment* — as non-commitment would result in huge overheads to maintain large numbers of recovery points. There is also a third kind of commitment which is *accidental commitment*. As the name suggests, this occurs when due to some unexpected event, recovery data is damaged and rendered unusable. Again, this brings up the issue of recovery in case of the shipping order example.

It is interesting to note that in all approaches that we have listed, a problem arises after commitment: how can committed resources, data, and interactions be modified and corrected? Even without considering errors and recovery, corrections might

[2] Information flow techniques try to use knowledge of interaction points among processes to identify suitable recovery point during execution. For example, a good recovery point would be before a process asks for a resource from another process so that in case the latter fails, the former can recover to exactly the point before the interaction.

[3] Recovery points of different processes must contribute towards a common global state and hence they have to be consistent in the sense that the recovered global state has to respect all applicable consistency constraints.

still need to be carried out due to, for example, a user modifying an order placed earlier. Therefore, this problem is not strictly speaking caused by the need of error recovery but rather by a combination of challenges due to the need of correctly handling concurrent access to data, and the need to be able to make consistent corrections to data. The problem becomes even more intricate with processes which take a long time to complete and for which it may thus not be desirable to keep recovery information for such long periods of time. Even worse, it may not even be reasonable to prevent the process (during its long execution) from interacting with the outside world. The first case — that of recovery data being discarded due to the length of the transaction — corresponds to *explicit commitment*, while the second case — that of allowing interactions with the "outside world" — corresponds to *interaction commitment*. Both commitments, as explained above would render backward recovery impossible. This is why compensations are necessary — providing a means of reversing operations even beyond their commitment. In the following subsections we expand further on compensations and give an example to illustrate a number of advanced features of compensations.

2.3 Compensations

Compensations have been proposed almost four decades ago [Davies 1973] as a form of forward recovery [Randell et al. 1978] which attempts to correct the state of a system given some knowledge of the previous actions of the system. For example, consider a bookshop which is processing an order — as long as the bookshop's computer system does not interact with the outside world, say, the shipping agent, then backward recovery would be possible because all processes involved are under the control of the bookshop. However, as soon as the bookshop places the shipping order, an interaction commitment would have taken place as the bookshop does not have access to backward recover the shipping order at the shipping agent's site (and even if the shipping agent was part of the bookshop system, the order might have already started being carried out, i.e. an interaction has taken place with another process outside the control of the system — the physical process). If a client decides to cancel an order, then a special forward recovery, termed a *compensation*, has to be carried out to check whether the shipment is still in time to be cancelled. If the shipment is successfully cancelled, the client is possibly charged a fee and notified of the cancellation. On the other hand, if it is too late to cancel the shipment, an apologetic message is sent to the client explaining the situation. Note that although at a high level of abstraction cancelling the order might be considered as a backward recovery (cancelling the shipping order), in actual terms the order has not been undone but rather a "counter"-transaction took place whether or not cancellation succeeded. Expanding the same concept, [Gray 1981] presents compensating transactions, later as sagas [Garcia-Molina and Salem 1987], as an extra layer on top of ACID transactions. The argument for this addition is two-fold: (i) long-running transactions render ACID transactions impractical as locking resources for long periods of time is infeasible in a highly concurrent system; and (ii) ACID transactions do not support nesting of transactions (as committed ACID transactions cannot be undone) so using compensations as counter-transactions, transactions can be nested and composed into a saga — a form of higher-level transaction.

This historic account of compensations brings us to compensable transactions[4] as understood in contemporary literature. To further illustrate the concepts presented, in the following subsections we give a larger example with compensations and explain in more detail the concepts surrounding compensations.

2.4 Compensations by Example

To explain the concept of compensations and demonstrate how this is applied to realistic scenarios, we present an extended example of the online bookshop scenario.

Consider the process the bookshop undertakes upon receiving an order: The bookshop first checks whether the requested books are in stock. If this succeeds, the bookshop concurrently sends the books to be packed and charges the client's bank account. If both operations are successful, a courier is booked to deliver the order. In the eventuality of a failed activity, any completed activities are compensated by executing the associated compensating activities in order to remove the effects of the transaction. For example a bank charge is compensated by a refund, whereas packing is compensated by unpacking. Note that the compensation need not be the exact reverse of the normal activity: apart from incrementing the stock as a compensation to a stock decrement, an email is sent to the client as a notification of the transaction failure. It is usually important to decide in which order the compensations should be executed. In this case, if for any reason, the unpacking was unsuccessful then it does not make sense to increment the stock level. Generally, the order of executing compensations is the reverse order of their normal execution. For example, if the courier booking fails, then both the client charge and the packing need compensating (in parallel) followed by the compensation of the stock decrement. Of particular interest is the case when the client bank charge fails because the next action depends on whether or not the packing activity running in parallel has completed or not. In case of completion, first the order is unpacked and then the stock is increased; otherwise, only the latter is carried out.

Figure 1 represents the bookshop scenario with the upper half of the boxes representing forward behaviour and those below representing the associated compensation. The arrows with a filled head represent successful execution flow control, while the others represent a fault which triggers compensation execution. Once compensation is triggered, the control flow continues in reverse order of the forward behaviour. For example, failure when processing the client payment will trigger an email to increase stock, but also the unpacking of the order, if that was completed before the processing of the payment failed. If a compensation fails, a human operator is notified (not shown in the diagram).
Note that the related backward actions are not intended to fix the related forward actions if it fails halfway through, but rather to undo it if it had previously been successfully completed.

2.4.1 *Compensation Life Cycle.* In practical terms, applying compensations involves the following steps: (i) the specification of the system is given, relating compensation actions to their counterparts (e.g. as in Figure 1); (ii) during system execution, upon succesful completion of each action, the related compensation is

[4] In the rest of the paper we use the term *transactions* to refer to compensable transactions.

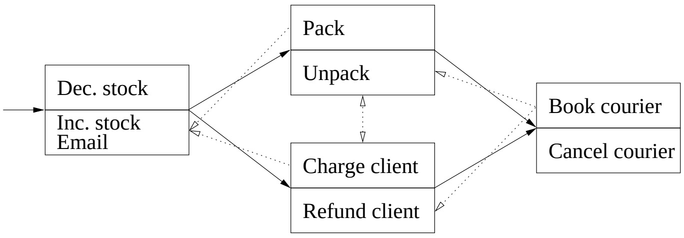

Fig. 1. A representation of the online bookshop example.

stored for potential invocation in case of failure later on — more technically it is said to be *installed* (e.g. upon successfully reducing the stock level, increasing the stock and emailing the client is installed as a compensation); (iii) subsequently, if the whole transaction succeeds, then all installed compensations are discarded; (iv) on the other hand, if a part of the transaction fails, the compensations installed earlier are executed (e.g. if the booking of the courier fails, then the compensations for the bank charge, the packing and the stock decrease, are executed); and (v) finally, if compensation fails, then further action might be necessary such as notifying a human operator to handle the situation.

2.5 Summary

Concurrency complicates the process of error recovery because interaction among processes causes failure domino-effects. By structuring process interactions localised backward error recovery is possible except for cases where interaction occurs with non-reversible processes such as real-life processes. This motivated compensations — providing a means of executing activities which semantically reverse actions which led to a failure — semantic backward recovery. Throughout the years since the conception of the idea of compensations, different philosophies have emerged which have been concretised in various models and formalisms. In the following section, we explore these differences, explaining and comparing the possible design options as suggested in the literature.

3. DESIGN OPTIONS IN COMPENSATIONS

Compensation-based approaches come in various shapes and flavours, with different approaches adopting different design options. In this section, we identify the major issues in compensation-based approaches, and the related design options. The aspects are organised according to the compensation life cycle stage (see Section 2.4.1) under which they fall, with each subsection corresponding to a different life cycle stage. The overview presented in this section extends the observations made in the following works: [Bruni et al. 2005; Chessell et al. 2002; Butler and Ferreira 2004; Li et al. 2007a; Li et al. 2007b; Arkin et al. 2007; Butler et al. 2004; Butler and Ripon 2005; Bocchi et al. 2003; Guidi et al. 2006; Guidi et al. 2008; Lanese and Zavattaro 2009; Laneve and Zavattaro 2005; Mazzara and Govoni 2005; Vaz et al. 2009; Lapadula et al. 2007b; 2008a; Bruni et al. 2004; Lanotte et al. 2006; 2008][5].

[5] We adopt the following terminology: a *transaction* is a long running transaction which can be composed *processes* and *activities* (used interchangeably with *actions*). A process can itself be a

3.1 Specification of Compensations

Clearly, in a compensation-based setting, a system must not only have its normal behaviour specified, but also the compensations of its constituent parts — requiring a notation for the description and composition of compensations. We start by identifying and discussing design options related to: (i) compensation operators — what operators does the notation support for specifying compensations; (ii) composition of transactions — whether a transaction can itself be made of other transactions; (iii) action compensations versus transaction compensations — whether a compensation is itself an action or a transaction; (iv) compensating for failure of compensations — whether compensations can have compensations; and (v) concurrent compensation scopes — how simultaneously active scopes can be dealt with.

— *Compensation Operators:* We have identified four main compensation operators used in the literature: (i) the scope operator, which is responsible for defining the boundaries up to which a compensation remains valid; (ii) the compensation installation operator, which is responsible for associating compensations to a scope — ready to be activated in case of a failure; (iii) the compensation discard operator which discards the currently installed compensations; and (iv) the compensations activation operator which triggers the execution of the currently installed compensations. The main design option with regards to compensation operators is whether or not to provide explicit operators (as opposed to implicit default behaviour) for these indispensable compensation operations. The only exception is the scope operator which cannot be implicit. However, compensation installation can be implicitly associated with the start of a transaction scope [Laneve and Zavattaro 2005; Bruni et al. 2004; Lanotte et al. 2008], the discard operator can be implicitly associated to the termination of a transaction scope [Butler et al. 2004; Butler and Ripon 2005; Laneve and Zavattaro 2005; Bruni et al. 2004] and compensation activation can take place automatically upon the occurrence of a fault [Bruni et al. 2005; Li et al. 2007b; Butler et al. 2004; Butler and Ripon 2005; Bocchi et al. 2003; Lanese and Zavattaro 2009; Laneve and Zavattaro 2005; Vaz et al. 2009; Lapadula et al. 2008a; Bruni et al. 2004; Lanotte et al. 2008].
The main options associated with this issue is which of the compensation operators are used explicitly in the syntax or handled implicitly by the semantics.

— *Composing Transactions:* An important decision is whether or not to allow transactions to be made up of other long running transactions. For example, a payment action may be seen as a single transaction, with a compensation related to it as a whole, and which is invoked if failure occurs later and the payment has to be revoked. However, such a transaction may involve multiple account transfers happening behind the scenes, each of which may come with its own compensation just in case failure occurs before the payment transaction terminates. Note that none of the formalisms under review allow ACID transactions to be explicitly part of long running transactions. However, a number of flow composition languages [Bruni et al. 2005; Li et al. 2007b; Butler et al. 2004; Butler and Ripon 2005] assume atomic actions as the building blocks (of long running transactions) which

transaction and can also be composed of other processes and activities.

can be used to represent ACID transactions. All the works being considered support nested long running transactions with the exception that [Laneve and Zavattaro 2005] flattens out nested transactions and treats them as parallel transactions. Various design decisions arise as to the interaction between a transaction and its parent. For example, is the accumulated compensation of a sub-transaction passed on to its parent when the sub-transaction completes? What happens to the parent transaction if one of its children fail? These design issues will be discussed in other sections further on.
The options presented are thus: (i) whether or not transactions can themselves be composed of other transactions; and (ii) whether or not the semantics (not only the syntax) supports transaction composition.

