When Things Go Wrong: Interrupting Conversations

Juliana Bowles[1] and Sotiris Moschoyiannis[2]

[1] School of Computer Science, University of St Andrews
Jack Cole Building, North Haugh, St Andrews KY16 9SX, UK
jkfb@st-andrews.ac.uk
[2] Department of Computing, University of Surrey
Guildford, Surrey GU2 7XH, UK
s.moschoyiannis@surrey.ac.uk

Abstract. This paper presents a true-concurrent approach to formalising integration of Small-to-Medium Enterprises (SMEs) with Web services. Our approach formalises common notions in service-oriented computing such as *conversations* (interactions between clients and web services), *multi-party conversations* (interactions between multiple web services) and *coordination protocols*, which are central in a transactional environment. In particular, we capture long-running transactions with recovery and compensation mechanisms for the underlying services in order to ensure that a transaction either commits or is successfully compensated for.

1 Introduction

Business transactions between open communities of SMEs have been highlighted as a key area within the emerging Digital Economy [1]. A business transaction can be a simple usage of a service (rare in Business-to-Business (B2B) relationships) or a mixture of different levels of composition of services from various providers. Within the database community the conventional definition of a transaction [2] is based on ACID (Atomicity, Consistency, Isolation, Durability) properties. However, in advanced distributed applications these properties often present considerable limitations and in many cases are in fact undesirable.

Business transactions typically involve interactions and coordination between multiple partners. The specification of a transaction comprises a number of *sub-transactions* or *activities* which involve the execution of several underlying services from different providers, some of which take minutes, hours or even days to complete - hence the term *long-running* transaction. Indeed a wide range of B2B scenarios correspond to long-lived business activities and may have a long execution period.

It is often the case that internal activities need to share results before the termination of the transaction (commit). More generally, dependencies may arise between activities inside a transaction due to the required ordering on the service invocations or, simply, due the sharing of data [3]. Further, many B2B scenarios require a transaction to release some results to another transaction, before it commits. That is to say, dependencies may exist across transactions due to the need for releasing *partial results* outside a transaction. Failure to accommodate this may lead to unacceptable delays in related transactions and, even worse, leave a service provider open to denial of service attacks (as data may be locked indefinitely in a non-terminating transaction). Thus, the Isolation property must be relaxed and this poses further challenges with regard to keeping track of the dependencies that arise between the corresponding service executions.

The multi-service nature of transactions makes Service-Oriented Computing (SOC) [4,5], whose goal is to enable applications from different providers to be offered as services that can be used, composed and coordinated in a loosely-coupled manner, the prevalent computing paradigm in a transactional environment. The actual architectural approach of SOC, called SOA, is a way of reorganising software applications and supporting infrastructure into an interconnected set of services, each accessible through standard interfaces and messaging (coordination) protocols.

In this paper we are concerned with modelling long-running transactions, and in particular the coordination of the underlying service executions. The challenges in exploiting the promise of SOA in a transactional environment requires a thorough understanding of the dependencies that arise from the complex interactions between services and the valid sequences of service invocations.

We have seen that business transactions involve interactions between multiple service providers which need to be orchestrated. Business transactions also need to deal with faults that arise at any stage of execution. In fact, a long-running transaction should either complete successfully (commit) or not take place at all. This means that there must be some mechanism/procedure in place, which in the event of a failure (service unavailable, network/platform disconnection, etc.) makes it possible to undo parts of the transaction that have actually happened so far. Therefore, in addition to formalising conversations in a transactional environment we also provide constructs for compensating (earlier parts of) conversations, in case some failure later on makes this necessary.

Our approach uses a true-concurrent model (labelled prime event structures) which can be obtained directly from UML 2.0 sequence diagrams describing multi-party conversations. The model is extended to capture possible faults that result in interrupting a conversation and taking compensating action. We show how our formal framework orchestrates conversations and associated compensations in order to achieve the desired effect - a transaction either commits or is successfully compensated for.

This paper is structured as follows. Section 2 outlines the use of sequence diagrams for modelling conversations within a transaction. Section 3 describes the formal model used for long-running transactions, and Section 4 extends it to describe interruptions and compensation whereby we distinguish between *rollback with memory* and *forgetful rollback*. A brief discussion on related work is included in Section 5 and the paper finishes with some ideas for future work given in Section 6.

2 Example

We consider a simple multi-party conversation within a long-running transaction and show how it can be modelled using UML2.0 [6]. The example has been simplified somewhat but still contains enough complexity to illustrate the key ideas behind our formal modelling approach.

We use sequence diagrams to represent conversations within long-running (multi-party service) transactions. We only model the participants of the conversations (web services) without explicitly representing a distinction between the initiator and a participant of a transaction. In sequence diagrams, we can indicate any exchange with a client or coordinator using *gates*. Gates correspond to the environment which we are not interested in capturing explicitly. Notice that this means that we are deliberately ignoring a choice of central or distributed coordination in our model. Indeed, our formalism works with both architectures. The choice of architecture very much depends on the target application - for example, in a digital business ecosystem involving SMEs a fully distributed solution may be most appropriate since the use of a centralised coordinator would violate local autonomy, as is the case with existing transaction models (e.g., BTP[13], WS-Tx[12] are briefly discussed in Section 5), and this is a barrier for the adoption of SOA by SMEs.

