Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 238 (2010) 63–79
www.elsevier.com/locate/entcs

Transaction Scripts: Making Implicit Scenarios Explicit*

Sotiris Moschoyiannis[1] Amir Razavi[2] Paul Krause[3]

Department of Computing, University of Surrey
Guildford, Surrey, GU2 7XH, UK

Abstract

We describe a true-concurrent approach for managing dependencies between distributed and concurrent coordinator components of a long-running transaction. In previous work we have described how interactions specified in a scenario can be translated into a tuples-based behavioural description, namely *vector languages*. In this paper we show how reasoning against order-theoretic properties of such languages can reveal missing behaviours which are not explicitly described in the scenario but are still possible. Our approach supports the gradual refinement of scenarios of interaction into a complete set of behaviours that includes all desirable orderings of execution and prohibits emergent behaviour of the transaction.

Keywords: transactions, interactions, dependencies, concurrency, UML 2.0 sequence diagrams, vector semantics

1 Introduction

The adoption of the internet and the emerging paradigm of *computing as interaction* has fostered an environment where many distributed services are available through different components. This has increased the demand to automate business activities and workflows among networked organisations and has increasingly placed focus on *long-running transactions* that correspond to conducting business activities involving a number of partners.

The specification of a transaction in this context typically comprises a number of activities which rely on the execution of several underlying services from different components of different providers, some of which may take minutes, or hours, or even days to complete - hence the term *long-running* transaction. This requires the

* This work was supported by the EU project OPAALS, Contract FP6-034824.

[1] Email: s.moschoyiannis@surrey.ac.uk
[2] Email: a.razavi@surrey.ac.uk
[3] Email: p.krause@surrey.ac.uk

doi:10.1016/j.entcs.2010.06.005

orchestration of the participating components and the underlying service executions as well as the the provision for compensating mechanisms which in case of a failure (network/platform disconnection or delay, service unavailable) make it possible to undo parts of the transaction that have actually already happened. We will not be concerned with recovery management in this paper - preliminary ideas can be found in [13].

In this paper we will be concerned with managing the dependencies that arise between services of concurrent and distributed components within a transaction, aiming to get a thorough understanding of the behaviour patterns the sequences of service invocations should exhibit to increase confidence in a successful outcome. Dependencies may exist due to the required ordering on service invocations (e.g. book a hotel only after booking the flight) or due to sharing of data (one service uses the results of another).

Current transaction models targeting web services such as the *Business Transaction Protocol* (BTP) [4] and *Web Services Transactions* (WS-Tx) [2] do not consider a formal model for the coordination of the services involved in the execution of a transaction. This makes it difficult to eradicate *emergent* behaviour - that is, behaviour not intended but resulting from the complex interplay of the interactions themselves, e.g. race conditions. For this reason both BTP and WS-Tx seem to be geared towards centralised control (using the WS-Coordination [2] that requires tight-coupling of the underlying services, which is against the basic premise of service-oriented computing (SOC) [11], and also some knowledge of the internal build-up of the participating components, which violates the local autonomy of the participants' platforms.

In previous work [8,9] we have described a true-concurrent model, based on *vector languages* [16], for capturing the behaviour of components in terms of their interactions and have shown how this behavioural description can be obtained directly from UML2.0 sequence diagrams describing scenarios of interaction. In this paper we use sequence diagrams to describe the service interactions between various coordinator components involved in a transaction and translate them into the formal language of the vector-based description of behaviour.

We describe how reasoning against order-theoretic properties of vector languages can identify missing behaviours that infer additional scenarios. These were simply unthought in the initial design specification or indicate emergent behaviour, e.g. due to the subtle interplay between concurrency and nondeterminism in the interaction. Our approach is effectively used to elaborate the initial scenarios of interaction to more comprehensive ones, which are gradually refined to exclude emergent behaviour and include all desirable orderings of execution.