— *Action Compensations versus Transaction Compensations:* A compensation may either be an action or a whole compensable transaction itself. The argument in favour of having only transactions as compensations [Butler and Ferreira 2004; Arkin et al. 2007; Bocchi et al. 2003; Lanese and Zavattaro 2009; Laneve and Zavattaro 2005; Vaz et al. 2009; Lapadula et al. 2008a; Bruni et al. 2004] is that this provides a uniform approach such that the compensation may itself fail and have its own compensations. On the other hand, a number of formalisms [Bruni et al. 2005; Butler et al. 2004; Butler and Ripon 2005; Li et al. 2007b] choose to allow basic actions to have actions as compensations while transactions have transactions as compensations, thus providing a clear correspondence between actions and their corresponding compensating actions. The problem with this might be that in reality a compensation for an action is not itself an action. For example, the compensation for a bank refund might involve the actual money transfer together with an email being sent as an explanation to the client. A third option is to allow only basic activities (or their composition) to be compensations [Lanotte et al. 2008]. This approach does not support programmable compensations as transactions cannot have transactions as compensations. Thus the compensation of a transaction is always the composition of installed compensations and the compensation cannot itself be compensable.
There are thus, three main design options: (i) all compensations are transactions themselves; (ii) actions have corresponding compensating actions while a transaction may have a compensating transaction (programmable compensation); or (iii) compensations are processes which do not support compensations.

— *Compensating for Failure of Compensations:* With nested compensations, we considered the possibility of having the forward part of a transaction being split into parts with smaller compensation units. The dual issue is whether the backwards actions — the compensations — can themselves include compensations in case *they* need to be undone. Since compensation is usually considered as backward behaviour, it is somewhat strange to have a backward behaviour of the backward behaviour. Indeed, in real life examples, one rarely, if at all, encounters examples where nested compensations are required. However, from a theoretical perspective enabling compensations to have compensations permit cleaner and more compositional syntax and semantics.
There are various design options which have been considered in the literature: (i) both the syntax and semantics prohibit compensations to have compensations

[Butler et al. 2004; Butler and Ripon 2005; Lanotte et al. 2008]; (ii) the syntax allows for nested compensations while the semantics ignores them [Li et al. 2007b]; and (iii) both the syntax and semantics support nested compensations [Bruni et al. 2005; Butler and Ferreira 2004; Arkin et al. 2007; Bocchi et al. 2003; Lanese and Zavattaro 2009; Laneve and Zavattaro 2005; Vaz et al. 2009; Lapadula et al. 2008a; Bruni et al. 2004].

— *Concurrent Compensation Scopes:* The scope of a compensation is the processing component during which the compensation is active, usually referred to as a transaction. Within a scope, compensations are accumulated together so that if something goes wrong, anything completed up to that point can be undone. There are two main options with respect to concurrent scoping: either having only a single active compensation scope at any one time to which compensations can be added or else having multiple active scopes (this feature is only provided in [Butler and Ferreira 2004]). The advantage of the latter is that it allows the programmer to choose to which compensation scope to install the compensation and eventually which scope to activate. This is useful, for example, when one wants to use compensations both for recovery and also for programming purposes. Consider a scenario where one is making provisional bookings at various establishments; for each booking two compensations are installed: the confirmation of the booking on one scope and the cancellation on the other. If all the bookings succeed then one would simply need to execute the scope with all the confirmations while in case of failure one runs all the cancellations.

3.2 Installation of Compensations

The installation of a compensation refers to the instant at which an activity is marked as a compensation ready to be used in case a fault is encountered later on. The first arising issue is what points during the execution of a transaction can be used as potential compensation installation points. Furthermore, installation can either occur implicitly — hiding the details of what is actually happening — or explicitly — having the user to explicitely state what and where to install compensations. Another important question when it comes to installing compensations is the policy to be used in the ordering of compensations. The problem is more complex when there are parallel processes — should the respective compensations also be activated in parallel or in the order in which execution has actually taken place at runtime? This is a delicate question to which one finds a variety of answers in the literature.

— *Compensation Installation Points:* Usually, compensation installation occurs upon the completion of an activity [Bruni et al. 2005; Li et al. 2007b; Butler et al. 2004; Butler and Ripon 2005] since a compensation can be thought of as the reverse of that activity. However, sometimes one would like to compensate for a whole transaction rather than a single activity and for this reason it is also reasonable to install a compensation upon the completion of a transaction [Arkin et al. 2007; Bocchi et al. 2003; Lapadula et al. 2008a]. Still there are various other options of allowing compensation installation at particular points in a transaction (e.g. after interactions [Vaz et al. 2009]) or possibly allowing the installation of compensations anywhere during the execution of a transaction [Butler and Ferreira 2004; Lanese and Zavattaro 2009]. Finally, although it is somewhat contrary to intuition, there are examples in the literature where compensation is installed at the start of a transaction [Laneve and Zavattaro 2005; Bruni et al. 2004; Lanotte et al. 2008]. This is only possible when the execution of the compensation does not depend on the execution history — a condition which is satisfied in particular weak forms of compensation.

In summary, the design options considered are: (i) install compensations upon completion of an activity; (ii) install compensations upon completion of a transaction; (iii) install compensations after interactions; (iv) allow compensation installation at any point; and (v) compensation is installed at the start of a transaction.

— *Composing Compensations:* When installing a compensation, usually, this is composed in some way to the already accumulated compensations for that particular scope, defining the order in which compensations are executed (if later activated). There are theoretically many other possible orderings than the ones presented here, but these are the approaches found in the literature. Furthermore, there seems to be no ordering which is generally more preferred than another.

The four main options of ordering compensations are: (i) install compensations always in parallel [Bocchi et al. 2003; Vaz et al. 2009]; (ii) install compensations always in sequence — achieving reverse order of execution [Butler and Ferreira 2004; Arkin et al. 2007]; (iii) install compensations such that they match the forward behaviour, i.e. parallel compensation for parallel activities and sequential compensation for sequential activities [Bruni et al. 2005; Li et al. 2007b; Butler et al. 2004; Butler and Ripon 2005; Lanotte et al. 2008]; or (iv) user-given [Lanese and Zavattaro 2009].

3.3 Replacing and Discarding Compensations

Once the end of a compensation scope is reached, a decision is required regarding the accumulated compensation: this may either be maintained, discarded or replaced. Discarding compensation without replacement is somewhat counterintuitive as the possibility of compensating for completed transactions is one of the distinguishing factors between ACID transactions and compensable transactions. Having said this, an outer scope can still semantically compensate for its sub-scopes by taking into consideration whether they have succeeded or not (without actually using the sub-scopes' compensations). Replacing compensations, i.e. programmable compensations, is more a question of flexibility since the fine-grained compensations may need to be replaced by a more simple action once the transaction has been completed. To introduce even more flexibility, another possible design option is to provide the possibility of discarding, replacing and modifying the stored compensation at any point of transaction execution [Butler and Ferreira 2004; Guidi et al. 2008; Lanese and Zavattaro 2009; Lanese et al. 2010]. However, this approach is not mainstream in compensation literature, possibly because such degree of freedom is generally unnecessary, and requires substantial syntactic overhead in the specification.

The options associated with this design issue are various: (i) compensations of nested scopes are executed as part of the compensation of the outer scope [Lanotte et al. 2008]; (ii) propagate the compensation to the next outer scope whereupon the relevant installation policy (see previous subsection) is applied [Arkin et al. 2007; Vaz et al. 2009]; (iii) discard the compensation of that scope [Laneve and Zavattaro 2005; Bruni et al. 2004]; (iv) discard the compensation of that scope with the possibility of installing a coarser-grained compensation in the next outer scope [Butler et al. 2004; Butler and Ripon 2005; Bocchi et al. 2003]; (v) propagate the compensation to the next outer scope only if no coarser-grained replacement is available [Bruni et al. 2005; Li et al. 2007b]; or (vi) preserve the compensation of the scope beyond its completion so that in the future the scope can still be compensated without any connection to the compensation of the outer scope [Butler and Ferreira 2004; Lanese and Zavattaro 2009; Lapadula et al. 2008a].

3.4 Activation of Compensations

Once a compensation is installed, it can be activated and executed to compensate for already completed activities. There are various reasons why a compensation may be activated: (i) either the programmer chooses (explicitly) to execute the compensation at some point; (ii) an exception occurs within the transaction, causing it to compensate; or (iii) because an external fault (e.g. the failure of a parallel transaction) forces the transaction to terminate (forced termination).

These possibilities give rise to various design issues: (i) whether or not compensation is triggered automatically upon failure; (ii) if a transaction fails, how does failure propagate from child to parent? (iii) how does failure propagate from parent to child? (iv) handling interruption — how forced termination is handled; and (v) termination coordination — how concurrent compensations are executed.

— *Implicit versus Explicit Compensation Activation:* This design aspect is particularly crucial because it touches on how compensations are ideally viewed: whether or not compensations should be strictly related to failures. In general, compensations have been proposed as a means of handling failure but when compensations can be activated at any point of execution (irrespective of failure), then compensation becomes more of a programming pattern rather than a failure handling mechanism. This issue is further discussed in Section 8.2.
There are three main compensation activation modes: (i) compensation is activated automatically upon a failure [Bruni et al. 2005; Li et al. 2007b; Bocchi et al. 2003; Bruni et al. 2004; Lanotte et al. 2008]; (ii) compensation is activated explicitly (by the programmer) [Butler and Ferreira 2004; Butler et al. 2004; Butler and Ripon 2005; Lanese and Zavattaro 2009; Laneve and Zavattaro 2005; Vaz et al. 2009; Lapadula et al. 2008a]; or (iii) compensation is activated automatically unless an explicit activation is specified [Arkin et al. 2007].

— *Upward Abortion Propagation:* When a transaction fails during normal behaviour[6], it usually affects other transactions such as the parent-transaction and its child-transactions. In this point, we focus on the former (which we refer to as upward abortion propagation) while we discuss the latter (downward abortion propagation) in the following point. Another important distinction is that we are

[6] We distinguish between failure during forward behaviour and failure during backward behaviour (i.e. during compensation). In this section, we focus on the former while we leave the discussion of the latter for Section 3.6.

discussing failure propagation amongst transactions not among concurrent activities within a transaction. The fact that an activity failure causes the whole transaction to fail is inherent in the definition of a transaction which either succeeds fully or fails completely.
In general, in the literature (except [Li et al. 2007b; Arkin et al. 2007]), failure of a transaction whose compensation succeeds is not propagated upwards to the transaction's parent. This option favours the notion that a successful compensation is not an exception but rather part of the normal logic. Therefore, a transaction need not be aware if a sub-transaction has successfully compensated.

— *Downward Abortion Propagation:* When a transaction encounters an exception, an important decision needs to be taken as to how any (concurrently) executing sub-transactions should be handled. The main issue here is whether a failed branch of the transaction should cause the rest of the concurrent branches to fail. Therefore this can be considered as a question of whether to have a weak or a strong parallel composition. In the case of the latter, another question arises: is failure propagated in a top-down fashion where the transaction signals all its branches to terminate, or does the (failing) branch itself cause its siblings to terminate?
In other words, the options are: (i) leave child-transactions to continue unaffected [Bocchi et al. 2003; Laneve and Zavattaro 2005; Butler and Ferreira 2004; Lanotte et al. 2008]; (ii) propagate the exception in a centralised top-down fashion [Arkin et al. 2007; Vaz et al. 2009; Bruni et al. 2004]; or (iii) propagate exception to other sibling-processes and transactions running in parallel [Bruni et al. 2005; Li et al. 2007b; Butler et al. 2004; Butler and Ripon 2005; Lanese and Zavattaro 2009; Lapadula et al. 2008a].

— *Handling Interruption:* Exception propagation among transactions assumes a form of external interruption, also referred to as forced termination. Naïvely stopping a process at any point of execution is dangerous as it might leave the system in an invalid state in the middle of an operation. The preferred approach would depend on how sensitive the system is to forced termination.
Various approaches have been proposed in the literature: (i) by default a transaction does not yield to forced termination but the programmer can indicate yielding points at which the transaction yields in case of a forced termination [Butler et al. 2004; Butler and Ripon 2005]; (ii) by default a transaction yields at any point to forced termination but a protection mechanism can be used to protect particular parts of a transaction from interruption [Butler and Ferreira 2004; Arkin et al. 2007; Lanese and Zavattaro 2009; Vaz et al. 2009; Lapadula et al. 2008a]; or (iii) allow an interrupted transaction to execute a termination handler to ensure graceful termination [Arkin et al. 2007; Lanese and Zavattaro 2009].

