True Concurrency in Long-running Transactions for Digital Ecosystems

Sotiris Moschoyiannis and Paul J. Krause

Department of Computing
Faculty of Engineering and Physical Sciences
University of Surrey, Guildford, Surrey
GU2 7XH, UK
{s.moschoyiannis, p.krause}@surrey.ac.uk

Abstract

The concept of a digital ecosystem (DE) has been used to explore scenarios in which multiple online services and resources can be accessed by users without there being a single point of control, which can be used to effectively serialise their interactions. We argue in this paper that this weak coupling between services places additional demands on the modelling of compensation and recovery management in long-running transactions over traditional SOC related formalisms. We describe an adaptation of Shields’ vector languages, in that the synchronisation constraint is removed (no shared actions), as a formal semantics for a transaction in terms of the common ordering constraints on the underlying interactions between its participants. The notation afforded by the so-called transaction languages captures the invocations on each participant service (online resource), and at each point during execution, across the whole transaction. Concurrency is modelled explicitly through a notion of independence, which is lifted onto tuples of sequences (one for each participant of the transaction) rather than individual sequences, as in Mazurkiewicz trace languages or events, as in the event structures model. Participating subcomponents execute concurrently and failure of one or more causes the recovery of the whole transaction. Compensations are triggered immediately upon failure and concurrent forward actions are compensated concurrently. We highlight the benefits of our true-concurrent approach in the context of DEs and outline connections of transaction languages to other partial order models. Further, we discuss how our approach supports forward recovery in that recovering the whole transaction is avoided wherever possible.

1 Introduction

There has been a growing interest in applications that are built up as networks of collaborating services and online resources, which are distributed within and across organisational boundaries. Business requests can be defined in terms of finer-grained subtasks that are available as loosely-coupled online services. Central aspects that come into this paradigm of building collaborative applications for fulfilling business requests have to do with coordinating online service interactions, providing transactional guarantees in meeting a compound business request, and considerations about the underlying digital infrastructure.
1.1 Digital Ecosystems and long-running transactions

The concept of a Digital (Business) Ecosystem has been in circulation since 2002 [39]. This was in response to the observation that although there were at the time over 19 million Small and Medium Enterprises (SMEs) in Europe, they were lagging behind in the adoption of e-business as a distribution channel. We have seen a steady improvement in this situation since 2002 [16], but the e-market place is still heavily dominated by a relatively small number of "keystone" organisations.

A key vision of many working in the Digital Ecosystem (DE) community is that a DE be a "self-organising agent environment" [5]. In contrast to a tightly coupled organisation in which agents have pre-defined roles, the focus in a DE is very much on respecting the autonomy of individual agents and enabling the global properties and institutions of such an ecosystem to emerge (primarily) through self-organisation. In [26] we proposed the following definition:

"A digital ecosystem is an interactive system established between a set of active agents and an environment within which they engage in common activities."

"Agents" include (but might not be limited to) providers of software services, information sources, and human agents. The environment is a combination of a socio-economic context and a digital infrastructure. We argued in [26] that the nature of the latter, the digital infrastructure, can impact (undesirably at present) on the properties that emerge in the ecosystem. This is particularly noticeable in the context of the support for transactions involving a composition of web services from disparate organisations. The WS-* web services efforts [54] to realise the promise of Service-Oriented Computing (SOC) [43], also known as Big Web Services [48], are rooted in enterprise systems vendors and the history of that industry.

In recent years most of the implementation of WS-* technologies has been limited to the interior of the corporate firewalls, having failed to convince practitioners on the open Web. For instance, the remote procedure call (RPC) style of the main protocols in the WS-* architecture, such as the Simple Object Access Protocol (SOAP), the Web Services Description Language (WSDL) and the Universal Description Discovery and Integration (UDDI), leads to a tighter coupling between services and their respective host organisations than is appropriate for a DE. In addition, the modelling of business transactions has focussed on the perspective of a single (coordinating) organisation. This militates against meeting the vision of a DE in which online services collaborate to meet a common goal (business request), without the need for a central portal or controller.

In [30] we propose an alternative architecture to WS-* for large-scale distributed applications built on RESTful Web Services, and Semantics of Business Vocabulary and Rules (SBVR) [41] as a modelling language. REST stands for Representational State Transfer and refers to an architectural style that focuses on a coordinated set of constraints applied on interacting components within a distributed hypermedia system. The World Wide Web represents the largest implementation of a system conforming to REST. The goal in our work is to enable the dynamic composition of a long-running transaction involving services from participating businesses in response to a compound business request from a consumer. The business request would be launched as a query on a pool of resources described using SBVR (hence, steering away from WSDL and UDDI). The response to this query would be a candidate transaction that could then coordinate the invocation of the RESTful interfaces of the resources needed to satisfy the request. Invocations can be implemented using REST over HTTP, which provides a uniform interface (GET, PUT, POST, DELETE operations of the HTTP protocol), hence removing the need for RPC-style invocations which compromise the
weak coupling between collaborating services. We omit further details and refer to [30] for a
detailed treatment of the proposed RESTful architecture for distributed applications.

The issue of coordinating the corresponding service interactions has been approached in
standardisation proposals from two different viewpoints. One is concerned with the descrip-
tion of the interaction from an individual participant’s viewpoint in what is referred to as
service orchestration. The focus here is on describing the interactions in which a given service
can engage together with the internal steps between these interactions (e.g., data transfor-
mations). The other viewpoint is concerned with the description of the interaction scenario
from a global perspective in what is referred to as service choreography. The focus here is on
describing the interactions in which the participating services engage, often in a peer-to-peer
(P2P) fashion, and the dependencies that arise, including control flow dependencies (e.g., a
given service invocation must occur before another one), time constraints, transactional de-
dendencies, message correlations, etc. A choreography does not describe any internal action
(e.g., internal computation, data transformation) within a participating service.

Standardisation proposals for service orchestration have culminated in the Web Services
Business Process Execution Language (WS-BPEL) [19]. Standardisation activities for chore-
ography have resulted in W3C’s Web Services Choreography Description Language (WS-CDL)
[56]. We note that the need for transactional support in service choreography is mentioned
in WS-CDL, but the standard lacks an associated formal semantics for the (XML) constructs
it defines, e.g., exception work units. A critical overview of WS-CDL can be found in [2].

Web transaction protocols such as the Business Transaction Protocol (BTP) [17] and WS-
Transaction [18] use long-running transactions as a mechanism for describing loosely-coupled
(online) activities but do not support a definitive mechanism for recovering a transaction in
the event of a failure. The Web Services community has not reached an agreement on a unique
notion for long running transactions [29]. As a result, in practice this is typically dealt with
in ad-hoc ways by the programmer. We discuss such aspects and a number of surrounding
issues in the related work section (Section 5).

1.2 Formal models for long-running transactions

Current approaches to giving a formal semantics to long-running transactions seem to be
g geared towards service orchestration while choreography has received much less attention.
Traditional SOC-related formalisms such as [4, 6, 7, 11, 12, 22, 28, 33] have targeted dis-
tributed applications that are controlled by a single entity. It may be appropriate to do so for
orchestration, which restricts to describing the interaction from a given service’s viewpoint,
but is not entirely suitable for the choreography of distributed services where the focus is on
describing the conversation across the different participating services. Furthermore, there is
a range of scenarios requiring applications that do not rely on a central controlling instance,
such as those considered by the DE community and the open web, but also in P2P systems,
Cloud Computing, and mobile ad-hoc networks (MANETs).

In addition, formal work to date has focused on modelling the behaviour of a transaction
using an interleaving semantics. This is perhaps in line with targeting service orchestration
and assuming a single controlling instance, which can be used to effectively serialise concurrent
interactions (invocations). However, when considering choreography of services from disparate
organisations, in the absence of a single point of control, a number of inherent assumptions
in current formalisms become questionable.

The approaches to concurrency proposed in the literature can be classified in two main
categories [58]: the so-called interleaving semantics, which is based on the assumption that concurrency between actions can be reduced to the nondeterministic choice between their possible sequentialisations, see for example the influential contributions by Milner [34] and Hoare [24]; and the so-called true concurrency or partial order semantics in which concurrency is considered as a primitive notion and is modelled explicitly by means of causal independence, see for example the seminal work by Petri [45] and Mazurkiewicz [32], Shields [49, 50, 52], Winskel [40]. The basic difference between the two is often highlighted by means of a simple example (e.g., see [13], which also discusses it in view of refinement of atomic actions in the different approaches). Consider a system concurrently performing actions $a$ and $b$. From the point of view of an interleaving semantics, $a|b$ and $a;b + b;a$ would be considered equivalent behaviours of such a system (we use CSP-like expressions here, where '$|$' denotes parallel composition, '+' for choice, and ';' for sequential composition). Even though one says that $a$ and $b$ happen concurrently and the other says that there is a choice between doing $a$ and then $b$ or doing $b$ and then $a$, these behaviours are identified as the same, even by bisimulation [34]. In a true concurrency semantics, the two behaviours are distinguished, i.e., $a$ and $b$ are either concurrent or there is a choice between doing $a$ and then $b$ or doing $b$ and then $a$. This is because concurrency is identified explicitly in this semantics instead of being reduced to the nondeterministic choice between the interleavings of the actions involved.

In a transactional setting it is important to be able to distinguish between conflicting (choice) and concurrent actions in the behavioural description of a transaction. This is because the way the forward actions are modelled (while the transaction is executing successfully) has a bearing on the series of compensating actions that need to be performed, in the reverse order, whenever a failure makes this necessary. In other words, given failure of any subcomponent of the transaction, the transaction must recover cleanly without locking up additional resources (which could lead to financial loss in terms of opportunity cost). The true concurrency semantics in our approach allows to compensate concurrently for previous successful concurrent actions (invocations between participating services). Also, since the multi-party conversation involves services from disparate organisations, which collaborate in the absence of a central controller, it seems appropriate that a formal model for long-running transactions should handle true concurrency.

Furthermore, the model should be able to express the common ordering conditions on externally visible actions across the multi-party conversation, rather than from (or, in addition to) an individual service’s viewpoint. That is to say that the behaviour of a transaction is subject to a set of sequential constraints, each constraint determining the orderings of the invocations on a specific participating service. The question is how we can check that the transaction as a whole respects the constraints of each participant.

In order to answer this we need to look across the multi-party conversation, at each point during transaction execution. This is something that is naturally provided by Shields’ vector languages [52]. In this semantics, the behaviour of a family of processes is described by tuples of sequences, which record the actions performed by individual processes in the behaviour. The ordering constraints on invocations between different services are manifested in the order structure of such a set of vectors (cf Section 3.1, Fig. 2).