This paper is structured as follows. In Section 2 we describe how to model transactions, focusing on their structure and interactions. In Section 3 we present a formal language for describing interactions between coordinator components of a transaction. In Section 4 we show how the formal language can be used to reason about the dependencies between services of different coordinator components and uncover implicit scenarios of execution in the initial sequence diagram. The paper

finishes with some concluding remarks and ideas for future work in Section 5.

2 Modelling long-running transactions

In this section we briefly describe the structure of a transaction and then show how service interactions between the coordinator components of the participants can be modelled using UML2.0 [10].

We have seen that long-running transactions involve interactions between multiple service providers, which need to be orchestrated in order to increase expectations of a successful outcome prior to deployment. The orchestration required needs to be performed in a way that respects the loose-coupling of the underlying services - a basic premise of SOA [11]. For this reason, each networked organisation (service provider / consumer) provides its services, and also requests services of others, through a *coordinator component* that manages the communication between different platforms and the deployment of the corresponding services. Without going into much detail, it can deploy (upon receiving a call) the services in its Local Service Repository and can make calls to services it requires from others components, within a given transaction, whose service descriptions (e.g. in WSDL) are listed in its Global Service Repository. The structure of a coordinator component is shown in Fig. 1.

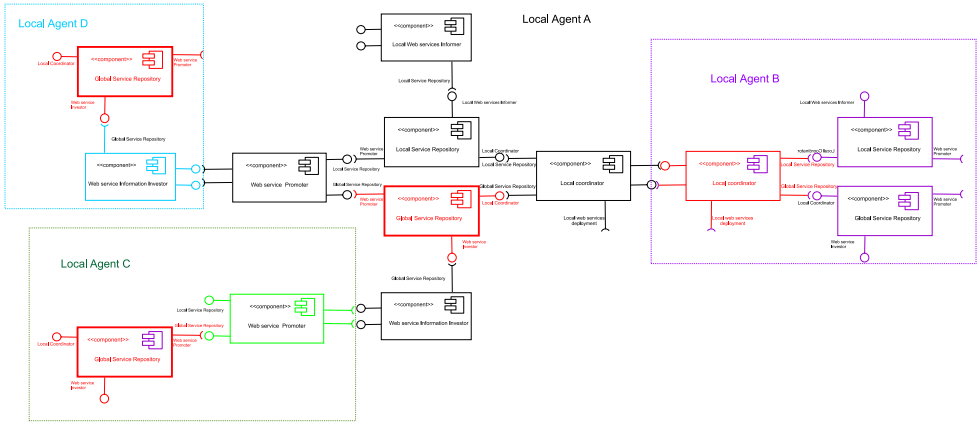

Fig. 1. Coordinator components in a transaction

In earlier work [14], we have described a transaction model for the distributed coordination of multi-service transactions. This is based on log structures, given in the form of directed graphs, which allow the coordinator component of each platform to only need to know about its own services and their dependencies on other components' services - in fact, it needs to know only what happens immediately *before* and *after* deployment of its own services. Due to space limitations we omit further details.

A transaction in our approach is represented by a tree structure that captures nested subtransactions and exemplifies the local coordination that is required for the services involved to be performed in unison. Drawing upon the latest work on

an extended service-oriented architecture for a business environment [11], we have considered five different composition types or *composers* which allow for various modes of service interaction in our model.

Sequential: execution of a service is dependent on the previous one; this composer handles both Sequential with Commit Dependency (SCD) and Sequential with Data Dependency (SDD) process-oriented service composition.

Parallel: the services are executed concurrently; composer handles Parallel with Commit Dependency (PCD), Parallel with Data Dependency (PDD) and Parallel without Dependency (PND) process-oriented service composition.

Sequential Alternative: the services will be attempted in succession until one produces the desired outcome, as specified by some criterion (e.g. cost).

Parallel alternative: alternative services are executed in parallel and once a service produces the desired outcome, the rest are aborted.

Data-oriented: this composer handles data-oriented service composition and deals with released data items both within and outside a transaction.

Delegation: this composer allows (part of) a transaction to be delegated to another platform, e.g. to overcome bottlenecks or low bandwidth connections.