— *Termination Coordination:* As discussed in previous points, upon the occurrence of a failure in a strong parallel composition, sibling processes are terminated (interrupted) and compensated. This raises another design issue: does each parallel process immediately start compensating, or does it wait for the other siblings so that all compensations start at the same time? The former is referred to as *distributed interruption* while the latter is known as *coordinated interruption* [Bruni et al. 2005]. The choice depends on whether the compensations of the interrupted processes are dependent on each other and whether it is preferable to have a cen-

tralised mechanism which coordinates the compensations. [Bruni et al. 2005; Li et al. 2007b] adopt distributed interruption, while [Arkin et al. 2007; Butler et al. 2004; Butler and Ripon 2005; Vaz et al. 2009] adopt coordinated interruption.
In summary the options are: (i) distributed interruption where compensations start independently; and (ii) coordinated interruption where compensations start together.

3.5 Compensation Execution

In the theoretical literature reviewed, compensations are executed as normal activities but in practical terms various issues arise as regards to state management during compensation execution. For example, what system state should be used to start compensation execution? Should the compensation be executed on the current state of the system, the state at the moment of compensation installation, or on the state at the moment of completion of the corresponding activity. BPEL, as an example, records a *scope snapshot* which records the state of a scope exactly after completion of the scope. This is then loaded before executing the corresponding compensation. Another issue is to choose the scope of the compensation execution, i.e. which part of the state of the system the compensation can access. Referring again to BPEL as an example, the compensation execution does not only have access to the snapshot but also to the state of the scope (and the enclosing scopes) which invoked the compensation. After the completion (or failure) of compensations new issues arise, which we will discuss in the next subsection.

3.6 Post-Compensation Execution

Upon completion of a compensation execution, the point from which execution is to resume is an important choice. This obviously takes into account whether compensation was successful or not. The possible design options identified in the literature involved in this phase are: (i) how the transaction continues after a successful compensation; and (ii) how the transaction continues after a failed compensation.

— *After a Successful Compensation:* Recall that when a transaction compensates successfully, the parent transaction is generally not even aware that a compensation has taken place (see Section 3.4). Therefore, this is one possibility of what happens after successful compensation, i.e. execution continues as normal forward behaviour in the parent scope. On the other hand, if compensation is propagated to the next outer scope, then after a successful compensation, execution continues by executing the compensation of the next outer scope. Apart from these options, there is however another possibility: *alternative forwarding* — after a successful completion of a compensation, alternative forwarding provides another behaviour which is semantically equivalent to the one which has been compensated (e.g. if courier booking fails, another courier which provides an equivalent service may be tried out). This feature is explicitly provided in [Bruni et al. 2005; Li et al. 2007b].
In summary, the options of continuation following a successful compensation are: (i) execution continues normally in parent transaction; (ii) execution continues by triggering the parent's compensation; (iii) execution continues by executing alternative behaviour — alternative forwarding.

— *After a Failed Compensation:* Recall (from Section 3.1) that in many cases, a compensation may itself be a transaction. Therefore, a failed compensation usually has its own ways of coping with failure. However, if after attempting to handle itself, the compensation still fails, then the transaction is considered just as a failed transaction. In more practical terms, a failure during compensation becomes a "normal" failure within the scope which initiated the compensation. In this sense, there is no distinction between a failure during normal behaviour and a failure during compensation. A more complex approach arises when compensations are not necessarily transactions. In this case, a failed compensation is considered as a special exception which leaves the system in an inconsistent state and therefore must be handled by exception handling. If an exception handler is not provided, then a failed compensation causes a failure which propagates upwards to the transaction's parents, causing the whole transaction to end in failure unless the exception is caught at some level.

In summary, the options after a failed compensation are: (i) either to treat the failure as a normal failure within the scope which triggered the compensation, possibly triggering higher-level compensations; or (ii) the failure is treated as a special failure which can only be handled by exception handling.

3.7 Conclusion

The design issues in compensations are considerable and each issue is surrounded by numerous options. In the next section, we consider various models of compensations from the literature, with each model representing a particular combination of design options from the above.

4. MODELS OF COMPENSATIONS

There are various attempts of formalising the notion of compensation, mainly motivated by the need of clear semantics and the possibility of applying verification techniques. Among these formalisms there are notable differences in the way they handle compensations, thus providing a rich background for comparing and contrasting the various aspects of compensations.

In the literature, one finds three main approaches to modelling compensations — (i) *flow composition languages* which primarily focus on the control structure and flow of the process under analysis; (ii) approaches, usually based on process calculi, which focus on the communication and interaction between entities making up the process; and (iii) automata-based approaches. We start by briefly describing the interesting characteristics of a number of control-flow approaches:

— Sagas [Bruni et al. 2005] is a flow composition language based on the idea of sagas introduced in [Garcia-Molina and Salem 1987] where, in essence, a sequential saga is a sequence of activities which either succeed in totality or else the completed activities should be compensated (giving the illusion that none of them were successful). Sagas automatically installs compensation actions corresponding to the forward behaviour — in sequence if the forward behaviour is in sequence and in parallel if the forward behaviour is in parallel. Starting from this basic structure of a sequence of activities, Bruni et al. go on to provide a hierarchy of extensions including parallel composition of activities, nesting of transactions, programmable compensations and various other highly expressive features including alternative forwarding and speculative choice. Sagas provides both exception and compensation handling with the former taking over when the latter fails. A limitation of Sagas is that it does not provide a mechanism for handling forced termination, i.e. if a process fails, other processes running in parallel are interrupted without any precautions.

— Compensating CSP (cCSP) [Butler et al. 2004; Butler and Ripon 2005] is an extension to CSP [Hoare 1985] with the aim of providing support for long-lived transactions. In cCSP, all basic activities succeed and failures are explicitly programmed using a special *THROW* activity which always fails. Compensations in cCSP can be programmed at two levels: a compensating action can be associated with an action while a compensating transaction can be associated with a transaction. Similar to Sagas and t-calculus[Li et al. 2007a; Li et al. 2007b], cCSP installs compensations automatically in such a way as to reflect the forward behaviour. A special feature of cCSP is that once a transactions completes, installed compensating actions are automatically discarded. Therefore, by associating a compensation to a transaction one would effectively be replacing the automatically accumulated compensations with a coarser-grained compensation for the whole transaction (programmable compensation). Another particular feature of cCSP is that in case of a process failing within a parallel composition, it allows the programmer to decide at which point the other processes (in the composition) can be interrupted by the failure. It is also interesting how fault handling and compensations are intermixed in cCSP: when a failure occurs, the compensation is only triggered if the exception handler fails (or no exception handler is available).

— StAC [Butler and Ferreira 2000; Chessell et al. 2002; Butler and Ferreira 2004] separates the compensation mechanism from failure handling and thus compensations can be used freely as any other programming construct. Compensations in StAC are stored in so called *compensation stacks* such that compensations can be installed, executed and discarded through stack operations. Although this approach makes automation of compensation installation impossible, it provides total freedom to the programmer to use compensations as deemed necessary. StAC$_i$ is an extension of StAC supporting concurrent compensation stacks, implying that several compensation scopes can be maintained concurrently during the execution of a process. This is useful when for example, one wants to be able to execute a particular "compensation" if all bookings succeed (e.g. confirming all bookings) and a different compensation if one of the bookings fail (e.g. cancelling all bookings). Additionally, StAC$_i$ (but not StAC) provides a mechanism for protecting a process from early termination originating from another process. This guarantees that processes are interrupted only when it is safe to do so.

— Transaction calculus (t-calculus) [Li et al. 2007a; Li et al. 2007b] is a highly expressive language which, building on the ideas of Sagas [Bruni et al. 2005], StAC [Butler and Ferreira 2004] and cCSP [Butler et al. 2004], provide an algebraic semantics for transactions. The transaction calculus is notorious for providing two exception handling constructs apart from the compensation mechanism: forward and backward handling. The difference is that the forward mechanism treats the handler as a successful alternative forward behaviour to the failure, i.e. after successful forward exception handling, execution continues as if everything was successful.

On the other hand, the backward mechanism uses the handling logic as a reversal of any execution which occurred before the exception, i.e. after successful backward exception handling, execution has to restart the failed process. Unlike StAC but like Sagas and cCSP, t-calculus uses exception handling only when compensation fails. Similarly to cCSP, the calculus is based on atomic actions which always succeed and each action must have an associated compensation action which can be the empty process or the process which always fails. Similar to Sagas, t-calculus provides native advanced operators including speculative choice, alternative forwarding and programmable compensation. Also, it does not provide a means of termination handling.

— Amongst the suggested languages to tackle programming complex services and interactions across organisational boundaries are Microsoft's XLANG [Thatte 2001] and IBM's WSFL [Leymann 2001]. The Business Process Execution Language for Web Services, known as BPEL4WS [Andrews et al. 2003] and more recently WS-BPEL [Arkin et al. 2007] (or BPEL for short) is the offspring of both XLANG and WSFL, integrating XLANG's structured nature with WSFL graph-based approach. Due to the widespread application of BPEL in industry and its incorporation of compensations, it has been given much attention in the area of compensations. Various attempts have been made to formalise the semantics of BPEL since these have been originally given in textual descriptions. Among these attempts, several [Hamadi and Benatallah 2003; Arias-Fisteus et al. 2004; Koshkina and van Breugel 2004; Haddad et al. 2004; Fu et al. 2004; Pistore et al. 2004; Wombacher et al. 2004; Viroli 2004; Verbeek and van der Aalst 2005; Pistore et al. 2005; Baldoni et al. 2005; Kazhamiakin and Pistore 2005; Yang et al. 2006; Pu et al. 2006; Nakajima 2006; Weidlich et al. 2007; Mateescu and Rampacek 2008; yun Long and shi Li 2009] do not take compensations into consideration. In a more detailed report [Colombo and Pace 2011] we delve deeper into the literature [Farahbod 2004; Farahbod et al. 2005; Fahland and Reisig 2005; Fahland 2005; Butler et al. 2005; Lucchi and Mazzara 2007; Lohmann 2007; Ouyang et al. 2007; He et al. 2008; Ferrara 2004a; 2004b; Pu et al. 2006; Foster et al. 2006; Foster 2006; Eisentraut and Spieler 2008; Abouzaid and Mullins 2008; Lapadula et al. 2008b; Coleman 2004] which formalises BPEL including its compensation mechanism. These formalisms include abstract state machines, Petri Nets, and process algebras.
The main construct of BPEL scoping, which acts as a process container providing a fault, a compensation and a termination handler for the scope-enclosed process. In BPEL, it is always the responsibility of the fault handler to handle faults (and never of the compensation handler). However, the default fault handler invokes the compensation handler so that any actions completed before the occurrence of the fault can be compensated. Note that the fault handler can be customised and need not call the compensation handler. Unlike Sagas and t-calculus but like StAC, BPEL always installs compensations in sequence (never in parallel) irrespective of whether the forward behaviour was executed in parallel. Note that BPEL provides a termination handler which allows the programmer to decide what actions need to complete before a process is forced to terminate.

Variants of process algebras have also been extensively used to model business processes and web services. Unlike flow compensation approaches, they focus primarily on the communication taking place amongst processes, which is particularly salient when modelling systems scattered across a network.

— $\pi\mathtt{t}$ calculus [Bocchi et al. 2003] is an extension of the π-calculus [Milner 1999] having a transaction construct. A transaction includes four components: a main process, a failure manager, a compensation store, and a compensation process. The failure manager is a process which is executed in case the transaction fails, the compensation store is a collection of compensations which compensate for previously completed nested transactions, while the compensation process is a process which will be added to the compensation store if the current transaction succeeds. $\pi\mathtt{t}$ calculus is particular in that the failure manager is always executed after the compensation store is executed. This allows the programmer to decide how to continue after compensation. A characteristic which is only present in $\pi\mathtt{t}$ calculus is that it does not force processes to terminate when a process fails in a parallel composition. Furthermore, compensations can only be installed in parallel — never in sequence — meaning that irrespective of the order in which actions were executed, the order of their compensation is arbitrary.