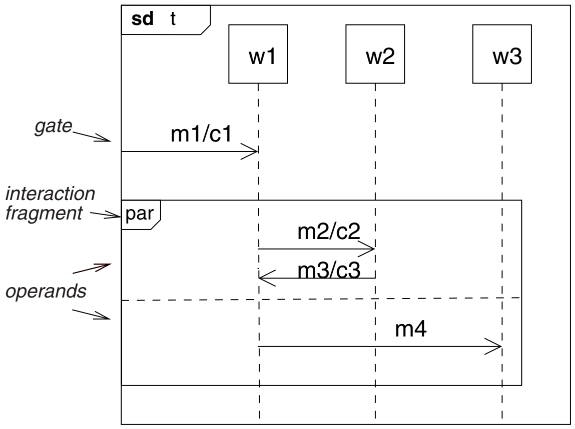

Fig. 1. A multi-party conversation

Fig. 1 shows three participants and the messages exchanged between them for transaction `t`. All messages are exchanged asynchronously. The `par` fragment indicates that the operands are executed in paralell. If a particular operation invocation needs to be compensated for in a specific way this is given by the application developer and is written after the name of the operation. For instance, the construct `m1/c1` indicates that `m1` is the operation being invoked and `c1` is the corresponding compensation. The compensation for an operation invocation can be complex and impose a sequence of message exchanges between services and the environment. For example, a lock mechanism might be required to ensure consistency of data used in the conversation during recovery (e.g. see [3]). If this is the case, `c1` can itself be represented by a sequence diagram with the same name (and similarly for all compensations).

We use sequence diagrams to model conversations showing only protocol-specific message exchanges (e.g., `m1`, `m2`, and so on, in Fig. 1) and ignore initial activation and registration exchanges to initiate the execution of a conversation. However, within the duration of a long-running transaction there are other messages that can be sent between the participating web services and a coordinator. These messages are sent to indicate the status of the execution of the transaction and allow for compensating action to be taken whenever necessary. It may be instructive to note that in a distributed solution, each service provider will have its own coordinator that is responsible for the services it offers and has knowledge of their dependencies (on other coordinators' services that execute immediately before or after its own). An overview of a transaction model with local coordination can be found in [7]. In a transaction model with centralised coordination, services from each service provider communicate through the central coordinator which is typically controlled by the network provider.

Messages used for providing a transaction processing system with compensating capability include `faulted`, `compensate` and `forget`. A web service may fail the execution of its invoked operation in which case it sends a message `faulted` to the coordinator (or, its coordinator). All other participants need to be informed in the next step as they will need to `compensate` or `forget` their parts of the execution of the conversation so far. If a web service participating in the conversation receives a message `compensate`, then the normal flow of messages should stop immediately and all its previous actions (effects of previous message exchanges) within the conversation need to be undone. This compensation does not necessarily leave the web service in exactly the same state it had at the beginning of the conversation (we call it *rollback with memory*). By contrast, if a participant receives a `forget` message, again the normal flow of messages is interrupted abruptly, but this time the participating service returns to the same state as in the beginning of the conversation and all its previous actions are simply ignored (we call it *forgetful rollback*).

We do not represent these additional messages when modelling the (multi-party) conversation in Fig. 1 as it would complicate the model unnecessarily. All possible faults and consequences are, however, considered in our formal model as we will see in Section 4.

One example of a possible faulted scenario leading to the abortion of the long-running transaction of our example is given in Fig. 2. In this case, `w3` fails whilst executing `m4` and sends a `faulted` message to the corresponding coordinator. This leads to two further messages being sent from the coordinator, namely a `forget` to `w3` and a `compensate` to `w1` (the later one will possibly trigger a further `compensate` message to `w2` depending on the state of the execution of the interaction in the first operand). Notice that in this particular scenario the

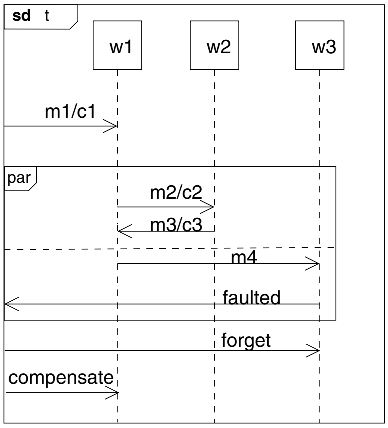

Fig. 2. A multi-party conversation with faults

`faulted` message is sent in parallel to the conversation between `w1` and `w2` and it is therefore possible that this sub-interaction has not happened and does not need to be compensated for. This can be made explicit in our formal model of conversations, which is described in the following sections.

3 The Model

We have used labelled event structures as an underlying model for sequence diagrams in UML 2.0[8,9]. In this paper, we use labelled event structures to capture conversations and coordination protocols.

3.1 Event Structures: Basic Notions

We recall some basic notions on the model we use, namely *labelled prime event structures* [10].

Prime event structures, or event structures for short, allow the description of distributed computations as event occurrences together with relations for expressing causal dependency and nondeterminism. The first relation is called *causality*, and the second *conflict*. The causality relation implies a (partial) order among event occurrences, while the conflict relation expresses how the occurrence of certain events excludes the occurrence of others. Consider the following definition of event structures.

Event Structure. An *event structure* is a triple $E = (Ev, \rightarrow^*, \#)$ where Ev is a set of events and $\rightarrow^*, \# \subseteq Ev \times Ev$ are binary relations called *causality* and *conflict*, respectively. Causality $\rightarrow^*$ is a partial order. Conflict $\#$ is symmetric and irreflexive, and propagates over causality, i.e., $e\#e^{'} \wedge e^{'} \rightarrow^* e^{''} \Rightarrow e\#e^{''}$ for all $e, e^{'}, e^{''} \in Ev$. Two events $e, e^{'} \in Ev$ are *concurrent*, $e \; co \; e^{'}$ iff $\neg(e \rightarrow^* e^{'} \vee e^{'} \rightarrow^* e \vee e\#e^{'})$.

From the two relations defined on the set of events, a further relation is derived, namely the *concurrency* relation *co*. As stated, two events are concurrent if and only if they are completely unrelated, i.e., neither related by causality nor by conflict.