Further details on the composition types considered in our model can be found in [14]. A schema using Netbeans 6 for generating XML descriptions of transaction contexts, as specified by the composers used in the corresponding transaction tree, is given in Fig. 2.

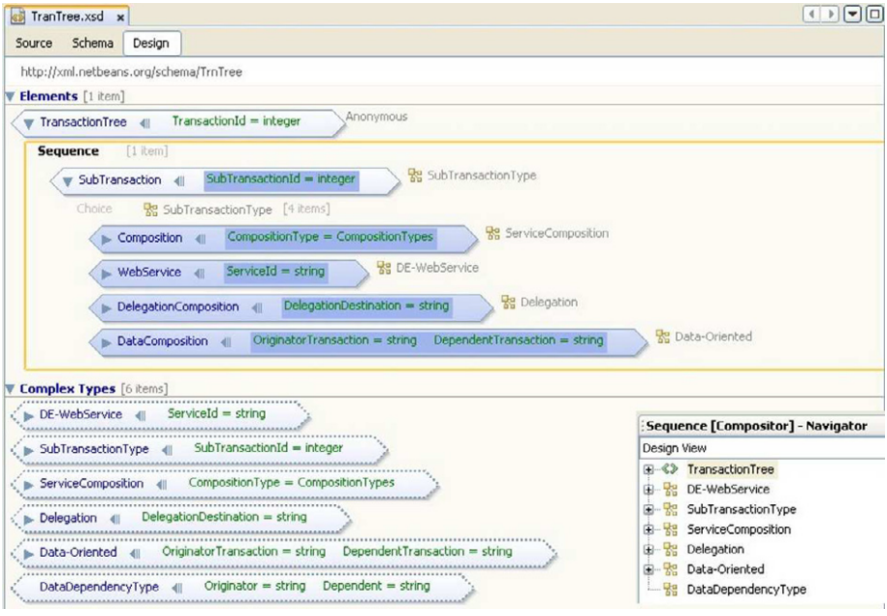

Fig. 2. XML schema for describing transaction contexts

In this paper we will be concerned with the sequential, parallel and sequential-alternative composition types. Fig. 3 shows a transaction tree with four basic services - $a1$ and $a2$ of a local paltform with coordinator component $CC1$, $b1$ of $CC2$, and $c1$ of $CC3$ - whose order of execution is determined by the correspond-

ing composition types. For example, the tree specifies that the service calls $a2$ and $c1$ are children of a sequential composer and hence $c1$ can only happen after $a2$. The connecting lines on this composer indicate data dependencies between the corresponding services' deployment. The corresponding XML description for the example transaction tree is derived from the schema of Fig. 2 and can be found following [1].

The scenario described in Fig. 3 has appeared in [14] and has been simplified somewhat here, but is still complicated enough to illustrate the key ideas. The service interactions implied by a transaction tree can be modelled using a UML sequence diagram. Fig. 4 shows the three coordinator components of the transaction and the messages (service invocations) exchanged between them during the execution of the transaction in our example.

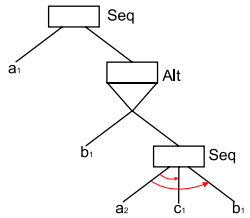

Fig. 3. A simple transaction in a tree structure

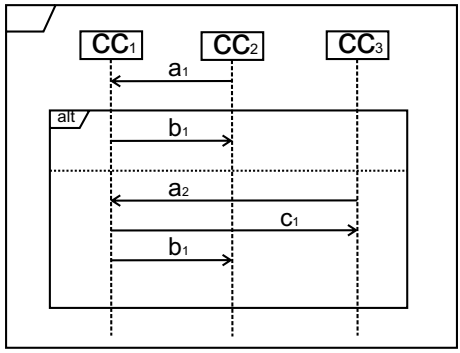

Fig. 4. Behavioural scenario of a simple transaction

It can be seen that the behavioural scenarios, as given by the corresponding UML sequence diagram, determine the order of execution of the participating components' services. In the remainder of the paper we will be concerned with a formal reasoning approach aiming to identify missing behaviours (if any) in the initial scenario-based specification of the transaction context. We shall see that missing behaviours may indicate emergent behaviour (e.g. due to race conditions) or scenarios of execution that were simply unthought in the initial design specification. First, we need to describe a transaction more formally in order to get a thorough understanding of the behaviour patterns the underlying service invocations should exhibit.