— SOCK [Guidi et al. 2006; Guidi et al. 2008; Lanese and Zavattaro 2009] is aimed specifically as a calculus for service oriented computing. For example, SOCK provides the request-response mode of communication through which a client can request an activity on a server and receive back the output of the activity. For such a scenario, SOCK also offers other notions such as the concept of a location — a process is not only distinguished by its definition but also on the location where it is running. SOCK provides three error handling mechanisms: fault handling, termination handling and compensation handling — all centred around a process container called a *scope*. The scope associates fault names with fault handlers and the scope's own name is associated with the termination handler. If the scope terminates and the termination handler has not yet been invoked, then the termination handler becomes the compensation handler for that scope (as the scope has been successfully completed). Subsequently, the name of the scope can be used to trigger the compensation handler in case the successful scope needs to be compensated. In SOCK, handlers (any type) can be modified at any point of execution. This provides a high degree of flexibility and does not impose any predefined policy on the programmer. A special feature of SOCK is that it provides a mechanism for distributed compensation. This is achieved by allowing a server to send a failure handler to the client. Thus, if the operation on the server fails, the client is informed by the server how compensation can take place.

— $\mathtt{web}\pi$ [Laneve and Zavattaro 2005; Mazzara and Govoni 2005] is an extension of asynchronous π-calculus [Hennessy 2007] with a special process for handling transactions. A notable aspect of $\mathtt{web}\pi$ is that it has the notion of a timeout: if a transaction does not complete within a given number of cycles, then it is considered to have failed. A variation of $\mathtt{web}\pi$ which abstracts from time is $\mathtt{web}\pi_\infty$ [Mazzara and Lanese 2006]. When the notion of time is abstracted, a transaction which has not timed out is a transaction with normal behaviour, while otherwise it is a failure. Both flavours of $\mathtt{web}\pi$ provide the notion of mobility of processes across machines which is particularly useful when modelling web services. $\mathtt{web}\pi$ differs from all other models of compensation in that transactions composed of other transactions are treated as the parallel composition of the parent and child transactions. The implication is that no automatic error propagation occurs in webπ, i.e. if a transaction fails, no other transaction is affected. In all other models, when a transaction fails (unless the failure is corrected) the failure propagates to the parent transaction. However, failure propagation can still be programmed manually in webπ through channel communication. Another minimalistic aspect of webπ is that exception handling and compensation handling are one and the same thing. This implies two serious limitations in the expressivity of compensations supported: (i) to compensate for completed transactions one has to manually keep the transaction alive to preserve the compensation handler; and (ii) the order of compensation execution cannot take into account the execution history.

— dcπ [Vaz et al. 2009] is another process calculus based on asynchronous π-calculus. It differs from other models in that instead of associating compensations to a scope, it associates compensations with channel input, i.e. compensation installation occurs upon an input on a channel. Another peculiarity of dcπ is that it guarantees that installations and activations of compensations take place through proofs on the semantics. In dcπ each transaction is assigned a unique identifier through which it can be signalled to start executing its compensation. Thus, any process which is allowed to cause a transaction t to compensate, should be given access to a corresponding channel t. For this purpose, scope extrusion[7] is heavily used in dcπ to pass on access to such channels to the relevant transaction process. Another feature of dcπ is that after their completion, transactions still preserve their compensations so that these can be executed at any time later on. dcπ also provides a construct for protecting transactions (or compensations) from being discarded due to forced termination. A remarkable limitation is that dcπ only allows parallel composition upon compensation installation. This prevents the ordering of compensations from being based on the execution history.

— COWS [Lapadula et al. 2007b; 2008a] is specifically targeted to formally model web services, providing a way of applying formal methods to web services. In COWS, an endpoint which receives or sends service invocations is a tuple: a partner (conceptually similar to a location) and an operation. Thus, each partner can be involved in various concurrent operations. Notably, COWS provides a very restricted basic syntax, using which, richer syntax is defined, providing constructs for modelling failure, compensation, and scope. The main three basic operators which are crucial as building blocks are (i) a delimitation operator which defines the scope of a variable; (ii) a *kill* operator which terminates a scope; and (iii) a protection operator which protects a scope from the kill operator. Using these operators a process can be enclosed in a special scope with the possibility of associating failure and compensation handlers to it. Similar to webπ, in COWS, compensations are statically defined as part of the scope and thus cannot take the execution history into account. However, unlike webπ and like dcπ, COWS allows transactions to be compensated after their termination.

— The committed join (cJoin) [Bruni et al. 2004] is an extension of the join

[7] Informally, scope extrusion occurs when a restricted (local) channel is passed on to another process (outside of the scope of the channel).

calculus [Fournet and Gonthier 1996], enabling it to handle compensations. The join calculus has been devised to provide an abstract foundation for object-based languages and it follows the idea of chemical abstract machines which attempt to emulate chemical processes as a means of computation. Similar to process algebras, chemical abstract machines are apt to model interactions among processes and how processes evolve during execution. These aspects, together with a *negotiation* construct — a kind of scope construct which relates failure handling process to another process — make cJoin capable of modelling transactions with compensations. Like webπ and COWS, cJoin also supports only static compensation which is defined as part of the scope. Furthermore, like webπ, cJoin does not separate compensation and fault handling. Moreover, compensation is automatically discarded upon transaction termination and therefore completed transactions cannot be compensated.

A number of other calculi such as RCCS [Danos and Krivine 2005], τ-calculus [Field and Varela 2005], and TransCCS [De Vries et al. 2010a; 2010b] also enable the formalisation of transactions. In particular, TransCCS supports the specification and verification of safety and liveness properties in the context of transactions. These calculi differ from the others overviewed above in that the former support rollback of transactions rather than compensations. The essential difference lies in the fact that transactions are reversed automatically by reverting to a savepoint while compensations have to be programmed explicitly.

Finally, automata have also been used to model compensating transactions. Apart from the advantage of being a graphical notation, a lot of work has already been done in automata particularly in the area of verification which can then easily be adapted for compensations.

— Communicating hierarchical transaction-based timed automata (CHTTAs) [Lanotte et al. 2006; 2008] are communicating hierarchical machines [Alur et al. 1999] enriched with time (similarly to timed automata [Alur and Dill 1994]), and with other slight modifications to accommodate the representation of transactions. Two appealing features of CHTTAs (apart from the inherent graphical aspect) is that they support the notion of time and can be reduced to timed automata and hence are model-checkable. Long running transactions (LRTs) are defined over and above CHTTAs such that a CHTTA can be specified as the compensations of another CHTTA. Furthermore, LRTs can also be nested or composed in parallel or sequentially. Similarly to a number of other approaches, the order of compensation execution in LRTs is in reverse order in case of sequence and in parallel in case of a parallel composition. Also, in the case of successfully aborted nested transactions, the parent transaction is not aware of abortion and continues unaffected. A limitation of LRTs is that they do not show clearly (graphically) which compensation corresponds to which component and it is assumed that compensations succeed. The latter limitation can be lifted by introducing exception handling which is completely absent in LRTs. Another mechanism which LRTs do not provide is forced termination and consequently neither termination handling.

4.1 Comparison by Example

To illustrate some of the differences between the different approaches, we encode the example given in Section 2.4 in Sagas, SOCK and BPEL. The example can be found encoded in other compensation modelling languages in [Colombo and Pace 2011].

— *The Example in Sagas:* Apart from the usual sequential and parallel composition operators, Sagas employs a number of compensation-related ones. For example, in order to associate actions to their compensations, Sagas uses the $\div$ operator, e.g. $\mathit{DecStock} \div \mathit{IncStock}$ represents the fact that increasing stock is the compensation of increasing it. Furthermore, **try** a **with** b is used to associate exception handling b to a compensable unit a so that if a failure occurs during compensation, the handler b is triggered. Finally, $\{\![\ ]\!\}$ are used to delineate the scope of compensations such that if a failure is successfully compensated, then compensation is not propagated beyond the scope. Using these operators, the example can be encoded as follows:

$$\begin{aligned}\mathit{Transaction} \overset{\text{def}}{=} \textbf{try}\ \{\![\ & (\mathit{DecStock} \div \mathit{IncStock})\,;(0 \div \mathit{Email})\,;\\ & ((\mathit{Pack} \div \mathit{Unpack}) \mid\mid (\mathit{Credit} \div \mathit{Refund}))\,;\\ & (\mathit{Courier} \div \mathit{Cancel})\ ]\!\}\ \textbf{with}\ \mathit{Operator}\end{aligned}$$

Note that in Sagas actions *DecStock*, *IncStock*, etc, are not decomposed further but are assumed to either succeed or fail according to their context. This is in line with the big-step semantics in which the semantics of Sagas is given. Furthermore, a compensation in Sagas is a single activity and as such the sending of an email had to be encoded as the compensation of the inert process — $0 \div \mathit{Email}$. By so doing, we have not strictly kept to the specification because the stock update and the sending of the email are supposed to be done in parallel. The alternative would have been to amalgamate both activities as a single basic activity. In this example, we have also deviated from the original specification as the interaction among the various entities involved (e.g. bookshop, warehouse, etc) in the transaction is not explicitly modelled.

— *The Example in SOCK:* Unlike Sagas, SOCK models interaction among processes as channel communication with c and $\overline{c}$ representing input and output on channel c, respectively. Another major difference is that compensation and failure handlers are installed as processes attached to handler and scope names. For example $s \mapsto cH \mid\mid \mathit{IncStock} \mid\mid \mathit{Email}$ represents the fact that in parallel with anything already associated with handler s (represented by cH), two additional actions are composed: increasing the stock and sending an email to the client. The example modelled in SOCK is given below:

$$\begin{aligned}\mathit{Store} &\overset{\text{def}}{=} \mathit{order}\,;(\overline{x} \mid\mid (x) + (x\,;\mathit{throw}(st))) \mid\mid \mathit{restock}\,;(\overline{x} \mid\mid (x) + (x\,;\mathit{throw}(rs)))\\ \mathit{DecStock} &\overset{\text{def}}{=} \overline{\mathit{order}}([s \mapsto (cH \mid\mid \mathit{IncStock} \mid\mid \mathit{Email}), st \mapsto \mathit{throw}(f)])\\ \mathit{IncStock} &\overset{\text{def}}{=} \overline{\mathit{restock}}([rs \mapsto \mathit{throw}(g)])\\ \mathit{Transaction} &\overset{\text{def}}{=} \{\mathit{inst}([f \mapsto \mathit{comp}(s), g \mapsto \mathit{Operator}]) \mid\mid \mathit{IncStock}\,;(\mathit{Pack} \mid\mid \mathit{Credit})\}_s\end{aligned}$$

Modelled in a service-oriented fashion, the *IncStock* activity contacts the *Store* through channel *order* and if the activity succeeds (modelled as a non-deterministic choice using the + operator), then the activity exits, otherwise, it throws fault *st*. This fault is handled by *DecStock* and rethrown as fault f which is in turn handled by *Transaction* — triggering the compensation for scope s. If the compensation (*IncStock*) fails, fault g is triggered instead of f, causing the *Operator* activity to start. The activities which are not given above should be modelled in a similar fashion to *DecStock* and *IncStock*. Note that in SOCK we have kept to the specifications and installed compensations in parallel.

— *The Example in BPEL:* A BPEL process is a composition of scopes where each scope performs some kind of function and may have up to three handlers associated to it: a fault handler, a compensation handler and a termination handler. A fault handler intercepts faults and handles them, possibly by executing other handlers such as a compensation handler. As a scope can only be compensated upon successful completion, the outermost scope can never be compensated and therefore it cannot have an associated compensation either. The following is a skeleton of the BPEL specification of the bookshop example:

```
<process name="bookSellingTransaction">
  <documentation xml:lang="EN"> A process for handling orders </documentation>
  <faultHandlers> <catchAll> <compensate /> </catchAll> </faultHandlers>
  <sequence>
    <scope name="stockUpdate">
      <documentation> This is the scope for handling the stock level </documentation>
      <faultHandlers> <!-- This is the default fault handler>
        <catchAll> <sequence> <compensate /> <rethrow /> </sequence> </catchAll>
      </faultHandlers>
      <compensationHandler> ... reupdate stock levels and send email </compensationHandler>
      ... check stock and decrease
    </scope>
    <flow>
      <scope name="packOrder">
        <documentation> This is the scope for packing the order </documentation>
        <compensationHandler> ... unpack </compensationHandler>
      <!-- This is the default termination handler>
        <terminationHandler> <compensate /> </terminationHandler>
        ... pack
      </scope>
      <scope name="chargeCustomer">
        <documentation> This scope is for charging the bank account </documentation>
        <compensationHandler> ... refund customer </compensationHandler>
         ... charge customer's bank account
      </scope>
    </flow>
  </sequence>
</process>
```

The encoding in BPEL is done using three scopes — one for each activity having a compensation. Scopes are the only mechanism in BPEL through which compensations can be associated to activities. Note that in all scopes, the default fault handling mechanism is utilised (only explicitly shown for the scope *stockUpdate*; it is implicit in the other cases). This simply executes the installed compensations (using `<compensate />`) and rethrows the fault to the next higher level scope. Similarly, we only explicitly declare the default termination handler in scope *packOrder* so that, if for example the bank account transfer fails while the packing is still executing, compensation is executed (assuming packing has a further nested

scope).
The rest of the details (which are left out) can be implemented using BPEL processes such as `assign` to update stock values, and `invoke` and `receive` to communicate with the bank and the packing department.