Concurrency is modelled explicitly in the model using the notion of causal independence. It is essentially a partial order model of concurrency (cf Definition 3.5, Proposition 3.1). The construction is lifted onto tuples of sequences (one sequence for each participant of the transaction) rather than individual sequences, as in Mazurkiewicz trace languages [32], or events, as in event structures [40]. While Mazurkiewicz trace languages can be understood
as a subset of a semi-commutative monoid whose generators are sequences, vector languages can be seen as subsets of a monoid whose generators are *tuples of sequences*. In both cases, the commuting of generators models potential concurrency.

There is a subtle difference between the vector languages considered in this paper and the original ones studied in [52] and elsewhere (e.g., seminal paper on adequate path expressions in [49]). The vector semantics in [52] considers shared actions between different participants. This means that the corresponding sequences cooperate in a specific way, following a synchronisation constraint similar to the path composition rule in COSY [25] or the ‘||’ operator of CSP or the synchronisation vectors in Vector Controlled Concurrent Systems [38]. In the context of long-running transactions in a DE web-oriented architecture, actions or invocations on interfaces of online resources can be done asynchronously using REST over HTTP. They are not necessarily RPC-style invocations, which typically require (some form of) synchronisation and result in a tighter coupling between services, as discussed before. Our adaptation of vector languages does not consider shared actions, and hence the sequences appearing in the so-called *transaction languages* described here do not follow a synchronisation constraint.

In previous work [36], we have also shown how these languages can be used to reveal missing, and potentially faulty, behaviours in the forward execution of a transaction.

In this paper, we present our asynchronous version of vector languages as a semantics for modelling the forward and compensating behaviour of a long-running transaction that executes in an open distributed environment, like the open web, which does not depend on a central controlling instance. We describe how our formal model can be used to provide a global definition of the common ordering constraints among collaborating services while the transaction is executing successfully (*forward behaviour*). We then extend the model with cancellation operations and show how this is used to describe *compensating behaviour* - going backwards, upon failure, and effectively 'undoing' all successful forward actions.

We discuss the benefits of our true-concurrent approach when it comes to compensating for concurrent forward actions. These build on the formal characterisation of transaction *execution histories* (cf Corollary 4.1, Definition 4.5) and exploiting the additional structure provided by the notion of independence between transaction vectors (Definition 3.7) to characterise *equivalent execution histories* (Definition 4.6). We also highlight the provision for *forward recovery* by exploiting the conflict structure present (cf Fig. 2), which is possible in our model precisely because the notation offered by transaction languages captures *snapshots of the actions* of each participant, at each point during execution.

This paper is structured as follows. Section 2 outlines the design of a transaction using SOC concepts and UML, with reference to choreography. Section 3 presents a true concurrency semantics for long-running transactions that captures *forward behaviour*. Section 4 introduces a cancellation semantics for expressing *compensating behaviour* in light of transaction recovery management. In Section 5 we discuss related work, focusing on SOC transaction formalisms. Section 6 contains some concluding remarks and possible extensions of this work.

## 2 Designing a long-running transaction

We have seen that service choreography aims to provide a global view of the interactions between multiple online services in order to achieve a common goal. The WS-CDL choreography standard [56] includes a *declarative* part, which defines the participants and the set of operations that can be invoked on each, and a *conversational* part, which provides the global
definition of the common ordering conditions and constraints within a conversation between services from participating businesses, in response to a compound request.

In [30] we have described how a business request launched as a query on a pool of resources described in SBVR results in the generation of a transaction tree. This can be drawn using SOC notation [44] and is intended to describe the participants of the transaction and the coordination of the underlying service invocations. In this sense, it sets the context of the conversation to follow. In [36] we have provided a schema for describing transaction contexts. Fig. 1 shows a transaction tree involving five service invocations on online resources - \( a_1 \), \( a_2 \) and \( a_3 \) provided by a local platform with coordinator component \( CC_1 \) (e.g., flight reservations), \( b_1 \) and \( b_2 \) of \( CC_2 \) (hotel bookings), and \( c_1 \) of \( CC_3 \) (transport tickets) - whose order of execution is determined by the associated composition types on the corresponding nodes.

![Figure 1: A transaction tree (SOC notation) and the corresponding behavioural scenario](image)

We note that the example of a transaction given in Fig. 1 can be found in BTP [17]. It has also been considered in [36] and it will be extended in this paper to illustrate the key ideas. The service interactions implied by a transaction tree, whether these are service calls (RPC-style) or invocations of RESTful APIs, can be described in a UML model [42]. The UML 2 sequence diagram in Fig. 1 shows the coordinator components of the participants and the required invocations between them. The behavioural scenario given by the corresponding sequence diagram determines the order of the invocations on the participating components’ services.

It can be seen that a long-running transaction is associated with a set of coordinator components \( C \) executing concurrently. Each subcomponent of the transaction engages in a specific set of actions. The conversational part of the WS-CDL choreography specification [56] for instance includes the Interaction Activity, under Basic activities, which is intended to describe the exchange of information between the participants in terms of making a request, or making a response (respond), or making a request that requires a response (request-respond). It is worth noting that WS-CDL focuses on the receiver of the information, hence WS-CDL Interaction Activity descriptions refer to the activity performed when the information is received. Our interest is in the observable events on coordinator components and thus actions can be understood as invocations on the RESTful interfaces of the online resources, or, more generally, service invocations on coordinator components as shown in the scenario of Fig. 1.

Although our objective in this paper is not to give a formal semantics to WS-CDL, we
note that there are three types of control-flow activities in the standard: the Sequence activity, the Choice activity, and the Parallel activity. These different modes of interaction are captured in the generated transaction tree and, perhaps even more clearly, in the corresponding behavioural scenario. It is worth bearing in mind that the Parallel activity in WS-CDL is defined to "describe one or more activities that can be executed in any order or at the same time" (see [56]). As will be shown in Section 3.1, the notation offered by transaction languages proposed here, can express formally this interpretation of parallelism.

3 Formal description of a long-running transaction

In the context of long-running transactions a successful outcome is either the execution of the whole interaction scenario (forward behaviour) or the ‘undoing’ of previously completed parts of the interaction, via compensating actions, if some failure later on in the transaction makes this necessary (compensating behaviour). First, we give a formal description of a transaction and then show how the semantics can be used to model forward and compensating behaviour.

3.1 Vector semantics for long-running transactions

We have seen in the previous section that a transaction is associated with a set of actions $M$ and each subcomponent in $C$ is associated with a subset of these actions, which correspond to receiving a request to deploy services or operate on resources provided by its own platform or web application. This is captured in the following which formalises transaction contexts.

**Definition 3.1 [Transaction signature]** A transaction signature is given by a pair $\Sigma = (C, \mu)$, where $C$ is the set of coordinator components, and $\mu$ is given by $\mu : C \rightarrow \wp(M)$ such that $\forall i, j \in C, i \neq j, \mu(i) \cap \mu(j) = \emptyset$ and $M = \bigcup_{i \in C} \mu(i)$.

By $\wp(M)$ we denote the powerset of $M$, i.e., the set of all subsets of $M$. Hence, $\mu(i)$ is the set of actions associated with component $i$, for each $i \in C$. The conditions required of $\mu$ mean that the intersection of any two distinct sets is empty (pairwise disjoint) and the union of all sets is equal to $M$. No set is empty, as it contains the actions (received requests) of each subcomponent. Hence, the sets of $\mu(i)$, for all $i$, partition the set of actions $M$ of the transaction. As shown in Fig. 1, a transaction has a number of activation or access points, namely the interfaces of the participating coordinator components.

Each participant will have its own set of constraints that dictate the order in which the actions (in its $\mu(i)$) are allowed to occur. The behaviour of each participant in the conversation involved in the transaction can be given in terms of a deterministic sequential process recording the order in which the actions have been performed. This is the individual participant’s viewpoint, which is of interest in service orchestration, e.g., see WS-BPEL [19], and can be described in terms of sequences of action names. The set of all behaviours of participant $i$ is given by the set of all string prefixes of the elements in its $\mu(i)$.

The behaviour of the transaction as a whole, from a global perspective, which is of interest in service choreography, e.g., see WS-CDL [56], can be described using a tuple of strings, one string for each participant, recording the observable actions across the multi-party conversation.

**Definition 3.2 [Transaction vectors]** Let $\Sigma = (C, \mu)$ be the signature of a transaction. We define $V_\Sigma$ to be the set of all functions $\nu : C \rightarrow M^*$ such that $\nu(i) \in \mu(i)^*$, for all $i$. 7
By $\mu(i)^*$ we denote the set of all finite sequences formed over $\mu(i)$. The vector $v$ satisfying $v(i) = \Lambda$, for each $i$, will be written $A_{\Sigma}$, where $\Lambda$ denotes the empty sequence.

When $C = \{1, ..., n\}$, we can represent a vector $v$ of $V_\Sigma$ by the tuple

$$(v(1), v(2), ..., v(n)) \in \mu(1)^* \times \mu(2)^* \times ... \times \mu(n)^*$$

where $v(1) \in \mu(1)^*$, $v(2) \in \mu(2)^*$, ..., $v(n) \in \mu(n)^*$.

That is to say that the set of vectors $V_\Sigma$ is the Cartesian product of the sets $\mu(i)^*$, for each $i$. Each transaction vector provides a snapshot of behaviour in that it describes what actions have taken place and on which coordinator components of the transaction.

**Example 3.1** Let $C = \{CC_1, CC_2, CC_3\}$ with $\mu(CC_1) = \{a_1, a_2, a_3\}$, $\mu(CC_2) = \{b_1, b_2\}$ and $\mu(CC_3) = \{c_1\}$. We may write $(v(CC_1), v(CC_2), v(CC_3))$ for $v \in V_\Sigma$, effectively assigning a vector coordinate to each coordinator component of the transaction in Fig. 1.

Consider $v$ given by $v(CC_1) = a_1a_2$, $v(CC_2) = b_1$ and $v(CC_3) = \Lambda$. This transaction vector will normally be written $v = (a_1a_2b_1, \Lambda)$. The vector $u = (a_1a_2b_1, \Lambda)$ for example is not a transaction vector for this transaction, since $u(CC_2) = a_2b_1 \notin \mu(CC_2)^*$.

We note that the $\mu(i)$, for each $i$, are pairwise disjoint by Definition 3.1 so action names are unique on subcomponents of a transaction. This is a subtle technical difference between the transaction vectors considered here (also in [36, 37]) and Shields’ behaviour vectors in [52], where common actions are allowed at the expense of a synchronisation constraint. In Shields’ original vector languages, processes synchronise on shared actions, i.e., an action can occur only if it occurs in every process having that action in its alphabet. In the context of transactions on the Web involving the execution of online services on resources from disparate organisations, and the more general setting of DEs which champion weak coupling and local autonomy as discussed in the Introduction (Section 1), it seems appropriate to defer from considering common actions and applying a synchronisation constraint.