3 Vector languages for transactions

In this section we introduce a formal language for long-running transactions that captures the dependencies between service executions of the various coordinator components involved, and enables formal reasoning about the interactions to uncover hidden scenarios. In a transactional environment there is a high degree of concurrency since real problems require a number of activities to take place in parallel. We will therefore find the general theory of non-interleaving representation of parallel behaviour found in [16] of great use in what follows.

The semantics is intended to describe the behaviour of a transaction in terms of its services at the deployment level, but not the low-level computations performed by the services themselves. In fact, in certain contexts such as a digital ecosystem for business services are offered by different service providers and it is important that we defer from interfering with the local state of service execution. This means it is appropriate to consider that any action within the transaction model has no significant duration, in the sense that (i) it either occurs as a whole or not at all; (ii) it occurs either wholly before, or wholly after, or wholly in parallel with, every other action.

As discussed in Section 2, a transaction involves a number of local agents acting on behalf of different parties (service providers, consumers, or both) which collaborate to perform a business activity. To preserve the local autonomy of each platform, they communicate via their coordinator components which manage the required service interactions (recall Fig. 3 and 4).

A transaction T is associated with a set of coordinator components C and a set of actions M. Our interest is in the observable events on the coordinator components and thus actions can be understood as service invocations between the participating components, as shown for example in the scenario of Fig. 4. Hence, each component in C is associated with a set of actions which correspond to deploying (its own) or requesting (others') services. We denote this set by $\mu(i)$, for each $i \in C$, where μ is given by $\mu : C \to \wp(M)$. Further, we require that $\bigcup_{i \in C} \mu(i) \subseteq M$.

As can be seen in Fig. 3, a transaction has a number of activation or access points, namely the interfaces of the coordinator components participating in the interaction. Thus, instead of modelling the behaviour of a transaction by a sequential process, which would generate a trace of a single access point, we consider a number of such sequences, one for each component, at the same time. This draws upon Shields' vector languages [16] and leads to the definition of the so-called *transaction vectors*.

Transaction vectors. Let T be a transaction. We define V_T to be the set of all functions $\underline{v} : C \to M^*$ such that $\underline{v}(i) \in \mu(i)^*$.

By $\mu(i)^*$ we denote the set of finite sequences over $mu(i)$. Mathematically, the set V_T is the Cartesian product of the sets $\mu(i)^*$, for each i. Effectively, are n-tuples of sequences where each coordinate corresponds to a coordinatro component in the transaction (hence, n is the number of components) and contains a finite sequence of actions that have occurred on that component.

When an action occurs in the transaction, that is to say when a service is called on a coordinator component, it appears on a new transaction vector and at the appropriate coordinate. For example, the vector $(a1, \Lambda, \Lambda)$ describes that portion of behaviour of the transaction in which an action $a1$ (e.g. service invocation) has taken place on the corresponding component allocated to the first coordinate. The vector $(a1, b1, \Lambda)$ describes that portion of behaviour in which both $a1$ and $b1$ have happened on the corresponding components while the vector $(a1a2, b1, \Lambda)$ describes an occurrence of $a1$ and an occurrence of $a2$ on the component corresponding to the first coordinate, and an occurrence of $b1$ on the second coordinate - nothing has happened on the component corresponding to the third coordinate.

It can be seen from the example given above that there is already an ordering among actions on a particular access point or component, e.g. $a1$ followed by $a2$. This vector-based behavioural description of transactions can also capture the orderings between service invocations on different components, which amounts to actions appearing on different vector coordinates. This requires however a more careful consideration of the mathematical properties of such vectors which is described in the sequel.