In our approach to inter-object behaviour specification, we will consider a restriction of event structures sometimes referred to as *discrete* event structures. An event structure is said to be *discrete* if the set of previous occurrences of an event in the structure is finite.

Discrete Event Structure. Let $E = (Ev, \rightarrow^*, \#)$ be an event structure. E is a *discrete event structure* iff for each event $e \in Ev$, the *local configuration* of e given by $\downarrow e = \{e^{'} \mid e^{'} \rightarrow^* e\}$ is finite.

The finiteness assumption of the so-called local configuration is motivated by the fact that system computations always have a starting point, which means that any event in a computation can only have finitely many previous occurrences.

Consequently, we are able to talk about immediate causality in such structures. Two events e and $e^{'}$ are related by *immediate* causality if there are no other event occurrences in between. Formally, if $\forall_{e^{''} \in Ev}(e \rightarrow^* e^{''} \rightarrow^* e^{'} \Rightarrow (e^{''} = e \vee e^{''} = e^{'}))$ holds. If $e \rightarrow^* e^{'}$ are related by immediate causality then e is said to be an *immediate predecessor* of $e^{'}$ and $e^{'}$ is said to be an *immediate successor* of e. We may write $e \rightarrow e^{'}$ instead of $e \rightarrow^* e^{'}$ to denote immediate causality. Furthermore, we also use the notation $e \rightarrow^+ e^{'}$ whenever $e \rightarrow^* e^{'}$ and $e \neq e^{'}$.

Hereafter, we only consider discrete event structures.

Configuration. Let $E = (Ev, \rightarrow^*, \#)$ be an event structure and $C \subseteq Ev$. C is a *configuration* in E iff it is both (1) conflict free: for all $e, e^{'} \in C$, $\neg(e \# e^{'})$, and (2) downwards closed: for any $e \in C$ and $e^{'} \in Ev$, if $e^{'} \rightarrow^* e$ then $e^{'} \in C$. A maximal configuration denotes a run. A run is sometimes called life cycle.

Finally, in order to use event structures to provide a denotational semantics to languages, it is necessary to link the event structures to the language they are supposed to describe. This is achieved by attaching a labelling function to the set of events. A generic labelling function is as defined next.

Labelling Function. Let $E = (Ev, \rightarrow^*, \#)$ be an event structure, and L be an arbitrary set. A *labelling function* for E is a total function $l : Ev \rightarrow L$ mapping each event into an element of the set L.

An event structure together with a labelling function defines a so-called labelled event structure.

Labelled Event Structure. Let $E = (Ev, \rightarrow^*, \#)$ be an event structure, L be a set of labels, and $l : Ev \rightarrow L$ be a labelling function for E. A *labelled event structure* is a pair $(E, l : Ev \rightarrow L)$.

Usually, events model the occurrence of actions, and a possible labelling function maps each event into an action symbol or a set of action symbols. In this paper, we use labelled event structures in the context of long-running transactions. As we will see, in our case, and since we use UML models of conversations within a transaction, the labelling function indicates whether an event represents sending or receiving a message, the beginning or end of an interaction fragment.

3.2 Event Structures for Transactions

In this paper, we use sequence diagrams to model transactions. We first need to understand how to obtain a labelled event structure for the sequence diagram (without faults), and then can move to show how compensation mechanisms can be integrated. In [8] we have shown how labelled event structures can be used to provide a model for sequence diagrams. Here we only provide the general idea.

To obtain the corresponding event structure model, we want to associate events to the *locations* of the diagram and determine the relations between those events to reflect the meaning of the diagram. Fig. 3 shows the relation between the locations in a simple sequence diagram (which could for example correspond to the interaction between a client and a service) and the corresponding event structure model (where we depict immediate causality). Asynchronous communication is captured as immediate causality as well, hence the two locations for sending and receiving message `m2/c2` are captured by two consecutive events. The labels become clearer later when we define the labelling function used.

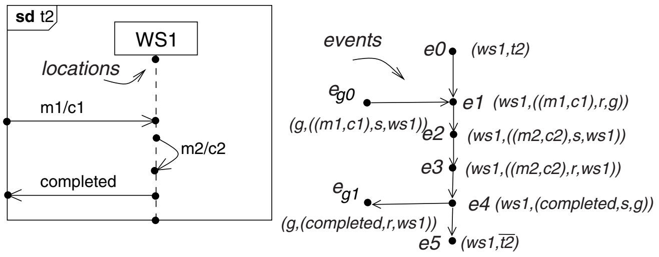

Fig. 3. A simple sequence diagram and its corresponding model

However, for more complex diagrams with fragments the correspondence between locations and events is not always so obvious.

The locations within different operands of an **alt** fragment are naturally associated to events in conflict. However, the end location of an **alt** fragment is problematic. If it corresponded to one event then this event would be in conflict with itself due to the fact that in a prime event structure conflict propagates over causality. This would, however, lead to an invalid model since conflict is irreflexive. We are therefore forced to copy events for locations marking the end of **alt** fragments, as well as for all locations that follow. Events associated to locations that fall within a **par** fragment are concurrent. Synchronous communication is denoted by a shared event whereas asynchronous communication is captured by immediate causality between the send event and receive event.

As mentioned earlier, for representing sequence diagrams we use a labelling function to indicate whether an event represents sending or receiving a message, a condition, the beginning or end of an interaction fragment. The only considered fragments in this paper is **par** and **alt**. For more details on further fragments please see [8,9].