Although simplistic, the example presented in this subsection should illustrate the main differences among the different models used to model compensations: mainly that flow composition languages generally provide a more concise representation of the logic with a clear delineation of activities and compensations, process algebraic approaches focus on the communication aspect, while BPEL is intended as an implementation to be executed on a BPEL engine.

4.2 Conclusion

The formalisms presented in this section provide varied approaches to the design issues of compensations. Undoubtedly, there are various other formalisms catering for compensations. For example, [Acu and Reisig 2006] extends the work on workflow nets in [van der Aalst 1998] to include compensations. However, the concept of compensation employed is similar to that of Sagas [Bruni et al. 2005] where the order of the compensations depends on the order of execution, establishing a partial order among activities. Therefore, we opt to leave out further details of such work. Similarly, we leave out others works which we feel are already in some way represented in the notations tackled in this section.

Although we have reviewed various compensation models, we have not presented what it means for a compensation to be correct in a given context. In the next section, we give an overview of the literature which tackles this issue.

5. FORMALISING COMPENSATION CORRECTNESS

The complexity of compensation semantics raises the question of the soundness of a given language or approach as to whether it really deals with compensations correctly. In this section we review two main approaches of compensation correctness appearing in the literature. The first approach assumes that compensations are perfect, i.e. running a compensation after the compensated action would result as if no action has been taken at all (all-or-nothing). This assumption facilitates reasoning about compensable transactions as one would simply have to ensure that either the transaction succeeds fully or not at all (up to compensations). The second approach revolves around algebra of programs, i.e. how compensation activities would fit within a program and what laws apply in general to programs with compensations. Such generic laws can then be used to manipulate any program with compensations.

5.1 All-or-Nothing Correctness Principle

The approach usually taken to prove sanity of compensation semantics is that *a system built from atomic actions all of which have a perfect compensation*[8] *associated to them, will either successfully complete its behaviour, or will be aborted*

[8] How *perfect* a compensation is, depends on the level of abstraction one is viewing the behaviour. For example *delete* is a perfect compensation for *insert* if one views an index at the level of abstraction of an index. It is highly improbable though, that at the bit level the original index

but will not leave any side effect on the system state. Note the strong assumption of perfect compensations is rarely the case in practice, but is only used to ensure that the order of execution of triggered compensations is as expected. Different approaches have been taken in the literature:

— *Perfectly compensated programs:* The cancellation semantics given for cCSP [Butler et al. 2004] are an attempt to formalise the correctness criterion that a compensable transaction should either succeed or be equivalent to the inert process *skip*, i.e. it leaves no side effects. The result assumes that the compensating actions are perfect inverses of the compensated actions and that compensations are independent of each other (they commute with respect to sequential composition). The approach adopted to define soundness, is to consider perfect compensation of basic activities, and proving that the formal semantics guarantee that the derived compensation of a transaction (built compositionally from basic activities) is still a perfect compensation of the transaction. The semantics of a compensable process in cCSP is given as sets of pairs of traces — the forward behaviour, and the accumulated backward (compensating) behaviour. The sanity check for compensating programs is thus that the trace semantics of a transaction block, may only include: (i) successfully terminating traces; and (ii) traces which after applying commutativity (to enable moving of compensations forward and backward through sequential composition) and cancellation (a basic action A and its compensation A' satisfy $A; A' = \mathit{skip}$) are equivalent to *skip*.

— *Relaxing equivalence:* The work in [Caires et al. 2008] weakens the strong assumption of equivalence of behaviour by parametrising the verification up to a given equivalence relation $\bowtie$. Rather than assuming that each atomic action A is associated with its perfect compensation A', it is assumed that A' *reverts* A — that executing A followed by A' will yield a state equivalent (up to $\bowtie$) to the original one. It is assumed that compensation execution is always successful.
The *atomicity property* of $\bowtie$-consistent programs (in which basic actions are reverted by their compensations) states that the resultant behaviour of a compensable program, P, simulates a non-deterministic choice between the successful forward behaviour of the program P^+ (dropping accumulated compensations), and the throwing of an error but with no further side effect *throw*: $P \sqsubseteq P^+ \oplus \mathit{throw}$.

— *Transactions dependencies:* The notion of compensation correctness in [Korth et al. 1990] revolves around the effect of a transaction on other transactions. Rather than looking at a transaction behaviour locally, the sanity condition is that from the point of view of transactions dependent on a particular transaction T, the behaviour of T failing and compensated for, is indistinguishable from the case when T has never been executed. The model focuses on low-level database actions as the basic atomic building blocks.
Seeing a computation history as a function over system states, and using a notion of transaction dependency (where the set of transactions dependent on transaction T is defined to be those which use values which T can write to), they define the notion of compensation soundness as: *A transaction T (with dependant transactions D_T) and*

is identical to the index following an insert and a compensating delete. This obviously has an impact on the notion of correctness.

its compensation T' are said to be sound if at any point in the execution, the history of D_T is equivalent to the combined histories of T, T' and D_T. In practical terms, the outcome of the transactions depending on T is indistinguishable whether (i) T is never executed; or (ii) T occurs followed by its compensation. More succinctly, the transactions which depend on T are not affected if T is run and undone using its compensation T'.

The principle of compensation correctness is similar in all approaches, namely that of giving the illusion of atomicity — that either the transaction succeeds in its entirety or it fails and leaves no trace of its execution. The differences lie in the level of abstraction at which they tackle the issue and in the underlying assumptions. All approaches take the assumption that compensations succeed and thus show that perfect compensation is compositional. Atomicity of execution is, however, just one of the desired properties of compensation logics and languages. Other issues include, for instance, the effect of compensation replacement — ensuring that replacing a compensation only affects the compensation stack locally — and the effect of failures in compensations — ensuring that triggering backward execution multiple times ensures overall failure resolution. Although most of the logics deal with these issues directly by being direct consequences of the semantics, there still lacks a complete formalisation of the properties expected to hold of compensation-aware languages.

5.2 Algebraic Properties of Compensable Programs

The algebraic approach to programming proposed by Hoare *et al.* [Hoare et al. 1987] used the notion of *program quotients*, where the quotient[9] of program P with respect to program (or specification) Q (written P/Q), returns the weakest specification or program Q' such that executing Q and Q' sequentially is a refinement of the original program P (P/Q is the weakest X such that $Q; X \sqsubseteq P$). Although the original paper makes no mention of compensations, this operator enables one to express the notion of a perfect immediate compensation of a program P as $\texttt{skip}/P$ (where `skip` is the program that has no effect on the state), since it corresponds to the weakest program X such that $P; X \sqsubseteq \texttt{skip}$.

These algebraic laws of programming were extended to reason about compensable programs [Jifeng 2007], where laws of a programming model with a notion of compensations which are composed in reverse sequential order are given. These laws also correspond to soundness criteria for compensation handlers, as for example, the property that a pure compensation installer commutes (over sequential composition) with assignment and that installing two compensations in sequence is equivalent to installing them as a single compensation but with their order switched.

5.3 Conclusion

Formal reasoning based on the all-or-nothing principle provides a mechanism to reason about the sanity of compensation formalisms while algebraic laws provide a means of reasoning about compensable programs. A considerable limitation of

[9] In fact, [Hoare et al. 1987] introduces both left and the right quotient operators. In this paper, only the right quotient is relevant to the discussion.

these works is that the assumptions involved are quite rigid vis-à-vis practical challenges. For example, what if an action cannot be completely compensated or one consciously wants to compensate a part of the transaction later on (see [Greenfield et al. 2003] for more examples)? This is potentially a direction of future work in the area of compensations, enabling a more flexible reasoning approach.

6. VERIFICATION OF COMPENSATING TRANSACTIONS

Apart from checking the sanity of compensating transactions through notions of correctness, one would also like to be able to specify and verify more specific behavioural properties about transactions. For example, can it be guaranteed that if the credit check fails, the courier booking has not taken place? In order to verify such transaction behaviour, one would need an abstraction of transaction implementation and a notation to describe the set of acceptable transaction behaviours, i.e. a specification. Three main approaches in the literature for compensable transaction verification use different levels of implementation abstraction. At the lower end of the spectrum, the proposed abstraction is a transaction trace which preserves both order and repetition of transaction actions. On the other end, transaction actions are represented as a set of completed actions, thus discarding order and repetition information. The higher the level of abstraction, the more behavioural information is lost but verification becomes cheaper as the range of possible verifiable behaviours becomes more limited. For example temporal properties cannot be verified on implementation abstractions which discard order and repetition details. More details are given below, starting with the highest level of abstraction.

— *Set Consistency:* The set consistency approach [Fischer and Majumdar 2007] discards ordering and repetition of transaction actions and represents a transaction execution in terms of a set of successfully completed actions. Any actions which have either not been executed or have failed are not members of the set. For example, if in the case of the bookshop, courier booking failed, then a possible set of executed actions is: $S = \{\mathit{DecStock}, \mathit{Credit}, \mathit{Pack}, \mathit{Unpack}, \mathit{Refund}, \mathit{IncStock}\}$. A property is then defined in terms of which actions should and should not be included in the set. The example property that unless the payment succeeds, courier booking should not take place can be specified as: $\mathit{Credit} \notin S \Rightarrow \mathit{Courier} \notin S$. This is clearly satisfied by the transaction execution above.

— *Acceptable Terminating States:* A similar approach to the set consistency approach is the set of *acceptable termination states* (ATS) [Li et al. 2008] approach. The main difference is that the set elements do not simply show whether an action has completed (through membership) or failed (through omission). Rather, a special tag can be associated with each action where tags are based on the idea that a transaction goes through the following life cycle: it starts as *idle*, is *activated*, possibly encounters an error and either successfully **aborts** or **fails**, otherwise if **successful**, if might be later **compensated** or **half-compensated** if compensation fails. Amongst these states, the set of possible final states are abbreviated as: $\Delta = \{\mathit{abt}, \mathit{fal}, \mathit{suc}, \mathit{cmp}, \mathit{hap}\}$ (reflecting the states in bold). Using these as tags, the following is a possible termination state of the bookshop example: $\{\mathit{IncStock}.\mathit{cmp}, \mathit{Pack}.\mathit{cmp}, \mathit{Credit}.\mathit{cmp}, \mathit{Courier}.\mathit{abt}\}$, meaning that the courier booking failed and all the previously completed operations were successfully compensated. Specifying properties in this approach involves specifying the set of termination states which the transaction is allowed to reach. If the set of acceptable (allowed) termination states all fall within the set of possible termination states of the transaction, then the specification is satisfied. Taking the above example, the ATS might include: $\{\mathit{IncStock.cmp}, \mathit{Pack.cmp}, \mathit{Credit.cmp}, \mathit{Courier.abt}\}$ which signifies that it is allowed to complete the payment and later fail the courier booking. This behaviour can be generated by the current transaction specification and thus verification succeeds.