So while processes in behaviour vectors in [52] are composed following a synchronisation rule familiar to the parallel composition operator ‘||’ of CSP [24] or the path composition rule in COSY [25] or the control mechanism in VCCS [38], the way the sequential processes in transaction vectors are handled resembles the ‘||’ operator of CSP. This adaptation of vector languages still allows to model true concurrency between actions on distinct coordinates of transaction vectors and hence on different subcomponents of a long-running transaction.

Transaction vectors come with a range of algebraic properties which can be exploited in the formal analysis and reasoning about the complex interactions involved in a transaction. In what follows we outline the basic properties of transaction vectors, focusing on their order structure since the orderings between vectors in a given subset of $V_\Sigma$ (cf Definition 3.8) is what determines the patterns the underlying service invocations in the corresponding transaction should follow and is what constitutes the so-called transaction history [21].

First, we consider a specific kind of transaction vector, which is used in our model to describe actions (e.g. service requests, invocations on RESTful interfaces) within a transaction.

**Definition 3.3 [Action vectors]** Let $V_\Sigma$ be a set of transaction vectors over a signature $\Sigma$. Define

$$A_\Sigma = \{a \in V_\Sigma \mid \{A_\Sigma\} \mid |a(i)| \leq 1\}$$

where $|x|$ is used to denote the length of the sequence $x$.
Thus, action vectors are themselves transaction vectors, but have the additional constraint that every coordinate is either the empty sequence or contains a single action - at least one coordinate in an action vector must be non-empty. For example, the action vector $\alpha_1 = (a_1, \Lambda, \Lambda)$ represents a service invocation $a_1 (a_1 \in \mu(CC_1))$ on the subtransaction associated with the first coordinate. The action vector $\alpha_2 = (a_2, \Lambda, \Lambda)$ represents a service invocation $a_2 (a_2 \in \mu(CC_1))$ also on the subtransaction associated with the first coordinate. On the contrary, the vector $u = (a_1 a_2, \Lambda, \Lambda)$ is not an action vector, it is a transaction vector which captures that part of behaviour of the transaction in which service invocation $a_1$ followed by service invocation $a_2$ have occurred, and both occurred on the subtransaction associated with the first coordinate.

Note that an action vector may be representing more than one occurrence of an action. In the version of vector languages considered here, this can only be the case for different coordinates, and hence for different actions. Remember that the alphabets associated with each participant service are disjoint (by Definition 3.1). If two or more coordinates of an action vector are non-empty, then the corresponding actions occur simultaneously and we talk about a simultaneity class of action occurrences.

In the context of modelling the behaviour of a long-running transaction the interest is in modelling invocations on different service interfaces, which happen concurrently. Therefore, in the remainder of this paper, action vectors will contain a single action but the fact the notation is sensible to simultaneity classes of action occurrences allows to draw connections to models using step semantics or pomsets. We will have more to say on this in the concluding section of the paper.

By definition, transaction vectors are tuples of sequences. This means that operations on vectors can be defined in terms of well known operations on sequences.

**Definition 3.4 [Concatenation]** For $u, v \in V_\Sigma$, we define $u \cdot v$ to be the unique vector $w$ such that $w(i) = u(i) \cdot v(i)$, for each $i \in C$.

Hence, concatenation on vectors is defined in terms of concatenation of the sequences appearing on their respective coordinates. For example, $(a_1, b_1, \Lambda) \cdot (a_2, \Lambda, \Lambda) = (a_1 a_2, b_1, \Lambda)$.

**Definition 3.5 [Prefix ordering]** For $u, v \in V_\Sigma$, define $u \leq v$ iff $u(i) \leq v(i)$, for each $i \in C$.

Hence, the ordering amongst vectors is defined in terms of the usual prefix ordering operation on the sequences appearing on their respective coordinates. For example, for $(a_1, b_1, \Lambda)$ and $(a_1 a_2, b_1, \Lambda)$ we have, $(a_1, b_1, \Lambda) \leq (a_1 a_2, b_1, \Lambda)$ since $a_1 \leq a_1 a_2$ and $b_1 \leq b_1$ and $\Lambda \leq \Lambda$.

The result in the following proposition may be deduced by arguing coordinate-wise. For example, $u \cdot (v \cdot w) = (u \cdot v) \cdot w$ because $u(i) \cdot (v(i) \cdot w(i)) = (u(i) \cdot v(i)) \cdot w(i)$, for all $i$.

**Proposition 3.1** A set of transaction vectors $V_\Sigma$ formed over a signature $\Sigma$ is:

1. a monoid with binary operation $\cdot$ and identity $\Lambda_\Sigma$.
2. a poset with partial order $\leq$ and bottom element $\Lambda_\Sigma$.

**Proof.** Straightforward, by arguing coordinate-wise. See also proof of Prop. 9.2.7 in [52]. □
Point (1) of the proposition says that under the operation of concatenation the set \( V_\Sigma \) is a monoid (a semi-group with identity) where the identity element is the empty vector \( \Lambda_\Sigma \). We will see that transaction vectors are formed by concatenation, or more precisely, by a series of concatenations, with action vectors starting from the empty vector (cf Section 4.2). In giving a formal semantics to a long-running transaction, our interest is in the monoid generated by the action vectors in \( A_\Sigma \) of the transaction in question, namely the set of all products of the form \( \alpha_1, \alpha_2, \ldots, \alpha_n \), where \( \alpha_i \in A_\Sigma \), for \( i = 1..n \), together with the empty vector \( \Lambda_\Sigma \). The set of these products comprises \( V_\Sigma \) for a given transaction.

Point (2) of the proposition says that under the operation of prefix ordering, the set of vectors \( V_\Sigma \) is a partially ordered set where the bottom element is the empty vector \( \Lambda_\Sigma \). This implies that some vectors are comparable, so \( u \leq v \) means that \( u \) describes an earlier part of behaviour than \( v \) does, or that the behaviour described by \( u \) is causally related to that described by \( v \) (causality). It also implies that some vectors may be incomparable in which case these vectors express either alternative behaviour (conflict) or concurrent behaviour (concurrency).

Incomparable vectors in \( V_\Sigma \) either describe a choice between actions or describe concurrent actions on the respective coordinator components of the transaction. Whether the incomparable vectors describe conflict or concurrency is determined by whether they lead to the same behaviour or not (cf Figure 2). In order-theoretic terms, this comes down to whether they are bounded above by another vector in \( V_\Sigma \) or not.

**Definition 3.6** For \( u, v \in V_\Sigma \), we define

- **[Greatest lower bound]** \( u \sqcap v \) to be the vector \( w \) which satisfies \( w(i) = \min(u(i), v(i)) \), for each \( i \).

- **[Least upper bound]** We also define \( u \sqcup v \) (if it exists) to be the vector \( w \) which satisfies \( w(i) = \max(u(i), v(i)) \), for each \( i \).

The operations \( \sqcap \) and \( \sqcup \), which we note are also applied coordinate-wise, give the greatest lower bound and the least upper bound of \( u, v \in V_\Sigma \), respectively, in the usual sense of lattices and domain theory [14]. We shall see that these operations are central to the treatment of concurrency in our approach, and also have an important role to play in defining the order-theoretic properties that allow the formal reasoning in determining the complete set of behaviours that are possible in a given interaction scenario, as shown in [36].

Concurrency in our approach is represented explicitly by means of an independence relation on the set of actions in the transaction. This true concurrency semantics is rooted in Mazurkiewicz’s trace languages [32] and Shields’ vector languages [52].

Let \( A \) denote a (finite) set. A concurrent alphabet is an ordered pair \( (A, \iota) \) where \( \iota \subseteq A \times A \) is symmetric \( (a, b \in A, a \iota b \rightarrow b \iota a) \) and irreflexive \( (a, b \in A, a \iota b \rightarrow a \neq b) \). Symmetry requires that concurrency is always mutual while irreflexivity prohibits an action being concurrent with itself. Given a concurrent alphabet \( (A, \iota) \), a relation \( \equiv_\iota^{(1)} \) can be defined on \( A^* \) by

\[
x \equiv_\iota^{(1)} y \iff \exists u, v \in A_\Sigma, a, b \in A_\Sigma : a \iota b \land x = u.a.b.v \land y = u.b.a.v
\]

Now let \( \equiv_\iota \) be the least equivalence relation that includes \( \equiv_\iota^{(1)} \), i.e., the reflexive, transitive closure of \( \equiv_\iota^{(1)} \). We denote the equivalence class of \( x \in A^* \) by \( < x >_\iota \). The set of equivalence classes of \( A \) with independence relation \( \iota \) is denoted by \( A^*_\iota = \{ < x >_\iota \mid x \in A^* \} \). Any subset \( L \) of \( A^*_\iota \) is a Mazurkiewicz trace language [32].
Therefore, the independence relation $\iota$ on $A$ gives rise to an equivalence relation $\equiv_\iota$ on $A^*$. Intuitively, the equivalence relation on sequences of actions says that any two consecutive actions are allowed to permute, providing they are independent, and the resulting sequences are equivalent in that they describe the same behaviour. We make use of this construction in transaction vectors and it is central to the way concurrency is handled in our approach. Drawing upon the extension of the independence relation $\iota$ to behaviour vectors in [52], the notion of independence can be readily interpreted into transaction vectors in our approach.

**Definition 3.7 [Independence]** For $u, v \in V_\Sigma$ we define

$$u \text{ ind } v \iff \forall i \in C : u(i) > \Lambda \Rightarrow v(i) = \Lambda$$

If $u \text{ ind } v$, then we say that $u$ and $v$ are independent.

Thus, two transaction vectors are independent if the behaviours they describe involve actions that occur on different coordinator components of the transaction. In the case of action vectors $\alpha_1, \alpha_2 \in A_\Sigma$, independence captures the fact that the action appearing in one can occur independently of that appearing in the other.

Note that if there are no independent action vectors or no two independent action vectors are enabled after the same behaviour of the transaction, then no action vectors can commute in the series of concatenations that form the corresponding transaction vectors. This amounts to the understanding of sequential systems, the behaviour of which can be captured using the interleaving simplification of representing concurrency as the nondeterministic choice between all possible sequentialisations.

**Example 3.2** The action vectors $\alpha = (a, \Lambda)$ and $\beta = (\Lambda, b)$ describing actions of two subcomponents of a transaction, are independent. Whenever the actions $a$ and $b$ of each subcomponent are both enabled (can both occur at some point in the conversation, after some behaviour) and happen consecutively, then the corresponding action vectors commute when concatenated, i.e., $\alpha \cdot \beta = (a, b) = \beta \cdot \alpha$. If they were both enabled after, say $\Delta_{\Sigma_{ex}} = (\Lambda, \Lambda)$, then in the resulting behaviour captured by transaction vector $v = (a, b)$, the actions $a$ and $b$ happened concurrently.

If both actions are not enabled or do not happen consecutively, then they are not concurrent (even if they, and their corresponding action vectors, are independent). The distinction may be seen more clearly perhaps in the associated Hasse diagrams of Fig. 2.