Let D be a set of diagram names corresponding to conversations, transactions or the detailed description of compensations, W_t be the set of web services participating in the interaction described by $t \in D$, and g denote a gate or the environment (i.e., a client or coordinator) with $g \in W_t$ for all $t \in D$. Let $C_t \subset D$ be the set of compensations associated to messages in $t \in D$. Let $F_t = \{t, par, alt\} \cup C_t$ with $t \in D$ and $\overline{F_t} = \{\overline{t}, \overline{par}, \overline{alt}\} \cup \overline{C_t}$ where $\overline{C_t} = \{\overline{c} \mid c \in C_t\}$. We use par (or $\overline{par}$) as a label of an event associated to the location marking the beginning (or end) of a **par** fragment. In particular, events associated to initial (or end) locations of a diagram t have labels t (or $\overline{t}$). Similarly for compensation diagrams (i.e., diagrams representing the behaviour of compensations). Let M_{web} be the set of messages exchanged between web services and/or the environment, and M_{env} be a predefined set of messages exchanged only between a service and the environment. M_{env} consists of messages such as `exited`, `completed`, `faulted` (sent by a web service to the environment), and `close`, `complete`, `compensate` and `forget` (sent by the environment to a web service). Let $Mes_t = M_{web_t} \times (C_t \cup \{-\}) \cup M_{env_t}$ be the complete set of message labels for $t \in D$. The labelling function for diagram t is a total function defined over events as follows:

$$\mu_t : Ev \rightarrow W_t \times (Mes_t \times \{s, r\} \times W_t \cup F_t \cup \overline{F_t})$$

Each event is associated to a unique web service involved in t and can denote sending a message, receiving a message or indicating the beginning/end of a fragment. This labelling function has been simplified to capture only asynchronous messages as this is the only form of communication we use in this paper. In the example of Fig. 3, event e_1 has label $(ws1, ((m_1, c_1), r, g))$ indicating it belongs to service $ws1$ and corresponds to the receipt of message m_1 with compensation c_1 from g. Certain operation invocations may not have a compensation defined in case of failure, in which case the label is written $(s_1, ((m, -), r, s_2))$. An example of a label for predefined messages is $(ws1, (\mathit{completed}, s, g))$. We write $(\mu_t(e))_1$ to indicate the first projection of the label for e (e.g., associated service).

Finally, for $t \in D$, a model is a labelled event structure $M_t = (E_t, \mu_t)$.

4 Modelling Interruptions

In the previous section, we have seen how for a (multi-party) conversation of a long-running transaction captured as a sequence diagram, we can obtain the underlying formal model as a labelled event structure. In this section, we are going to see how the model can be extended to incorporate possible faults in transaction executions.

Recall the sequence diagram of Fig. 1 showing the interaction between three web services `w1`, `w2` and `w3`. The labelled event structure that models the behaviour represented in the sequence diagram is given by Fig. 4. This model only takes into account the correct behaviour of all parties involved in the interaction. However, any of the operations invoked (`m1`, `m2`, `m3` or `m4`) could possibly fail during execution, for example, as in the scenario of Fig. 2. If that happens, the corresponding service would need to inform its coordinator (in Fig. 2, `w3`

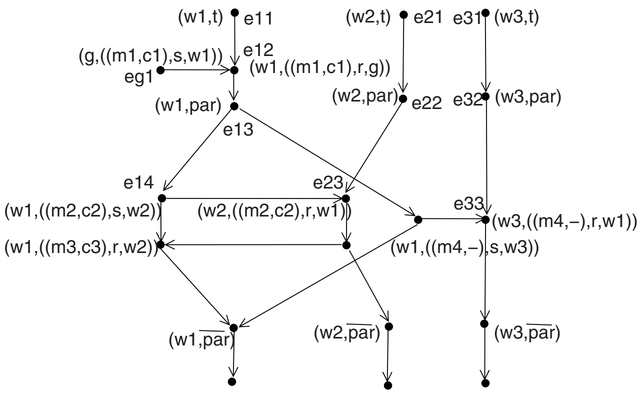

Fig. 4. LES for the multi-party conversation of Fig. 1

sends a message `faulted` to the environment) which would need to inform all parties of the correct compensation mechanism (in Fig. 2, the environment sends a message `forget` to `w3`, and `compensate` to `w1`). Consequently, we need to be able to represent the faults that can occur during an interaction and as well as their effects.

Initial Configuration. Let $M_t = (E_t, \mu_t)$ be a model for a transaction t. An initial configuration for M_t is $Q_0 = \{e \in Ev_t \mid \mu_t(e) = (w, t), w \in W_t\}$.

The initial configuration basically corresponds to the set of events associated to the initial location of every web service participating in the interaction. In Fig. 4, the initial configuration corresponds to the set of events $\{e_{11}, e_{21}, e_{31}\}$.

Immediate Configurations. Let $M_t = (E_t, \mu_t)$ be a model for a transaction t. For any two configurations $Q_1 \neq Q_2$ in E_t, we say that Q_1 and Q_2 are *immediate configurations* (or Q_2 is an *immediate postconfiguration* for Q_1, and Q_1 is an *immediate preconfiguration* of Q_2) iff (1) $Q_1 \subset Q_2$ and for any $e_1, e_2 \in Q_2 \setminus Q_1$ e_1 *co* e_2, and (2) if $(\mu_t(e))_1 = w$ for $w \in W_t$ and $e \in Q_2 \setminus Q_1$, then there is a maximal event $e^{'} \in Q_1$ such that $e^{'} \rightarrow e$ and $(\mu_t(e^{'}))_1 = w$.