— *Temporal Constraints:* A verification approach which does not abstract the order and repetition of executed transaction actions is given in [Li et al. 2008] by applying verification on traces. A trace of a transaction is an ordered list of transaction states (see previous point) which the sub-transactions have gone through. For example, taking the bookshop example and assuming that the packing failed, the following would be the generated trace: $\langle(\mathit{DecStock}, \mathit{suc}), (\mathit{Credit}, \mathit{suc}), (\mathit{Pack}, \mathit{abt}), (\mathit{Credit}, \mathit{cmp}), (\mathit{DecStock}, \mathit{cmp})\rangle$. This allows for more expressive properties to be verified, particularly temporal properties for which order is crucial. Using the temporal specification language supported in [Li et al. 2008], one can specify constraints such that eventually an action should occur, or that an action should always be preceded by another. For example the property that a successful courier booking should always be preceded be a successful payment can be written as $(\mathit{Credit}, \mathit{suc}) \ll (\mathit{Courier}, \mathit{suc})$. A transaction satisfies such a property if all the transaction traces fall with the set of all traces which respect the temporal property.

6.1 Conclusion

The first two approaches outlined in this section are quite limited in terms of the expressivity of supported properties because they discard all ordering information. On the other hand, the third approach allows temporal constraints on traces which are however more expensive to check due to the greater number of possible traces as opposed to possible sets. To the best of our knowledge only the consistency set approach has been implemented and applied to realistic examples. Note that there were other attempts at verifying business transactions [Pistore et al. 2004; Koshkina and van Breugel 2004; Arias-Fisteus et al. 2004; Kazhamiakin and Pistore 2005] mainly by translating the transactions to existing formalisms which support verification. However, none of these works fully supports the verification of transactions having compensations.

7. PRACTICAL ASPECTS OF COMPENSATIONS

In this section, we aim to give insights to prospective users of compensations where and where not to apply compensations. After delineating the use of compensations from the use of reparations, we give examples of practical applications which employ compensations. Subsequently, if an application is fitting for the use of compensations and one wants to model it formally, then the right formalism must be chosen. For this reason, we give an outline of the main features of the formalisms presented in this survey.

7.1 When to use Compensations

The advantages of using compensations over forward-recovery mechanisms, or reparations, mainly come in when the application domain has: (i) a transaction format such that when an activity fails, the previously completed activities require reversal; and (ii) clear corresponding compensations for its actions. In such cases, the compensation mechanism provides a structured way of organising the activities and their respective compensations. This is not to say that reparations cannot be used to encode compensations in many circumstances. However, in general compensations are more expressive than reparations [Lanese et al. 2010]. Since compensations are generated *dynamically* at runtime while reparations are defined *statically* at compile time, compensations have access to runtime information (such as execution ordering) which might not be available at compile time. Furthermore, another crucial difference between compensations and reparations lies in the state of a system following a failed activity A. If a reparation is executed after the failure, the resulting state is assumed to be equivalent to the state after a successful execution of A. On the other hand, if a compensation is executed, the state is assumed to be equivalent to the state before the start A.

The virtue of compensations is also their limitation: compensations follow a pattern of backward and forward behaviour which allows automatic composition of backward activities, reflecting forward ones. Various applications would not fit into such a pattern; particularly when the compensation is structurally different from the forward behaviour. For example consider a more complex online bookshop which in case some books are not in stock, it does not cancel the whole transaction but simply delivers the books which are in stock and deliver the other books when they are available in the future (see [Greenfield et al. 2003] for a similar but more extended example). In such a case the distinction between backward and forward behaviour is blurred and it would probably make more sense to model such behaviour in terms of reparations and be treated as forward behaviour.

It is not always clear when to use reparations and when to use compensations. Indeed, in a number of cases they offer a similar level of expressivity and the distinction between them may not be so clear. It is important to realise that compensations and reparations need not be substitutes. Rather, they can be seen as complementary tools, offering distinctive mechanisms to handle errors. As a matter of fact, it is not unusual to find them combined together for complex recovery situations — with reparation being triggered when a compensation fails, or with compensations applied when reparations do not have sufficient context information to fix the failure. The challenge is to use compensations and reparations in their rightful roles with the possibility of one complementing the other.

7.2 Applications of Compensations

Compensations usually accompany long running transactions to enable committed activities to be semantically undone. Therefore in areas of application of long running transactions one is bound to find compensations. Such applications have evolved through time [Wang et al. 2008]: (i) the first notion of compensation emerged when transactions started to become longer and more complex, frequently necessitating decomposition — the focus at this stage was to ensure that the application carries out the expected functions correctly; (ii) following this stage, the focus shifted towards workflows, i.e. how functions can be composed together to form reliable processes; and (iii) the latest stage shifts the focus on interactions across a number of geographically distributed participants, i.e. how distributed workflows can be composed reliably. Many examples in recent compensation literature fall under the latter stage: trip booking systems where a booking potentially includes a number of other bookings [Arkin et al. 2007; Bocchi et al. 2003; Bruni et al. 2004]; online sales systems where a number of parties are involved usually including the customer's bank and the courier [Bruni et al. 2005; Butler and Ferreira 2004; Li et al. 2007b; Butler et al. 2004; Arkin et al. 2007; Vaz et al. 2009]; and shipment example where potentially an order requires one or several shipments [Arkin et al. 2007; Lanese and Zavattaro 2009; Lapadula et al. 2007b]. Using more practical business terminology [Sarang et al. 2006; Bolie et al. 2006], long running transactions (and thus compensations) are most often used to solve a problem of integrating functionality across platforms and businesses. Although all the literature examples involve business-to-business interaction (B2B), this need not always be the case. Within a single business, frequently enterprise application integration (EAI) is used to provide middleware connecting various applications while enterprise resource planning (ERP) systems aim to provide a single readily integrated solution. However for various reasons such as legacy systems, mergers, etc, a business might end up having several EAIs and/or ERPs. In such a scenario, the service oriented architecture (SOA) provides the basic mechanism for integration by exposing functionality in a standardised way. Once services are available, these can be composed into business processes, i.e. long running transactions, to provide more sophisticated business solutions. As a real-life example consider a scenario [Bolie et al. 2006] where a business has both a Siebel and a SAP system, each with its own middleware: TIBCO and webMethods respectively. Using web services, a common interface can be provided by both systems (to both systems). Using an orchestration language such as BPEL, these web services can be orchestrated to provide a single interface to users while under the hood web services ensure that both systems remain synchronised, e.g. by coordinating inserts, updates and deletions. As a concrete example of a B2B integration, consider a travel agent's service [Sarang et al. 2006] where the travel agent receives travel requests and provides the user with a list of options ordered by price. Assuming the airlines have their functionality exposed as web services, the actual (potentially diverse) systems used by the individual airlines become irrelevant. Each airline would receive a service request and reply through a service response, supplying the necessary details. Once the travel agent receives the information from the airlines, this is communicated back to the client, ordered in some preferred way. Note that the emphasis in these examples are not compensations. However, compensations are a necessity to compose such loosely coupled services which interact with real-life actions and third-party activities which frequently cannot be simply undone.

From a practical point of view, BPEL is considered to be the de facto standard as a means of composing and executing web transactions and as such it probably constitutes the vast majority of use of compensations in industry. However, while the logics reviewed in this survey *focus* around the notion of compensation, BPEL is a language which has compensation as one of its constructs. The aim of compensation in BPEL is to have a practical and useful construct to be used in practice, and not to explore the choices or semantic issues with compensations. Compensations are thus but one (albeit important) part of BPEL and it would be misrepresenting BPEL to call it a compensation language. Still, if one compared BPEL from a compensation point of view of the other notations, there are various aspects in which BPEL is particular. For example, an error is always handled by exception handling and then it is the exception handler which triggers compensation execution. This approach allows full customisation of compensation during exception handling. Another important aspect which one has to keep in mind when adopting BPEL is that it is an orchestration approach. This means that interactions defined in BPEL are defined from the point of view of a single participant which is in charge of the interaction. This differs from the process algebra approaches presented earlier where each participant is equal in the interaction — the choreography approach. Therefore, before achieving an orchestration, the parties involved should already have agreed to participate in the interaction. For more details about how BPEL relates to other protocols we refer the reader to [Sauter and Melzer 2005; Kopp et al. 2009] and to [Van Der Aalst et al. 2003; van den Heuvel 2008] for surveys about business process specification.

Apart from these applications to which compensation are commonly associated, there are various other applications of compensations. Fundamentally, a compensation is a means of logically reversing the state of a system. As such, this is useful in any application where synchronisation between two or more parties requires some of the participants to go back some steps to be in line with the rest. A typical application are distributed games [Bernier 2001; Mauve et al. 2004] where all the players are participating in a global state and synchronisation is not instantaneous, meaning that state discrepancies are frequent. Compensations are then used to correct the local states to achieve a synchronised global state of the game. A similar application is the use of compensations in asynchronous monitoring of software [Colombo et al. 2010] where the system is allowed to proceed ahead of the monitor. When a violation is detected, compensations are used to revert the system back to the state exactly after the violation so that correction may take place.

7.3 Which Model to Choose

Having such a vast selection of formal notations to choose from makes it difficult for a user to choose the right formalism for a particular application. The models we have presented fall within the latter two evolution stages of compensations outlined in the introduction (of this section). These stages greatly facilitate the choice of a compensation model as most models are either flow composition languages which are useful to model applications which fall under stage (ii) while the rest focus on interaction among collaborating parties, facilitating modelling of applications of the kind which fall under evolution (iii).

In the rest of this subsection, we aim to highlight the main features of each formalism in an attempt to narrow down the choice process to individual models. Therefore, as in Section 4, we group the compensation models under flow composition models, process algebras, and automata. Note that we will treat BPEL separately in the next subsection.

— *Flow Composition Models:* Flow composition approaches usually have explicit operators for most compensation operations and they leave out notions of participants or communication among them. Therefore in this sense, understanding what is going on in terms of high-level composition of actions is generally easy when using flow composition models. Excluding StAC, the main difference among flow composition models is their focus or their style. Sagas can be considered as the most basic of the flow composition languages, providing all the basic compensation operators. This makes Sagas ideal for someone who would simply like to build a simple model by plugging different actions together to form a flow. Both cCSP and t-calculus are very similar to Sagas. However, cCSP provides the yield operator which allows processes to decide when to allow external interruption. This is useful when one needs to build a more realistic model of workflows. t-calculus does not support handling of external interruption but provides an extra operator for exception handling. This is useful when one needs to model a system where exception handling can take the state either backward or forward. StAC is very different from the rest as it allows compensation to be used possibly with no connection to error handling. This makes it ideal for someone using compensations as a programming construct and not necessarily in the context of transactions.

— *Process Algebras:* While flow composition languages totally omit interaction, this is the main appeal of process algebras where activities are represented by channel communication. Anyone choosing a process algebra to model a system should be aware of a fundamental distinction separating webπ, COWS, and cJoin from πt calculus, SOCK, and dcπ. The former three support only statically defined compensations, while the latter three allow compensations to be composed dynamically through installation. Furthermore, πt calculus and dcπ only support parallel installation of compensations. Considering these aspects one can already narrow down the choice according to the system to be modelled. πt calculus is the only notation which always executes an exception handler after executing compensations. On the other hand, dcπ is the only notation which associates compensation installation to channel communication. This might be useful for someone modelling a system where channel input represents the completion of an action by some participant. cJoin is particular as it is based on a different interaction model — it is based on chemical abstract machines rather than the π calculus.

Another distinction among process algebras is that webπ, SOCK, and COWS have been specifically aimed for modelling web services. Hence someone wishing to model web services formally, should consider one of these three. Among these three, webπ and a timed version of COWS [Lapadula et al. 2007a] are the the only formalisms which support a notion of a timeout. SOCK should be chosen if a user wants a process algebra with high flexibility when installing compensations. COWS is particular as it models communication endpoints as a tuple including a partner and an operation. This allows the invocation of the same operation on different partners or the invocation of different operations on the same partner. SOCK offers something similar by allowing the specification of engines which support multiple instances of the same service.

— *Automata-Based:* The automaton-based notation, CHTTAs presented in this survey, is very similar to flow composition languages in that automata are composed together to form more complex flows. CHTTAs do not support advanced compensation features such as forced termination or handling of errors during compensation. The main appeal of CHTTAs is their support of the specification of real-time constraints which is absent in all the other notations. Furthermore, it is useful for anyone wanting to model check the compensation model as these automata are reducible to timed automata and hence model checkable.