![Hasse diagrams](image)

**Figure 2:** Causality, concurrency, and conflict in transaction languages

In (i) actions $a$ and $b$ are **ordered** ($b$ is only invoked after $a$), in (ii) they are **concurrent** ($a$ and $b$ are invoked concurrently), and in (iii) $a$ and $b$ are **mutually exclusive** (there is a choice between invoking $a$ or invoking $b$).
Notice that the set of vectors shown in (i) of Fig. 2 does not include \((\Lambda, b)\), which means that \(b\) never occurs without \(a\) having occurred first. In (iii), it does not include \((a, b)\), which means that there is no valid behaviour of the transaction in which both \(a\) and \(b\) have occurred on the respective subcomponents. In (ii) it includes all four vectors, which means that \(a\) and \(b\) are unordered, and happen concurrently. This is indicated by the familiar diamond, found in *Asynchronous Transition Systems* (ATS) [50], that shows the corresponding order structure exhibits the characteristic structure of a finite distributive lattice [14]. The order structure of a set of transaction vectors depends on context - on what other vectors are in the set, as illustrated in Fig. 2.

It might be instructive to relate this to the associated order-theoretic properties of transaction vectors involved in describing concurrent actions. The vector \(\Delta_{\Sigma_{\alpha,\beta}} = (\Lambda, \Lambda)\) is the greatest lower bound of the vectors \(\Delta_{\Sigma_{\alpha}} = (a, \Lambda)\) and \(\Delta_{\Sigma_{\beta}} = (\Lambda, b)\), while the vector \(v = (a, b) = \Delta_{\Sigma_{\alpha}} \cdot \Delta_{\Sigma_{\beta}} = (\Lambda, \Lambda)\), to which their parallel execution leads (and notice the permutation of action vectors \(\alpha\) and \(\beta\)), is their least upper bound (recall Definition 3.6). The presence of the least upper bound is important as it is this vector that describes the common behaviour the two actions \(a, b\) lead to, and allows to distinguish between \(a\) and \(b\) happening concurrently and \(a\) and \(b\) being in conflict.

Finally, notice that there is a concept of *locality* in expressing concurrency - if the actions in question are not enabled at a given point, after some behaviour of the transaction, then nothing can be said about them with respect to concurrency. However, if they are enabled we may then express concurrency as a dual concept to conflict and causality. We return to this point in the concluding discussion (Section 6).

### 3.2 Modelling forward behaviour of a transaction

In describing the behaviour of a transaction we are not interested in all possible vectors \(V_{\Sigma}\), over a given \(\Sigma\), but rather in an appropriate subset which describes the desired behaviour of the transaction in terms of the orderings of the service invocations that take place during its execution. This provides a global definition of the common ordering conditions and constraints within the multi-party conversation among collaborating online services in a long-running transaction - see the *conversational* part of the WS-CDL specification for choreography [56].

In previous work [35] we have described a formal translation of UML 2 sequence diagrams for obtaining the appropriate subset of all possible vectors \(V_{\Sigma}\) in our behavioural description, incidentally giving a true concurrency semantics to the **par** interaction fragment found in UML 2 [42]. This can be readily applied to obtain the appropriate subset of \(V_{\Sigma}\) for a long-running transaction. The formal construction of the translation takes up ideas found in [27], where sequence diagrams were interpreted over *labelled prime event structures* [57], and uses the concept of a *location* along lifelines in a sequence diagram to map the receiving of a message (invocation / request) at each location onto the vector(s) that result from the concatenation of the corresponding action vector and the vector(s) at the preceding location. The very first location corresponding to the start of the lifelines is mapped onto \(\Delta_{\Sigma}\).

The interactions described in a UML 2 sequence diagram are translated into a set of transaction vectors, as shown in Fig. 3. Due to space limitations we omit further details and refer to [35] for a comprehensive treatment of the formal translation.

The key outcome of the translation is that the orderings between actions on different coordinator components of a transaction manifest themselves in the order structure of the set of vectors obtained from the interaction scenario described in the UML 2 sequence diagram.
We use the term *transaction language* to refer to such a subset of transaction vectors over a given signature. We may now formally define a long-running transaction as follows.

**Definition 3.8 [Transaction]** A transaction $T$ is a pair $(\Sigma, V)$ where $\Sigma$ is the signature of $T$, and $V \subseteq V_\Sigma$ is the transaction language of $T$.

The idea is that the transaction language associated with a given transaction, expresses the ordering constraints necessary in the coordination of the underlying services. For example, in the transaction of Fig. 3, following the service invocation $a_1$, there is a choice between invoking $b_1$ on the coordinator component $CC_2$ and invoking $a_2$ on $CC_1$. Whenever $b_1$ happens, it is followed by $a_3$ on $CC_1$ whereas $a_2$ is followed by $c_1$ and $b_2$, which are invoked concurrently on $CC_3$ and $CC_2$ respectively.

The order structure of the transaction language determines the pattern the underlying service invocations between the coordinator components in the transaction should follow. The idea is that starting with the empty vector, and following the pattern of Fig. 3, each subsequent action (or concurrent actions) takes place in going forward until the transaction as a whole completes successfully. In the sequel we show how this is driven by the mathematical construction underlying the formal semantics for the long-running transaction. In the event of a failure, the order structure again determines the pattern the compensating actions should follow in returning to the empty vector, as we will see in Section 4.2.

In [36] we provided a schema for deriving the XML description of the order structure of a transaction language. When these so-called *transaction scripts* are parsed they provide the full *transaction history*, which is the result of the actual deployment of the transaction, as reflected on the associated formal semantics.

We have seen that transaction vectors are obtained by coordinate-wise concatenation. Hence, a transaction language can be seen to be built up starting from the empty vector by a series of concatenations with action vectors which represent actions. The study of vector semantics in [52] shows that this is the case when the language satisfies certain properties.

In short, these relate to finiteness in discrete event systems and ensuring that each earlier part of a behaviour is itself a behaviour. Such requirements for the well-formedness of the behavioural description of a transaction can be expressed in terms of order-theoretic properties of transaction languages as follows.

**Definition 3.9 [Discreteness]** Let $V \subseteq V_\Sigma$ be a transaction language, then $V$ is discrete iff $\Delta_\Sigma \in V$ and whenever $u, v, w \in V$ such that $u, v \leq w$ then (i) $u \cup v \in V$ and (ii) $u \cap v \in V$. 

![Figure 3: Translating behavioural scenarios into transaction vectors](image)
Note that \( u \sqcup v \) is understood as asserting that \('\sqcup'\) is defined. The property restricts to transaction languages that are **consistently complete posets**, in the usual sense of domain theory [14]. It further requires that the glb '\( \sqcap \)' belongs to \( V \). The condition that \( \Lambda_\Sigma \in V \) excludes infinite descending chains of actions with respect to time ordering, and also reflects that there is an initial point in transaction execution where nothing has happened yet.

Discreteness imposes a finiteness constraint in the sense that only a finite number of actions can occur within a finite amount of time. It is reminiscent of the finiteness assumption in **prime event structures** [57]. In fact, it ensures that situations like those resulting in Zeno-type paradoxes will not arise in the obtained behavioural description. The presence of one start and one or more end points in execution is guaranteed and this is important in asserting completion. As we will see in Section 4, completion of a transaction is understood as either arriving at (one of) the maximal vectors (successful termination) or the empty vector \( \Lambda_\Sigma \) (recovery upon failure).

We further require that every occurrence of an action (e.g., service invocation) is recorded in the set of vectors associated with the transaction.

**Definition 3.10 [Local left-closure]** Let \( V \subseteq V_\Sigma \), \( i \in C \) and \( x \in \mu(i)^* \), then \( V \) is locally left-closed iff, whenever \( v \in V \) and \( \Lambda < x \leq v(i) \) then there exists \( u \in V \) such that \( u \leq v \) and \( u(i) = x \).

The local left-closure property guarantees that any earlier part of behaviour is itself a behaviour and is intended to resolve ambiguities that may arise from considering ‘glued’ actions and not having enough vectors in the language to describe the course of the behaviour in question. It is defined on each coordinate of the vectors in a transaction language, hence we refer to it as **local** left-closure. A locally left-closed transaction language ensures that it is appropriate to talk about the full **history** of the transaction.

**Definition 3.11 [Normal transaction languages]** A transaction language \( V \subseteq V_\Sigma \) over a signature \( \Sigma \) of a transaction \( T \) is normal iff it is locally left-closed and discrete.

This reflects the fact that the guarantees that accrue from these properties are embedded in the behaviour of the corresponding transaction. In fact, discreteness and local left-closure ensure the well-formedness of the behavioural description of a transaction in our model, and guarantee that this captures the transaction history, in the usual sense of [21]. This can then be exploited in recovering a transaction upon failure, as will be shown in Section 4.

Further, in checking against normality prior to deployment we may determine whether the transaction will exhibit the desired behaviour when executed or, on the contrary, other non-desirable or simply unthought scenarios of execution are also possible. This draws upon previous work on the kind of vector languages considered here and interaction scenarios in [35], which was subsequently applied to refining **transaction scripts** in modelling the forward behaviour of a transaction in [36].

**4 Recovery of long-running transactions**

In a digital business transaction environment, such as a DE, an additional dimension to the coordination of the underlying service invocations (multi-party conversation) has to do with **recoverability**. A long-running transaction that has been generated in response to a business request involving online services, e.g., as described in [30], will have a long execution period
and comprises a logical unit of work, which means that it should either be executed as a whole or not at all (atomicity, see [21]). If some action (e.g., service invocation, HTTP operation on a resource) fails during execution, then all previously successful actions should be compensated so that their effects on the corresponding participants of the transaction are effectively undone. The compensating actions are typically performed in the reverse order, much in the same way as done in conventional (data-centric) transactions on databases - this concept is central to Sagas [20], which underline the recovery schemes applied to transactional databases.

So far we have given a formal description of a long-running transaction and have shown how its forward behaviour can be modelled in terms of the orderings of the underlying service invocations (choreography of the online services involved). In standard transactional terms [21], this allows to determine the history of the transaction based on the associated transaction semantics. We may now proceed to show how this is used for the recovery of a long-running transaction whenever some failure during its execution makes this necessary.

4.1 Modelling compensations

It transpires that in order to address the recovery of a long-running transaction we need to precisely characterise the start of the transaction (or initial configuration) and the successful completion of the transaction (when all forward actions in the corresponding multi-party conversation have occurred). A failure may occur at any point during its execution.

In terms of the sequence diagram in Fig. 1 shown earlier, the initial configuration of the transaction corresponds to the very start of the lifeline (the dotted vertical line on the diagram) for each participant while the successful completion of the transaction execution corresponds to reaching the very end of the lifeline for each participant (end of diagram).

In terms of the corresponding transaction language, whose order structure is shown in Fig. 3 (right), the initial configuration is represented by the empty vector $\Lambda \Sigma$ while the successful completion of the transaction is described in our semantics by (one of) the maximal vectors in the language.