A configuration can have more than one possible immediate postconfiguration due to conflict or the possible concurrency between events corresponding to different services (for example, events e_{22}, e_{32} and e_{g1}). From one configuration to the next we can add either a single event or a set of events in concurrency. If two or more concurrent events appear in the same immediate postconfiguration we say they occur *simultaneously*. Otherwise their occurrence is effectively being interleaved. The initial configuration in Fig. 4 has seven immediate postconfigurations given by $\{e_{g1}, e_{11}, e_{21}, e_{31}\}$, $\{e_{11}, e_{21}, e_{22}, e_{31}\}$, $\{e_{11}, e_{21}, e_{31}, e_{32}\}$, $\{e_{g1}, e_{11}, e_{21}, e_{22}, e_{31}\}$,$\{e_{g1}, e_{11}, e_{21}, e_{31}, e_{32}\}$, $\{e_{11}, e_{21}, e_{22}, e_{31}, e_{32}\}$ and finally $\{e_{g1}, e_{11}, e_{21}, e_{22}, e_{31}, e_{32}\}$. Notice that $\{e_{11}, e_{12}, e_{21}, e_{31}\}$ is not a configuration (it does not satisfy the condition of being downwards closed) and can thus not be an immediate postconfiguration for the initial configuration. For the same reason, we know that receiving a message can never happen before the corresponding send.

Configuration Path. Let $M_t = (E_t, \mu_t)$ be a model for a transaction t. A *configuration path* in M_t is a sequence of immediate configurations $Q_0 \cdot Q_1 \cdot Q_2 \cdots Q_n$ in E_t starting with the initial configuration and ending in a maximal configuration or run.

Given our definition of immediate configurations and provided there is concurrency in the model, we always have several paths starting from an initial configuration and leading to a maximal configuration. In our example of Fig. 4 we have several paths but only one maximal configuration and consequently all paths lead to the same final configuration. If a model has conflict (due to alternatives given as **alt** fragments in the sequence diagram) we have as many maximal configurations as there are alternatives.

Models M_t for transactions that have more than one maximal configuration can be useful for providing *forward recovery* - that is, including capability for completing the transaction following an alternative path of execution rather than aborting in the event of a failure in some service of the corresponding conversation. We return to this discussion in the concluding section of the paper.

The difference between two immediate configurations is given by a set of events which corresponds to the occurrence of some action, i.e., sending/receiving a message or entering/exiting an interaction fragment. This can be seen as a transition between the two configurations labelled by the set of occurring action(s).

Configuration Transitions. Let $M_t = (E_t, \mu_t)$ be a model for a transaction t, and Q_1, Q_2 be immediate configurations in E_t with Q_1 the preconfiguration of Q_2. Let $Q_2 \setminus Q_1 = \{e_1, e_2, \ldots, e_n\}$ and $\mu_t(e_i) = l_i$ with $1 \leq i \leq n$. A *transition* from Q_1 to Q_2 is labelled by $\langle l_1, l_2, \ldots, l_n\rangle$ and written $Q_1 \overset{\langle l_1, l_2 \ldots l_n\rangle}{\longrightarrow} Q_2$.

Consider the model of Fig. 4. The transition between $Q_0 = \{e_{11}, e_{21}, e_{31}\}$ and $Q_1 = \{e_{g1}, e_{11}, e_{21}, e_{31}\}$ is labelled by $\langle(g, ((m_1, c_1), s, w_1))\rangle$. To simplify the label notation, we sometimes write $(w, m/c!, q)$ to denote w sending message m/c to q, and $(q, m/c?, w)$ to denote q receiving message m/c from w. Labels denoting entering/exiting fragments are often omitted (written $\langle - \rangle$). A possible path in the model of Fig. 4 could be given by the following transitions:

$$Q_0 \overset{\langle(g,m_1/c_1!,w_1)\rangle}{\longrightarrow} Q_1 \overset{\langle(w1,m_1/c_1?,g)\rangle}{\longrightarrow} Q_2 \overset{\langle - \rangle}{\longrightarrow} Q_3 \overset{\langle(w1,m_4!,w_3)\rangle}{\longrightarrow} Q_4$$
$$Q_4 \overset{\langle(w_3,m_4?,w_1)\rangle}{\longrightarrow} Q_5 \overset{\langle(w_1,m_2/c_2!,w_2)\rangle}{\longrightarrow} Q_6 \overset{\langle(w_2,m_2/c_2?,w_1)\rangle}{\longrightarrow} Q_7$$
$$Q_7 \overset{\langle(w_2,m_3/c_3!,w_1)\rangle}{\longrightarrow} Q_8 \overset{\langle(w_1,m_3/c_3?,w_2)\rangle}{\longrightarrow} Q_9 \overset{\langle - \rangle}{\longrightarrow} Q_{10} \overset{\langle - \rangle}{\longrightarrow} E_t$$

We now want to add to the model possible faults that can happen in the execution of transactions. This corresponds to adding transitions labelled by the predefined messages seen earlier between a web service and the environment or vice versa (e.g., `faulted`, `compensate`, `forget`).

The execution of a message m for a web service w can only fail after m was invoked (the message was received), and before a subsequent message is sent by w. For the path in our example, m_4 could fail after configuration Q_5. The question now is what is the result of such a transition, namely what immediate postconfiguration(s) do we get for Q_5 that satisfy $Q_5 \overset{\langle(w_3,faulted!,g)\rangle}{\longrightarrow} Q_6' \overset{\langle(g,faulted?,w_3)\rangle}{\longrightarrow} Q_7'$. We assume that faulted messages have priority and thus the send is always followed by the receipt. Moreover, once a path is faulted no further normal transitions are allowed. Notice, that after a faulted message occurs the configurations are no longer configurations in M_t but in an extended model for t with fault events.