After considering whether compensations can be useful for a task at hand, an appropriate theoretical notation can be useful as a tool for reasoning about the solution. Once a formal model has been built and validated, it can be implemented (or mapped directly) in an executable language such as BPEL. However, since BPEL does not have a formal semantics, in case of an implementation (as opposed to a direct mapping), one should be aware of what exactly happens under the hood (for example see execution details in Section 3.5). Otherwise, the resulting behaviour might be different from the intended behaviour.

8. SUMMARY AND CONCLUSIONS

Fuelled by the streamlining of complex business logic, the increase in electronic financial operations and of cross-entity interaction, modern day transactions require a robust framework, able to deal efficiently with failures. Furthermore, transactions became longer due to the communication involved with different parties, rendering ACID transactions inappropriate. The compensation mechanism was introduced to transactions, enabling them to handle the new challenges. Later, compensable transactions evolved and were integrated with more complex models involving amongst other aspects: parallelism, exception handling, transaction composition and communication amongst activities. A variety of approaches and models have emerged, providing different solutions to the design issues involved. After briefly describing a number compensation models, in this paper, we have compared them in depth with respect to many issues concerning compensations.

8.1 Trends in Compensation Research

Although compensable transactions started off as a means of handling complex database transactions, the focus has now shifted to modelling service-oriented interactions. Particularly, considerable attention has been given to the technology-supported language, BPEL, in numerous attempts to formalise it. Furthermore, apart from the work on BPEL, several formalisms have been devised to handle compensable transactions diverging in their approach from BPEL. This has led to a significant number of proposals and approaches, giving rise to a healthy spectrum of ideas. What is still significantly lacking is the application of various formal models to real-life scenarios. Moreover, another significant shortcoming in the area of compensations is that little work has been done to provide mathematical frameworks to reason about system states before and after the compensation is applied. Although theory in this respect has been proposed in [Caires et al. 2008], more work still needs to be done to apply the idea to a practical case study. Furthermore, more needs to be done to mathematically specify the properties which compensating transactions should satisfy. Future work can also be directed to apply the notion of compensations to other areas; possibly to areas such as automatic contract enforcement, and weak synchronisation in distributed systems. Finally, apart from a notion of a timeout event in BPEL (and some formalisms modelling it, e.g. [Pu et al. 2006; Haddad et al. 2008]) and work [Lanotte et al. 2006; 2008] which integrates compensations into hierarchical timed automata, the notion of real-time is lacking in the area of compensable transaction.

8.2 Final Remarks

Although compensations offer a rich construct for fault handling, the same aspect which enables automatic configuration of actions to generate a compensation in terms of lower-level activities, is the same characteristic which restricts their flexibility. In particular case studies, the compensation mechanism does not achieve an elegant solution, or possibly no solution at all (see for example [Greenfield et al. 2003]). Having said this, compensations are still useful both as a way of modelling many real-life scenarios as the current interest in compensations confirms.

An open discussion is whether compensations should be limited to failure recovery. In practice, a program with compensations is simply a program which remembers the execution trace in a stack so that when required, the program can perform particular actions by going through the stack. For example, consider resource management at the end of a program execution, the programmer might use the compensation stack to remember what resources have been claimed and automatically releases the resources by executing the stored compensations. However, if compensations are not used for recovery purposes, one might rightly question the appropriateness of the term "compensations".

ACKNOWLEDGMENTS

We wish to thank Adrian Francalanza for his feedback on preliminary versions of this paper, and the anonymous reviewers who provided concrete feedback on the content and presentation of the paper.

REFERENCES

ABOUZAID, F. AND MULLINS, J. 2008. A calculus for generation, verification and refinement of BPEL specifications. *Electr. Notes Theor. Comput. Sci. 200,* 3, 43–65.

ACU, B. AND REISIG, W. 2006. Compensation in workflow nets. In *Petri Nets and Other Models of Concurrency (ICATPN).* Lecture Notes in Computer Science, vol. 4024. Springer, Berlin, Heidelberg, 65–83.

ALUR, R. AND DILL, D. L. 1994. A theory of timed automata. *Theor. Comput. Sci. 126,* 2, 183–235.

ALUR, R., KANNAN, S., AND YANNAKAKIS, M. 1999. Communicating hierarchical state machines. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming.* ICAL '99. Springer-Verlag, London, UK, 169–178.

ANDREWS, T., CURBERA, F., DHOLAKIA, H., GOLAND, Y., KLEIN, J., LEYMANN, F., LIU, K., ROLLER, D., SMITH, D., THATTE, S., TRICKOVIC, I., AND WEERAWARANA, S. 2003. Business process execution language for web services v1.1. Available at: http://download.boulder.ibm.com/ibmdl/pub/software/dw /specs/ws-bpel/ws-bpel.pdf (Last accessed: 2010-02-17).

ARIAS-FISTEUS, J., FERNÁNDEZ, L. S., AND KLOOS, C. D. 2004. Formal verification of bpel4ws business collaborations. In *E-Commerce and Web Technologies (EC-Web).* Lecture Notes in Computer Science, vol. 3182. Springer, Berlin, Heidelberg, 76–85.

Arkin, A., Askary, S., Bloch, B., Curbera, F., Goland, Y., Kartha, N., Liu, C. K., Thatte, S., Yendluri, P., and Yiu, A. 2007. Web services business process execution language version 2.0. OASIS Standard. Available at: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf (Last accessed: 2010-02-17).

Baldoni, M., Baroglio, C., Martelli, A., Patti, V., and Schifanella, C. 2005. Verifying the conformance of web services to global interaction protocols: A first step. In *Formal Techniques for Computer Systems and Business Processes, European Performance Engineering Workshop, EPEW 2005 and International Workshop on Web Services and Formal Methods, WS-FM 2005*. Lecture Notes in Computer Science, vol. 3670. Springer, Berlin, Heidelberg, 257–271.

Bernier, Y. 2001. Latency compensating methods in client/server in-game protocol design and optimization. In *Proceedings of Game Developers Conference*.

Bocchi, L., Laneve, C., and Zavattaro, G. 2003. A calculus for long-running transactions. In *FMOODS*. Lecture Notes in Computer Science, vol. 2884. Springer, Berlin, Heidelberg, 124–138.

Bolie, J., Cardella, M., Blanvalet, S., Juric, M., Chandran, S. C. P., Coene, Y., and Geminiuc, K. 2006. *BPEL Cookbook: Best Practices for SOA-based integration and composite applications development: Ten practical real-world case studies combining business process management and web services orchestration*. Packt Publishing, Birmingham, UK.

Bruni, R., Butler, M. J., Ferreira, C., Hoare, C. A. R., Melgratti, H. C., and Montanari, U. 2005. Comparing two approaches to compensable flow composition. In *Concurrency Theory, 16th International Conference (CONCUR)*. Lecture Notes in Computer Science, vol. 3653. Springer-Verlag, London, UK, 383–397.

Bruni, R., Melgratti, H., and Montanari, U. 2005. Theoretical foundations for compensations in flow composition languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 209–220.

Bruni, R., Melgratti, H. C., and Montanari, U. 2004. Nested commits for mobile calculi: Extending join. In *IFIP TCS*. Kluwer, Hingham, MA, USA, 563–576.

Butler, M. J. and Ferreira, C. 2000. A process compensation language. In *Proceedings of the Second International Conference on Integrated Formal Methods*. LNCS. Springer-Verlag.

Butler, M. J. and Ferreira, C. 2004. An operational semantics for stac, a language for modelling long-running business transactions. In *COORDINATION*. Lecture Notes in Computer Science, vol. 2949. Springer, Berlin, Heidelberg, 87–104.

Butler, M. J., Ferreira, C., and Ng, M. Y. 2005. Precise modelling of compensating business transactions and its application to BPEL. *J. UCS 11,* 5, 712–743.

Butler, M. J., Hoare, C. A. R., and Ferreira, C. 2004. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 133–150.

Butler, M. J. and Ripon, S. 2005. Executable semantics for compensating csp. In *EPEW/WS-FM*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 243–256.

Caires, L., Ferreira, C., and Vieira, H. T. 2008. A process calculus analysis of compensations. In *Trustworthy Global Computing (TGC)*. Lecture Notes in Computer Science, vol. 5474. Springer, Berlin, Heidelberg, 87–103.

Chessell, M., Griffin, C., Vines, D., Butler, M., Ferreira, C., and Henderson, P. 2002. Extending the concept of transaction compensation. *IBM Syst. J. 41,* 4, 743–758.

Coleman, J. W. 2004. Features of BPEL modelled via structural operational semantics. MPhil Thesis, University of Newcastle.

Colombo, C. and Pace, G. J. 2011. A compensating transaction example in twelve notations. Tech. rep., Department of Computer Science, University of Malta. Technical Report CS2011-01.

Colombo, C., Pace, G. J., and Abela, P. 2010. Compensation-aware runtime monitoring. In *Runtime Verification - First International Conference, (RV)*. Lecture Notes in Computer Science, vol. 6418. Springer, Berlin, Heidelberg, 214–228.

Danos, V. and Krivine, J. 2005. *Transactions in RCCS*. Springer-Verlag, London, UK, 398–412.

Davies, Jr., C. T. 1973. Recovery semantics for a db/dc system. In *Proceedings of the ACM annual conference*. ACM '73. ACM, New York, NY, USA, 136–141.

De Vries, E., Koutavas, V., and Hennessy, M. 2010a. Communicating transactions. In *Proceedings of the 21st international conference on Concurrency theory*. CONCUR'10. Springer-Verlag, Berlin, Heidelberg, 569–583.

De Vries, E., Koutavas, V., and Hennessy, M. 2010b. Liveness of communicating transactions. In *Proceedings of the 8th Asian conference on Programming languages and systems*. APLAS'10. Springer-Verlag, Berlin, Heidelberg, 392–407.

Eisentraut, C. and Spieler, D. 2008. Fault, compensation and termination in WS-BPEL 2.0 - a comparative analysis. In *Web Services and Formal Methods (WS-FM)*. Lecture Notes in Computer Science, vol. 5387. Springer, Berlin, Heidelberg, 107–126.

Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 624–633.

Fahland, D. 2005. Complete abstract operational semantics for the web service business process execution language. Informatik-Berichte 190, Humboldt-Universität zu Berlin.

Fahland, D. and Reisig, W. 2005. ASM-based semantics for BPEL: The negative control flow. In *12th International Workshop on Abstract State Machines, ASM 2005*. 131–152.

Farahbod, R. 2004. Extending and refining an abstract operational semantics of the web services architecture for the business process execution language. M.S. thesis, School of Computing Science, Simon Fraser University.

Farahbod, R., Glässer, U., and Vajihollahi, M. 2005. A formal semantics for the business process execution language for web services. In *Web Services and Model-Driven Enterprise Information Services (WSMDEIS 2005)*. INSTICC Press, Setubal, Portugal, 122–133.

Ferrara, A. 2004a. Web services: a process algebra approach. In *Service-Oriented Computing - ICSOC 2004*. ACM, New York, NY, USA, 242–251.

Ferrara, A. 2004b. Web services: a process algebra approach. Tech. rep., Università di Roma "La Sapienza", Italy.

Field, J. and Varela, C. A. 2005. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. *SIGPLAN Not. 40*, 195–208.

Fischer, J. and Majumdar, R. 2007. Ensuring consistency in long running transactions. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ASE '07. ACM, New York, NY, USA, 54–63.

Foster, H. 2006. A rigorous approach to engineering web service compositions. Ph.D. thesis, University Of London.

Foster, H., Uchitel, S., Magee, J., and Kramer, J. 2006. LTSA-WS: a tool for model-based verification of web service compositions and choreography. In *28th International Conference on Software Engineering (ICSE 2006)*. ACM, New York, NY, USA, 771–774.

Fournet, C. and Gonthier, G. 1996. The reflexive CHAM and the join-calculus. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 372–385.

Fu, X., Bultan, T., and Su, J. 2004. Analysis of interacting BPEL web services. In *international conference on World Wide Web, WWW 2004*. ACM, New York, NY, USA, 621–630.

Garcia-Molina, H. and Salem, K. 1987. Sagas. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 249–259.

Gray, J. 1981. The transaction concept: Virtues and limitations (invited paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, Los Alamitos, CA, USA, 144–154.

Greenfield, P., Fekete, A., Jang, J., and Kuo, D. 2003. Compensation is not enough. In *7th International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE Computer Society, Washington, DC, USA, 232–239.