At this stage it suffices to understand that each transaction vector provides a snapshot of behaviour that captures what actions have already occurred and on which part (component) of the transaction. In describing the behaviour of a transaction however we are interested only in those vectors describing (orderings of) actions that we expect the coordinator components to engage in during the course of the transaction execution. In other words, for a given transaction T we are interested in a particular subset of all possible vectors formed over T.

We will use the term *transaction language* to refer to an appropriate subset V of all possible vectors V_T formed over a given transaction T. The idea is that the particular subset of transaction vectors, for a specific transaction, expresses the ordering constraints necessary in the corresponding service orchestration. To identify the appropriate subset of vectors capturing *intended* behaviour only, we turn our attention to the corresponidng UML model that describes the required sequences of service interactions.

We outline how UML 2.0 sequence diagrams [10] can be translated into transaction vectors in Section 4 and show how formal reasoning against the order-theoretic properties of the transaction language (given next) can be used to determine the complete set of behaviours of a transaction and inform the refinement of the initial UML model.

3.1 Basics of transaction vectors

We now examine the basic mathematical properties of our formal construction so far. The discussion is restricted to those operations used in the remainder of the paper. A detailed mathematical treatment can be found in [7,16].

We start by introducing a specific kind of transaction vector, which is used in our model to describe actions (e.g. service invocations) within a transaction.

Column vectors. Let T be a transaction and V_T its set of transaction vectors.

We define $A_T = \{\underline{\alpha} \in V_T \setminus \{\underline{\Lambda}_T\} : i \in C \implies |\underline{\alpha}(i)| \leq 1\}$ where $|x|$ denotes the lenght of the sequence $|x|$.

Thus, column vectors are themselves transaction vectors, but have the additional constraint that each of their coordinates is either the empty sequence or a single action. For example, the vector $(a1, \Lambda, \Lambda)$ represents the occurrence of an action $a1$ on the component associated with the first coordinate.

We have seen that transaction vectors are essentially tuples of sequences. This can be exploited in defining operations on vectors in terms of well-known operations on sequences.

Operations on vectors. For $\underline{u}, \underline{v} \in V_T$, we define,

- $\underline{u}.\underline{v}$ to be the unique vector $\underline{w}$ such that $\underline{w}(i) = \underline{u}(i).\underline{v}(i)$, for each $i \in C$ (*concatenation*)
- $\underline{u} \leq \underline{v}$ iff $\underline{u}(i) \leq \underline{v}(i)$, for each $i \in C$ (*prefix ordering*)
- $\underline{u} \sqcap \underline{v}$ to be the vector $\underline{w}$ which satisfies $\underline{w}(i) = min(\underline{u}(i), \underline{v}(i))$, for each i
- $\underline{u} \sqcup \underline{v}$ (if it exists) to be the vector $\underline{w}$ which satisfies $\underline{w}(i) = max(\underline{u}(i), \underline{v}(i))$
- if $\underline{u} \leq \underline{v}$, then we define $\underline{v}/\underline{u}$ to be the unique element $\underline{z} \in V_T$ such that $\underline{u}.\underline{z} = \underline{v}$ (*right-cancellation*)

Thus, the operation of concatenation on vectors is defined in terms of the concatenation of sequences appearing on their respective coordinates. For example, $(a1, b1, \Lambda).(a2, \Lambda, \Lambda) = (a1a2, b1, \Lambda)$.

The ordering amongst vectors is defined in terms of the usual prefix ordering operation on sequences appearing on their respective coordinates. For example, $(a1, b1, \Lambda) \leq (a1a2, b1, \Lambda)$ since $a1 \leq a1a2$ and $b1 \leq b1$ and $\Lambda \leq \Lambda$. In other words, the second vector 'wins' on the first coordinate (since it has a sequence of greater length in this coordinate) while the two vectors 'draw' on all other coordinates. It is not hard to see that some vectors will be incomparable. It turns out that such vectors describe either parallel or alternative behaviours of the transaction in question, and this will be further discussed in the following sections.