Faulted Transitions and Faulted Path. Let $M_t = (E_t, \mu_t)$ be a model for a transaction t, and $Q_0 \cdot Q_1 \cdot Q_2 \cdots Q_n$ in E_t be a configuration path in M_t. Let $Q_i \overset{\langle (w_2, m/c?, w_1)\rangle}{\longrightarrow} Q_{i+1}$ be a transition. The *faulted transitions* associated to message $(w_2, m/c?, w_1)$ at Q_{i+1} correspond to $Q_{i+1} \overset{\langle (w_2, faulted!, g)\rangle}{\longrightarrow} Q^{'}_{i+2} \overset{\langle (g, faulted?, w_2)\rangle}{\longrightarrow} Q^{'}_{i+3}$ where the new configurations are defined as follows $Q^{'}_{i+2} = Q_{i+1} \cup \{e_2 \mid e_2 \notin Q_{i+1}, \mu_t(e_2) = (w_2, (faulted, s, g))\}$ with $e_1 \rightarrow e_2$ where e_1 is the maximal event in Q_{i+1} with $(\mu_t(e_1))_1 = w_2$; and $Q^{'}_{i+3} = Q_{i+2} \cup \{e_3 \mid e_3 \notin Ev_t \cup \{e_2\}, \mu_t(e_3) = (g, (faulted, r, w_2))\}$ with $e_2 \rightarrow e_3$. The sequence $Q_0 \cdot Q_1 \cdot Q_2 \cdots Q_i \cdot Q_{i+1} \cdot Q^{'}_{i+2} Q^{'}_{i+3}$ is a *faulted path* for extended M_t.

Once a faulted message has been sent to and received by the environment (in effect the coordinator of the failed web service), the environment can respond with one or more forget or compensate. Similarly to faulted messages, sending and receiving a forget or compensate always happen successively. The definitions below reflect that a forget/compensate does not have to happen immediately after a faulted transition (there can be j pairs of configurations in between).

Forget Transitions. Let $M_t = (E_t, \mu_t)$ be a model for a transaction t, and $Q_0 \cdot Q_1 \cdot Q_2 \cdots Q_{i-1} \cdot Q_i \cdot Q_{i+1}$ be a faulted path for extended M_t where the faulted transitions are $Q_{i-1} \overset{\langle (w, faulted!, g)\rangle}{\longrightarrow} Q_i \overset{\langle (g, faulted?, w)\rangle}{\longrightarrow} Q_{i+1}$. The pair of *forget transitions* associated to the faulted transitions at Q_{i+1} correspond to $Q_{i+2j-1} \overset{\langle (g, forget!, w)\rangle}{\longrightarrow} Q_{i+2j} \overset{\langle (w, forget?, g)\rangle}{\longrightarrow} Q_{i+2j+1}$ where $j \in \mathbb{N}$ and the new configurations are defined as follows $Q_{i+2j} = Q_{i+2j-1} \cup \{e_2 \notin Q_{i+2j-1} \mid \mu_t(e_2) = (g, (forget, s, w))\}$ with $e_1 \rightarrow e_2$ where e_1 is an event in Q_{i+2j-1} with $\mu_t(e_1) = (g, (faulted, r, w))$. Let $O = \{w_1, \ldots, w_n \in W_t \mid \exists_{e_{w_k}, e^{'}_{w_k} \in Q_i} (\mu_t(e_{w_k}))_1 = w_k, (\mu_t(e^{'}_{w_k}))_1 = w$ and $e^{'}_{w_k} \rightarrow^{*} e_{w_k}$ for all $1 \leq k \leq n\}$. $Q_{i+2j+1} = Q_{i+2j} \cup \{p_1, \ldots, p_n \notin Q_{i+2j} \mid \mu_t(p_k) = (w_k, t), w_k \in O$ for all $1 \leq k \leq n\}$ and $e_2 \rightarrow p_k$ for all $1 \leq k \leq n$.

After a service w receives a forget message, all its previous executions are undone and it returns to the configuration it had at the beginning. The other services remain unaffected, unless a service $w^{'}$ received a request from w during the conversation ($w^{'} \in O$). If that is the case it also needs to reverse to its initial configuration. Further services that were not in (direct or indirect) interaction with w are unaffected and wait for a forget or compensate message from the environment. For space reasons, we omit the proof that the obtained configuration Q_{i+2j+1} after the forget transitions is a valid immediate configuration for Q_{i+2j}.

We now describe how to deal with compensation.

Compensate Transitions. Let $M_t = (E_t, \mu_t)$ be a model for a transaction t, and $Q_0 \cdot Q_1 \cdot Q_2 \cdots Q_{i-1} \cdot Q_i \cdot Q_{i+1}$ be a faulted path for extended M_t where the faulted transitions are $Q_{i-1} \overset{\langle (w1, faulted!, g)\rangle}{\longrightarrow} Q_i \overset{\langle (g, faulted?, w1)\rangle}{\longrightarrow} Q_{i+1}$. A pair of *compensate transitions* correspond to a sequence of configurations $Q_{i+2j} \cdot$

$Q_{i+2j+1} \cdot Q^{'}_{k_j} \cdots Q^{'}_{k_0}$ where $Q_{i+2j-1} \xrightarrow{\langle(g,compensate!,w2)\rangle} Q_{i+2j} \xrightarrow{\langle(w2,compensate?,g)\rangle} Q_{i+2j+1}$, $j \in \mathbb{N}$ and the new configurations are defined as follows:

– $Q_{i+2j} = Q_{i+2j-1} \cup \{e_2 \notin Q_{i+2j-1} \mid \mu_t(e_2) = (g,(compensate,s,w_2))\}$ with $e_1 \rightarrow^* e_2$ for $e_1 \in Q_{i+1}$ with $\mu_t(e_1) = (g,(faulted,r,w_1))$.
– $Q_{i+2j+1} = Q_{i+2j} \cup \{e_3 \notin Q_{i+2j} \mid \mu_t(e_3) = (w2,c)\}$ with $e \rightarrow e_3$ where e is the maximal event in Q_{i+2j} satisfying $(\mu_t(e))_1 = w_2$, and where $e^{'}$ is the maximal event in Q_k for the largest $k \leq i-1$ satisfying $\mu_t(e^{'}) = (w_2,((m,c),r,w))$ for some $w \in W_t$ and $(m,c) \in Mes_t$.
– Let $\{Q_{k_0},\ldots,Q_{k_p} \mid$ for every Q_{k_m} with $1 \leq k_m < i$ and $0 \leq m \leq p$ such that there is a maximal event e_{k_m} with $\mu_t(e_{k_m}) = (w_2,((m,c_{k_m}),r,w))$ for some $w \in W_t$ and $(m,c_{k_m}) \in Mes_t\}$. We define $Q^{'}_{k_m} = Q^{'}_{k_{m+1}} \cup \{e_{k_m} \notin Q^{'}_{k_{m+1}} \mid \mu_t(e_{k_m}) = (w_2,c_{k_m})\}$ with $e_{k_{m+1}} \rightarrow e_{k_m}$ and where $Q^{'}_{k_{p+1}} = Q_{i+2j+1}$.

If instead of a forget message the environment responds with a compensate message, then the affected service needs to undo all its actions in reverse order by doing all associated compensation actions. We obtain a succession of new configurations for each compensation performed. We omit the proof that the obtained sequence is a valid sequence of configurations in an extended model M_t. Furthermore, if the compensations denote complex behaviour described in another sequence diagram, we can obtain the refined model by applying the categorical construction of [9].

If we go back to the example configuration path given earlier for the example of Fig. 4, we can obtain the faulted path in accordance with the scenario of Fig. 2 as follows. A fault can happen after the invocation of m_4 at configuration Q_5, namely after

$$Q_0 \xrightarrow{\langle(g,m_1/c_1!,w_1)\rangle} Q_1 \xrightarrow{\langle(w1,m_1/c_1?,g)\rangle} Q_2 \xrightarrow{\langle-\rangle} Q_3 \xrightarrow{\langle(w1,m_4!,w_3)\rangle} Q_4 \xrightarrow{\langle(w_3,m_4?,w_1)\rangle} Q_5$$

leading to

$$Q_5 \xrightarrow{\langle(w_3,faulted!,g)\rangle} Q^{'}_6 \xrightarrow{\langle(g,faulted?,w_3)\rangle} Q^{'}_7 \xrightarrow{\langle(g,forget!,w_3)\rangle} Q^{'}_8 \xrightarrow{\langle(w_3,forget?,g)\rangle} Q^{'}_9$$

and further

$$Q^{'}_9 \xrightarrow{\langle(g,compensate!,w1)\rangle} Q^{'}_{10} \xrightarrow{\langle(w1,compensate?,g)\rangle} Q^{'}_{11}$$

In this case, only message m_1 for w_1 needs to be compensated and the faulted path thus finished at $Q^{'}_{11}$ where a new event e_{11} has a label $\mu(e_{11}) = (w_1,c_1)$.

Once faults and associated forget/compensate mechanisms are done for all configuration paths over M_t, we can derive a complete model from both configuration paths and faulted paths for the conversation t.

5 Related Work

We have seen that the need for releasing *partial results* outside a transaction may not happen as often as sharing results inside a transaction, but is nevertheless a primary requirement if long-running transactions are to cover a wide range of business models. Conventional transaction models such as *Sagas* [11] or the more recent models targeting web services such as *Web Services Transactions* (WS-Tx) [12] and *Business Transaction Protocol* (BTP) [13] do not provide capability for partial results and inevitably make it the business process designer's responsibility. This often means that new transactions are added that do not reflect the exact needs of the business activity itself but rather are added to get round the problem. Further, existing transaction models seem to be geared towards centralised control (WS-Coordination framework [12]) which means that, especially during compensation, access to the local state of service execution is required. This violates the primary requirement of SOA for loosely-coupled services and may not be possible or acceptable in a business environment as it does not respect the local autonomy of participating SMEs. Further, there is no capability for forward recovery and no provision for covering omitted results.

Part of the problem seems to be that multi-party conversations are involved and such frameworks are lacking a formal model for the coordination of the underlying interactions between services. It is only recently that long-running transactions have received the attention of the formal methods community.

The authors in [14] define a set of primitives for long-running transactions in flow composition languages concerned with structured control flows, given in terms of sequencing and branching. Their approach to modelling long-running transactions is driven by the understanding of long-running transactions as in *Sagas* [11]. The Sagas model is a point of reference for long-lived database transactions, nevertheless its applicability in conducting long-running business transactions is questioned (e.g. see [15]).

Furthermore, the basic idea is that a long-running transaction is modelled using CSP sequential processes. The fact there is no communication between sequential processes that are composed in parallel in [14] means that the parallel composition operator simply generates all nondeterministic interleavings of the actions from each process, and this may cause unnecessary overhead in compensating for parallel processes. In fact, the extension of CSP with compensations to produce the so-called *compensating* CSP (cCSP) [16] appeals to a non-interleaving semantics [17] when performing the compensations for sequential processes that are composed in parallel.

In the approach taken in [16] a long-running transaction is modelled as a sequential process, with the usual operators for sequential composition, choice of action, parallel composition. The authors incorporate constructs for writing compensable processes and then introduce a cancellation semantics for compensable processes. The resulting cCSP framework provides a blueprint for a process algebra that models long-running transactions. The notion of a long-running transaction considered however draws upon the concept found in Sagas, and this comes with potential pitfalls, as mentioned before.