Guidi, C., Lanese, I., Montesi, F., and Zavattaro, G. 2008. On the interplay between fault handling and request-response service invocations. In *8th International Conference on Application of Concurrency to System Design (ACSD)*. IEEE, Washington, DC, USA, 190–198.

Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., and Zavattaro, G. 2006. SOCK: A calculus for service oriented computing. In *Service-Oriented Computing (ICSOC)*. Lecture Notes in Computer Science, vol. 4294. Springer, Berlin, Heidelberg, 327–338.

Haddad, S., Melliti, T., Moreaux, P., and Rampacek, S. 2004. Modelling web services interoperability. In *International Conference on Enterprise Information Systems (ICEIS)*. 287–295.

Haddad, S., Moreaux, P., and Rampacek, S. 2008. A formal semantics and a client synthesis for a bpel service. In *Enterprise Information Systems, 8th International Conference (ICEIS)*. Lecture Notes in Business Information Processing, vol. 3. Springer, 388–401.

Hamadi, R. and Benatallah, B. 2003. A petri net-based model for web service composition. In *ADC '03: Proceedings of the 14th Australasian database conference*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 191–200.

He, Y., Zhao, L., Wu, Z., and Li, F. 2008. Formal modeling of transaction behavior in WS-BPEL. In *International Conference on Computer Science and Software Engineering, (CSSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 490–494.

Hennessy, M. 2007. *A Distributed Pi-Calculus*. Cambridge University Press, New York, NY, USA.

Hoare, C. 1985. *Communicating Sequential Processes*. Prentice-Hall, New Jersey, USA.

Hoare, C. A. R., Hayes, I. J., Jifeng, H., Morgan, C. C., Roscoe, A. W., Sanders, J. W., Sorensen, I. H., Spivey, J. M., and Sufrin, B. A. 1987. Laws of programming. *Commun. ACM 30,* 8, 672–686.

Jifeng, H. 2007. Formal methods and hybrid real-time systems. Springer-Verlag, Berlin, Heidelberg, Chapter Compensable programs, 349–363.

Kazhamiakin, R. and Pistore, M. 2005. A parametric communication model for the verification of bpel4ws compositions. In *Formal Techniques for Computer Systems and Business Processes, European Performance Engineering Workshop, EPEW 2005 and International Workshop on Web Services and Formal Methods, WS-FM 2005*. Lecture Notes in Computer Science, vol. 3670. Springer, Berlin, Heidelberg, 318–332.

Kopp, O., Mietzner, R., and Leymann, F. 2009. The influence of an external transaction on a bpel scope. In *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems: Part I*. OTM '09. Springer-Verlag, Berlin, Heidelberg, 381–388.

Korth, H. F., Levy, E., and Silberschatz, A. 1990. A formal approach to recovery by compensating transactions. In *16th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, San Francisco, CA, USA, 95–106.

Koshkina, M. and van Breugel, F. 2004. Modelling and verifying web service orchestration by means of the concurrency workbench. *ACM SIGSOFT Software Engineering Notes 29,* 5, 1–10.

Lanese, I., Vaz, C., and Ferreira, C. 2010. On the expressive power of primitives for compensation handling. In *Programming Languages and Systems, 19th European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 6012. Springer, Berlin, Heidelberg, 366–386.

Lanese, I. and Zavattaro, G. 2009. Programming sagas in SOCK. In *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009*. IEEE Computer Society, Los Alamitos, CA, USA, 189–198.

Laneve, C. and Zavattaro, G. 2005. Foundations of web transactions. In *FoSSaCS*. Lecture Notes in Computer Science, vol. 3441. Springer, Berlin, Heidelberg, 282–298.

Lanotte, R., Maggiolo-Schettini, A., Milazzo, P., and Troina, A. 2006. Modeling long-running transactions with communicating hierarchical timed automata. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*. Lecture Notes in Computer Science, vol. 4037. Springer, 108–122.

Lanotte, R., Maggiolo-Schettini, A., Milazzo, P., and Troina, A. 2008. Design and verification of long-running transactions in a timed framework. *Sci. Comput. Program. 73,* 76–94.

Lapadula, A., Pugliese, R., and Tiezzi, F. 2007a. C-clock-ws: A timed service-oriented calculus. In *Theoretical Aspects of Computing (ICTAC)*. LNCS, vol. 4711. Springer, 275–290.

LAPADULA, A., PUGLIESE, R., AND TIEZZI, F. 2007b. A calculus for orchestration of web services. In *Programming Languages and Systems (ESOP)*. Lecture Notes in Computer Science, vol. 4421. Springer, Berlin, Heidelberg, 33–47.

LAPADULA, A., PUGLIESE, R., AND TIEZZI, F. 2008a. A calculus for orchestration of web services. Tech. rep., Dipartimento di Sistemi e Informatica, Univ. Firenze. http://rap.dsi.unifi.it/cows.

LAPADULA, A., PUGLIESE, R., AND TIEZZI, F. 2008b. A formal account of WS-BPEL. In *Coordination Models and Languages (COORDINATION)*. Lecture Notes in Computer Science, vol. 5052. Springer, Berlin, Heidelberg, 199–215.

LEYMANN, F. 2001. WSFL — web services flow language. IBM Software Group.

LI, J., ZHU, H., AND HE, J. 2007a. Algebraic semantics for compensable transactions. In *Theoretical Aspects of Computing (ICTAC)*. Lecture Notes in Computer Science, vol. 4711. Springer, Berlin, Heidelberg, 306–321.

LI, J., ZHU, H., AND HE, J. 2008. Specifying and verifying web transactions. In *Formal Techniques for Networked and Distributed Systems (FORTE)*. Lecture Notes in Computer Science, vol. 5048. Springer, Berlin, Heidelberg, 149–168.

LI, J., ZHU, H., PU, G., AND HE, J. 2007b. Looking into compensable transactions. *Software Engineering Workshop, Annual IEEE/NASA Goddard 0*, 154–166.

LOHMANN, N. 2007. A feature-complete petri net semantics for WS-BPEL 2.0. In *Web Services and Formal Methods (WS-FM)*. Lecture Notes in Computer Science, vol. 4937. Springer, Berlin, Heidelberg, 77–91.

LUCCHI, R. AND MAZZARA, M. 2007. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming 70,* 1 (January), 96–118.

MATEESCU, R. AND RAMPACEK, S. 2008. Formal modeling and discrete-time analysis of bpel web services. In *Advances in Enterprise Engineering I and 4th International Workshop EOMAS*. Lecture Notes in Business Information Processing, vol. 10. Springer, Berlin, Heidelberg, 179–193.

MAUVE, M., VOGEL, J., HILT, V., AND EFFELSBERG, W. 2004. Local-lag and timewarp: providing consistency for replicated continuous applications. *IEEE Transactions on Multimedia 6,* 1, 47–57.

MAZZARA, M. AND GOVONI, S. 2005. A case study of web services orchestration. In *Coordination Models and Languages, 7th International Conference, COORDINATION 2005*. Lecture Notes in Computer Science, vol. 3454. Springer, Berlin, Heidelberg, 1–16.

MAZZARA, M. AND LANESE, I. 2006. Towards a unifying theory for web services composition. In *Web Services and Formal Methods, Third International Workshop, WS-FM 2006*. Lecture Notes in Computer Science, vol. 4184. Springer, Berlin, Heidelberg, 257–272.

MELLIAR-SMITH, P. M. AND RANDELL, B. 1977. Software reliability: The role of programmed exception handling. *SIGSOFT Softw. Eng. Notes 2*, 95–100.

MILNER, R. 1999. *Communicating and Mobile Systems: The π Calculus*. Cambridge University Press, Cambridge, England.

NAKAJIMA, S. 2006. Model-checking behavioral specification of BPEL applications. *Electr. Notes Theor. Comput. Sci. 151,* 2, 89–105.

OUYANG, C., VERBEEK, E., VAN DER AALST, W. M. P., BREUTEL, S., DUMAS, M., AND TER HOFSTEDE, A. H. M. 2007. Formal semantics and analysis of control flow in WS-BPEL. *Sci. Comput. Program. 67,* 2-3, 162–198.

PISTORE, M., ROVERI, M., AND BUSETTA, P. 2004. Requirements-driven verification of web services. *Electr. Notes Theor. Comput. Sci. 105*, 95–108.

PISTORE, M., TRAVERSO, P., BERTOLI, P., AND MARCONI, A. 2005. Automated synthesis of composite bpel4ws web services. In *IEEE International Conference on Web Services (ICWS)*. IEEE Computer Society, Los Alamitos, CA, USA, 293–301.

PU, G., ZHAO, X., WANG, S., AND QIU, Z. 2006. Towards the semantics and verification of BPEL4WS. *Electr. Notes Theor. Comput. Sci. 151,* 2, 33–52.

PU, G., ZHU, H., QIU, Z., WANG, S., ZHAO, X., AND HE, J. 2006. Theoretical foundations of scope-based compensable flow language for web service. In *Formal Methods for Open Object-Based*

Distributed Systems (FMOODS). Lecture Notes in Computer Science, vol. 4037. Springer, Berlin, Heidelberg, 251–266.

Randell, B. 1975. System structure for software fault tolerance. *IEEE Trans. Software Eng. 1,* 2, 221–232.

Randell, B., Lee, P., and Treleaven, P. C. 1978. Reliability issues in computing system design. *ACM Comput. Surv. 10*, 123–165.

Sarang, P., Juric, M., and Mathew, B. 2006. *Business Process Execution Language for Web Services BPEL and BPEL4WS 2nd Edition*. Packt Publishing, Birmingham, UK.

Sauter, P. and Melzer, I. 2005. A comparison of ws-businessactivity and bpel4ws long-running transaction. In *Kommunikation in Verteilten Systemen (KiVS)*. Informatik Aktuell. Springer, Berlin, Heidelberg, 115–125.

Thatte, S. 2001. XLANG — web services for business process design. Microsoft Corporation.

van den Heuvel, W.-J. 2008. Survey on business process management. Tech. rep.

van der Aalst, W. M. P. 1998. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers 8,* 1, 21–66.

Van Der Aalst, W. M. P., Hofstede, A. H. M. T., and Weske, M. 2003. Business process management: a survey. In *Proceedings of the 2003 international conference on Business process management*. BPM'03. Springer-Verlag, Berlin, Heidelberg, 1–12.

Vaz, C., Ferreira, C., and Ravara, A. 2009. Dynamic recovering of long running transactions. *Trustworthy Global Computing: 4th International Symposium, TGC 2008, Barcelona, Spain, November 3-4, 2008, Revised Selected Papers 5474*, 201–215.

Verbeek, H. M. W. and van der Aalst, W. 2005. Analyzing BPEL processes using petri nets. In *Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*. 59–78.

Verhofstad, J. S. M. 1978. Recovery techniques for database systems. *ACM Comput. Surv. 10*, 167–195.

Viroli, M. 2004. Towards a formal foundation to orchestration languages. *Electr. Notes Theor. Comput. Sci. 105*, 51–71.

Wang, T., Vonk, J., Kratz, B., and Grefen, P. 2008. A survey on the history of transaction management: from flat to grid transactions. *Distrib. Parallel Databases 23*, 235–270.

Weidlich, M., Decker, G., and Weske, M. 2007. Efficient analysis of BPEL 2.0 processes using p-calculus. In *Proceedings of The 2nd IEEE Asia-Pacific Services Computing Conference, APSCC 2007*. IEEE, Washington, DC, USA, 266–274.

Wombacher, A., Fankhauser, P., and Neuhold, E. J. 2004. Transforming BPEL into annotated deterministic finite state automata for service discovery. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*. IEEE Computer Society, Washington, DC, USA, 316–323.

Yang, Y., Tan, Q., Xiao, Y., Liu, F., and Yu, J. 2006. Transform BPEL workflow into hierarchical CP-nets to make tool support for verification. In *Frontiers of WWW Research and Development (APWeb)*. LNCS, vol. 3841. Springer, Berlin, Heidelberg, 275–284.

yun Long, H. and shi Li, J. 2009. A process algebra approach of BPEL4WS. *Information and Computing Science 4,* 2, 93–98.

Received Month Year; revised Month Year; accepted Month Year