The operations '$\sqcap$' and '$\sqcup$' give the greatest lower bound and the least upper bound of $\underline{u}, \underline{v} \in V_T$, respectively, in the usual sense of lattices and domain theory [3]. As we will see, these operations are central to the treatment of concurrency in our approach.

The right cancellation operator '/' says that if $\underline{u}$ is a transaction vector describing an initial part of the behaviour described by $\underline{v}$ so that $\underline{u} \leq \underline{v}$, then $\underline{v}/\underline{u}$ is the continuation of $\underline{u}$ that extends it to $\underline{v}$. This operation is central to the treatment of compensations in our approach. We return to this discussion in the concluding section of the paper.

It is important to stress the fact that all operations on vectors are performed coordinate-wise and this simplifies the proofs but also makes the formal model feasible for implementation. We are now set to show how transaction vectors can be used to capture the dependencies between service interactions of coordinator components within a transaction.

3.2 Managing dependencies

The prefix ordering relation on transaction vectors can be viewed as an ordering on partial executions, where each vector corresponds to that portion of behaviour in which the transaction has already engaged in the actions appear-ing on its coordinates. This can be expressed more succinctly by saying that $\underline{u} \leq \underline{v}$ means that $\underline{u}$ is an earlier part of behaviour leading to $\underline{v}$.

A more careful examination of the mathematical construction shows that we can say more than that. Indeed, we find it useful to determine *immediate predecessors* (or *successors*) of a transaction vector.

Covers. Let $\underline{u}, \underline{v} \in V \subseteq V_T$. We say that $\underline{v}$ *covers* $\underline{u}$ in V, and we write $\underline{u} \lhd \underline{v}$ iff (i) $\underline{u} \leq \underline{v}$ and $\underline{u} \neq \underline{v}$ and (ii) If $\underline{z} \in V$ such that $\underline{u} \leq \underline{z} \leq \underline{v}$, then $\underline{z} = \underline{u} \vee \underline{z} = \underline{v}$.

Thus, whenever $\underline{u} \leq \underline{v}$, and we also have that $\underline{u} \lhd \underline{v}$, then the last actions that went into forming each vector have occurred consecutively - one after the other. This allows to model sequential dependency inside a transaction. Recall the example of Fig. 3 where service $c1$ can only be called after $a2$.

Our approach towards modelling concurrent actions, actions that can happen in parallel, draws upon the concepts in Shields' *vector languages* [15] and Mazurkiewicz *trace languages* [6] where concurrent actions are considered as being *unordered*, in contrast to CSP trace theory where concurrent events are understood to occur *in either* order (nondeterministic interleaving).

The treatment of concurrency within our formal model of transactions thus takes up on non-interleaving models of concurrency, which introduce additional structure into formal languages in order to describe non-sequential behaviour. The additional structure is given in terms of an independence relation over action symbols, which describes potential concurrency. Drawing upon the extension of the independence relation ι to *behaviour vectors* in [16], the notion of independence between actions in Mazurkiewicz traces can be readily interpreted into transaction vectors in our approach.

Independence. For $\underline{u}, \underline{v} \in V \subseteq V_T$ we define

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

This definition says that two transaction vectors are independent if the behaviours they describe engage distinct components (correspond to service invocations on different coordinator components) of the transaction. This means the behaviours described by $\underline{u}$ and $\underline{v}$ may occur independently.

In the case of column vectors, independence captures the fact that actions appearing in one vector may occur independently of those appearing in the other. If in addition the vectors representing these actions are adjacent in an expression (of the series of concatenations that went into forming the corresponding transaction vectors), then the actions are concurrent. Hence, whenever two actions are independent and are both enabled (can both occur at some point, after some behaviour) then, their corresponding column vectors commute, i.e. $\underline{\alpha}_1.\underline{\alpha}_2 = \underline{\alpha}_2.\underline{\alpha}_1$, and in the resulting behaviour the two actions are concurrent.