We have seen that $(V \Sigma, \leq)$ is a poset and hence will have one or more maximal elements. The largest vectors in $(V \Sigma, \leq)$ describe maximal behaviour of the transaction, in the sense that they do not describe an earlier part of behaviour than any other vector in the language does. This may be put formally in the following definition.

**Definition 4.1 [Maximal vectors]** Let $T = (\Sigma, V)$ be a transaction. A transaction vector $v \in V$ is maximal in $V \subseteq V \Sigma$ iff there is no other vector $u \in V$ such that $v \leq u$.

In other words, the forward execution of the whole scenario corresponds to reaching a maximal vector. In the transaction language of Fig. 3 there are two maximal vectors, namely $v_1 = (a_1a_3, b_1, \Lambda)$ and $v_2 = (a_1a_2, b_2, c_1)$. This means that one allowed ordering of invocations is $a_1 \rightarrow b_1 \rightarrow a_3$ and the other is $a_1 \rightarrow a_2 \rightarrow (b_2$ concurrently $c_1)$. Note that both allowed sequences end up in a maximal vector in $V$, which corresponds to the behaviour exhibited when the transaction executes successfully and terminates (transaction execution complete).

When the transaction is actually deployed, then only one of the allowed sequences of actions can occur. In our example, this would depend on how the choice between actions $a_2$ and $b_1$, after $a_1$ has occurred, is resolved in each run of the transaction. Note that in Fig. 3 which describes the interactions within the forward behaviour of the transaction in question,
while there seem to be three pathways there are actually only two different sequences of actions that lead to completion of the transaction execution, as signified by arriving at one of the two maximal vectors. We return to this point in Section 4.3 where we define transaction execution histories and formally characterise equivalent execution histories.

Fig. 4 shows the compensating actions that need to be taken when a failure occurs at the very last part of the interaction scenario described in the transaction of Fig. 3. The compensating sequence diagram highlights the fact that the compensating actions, denoted by /a for a forward action a, need to take place in the reverse order, as done in Sagas [20] for long-lived transactions in databases. Note that this does not necessarily mean that /a is also a single action or that it completely 'undoes' a. It may in itself be a sequence of actions, in which case our approach to modelling forward actions would again apply.

Figure 4: Compensating for the interaction of Fig. 1

In fact, the WS-CDL choreography defines a construct < /exception > (WS-CDL is an XML-based standard language), which allows to define an exception work unit that is executed when a fault is encountered - see Section 4.2.1 in the WS-CDL specification [56].

4.2 Modelling compensating behaviour of a transaction

In our approach, the recovery of a long-running transaction in the event of a failure, including failure in some subtransaction, comes down to compensating for the forward actions in the reverse order, as advocated in Sagas [20].

When considering services from disparate organisations collaborating in the absence of a central controlling instance, the exception work units for each online service are defined by each provider on their local platform. What the compensating semantics provides is a systematic way to record when such work units need to be executed, and at which participant, whenever a failure in the multi-party conversation makes this necessary.

We have seen that the prefix ordering relation on vectors (Definition 3.5) captures causality in the usual sense that if \( u \leq v \) then, \( u \) describes an earlier part of behaviour than that described by \( v \) (or in normal transaction languages, as we will see in Corollary 4.1, \( u \) is a prefix of the series of concatenations of actions vectors described by \( v \)). From this relation, a further ordering relation can be derived that captures immediate causality.
Definition 4.2 [Covers] Let \( u, v \in V \subseteq V_\Sigma \). We say that \( v \) covers \( u \) in \( V \), and we write \( u \prec v \) iff (i) \( u \leq v \) and \( u \neq v \) and (ii) If \( z \in V \) such that \( u \leq z \leq v \), then \( z = u \lor z = v \).

The covers relation provides an ordering among transaction vectors in a language \( V \) in which one is immediately beneath the other, allowing no other vector in \( V \) to exist in between them. In other words, it identifies immediate predecessors (dually, immediate successors) of a vector. Note that the covers relation is not defined globally, but for a specific subset \( V \) of \( V_\Sigma \), however we will omit the subscript \( V \) when this is clear from context. We will use \( \text{Pre}_V(v) = \{ u \in V | u \prec v \} \) to denote the set of immediate predecessors of the vector \( v \) in \( V \).

We may now introduce a cancellation operation on transaction languages that is central to the handling of compensations in our model. In fact, two forms of cancellation are required; one to identify the last action that went into forming a given vector in the language, and another to remove that last action from a vector, starting from the vector describing the last successful forward action before failure occurred in the transaction.

We start by defining a left-cancellation operation to determine the common suffix of ordered vectors in a transaction language. This is done by considering the usual operation of left-cancellation on sequences, which removes the first occurrence of an element in the sequence starting from the left hand side. This is lifted onto vectors, by applying it coordinate-wise, as follows.

Definition 4.3 [Left-cancellation] Let \( u, v \in V_\Sigma \). If \( u \leq v \), then we define \( v/u \) to be the unique element \( z \in V_\Sigma \) such that \( u \lor z = v \).

Effectively, the left-cancellation operation ' \( / \) ' isolates the behaviour that arises in between vectors, so that if \( u \) is a transaction vector describing an earlier part of the behaviour described by \( v \), i.e., \( u \leq v \), then \( v/u \) is the 'continuation' of \( u \) that extends it to \( v \).

Furthermore, with respect to uniqueness of \( z \), we state the following. Let \( z' \in V_\Sigma \) be another vector (so \( z' \neq z \)) such that \( u \lor z' = v \). Then, for each \( i \), we have \( u(i), z'(i) = v(i) \), so that \( z'(i) = v(i)/u(i) \). But we also have that \( v(i)/u(i) = z(i) \), for each \( i \). Thus, \( z'(i) = z(i) \), for each \( i \), which means that \( z' = z \), hence establishing uniqueness.

In performing the compensations of a transaction whenever a failure occurs, our interest lies with isolating the last action that went into forming each vector up to that point rather than the 'continuation' from any earlier part of behaviour. It turns out that in a normal (discrete, locally left-closed; Definition 3.11) transaction language, if \( u \) is an immediate predecessor of \( v \), i.e., \( u \in \text{Pre}_V(v) \), then the application of the left-cancellation ' \( / \) ' produces an action vector. In fact, left-cancellation isolates precisely the last action that went into obtaining \( v \) from \( u \). This result is summarised in the following proposition.

Proposition 4.1 Let \( u, v \in V \subseteq V_\Sigma \), where \( V \) is a normal transaction language over a signature \( \Sigma = (C, \mu) \). If \( u \prec v \), then \( v/u \in A_\Sigma \).

Proof.
Since \( u \prec v \), we have by Definition 4.2 that \( u \neq v \), hence \( v/u \neq A_\Sigma \). We also have by Definition 4.2 that \( u \leq v \), which means that the sequences appearing on some (at least one of the) coordinates of \( u \) are prefixes of those appearing on the respective coordinates in \( v \). We denote the set of such coordinates by \( C_1 \). Since \( u \leq v \), the two vectors agree on the rest of the coordinates, i.e., \( v \) cannot have a prefix of the sequence appearing on \( u \) on the respective
coordinate. Let us denote the coordinates on which the two vectors have the same sequence by \( C_2 \). Hence, \( C_1, C_2 \subseteq C \) and \( C_1 \cup C_2 = C \) and \( C_1 \cap C_2 = \emptyset \). Also, \( C_1 \neq \emptyset \).

We want to show that \( \nu/\mu \in A_\Sigma \).

If \( \nu/\mu \notin A_\Sigma \), then on at least one of the coordinates in \( C_1 \) the difference between the sequence appearing on \( \nu \) and its prefix appearing on \( \mu \) will be at least two elements. If on all sequences in \( C_1 \) the vector \( \nu \) has only one additional element, then \( \nu/\mu \in A_\Sigma \) and the proof is done. Thus, for some \( i \in C_1 \), we have \( \nu(i) = \nu(i).w_1.w_2 \), where \( w_1, w_2 > \Lambda \). For all \( i \in C_2 \), we have \( \mu(i) = \nu(i) \).

Now, for each \( i \in C_1 \) such that \( \nu(i) = \nu(i).w_1.w_2 \), where \( w_1, w_2 > \Lambda \), we have that since \( V \) is normal it is also locally left-closed, hence by Definition 3.10 (and take \( u(i).w_1 \) as \( x \), for each \( i \)) there exists \( w \in V \) such that \( \nu(i) \preceq w(i) \) and \( \nu(i) = \nu(i).w_1 \). For all \( i \in C_2 \), we have \( \nu(i) = \nu(i) \).

Next, we have that \( \mu \) is normal, and hence, by Definition 3.9 we have \( \zeta \in V \), and also \( \mu \preceq \zeta \). But now we also have that, for \( i \in C_1 \), we have \( \nu(i) \preceq \nu(i).w_1 \). Since \( \zeta = \mu \cup w \), by Definition 3.6 we have \( z(i) = \text{max}(\nu(i), w(i)) \), for each \( i \), from which we can deduce that \( \nu(i) \preceq z(i) \), for all \( i \in C_1 \), hence \( \nu(i) < z(i) \), for all \( i \in C_1 \). For all \( i \in C_2 \), we have \( \nu(i) = \nu(i) \) and also \( \mu(i) = \nu(i) \). Thus, we have that \( w(i) = z(i) \), for all \( i \in C \). Since \( \zeta = \mu \cup w \), we also have that \( \nu \preceq \zeta \) and \( \mu \preceq \zeta \).

This implies that \( \mu \preceq \zeta \preceq \nu \), and \( \mu \neq \nu \) and \( \zeta \neq \nu \), which is a contradiction (since \( \mu \preceq \nu \).

This result dictates that what takes a transaction vector and extends it to its successors is an action vector (e.g., service invocation between different participants) of the transaction. This result defines a transition structure which can be exploited in associating transaction languages with a class of automata that can capture true concurrency. These can be seen as an elaboration of Asynchronous Transition Systems (ATS) [3, 50] and specialisations of Hybrid Transition Systems [52]. This has been done for the adaptation of vector languages considered here in the context of component-based software design in [37].

For the purposes of modelling compensating behaviour in long-running transactions here, this result allows to deduce that vectors are built up by a series of concatenations with action vectors, starting from the empty vector \( \Lambda V \).

**Corollary 4.1** Let \( \mu, \nu \in V \subseteq V_\Sigma \), where \( V \) is a normal transaction language over a signature \( \Sigma \), and \( \mu, \nu \in V \) such that \( \mu \preceq \nu \), then there exists \( \alpha_1, \ldots, \alpha_n \in A_\Sigma \) such that:

1. \( \mu.\alpha_1 \ldots.\alpha_n = \nu \)
2. \( \mu.\alpha_1 \ldots.\alpha_i \in V \), \( i = 1, \ldots, n \)
3. \( \mu \preceq \nu.\alpha_1 \) and \( \nu.\alpha_1 \ldots.\alpha_{i-1} \preceq \nu.\alpha_1 \ldots.\alpha_i \), \( i = 2, \ldots, n \)

**Proof.** If \( \mu \preceq \nu \), then the corollary holds with \( n = 1 \) and \( \alpha_1 = \nu/\mu \) by Proposition 4.1.