In cCSP transactions are understood as sequences of isolated activities and no communication is allowed between internal activities of a transaction. The only communication allowed is that of synchronising on terminal events of sequential processes that have been composed in parallel. As a result of prohibiting communication, there is no provision for partial results but also it is not possible to trigger the compensating procedure in one process as soon as a failure occurs in some other process. This is not remotely satisfactory when modelling real problems which require activities within a transaction to be executed in parallel, since it may result in a situation where one process fails early on in its execution and the other processes have to complete their execution until they reach their terminal event in order to be notified (via synchronisation) that they need to compensate the activities performed due to a failure in the other process.

6 Conclusions

We have described a formal model for the coordination of multi-party conversations in the context of long-running transactions. In particular, we showed how to model interrupting conversations in the presence of faults and described the compensating sequences of operation invocations that are required to undo the (forward) operation invocations or conversations. We considered two ways in which to model interruptions: forgetful rollback and rollback with memory. Moreover, our approach allows communication between web services (multi-party conversation) of a transaction and these interactions may happen concurrently. The corresponding compensations, in case the conversation is interrupted, also take place concurrently.

The abortion of a transaction, even if it is successfully compensated for, can be very costly especially in a business environment where accountability and trust are major concerns. Rolling back the whole system may lead to chains of compensating activities that are time-consuming and impact on network traffic. For this reason it is important to add diversity into the system and allow for alternative paths of execution in cases where the path chosen originally encountered a failure. Our approach can be extended to handle *forward recovery* by examining the runs (maximal configurations) of a conversation and working out the extent to which a faulted path should be compensated until it reaches another configuration path that can lead to a run which allows the transaction to commit.

In previous work [18], which draws upon the translation of sequence diagrams in [8] outlined in this paper, we have looked at reasoning about scenario-based specifications using *vector languages* [17] and have shown how this can uncover additional scenarios which are potentially faulty (e.g. due to race conditions) or simply unthought in the initial design [19]. This provides interesting perspectives with regard to identifying the complete set of behaviours of a given multi-party conversation, and on that basis determine alternative scenarios of execution for the transaction.

Finally, we are currently extending our distributed temporal logic interpreted over labelled event structures (cf. [8]) to be able to express properties about (interrupted) conversations. The distributed nature of the logic is crucial in a context of loosely-coupled web services. With the logic we will also be able to analyse whether our extended models with faults are complete and possibly reveal further faulted paths.

References

1. Digital Business Ecosystem (DBE), EU-FP6 IST Integrated Project No 507953 (2006), http://www.digital-ecosystem.org
2. Date, C.J.: An Introduction to Database Systems, 5th edn. Addison-Wesley, Reading (1996)
3. Razavi, A., Moschoyiannis, S., Krause, P.: Concurrency Control and Recovery Management in Open e-Business Transactions. In: Proc. WoTUG Communicating Process Architectures (CPA 2007), pp. 267–285. IOS Press, Amsterdam (2007)
4. Papazoglou, M.P., Georgakopoulos, D.: Service-Oriented Computing. Communications of the ACM 46(10), 24–28 (2003)
5. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F., Kramer, B.J.: Service-Oriented Computing Research Roadmap. In: Dagstuhl Seminar Proc. 05462, Service-Oriented Computing (SOC), pp. 1–29 (2006)
6. O.M.G.: UML 2.0 Superstructure Specification. document ptc/04-10-02 (2004), http://www.uml.org
7. Razavi, A., Moschoyiannis, S., Krause, P.: A Coordination Model for Distributed Transactions in Digital Business Ecosystems. In: Digital Ecosystems and Technologies (DEST 2007), IEEE Computer Society Press, Los Alamitos (2007)
8. Küster-Filipe, J.: Modelling concurrent interactions. Theoretical Computer Science 351(2), 203–220 (2006)
9. Bowles, J.K.F.: Decomposing Interactions. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 189–203. Springer, Heidelberg (2006)
10. Winskel, G., Nielsen, M.: Models for Concurrency. In: Handbook of Logic in Computer Science, vol. 4, pp. 1–148. Oxford Science Publications (1995)
11. Garcia-Molina, H., Salem, K.: Sagas. In: ACM SIGMOD, pp. 249–259 (1987)
12. Cabrera, F.L., Copeland, G., Johnson, J., Langworthy, D.: Coordinating Web Services Activities with WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity (January 2004), http://msdn.micorsoft.com/webservices/default.aspx
13. Furnis, P., Dalal, S., Fletcher, T., Green, A., Ceponkus, A., Pope, B.: Business Transaction Protocol, version 1.1.0 (November 2004), http://www.oasis-open.org/committees/download.php/9836
14. Bruni, R., Melgatti, H., Montanari, U.: Theoretical Foundations for Compensations in Flow Composition Languages. In: Principles of Programming Languages (POPL 2005), pp. 209–220. ACM Press, New York (2005)
15. Furnis, P., Green, A.: Choreology Ltd. Contribution to the OASIS WS-Tx Technical Committee relating to WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity (November 2005), http://www.oasis-open.org/committees/download.php/15808
16. Butler, M., Hoare, A.C.R., Ferreira, C.: Trace Semantics for Long-Running Transactions. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. LNCS, vol. 3525, pp. 133–150. Springer, Heidelberg (2005)
17. Shields, M.W.: Semantics of Parallelism. Springer, London (1997)
18. Moschoyiannis, S., Krause, P., Shields, M.W.: A True Concurrent Interpretation of Behavioural Scenarios. In: FESCA 2007. ENTCS, Elsevier, Amsterdam (to appear)
19. Moschoyiannis, S.: Specification and Analysis of Component-Based Software in a Concurrent Setting. PhD thesis, University of Surrey (2005)