Otherwise, the set \( Y = \{ \nu \in V|A_\Sigma \leq \nu \preceq \nu \} \) of all vectors below \( \nu \) is non-empty (it contains \( A_\Sigma \) since \( V \) is normal, and hence also discrete, by Definitions 3.11 and 3.9), it is finite, and thus contains a maximal element \( w \), for which \( \mu \preceq w \). Hence, we have \( \mu \preceq \nu \preceq w \).

By induction, there exists \( \alpha_1, \ldots, \alpha_{n-1} \in A_\Sigma \) such that \( \nu.\alpha_1 \ldots.\alpha_{n-1} = w \), \( \nu.\alpha_1 \ldots.\alpha_i \in V \), for \( i = 1, \ldots, n-1 \), and \( \mu \preceq \nu.\alpha_1 \) and \( \nu.\alpha_1 \ldots.\alpha_{i-1} \preceq \nu.\alpha_1 \ldots.\alpha_i \), for \( i = 2, \ldots, n-1 \).
If we now let \( \alpha_n = v/w \), then by Definition 4.3 we have that there exists unique \( z \in V \subseteq V_\Sigma \) such that \( w.z = v \), and by Proposition 4.1 we have that \( z = \alpha_n \), and \( \alpha_n \in A_\Sigma \). The fact that \( w.\alpha_n = v \) means that \( u.\alpha_1...\alpha_n \in V \), and we also have that \( u.\alpha_1...\alpha_{n-1} = w \prec_V u.\alpha_1...\alpha_n = v \). We now see that \( \alpha_1,...,\alpha_n \) have the desired properties. \( \square \)

The corollary says that effectively vectors in normal transaction languages are obtained by repeatedly concatenating action vectors, and the vectors obtained in this way are also behaviours of the transaction, and each is the immediate continuation of the previous one.

Point (1) says that if \( u \) is an earlier part of the behaviour described by \( v \) in a transaction, then a series of concatenations with action vectors is what takes \( u \) and extends it to \( v \). Point (2) says that the vectors obtained after each concatenation, are also behaviours of the transaction. Point (3) then says that at each step, after each concatenation with an action vector, the resulting transaction vector *covers* (or, sits immediately on top of) the previous vector.

In order to address the recovery of a transaction whenever a failure occurs during its execution, we need to compensate for the forward actions up to that point, and do that in the reverse order. For this reason, we may now define a *right-cancellation* operation on a transaction vector to remove the last action (action vector) that went into forming it from its immediate predecessor(s). This is done by considering the usual operation of *right-cancellation* on sequences, which removes the first occurrence of an element in a sequence starting from the right hand side. This is lifted onto vectors, by applying it coordinate-wise, as follows.

**Definition 4.4 [Right-cancellation]** Let \( v \in V \subseteq V_\Sigma \), \( V \) is normal, and \( \alpha_1,...,\alpha_n \in A_\Sigma \). If \( v = u.\alpha_1...\alpha_n \), some \( u \in V \), then we define \( v \div \alpha_n = u.\alpha_1...\alpha_{n-1} \).

The corollary says that effectively vectors in normal transaction languages are obtained by repeatedly concatenating action vectors, and the vectors obtained in this way are also behaviours of the transaction, and each is the immediate continuation of the previous one.

Point (1) says that if \( u \) is an earlier part of the behaviour described by \( v \) in a transaction, then a series of concatenations with action vectors is what takes \( u \) and extends it to \( v \). Point (2) says that the vectors obtained after each concatenation, are also behaviours of the transaction. Point (3) then says that at each step, after each concatenation with an action vector, the resulting transaction vector *covers* (or, sits immediately on top of) the previous vector.

In order to address the recovery of a transaction whenever a failure occurs during its execution, we need to compensate for the forward actions up to that point, and do that in the reverse order. For this reason, we may now define a *right-cancellation* operation on a transaction vector to remove the last action (action vector) that went into forming it from its immediate predecessor(s). This is done by considering the usual operation of *right-cancellation* on sequences, which removes the first occurrence of an element in a sequence starting from the right hand side. This is lifted onto vectors, by applying it coordinate-wise, as follows.

**Definition 4.4 [Right-cancellation]** Let \( v \in V \subseteq V_\Sigma \), \( V \) is normal, and \( \alpha_1,...,\alpha_n \in A_\Sigma \). If \( v = u.\alpha_1...\alpha_n \), some \( u \in V \), then we define \( v \div \alpha_n = u.\alpha_1...\alpha_{n-1} \).

Note that right-cancellation is defined in a normal (discrete, locally left-closed; Definition 3.11) transaction language and removes the last in the series of concatenations with action vectors that formed \( v \) in the first place (during forward behaviour of the transaction). Successive applications of \( \div \) remove each action in turn from \( v \).

The fact that transaction vectors (behaviours) are built up by a series of concatenations with action vectors, starting from the empty vector \( \Lambda_\Sigma \), is used to model the forward behaviour of the transaction, as discussed in Section 3.2. It also means that successive applications of the cancellation operations (Definition 4.3, Definition 4.4) can be used to express the series of compensating actions required whenever some failure occurs during execution of the transaction.

In other words, in our approach, (coordinate-wise) concatenation of vectors is used to model the occurrence of a forward action by

\[ u.\alpha = v \]

and, instead of introducing a separate notation and associated semantics for the recovery of a transaction upon failure, left- and right-cancellation is used to express the compensating action needed at each point in returning to the initial configuration \( \Lambda_\Sigma \).

In particular, left-cancellation (Definition 4.3) is applied to the vector where failure occurred and its immediate predecessor(s) (which are given by Definition 4.2). By Proposition 4.1, in a normal transaction language, this produces an action vector which is the last one that went into forming \( v \). Hence, for \( u.\alpha \prec_V v \) we have,

\[ v/u = \alpha \]
Next, right-cancellation (Definition 4.4) is applied to the vector in question to remove this action. This produces the immediate predecessor of $v$. Hence,

$$v \div (v/u) = v \div \alpha = u$$

Now left-cancellation is applied between $u$ and its immediate predecessor(s), producing the last action that went into forming $u$. This is followed by right-cancellation on $u$ to remove the last action, and so on.

Fig. 5 shows the case where a failure occurs after the invocation $a_2$ on the online resource managed by component $CC_1$. This means that it is no longer possible for the transaction to complete its execution by reaching the maximal vector $(a_1a_2, b_2, c_1)$, as dictated by the allowed sequence of actions in its design (recall Fig. 1). Consequently, the transaction needs to be recovered and this implies returning to the configuration the system was in before the transaction started. As mentioned earlier, this is represented by the empty vector $\Lambda$ in our model. In returning to $\Lambda$, all previously successful forward actions need to be compensated for. This is done in the reverse order by successive applications of the ’/’ (left-) and ’\div’ (right-cancellation) operations defined earlier (Definition 4.3 and Definition 4.4, respectively).

![Diagram](image-url)

**Figure 5: Full recovery of a transaction**

The first application of the two cancellation operations moves the transaction back to $(a_1, \Lambda, \Lambda)$ and the next to $(\Lambda, \Lambda, \Lambda)$. Hence, the transaction is returned to $\Lambda$. In this case we talk about full recovery of the long-running transaction.

In further explanation, the first application of the cancellations is performed as follows. Since $\text{Prev}((a_1a_2, \Lambda, \Lambda)) = \{(a_1, \Lambda, \Lambda)\}$ by applying the left-cancellation operation on these two vectors we have,

$$(a_1a_2, \Lambda, \Lambda) / (a_1, \Lambda, \Lambda) = (a_2, \Lambda, \Lambda)$$

which means it is the action vector $\alpha = (a_2, \Lambda, \Lambda)$ that is the last that went into forming $(a_1a_2, \Lambda, \Lambda)$. The right-cancellation is now applied to this vector and the action vector obtained by ’/’, i.e., $(a_2, \Lambda, \Lambda)$, and we have,

$$(a_1a_2, \Lambda, \Lambda) \div (a_2, \Lambda, \Lambda) = (a_1, \Lambda, \Lambda)$$

Similarly, the next application of the two operations, ’/’ and ’\div’, on the vector obtained by the application of ’/’ now, i.e., $(a_1, \Lambda, \Lambda)$, will take the transaction back to $\Lambda$.  

20
4.3 Execution histories

We have seen that if some action (e.g., service invocation, HTTP operation on a resource) fails during execution of a transaction, then all previously successful actions should be compensated so that their effect on the different participants of the transaction are effectively undone. The compensating actions are typically performed in the reverse order. This means that in transaction recovery we need to know the sequence of forward actions that took place up to the point that failure made further progress impossible.

In terms of the corresponding transaction language, we have seen that the initial configuration (before the transaction was started) is represented by the empty vector $\Lambda \Sigma$ while the successful completion of the transaction is described in our semantics by one of the maximal vectors in the language (Definition 4.1). We have also seen that transaction vectors are built up by repeatedly concatenating with action vectors (Proposition 4.1, Corollary 4.1), following the order of the actions as these appear in the corresponding transaction tree and/or interaction scenario specification (recall Fig. 1, Fig. 3) as given by the practitioner. These sequences of concatenations with action vectors effectively describe the orderings of the forward actions in transaction execution.

**Definition 4.5 [Execution histories]** We define sequences such as $\alpha_1 \ldots \alpha_n$ in Corollary 4.1 as t-sequences from $u$ to $v$. In the case, $u = \Lambda \Sigma$ and $v$ is a maximal vector, we describe such sequences as execution histories of a transaction $T = (\Sigma, V)$.

Hence, the execution histories correspond to the series of concatenations with action vectors that take $\Lambda \Sigma$ and extend it to a maximal vector. Successive applications of cancellation operations (Definitions 4.3, 4.4) on a maximal vector allows us to work backwards along an execution history and correspond to the series of compensations that take $v$ back to $\Lambda \Sigma$.

The set of execution histories of a transaction records the sequences of forward actions that need to take place in completing the multi-party conversation involved in a long-running transaction. Note that we talk about a set because of the conflict structure that arises whenever there is a choice between actions in the design of a transaction.

We may now define equivalent execution histories in the model much in the same way that equivalent classes of traces are defined in Mazurkiewicz trace languages.

**Definition 4.6 [Equivalent execution histories]** Let $x, y$ be execution histories of a transaction $T = (\Sigma, V)$. We define $x$ and $y$ to be equivalent execution histories, and write $x \equiv_{ind} y$, where $\equiv_{ind}$ is the least equivalence relation that includes $\equiv^{(1)}_{ind}$ which is given by

$$x \equiv^{(1)}_{ind} y \iff \exists u, v \in A^* \Sigma, \exists \alpha, \beta \in A \Sigma : \alpha \ ind \beta \land x = u \alpha \beta v \land y = u \beta \alpha v$$

Hence, execution histories that differ only in the order of consecutive action vectors that are independent, describe the same behaviour. In this behaviour, the corresponding actions are concurrent. When a failure makes it necessary to apply transaction recovery, we only need to compensate for one of the execution histories in the corresponding equivalence class of histories $< x >_{ind}$ of the transaction.

Fig. 6 shows the case where $a_2$ occurred after $a_1$, and hence the right branch of the corresponding Hasse diagram was actually deployed. It can be seen that each vector in turn
is obtained by coordinate-wise concatenation with the appropriate action vector, according to the allowed sequence of actions along the particular execution history.

In Fig. 6 it appears that there are two execution histories corresponding to the rightmost branch of the associated Hasse diagram, but these are the result of concurrency (between $b_2$ and $c_1$). These actions are both enabled after the behaviour described by $(a_1 a_2, \Lambda, \Lambda)$ and occur consecutively from that point. Therefore, the execution histories $\alpha_1, \alpha_2, \beta_2, \gamma_1$ and $\alpha_1, \alpha_2, \gamma_1, \beta_2$ are equivalent (by Definition 4.6). This is because the series of concatenations with action vectors differ only in the order of the independent action vectors $\beta_2 = (\Lambda, b_2, \Lambda)$ and $\gamma_1 = (\Lambda, \Lambda, c_1)$, which occur consecutively, and consequently are concurrent. This means that they describe the same behaviour of the transaction and not two different behaviours.

The benefit of a model of true concurrency, and being able to identify equivalent execution histories, can perhaps be seen most clearly when it comes to compensation in transaction recovery, where it is only required to compensate once for all equivalent execution histories.

**Example 4.1** Consider the transaction of Fig. 3 and assume that failure happened after $(a_1 a_2, b_2, c_1)$. Our approach allows for the concurrent forward actions $b_2$ and $c_1$ to be compensated concurrently. This is because the vector $(a_1 a_2, b_2, c_1)$ has two immediate predecessors, and hence the independent action vectors $\beta_2 = (\Lambda, b_2, \Lambda)$ and $\gamma_1 = (\Lambda, \Lambda, c_1)$ that extended them to $(a_1 a_2, b_2, c_1)$ would be called to be compensated for concurrently. In other words, by applying left-cancellation (Definition 4.3) on $v = (a_1 a_2, b_2, c_1)$ and its immediate predecessors $u_1 = (a_1 a_2, b_2, c_1)$ and $u_2 = (a_1 a_2, \Lambda, c_1)$ we have,

$$v / u_1 = (a_1 a_2, b_2, c_1) / (a_1 a_2, \Lambda, c_1) = (\Lambda, b_2, \Lambda) = \beta_2$$

and also,

$$v / u_2 = (a_1 a_2, b_2, c_1) / (a_1 a_2, b_2, \Lambda) = (\Lambda, c_1) = \gamma_1$$

Now, right-cancellation (Definition 4.4) on $v$ and $\beta_2, \gamma_1$, respectively, takes the transaction back one step to the immediately preceding vector, i.e., we have

$$v / \beta_2 = (a_1 a_2, b_2, c_1) / (\Lambda, b_2, \Lambda) = (a_1 a_2, \Lambda, c_1)$$

and also,

$$v / \gamma_1 = (a_1 a_2, b_2, c_1) / (\Lambda, \Lambda, c_1) = (a_1 a_2, b_2, \Lambda)$$
The mechanism proceeds in the same way from \((a_1a_2, \Lambda, c_1)\) and \((a_1a_2, b_2, \Lambda)\) to return to \((a_1a_2, \Lambda, \Lambda)\), and so on.

The notion of independence between actions is what allows to identify equivalent behaviours in the model, something that is not possible when adopting an interleaving model, as done in \(cCSP\) \cite{12} or \(\pi\)-\(calculus\) \cite{4}. Concurrent actions in these approaches are modelled by considering all possible combinations of their sequentialisations. When it comes to compensating for them in the reverse order, the nondeterministic choice between all possible sequentialisations is again available, thus all traces need to be compensated for in transaction recovery. These approaches are discussed further in the related work section (Section 5).

### 4.4 Forward Recovery

We have seen how right-cancellation can be applied to generate compensating sequences of actions for the full recovery of a transaction. Full recovery however, can be costly in terms of resources, delays, even business relations. Additionally, in a highly transactional environment dependencies may also exist across transactions so the effect of a recovered transaction may be magnified. For this reason it is desirable to avoid full recovery wherever possible.

One way to do this is to design transactions with alternative scenarios of execution; in other words, allow for multiple execution histories in the corresponding transaction. In such cases, our approach supports the provision for **forward recovery** which is a mechanism for avoiding full recovery. The aim is to explore during the compensation of a given execution history whether there is a possibility for successfully completing the transaction following a different execution history to the one originally deployed, instead of compensating for the whole original execution history.

#### Figure 7: Forward recovery of a transaction

As shown in Fig. 7, in going backwards, and while compensating for one action (or a set of concurrent actions) at a time, we look at each point whether there is an alternative path from the vector we arrived on (after applying the compensation) leading to a maximal vector. This is possible in our approach because all the execution histories are captured in
the corresponding transaction language. In recovering a given execution history which failed (as described in Section 4.2), after the next forward action(s) is compensated for we check whether the resulting vector is part of a different execution history. If this is the case, then we attempt to go forward by performing the concatenations in accordance with the allowed sequence of forward actions in that execution history, as described in Section 3.2.

In further explanation of the algorithm sketch shown in Fig. 7, we may add the following. Upon failure at the point described by a transaction vector $u_0 = (a_1a_2,A,A)$ here, the set of immediate predecessors $\text{Pre}_V(u_0)$ (recall Definition 4.2) is checked to see whether it contains any vector that belongs to the downset of some other vector $v$ in $V$ (given by $\downarrow v = \{w \in V \mid w \leq v\}$) which is not smaller than $u_0$, i.e., $v \not\leq u_0$. If there is no such vector in $\text{Pre}_V(u_0)$, or no such $v$, then the compensation is applied to $u_0$ as usual (recall Section 4.2), and the check is repeated for its immediate predecessor(s) in the corresponding execution history. If there is such a vector, say $z$, then the forward action(s) that extend this vector $z$ to its immediate successor(s) (given by Definition 4.2) are performed in moving the transaction forward again. If another failure is experienced, then the whole process is repeated for another vector from the set $\text{Pre}_V(u_0)$. If the set is exhausted, then the full recovery of the transaction is unavoidable.

5 Related work

Despite the great interest in defining business requests in terms of finer-grained subtasks that are available as loosely-coupled online services, the Web Services community has not reached a common agreement on a unique notion of this form of long-running transaction. Standardisation proposals for service orchestration have culminated in the Web Services Business Process Execution Language (WS-BPEL) [19] which actually builds on Microsoft’s XLANG [53] and IBM’s WSFL. Standardisation activities for choreography are embodied in W3C’s Web Services Choreography Description Language (WS-CDL) [56], which builds on earlier work on the Web Services Choreography Interface (WSCI) [55], again by W3C.

WS-BPEL [19] (service orchestration) and WS-CDL [56] (service choreography), use the concept of a long-running or web transaction and include modelling constructs for failure and compensations (e.g., exception work units), but do not support a definitive mechanism for recovering a transaction. As a result, this has to be dealt with in ad-hoc ways by the application programmer. This is also evident in mainstream transaction protocols for business activities such as the Business Transaction Protocol (BTP) [17] and Web Services Transaction (WS-Tx) [18] (which comprises the WS-Coordination, WS-AtomicTransaction and WS-BusinessActivity specifications). A critical comparison between these protocols for long-lived business activities can be found in [29].

Overall, different proposals have different interpretations of a long-running transaction hidden in the informal nature of their documentation, e.g., WS-CDL is an XML-based language specification. This has resulted in a strand of research work on formal semantics for long-running transactions that involve the composition of online services. The work in [12] is motivated by XLANG [53] (Microsoft’s predecessor to WS-BPEL) and extends the trace semantics of CSP [24] to model compensations in long-running transactions. Preceding this was work on the StAC language [11], which was inspired by BPBeans, a framework for modelling business processes integrated into WebSphere. Compensations in this work need to be activated through special primitives however, which are not related directly to the failure or success of the activities of the underlying services.
The work in [4] uses \( \pi \)-calculus for modelling long-running transactions and the resulting \( \pi t \)-calculus targets BizTalk, the visual environment of XLANG [53]. [8] addresses short-lived transactions in BizTalk. A timed extension to \( \pi \)-calculus, \( \text{web}\pi \), is considered in [28], which does not target a particular standardisation initiative but attempts to formalise key concepts, like our approach does, but in orchestration rather than choreography of the services in a long-running transaction. The interleaving semantics used for this in [28] results in the characterisation of timed bisimilarity, which is aimed at dealing with latency in business activities. Other formal approaches to reasoning about the composition of web services and transaction mechanisms include [33] which uses interface automata [15] and LTL to describe and verify properties using SPIN, and [22] which proposes an event calculus extension. The latter includes a semantic notion of structural congruence but is interpreted over labelled transition systems, hence falling back onto an interleaving semantics. There are common features in the approaches to describing the behaviour of a transaction such as capturing the sequences of actions that take place, including alternative and concurrent behaviour, as well as performing the compensations in the reverse order, as in linear *Sagas* [20].

Of the formal approaches described above, compensating CSP (cCSP) [12] and \( \pi t \)-calculus [4] are of particular interest here because, like our approach, they consider subcomponents of a transaction or subtransactions within a larger transaction. In concurrent execution, [4] considers multiple copies of the same subtransaction executing concurrently inside a transaction (see 3.2 in [4]). The transaction is allowed to complete even if some subtransaction fails. This may be suitable for certain scenarios, e.g., copies of a subtransaction that await responses to a tender. However, a range of business scenarios in practice require that all subtransactions execute their activities to completion in order to meet the needs of the corresponding business request. For example, in the travel arrangement transaction considered here (this example is also considered in BTP [29]), if the subtransaction responsible for booking a flight fails, then there is no longer a need to book a hotel and schedule transport at the destination.

In cCSP [12], the failure of a concurrent subtransaction causes the recovery of the whole transaction, but the respective compensations are only triggered once all subtransactions have finished their forward actions. In short, synchronisation between processes occurs on terminal events only, there is no inter-process communication. This implies latency in applying the recovery mechanism and, as a result, resources may continue to be unnecessarily allocated to subtransactions which will have to be compensated eventually. The late cancellation of a booking, for example, may lead to payment of a fee so this may also have complicated financial implications, especially when the services involved are controlled by different providers (across organisations), as is the case in a digital business ecosystem environment (Section 1).

We note that [9], which applies a CSS-style interleaving model to formalise *Sagas* [20] for orchestration, avoids this problem by including explicit primitives in the formalism to force abortion of a branch whenever some other parallel branch fails.

Formal approaches that reason about correctness of a *choreography* of services, rather than *orchestration* of a service, are fairly limited in comparison. The work in [10] describes a formal language that is targeted at WS-CDL [56]. The semantics is given in terms of a notion of structural congruence and this is interpreted over transition systems. The language is used to model multi-party conversations but transactional aspects, and recovery management in the event of a failure, are not addressed in this work. To the best of our knowledge, the only other work that deals with service choreography is [7] which targets the Web Services Choreography Interface (WSCI) [55], a predecessor to WS-CDL where the behaviour of a participant is described in terms of the role it plays in the conversation. This work does not
address transactional aspects either. We note that [22] talks about choreography but the approach in fact formalises WS-BPEL, hence it targets orchestration.

It transpires that most work to date on formal semantics of long-running transactions are geared towards the WS-* realisation of online service collaborations and target distributed applications that are controlled by a single entity, which is where the Service Oriented Computing (SOC) promise seems to have been realised. This is in line with the fact that most approaches are concerned with orchestration, where it is more natural to assume a central coordinator (the orchestrator) that is responsible for invoking and combining the single sub-activities in the collaboration required to meet a given business request.

It is also worth pointing out that the interleaving semantics, where concurrency between actions is reduced to the non-deterministic choice between their possible sequentialisations, is perhaps adequate to express concurrency in this view of a collaboration between online services, since the central controlling instance can be used to effectively serialise concurrent interactions. However, such formalisms may not be entirely appropriate in open distributed environments, like the open web, where applications and business requests require a similar collaboration but across organisational boundaries, and do not depend on a central controller. In such scenarios the focus is shifted from the orchestration of a service to the choreography of the multi-party conversation.

For example, as discussed in [31] where we propose a different approach to service composition that builds on declarative technologies, a request for organising a trip in a DE setting would result in a long-running transaction that includes subtransactions for booking a flight, a hotel and local transport. These would execute concurrently, but would also be provided by different organisations (in fact, different ones even for each run of the corresponding transaction). This places stringent requirements in terms of handling concurrent invocations among collaborating services and online resources, which are only reinforced when it comes to compensation in transaction recovery.

When it comes to the series of compensations required in transaction recovery, the non-deterministic choice between all possible interleavings of the sequences of forward actions, considered in the interleaving approach, are again available. Thus, all need to compensated for despite the fact that only one was actually deployed in a given run of the transaction. In contrast, the true concurrency semantics presented in this paper allows to determine equivalent sequences of forward actions (Definition 4.6 of execution histories) hence, when a failure occurs after the behaviour described by $v$, it suffices to compensate for one of the transaction execution histories in the corresponding equivalent class $<v>_{ind}$.

6 Conclusions and future work

We have described a true concurrency semantics for modelling the behaviour of a long-running transaction that coordinates the invocation of multiple online services and resources in meeting a business request (forward behaviour) as well as the series of compensating actions that need to take place whenever some failure makes this necessary (compensating behaviour).

The formal model proposed in this paper is flexible enough to address a range of scenarios in environments such as those considered in digital (business) ecosystems, like the open web, where online services from disparate organisations collaborate to meet a business request without the need for a central portal or controller. Preserving the weak coupling between collaborating services and the absence of a central controller place additional demands on
the formal description of forward and compensating behaviour in their interaction scenarios, particularly with respect to the notation and the handling of concurrency.

The notation afforded by transaction languages allows us to capture the contribution of each participant to the behaviour, at each point during execution, across the whole transaction. This makes it suitable for choreography although our objective is not to give a formal semantics to the WS-CDL standard [56] but to formalise key transactional concepts.

Concurrency is modelled explicitly using the notion of causal independence in the model (Definition 3.7, Fig. 2). The additional structure offered by this allows us to determine equivalent execution histories in a transaction (Corollary 4.1, Definition 4.6). As a result, certain pitfalls that come with following an interleaving approach to modelling long-running transactions are addressed. We consider concurrent execution of subtransactions, where failure of one or more causes the recovery of the whole transaction. The recovery and compensation mechanism is triggered immediately when some subtransaction fails, without locking up additional resources (which could lead to financial loss in terms of opportunity cost). In the event of a failure, only one execution history in the equivalent class of the corresponding behaviour up to that point needs to be compensated. The compensating actions are performed in the reverse order (as in Sagas [20]) while concurrent forward actions are compensated concurrently.

The asynchronous version of vector languages described in this paper is in fact an adaptation of Shields vector languages [52] in that no shared actions are allowed and therefore no synchronisation constraint on the interactions is applied. Hence, \( \mu \) (of Definition 3.1) partitions the set of actions \( M \) instead of being an indexed cover. The work on Vector Controlled Concurrent Systems (VCCS) [38] considers systems in which a fixed number of sequential processes work concurrently but synchronise on shared actions. While the behaviour of a VCCS is described by a vector language consisting of those combinations of individual sequential computations that observe the synchronisation constraints, the behaviour of a transaction (involving concurrently executing web services) in our approach is described by a vector language consisting of those combinations of individual sequential computations (interactions) that observe the ordering constraints found in the design specification of the transaction.

For reasons discussed in the Introduction (autonomous services, range of business scenarios, web-oriented DE architecture, RESTful service interactions instead of RPC-style invocations) it is not reasonable to assume or enforce synchronisation between autonomous online services in long-running transactions executing in the open Web. Therefore, in the vector languages considered here we do not allow shared actions (and hence, there is no synchronisation). Instead of a synchronisation mechanism for shared actions, we consider the specification of the transaction (as given by practitioners in terms of a transaction tree and / or a UML design model, Fig. 1) as the 'mechanism' that imposes constraints on the individual sequential processes that work concurrently during transaction execution.

Transaction languages are a model of true concurrency, based on an independence relation. It is essentially a partial order model - the construction is lifted onto tuples of sequences (one for each participant of the transaction) rather than events, as in the event structures model [40], or individual sequences, as in Mazurkiewicz trace languages [32]. The semantics is based on well known operations on sequences: concatenation for modelling forward behaviour and left- and right-cancellation for compensating behaviour. Operations are applied coordinate-wise to tuples of sequences (transaction vectors), which makes the model amenable to implementation. In a certain important sense, vector languages are to trace languages what matrices are to linear transformations; they afford a more concrete representation which has advantages when it comes to computation or manipulation.
To the best of our knowledge the only other true concurrent model for long-running transactions is [6] where we have described the use of prime labelled event structures [58]. Transaction languages provide a finer-grained notation and a more natural way to describe service invocations between different participants in a transaction.

There is a connection between normal transaction languages and event structures [57]. This builds on a key result in [51] which underlines the relationship between language-theoretic objects (like vector languages) and order-theoretic objects (like event structures). It says that a prime algebraic and consistently complete poset gives rise to an order-theoretic object whose set of occurrences is the set of prime elements of the poset. The characterisation of prime elements in a normal transaction language comes down to vectors that describe a unique last occurrence of an action. For example, in (ii) of Fig. 2 the vector \((a, b)\) is the only vector that is not a prime element in the set. This has been studied in [52] in the context of behavioural presentations, which generalise the event structures model in allowing the ordering relation to be a pre-order (a transitive, reflexive relation) rather than a partial order, thereby allowing the representation of simultaneity as well as concurrency.

A connection to automata-theoretic models builds on the result in Proposition 4.1, which says that what takes one transaction vector and extends it to (one of) its immediate successors is an action vector. This defines a transition structure in a transition system where transaction vectors are states and action vectors are the labels on transitions. This results in class of automata [37] that can be seen as an elaboration of Asynchronous Transition Systems (ATS) [3, 50] and specialisations of Hybrid Transition Systems [52].

Point (2) of Corollary 4.1 is reminiscent of the notion of extensibility found in [46], a local liveness constraint condition that says that whatever the behaviour to date it should be possible for any active participant to progress in the future. In [46], Shields’ vector languages are used to model cyclic behaviour in asynchronous periodic systems and the extensibility property is defined locally (at the vector coordinate level). Having defined normality (Definition 3.11), the extensibility property is deduced to hold globally (at the vector level) for the adaptation of vector languages considered in this paper. If for any vector \(u\), which is not maximal in \(V\), there exists \(v \in V\) that extends it to a maximal vector, then we get a strong form of deadlock freedom. This is an aspect that deserves further attention.

Another aspect of this work that was highlighted in Section 3.2 has to with the fact that there is a concept of locality in representing concurrency - if the actions in question are not enabled at a given point, after some behaviour of the transaction, then nothing can be said about them being concurrent or not by doing only this analysis. In partial order models, concurrency can be regarded a dual concept to conflict and causality. The diamond in the order structure of a vector language with concurrent actions (recall Fig. 3) offers a way of expressing the so-called linearised concurrency (the dual of causality) while the simultaneity class of actions on a given action vector (recall Definition 3.3) offers a way of expressing immediate concurrency (the dual of conflict). The ability to express this subtle difference is central to defining modal logics with partial order models, as postulated in [23] where a separation fixpoint logic is described for true concurrency.

It is also worth bearing in mind in connection to this last point that the SBVR standard [41], which can be used to specify business requests, but also to describe resources and services, specifies a logical formulation for rules and this is typically interpreted in terms of first-order logic. However, SBVR also defines modalities for rules, e.g., obligation, necessity, prohibition. When included in the formal semantics, these would require some form of modal logic. It is known that certain finer-grained behavioural equivalences, e.g., hhp-bisimilarity,
are undecidable even for finite, bounded computations such as those appearing in long-running transactions (see [47] for a comprehensive survey of concurrent equivalences). It might be instructive to characterise decidable fragments of the logic, as described in [1] where different equivalences are induced by restricting the ‘causal depth’ of the future. Model-checking and decidability issues are challenging directions of future investigation.

Another aspect concerns the translation of business requests into the transaction tree. An SBVR model expresses business logic and rules, both static (which apply to all states) and dynamic (which express transitions between states) [41]. The modelling and reasoning capability afforded by our formal approach could be exploited not only in identifying conflicts and priorities in static rules, but also in capturing dynamic rules which can affect the evolution of the SBVR model itself [30]. The gradual elaboration of transaction design together with the practitioner is another direction for future work as it would make the transactional framework accessible to a wider audience and particularly smaller businesses.

Acknowledgement. We would like to thank Mike Shields for the insightful discussions about the formal aspects treated in this paper. Thanks are also due to the anonymous reviewers whose comments helped improve the manuscript.

This work was partly funded by the UK Research Council EPSRC, under the project Evolution and Resilience of Industrial Ecosystems (ERIE), Contract No. EP/H021779/1.

References