Based on the prefix ordering between transaction vectors in the set V we may

also model a choice between actions. That is, actions which are mutually exclusive in that occurrence of one excludes occurrence of the other. In discussing concurrent actions in a long-running transaction, we saw that the two incomparable transaction vectors represent parallel behaviour. The vector they both cover is in fact their greatest lower bound and is obtained by applying the operation '$\sqcap$' given earlier. The fact the two incomparable vectors represent concurrent actions is only because they are bounded above in the set (by the transaction vector which is their least upper bound, given by '$\sqcup$', and is sitting on top of the lozenge). Whenever this latter requirement does not hold we may talk about events in conflict.

It might be instructive to make the distinction in terms of pictures and associated Hasse diagrams. In the diagram of Fig. 5, $a1$ and $d1$ are sequential ($d1$ can only be invoked after $a1$) in Fig. 5(i), they are concurrent in Fig. 5(ii) while there is a choice between them (alternative) in Fig. 5(iii).

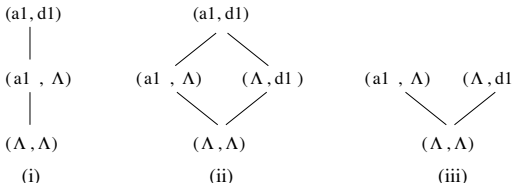

Fig. 5. Order structure of transaction languages

Notice that the set of vectors in (i) does not include $(\Lambda, d1)$, which means that $d1$ never occurs before $a1$ does; in (iii) it does not include $(a1, d1)$ which means there is no valid behaviour of the transaction processing system in which both $a1$ and $d1$ have taken place; in (ii) it includes all four vectors, which means that $a1$ on its own, $d1$ on its own, and $a1$, $d1$ together, are all valid observations of the behaviour of the transaction in which $a1$ and $d1$ happened concurrently. This is indicated by the familiar lozenge shape that shows the corresponding order structure exhibits the characteristic structure of a finite lattice.

In further explanation, the vector sitting at the bottom of the lozenge is the greatest lower bound '$\sqcap$' of the two incomparable vectors $(a1, \Lambda)$ and $(\Lambda, d1)$ sitting at the middle of the lozenge while the vector at the top is their least upper bound '$\sqcup$'. The lozenge as a whole describes that part of behaviour of the transaction in which $a1$ and $d1$ happened concurrently. Further details on how the ordering relations between actions are manifested in the order structure of the resulting set of vectors can be found in [7].

3.3 Well-formedness of the behavioural description

In describing the behaviour of a transaction we are interested in the actions (activations) on its various components. These are captured in our model using *column vectors*. Thus, instead of considering all possible transaction vectors we would like to be concerned with those obtained by concatenations with column vectors only. This gives the behaviour of the transaction in terms of actions of its coordinator components and can be used to enforce the coordination of the underlying services.

We have seen that transaction vectors are obtained by coordinate-wise concatenation. Hence, they can be seen to be built up starting from the empty vector by a series of concatenations with column vectors which represent actions. The study of vector languages in [16,9] shows that in order to ensure that vectors considered are the result of concatenations with column vectors only, the set of transaction vectors must satisfy certain properties. We introduce these properties next.

The first property captures the fact that a system's computations always have a starting point, and ensure that only a finite number of actions may occur within finite time. This turns out to be the case if whenever two vectors describe an earlier part of behaviour than a third, also in the set, then their least upper and greatest lower bounds are also in the set. This is formally put in the following definition.

Discreteness. Let $V \subseteq V_T$, then V is *discrete* iff $\underline{\Lambda}_T \in V$ and whenever $\underline{u}, \underline{v}, \underline{w} \in V$ such that $\underline{u}, \underline{v} \leq \underline{w}$ then (i) $\underline{u} \sqcup \underline{v} \in V$ and (ii) $\underline{u} \sqcap \underline{v} \in V$.

Note that $\underline{u} \sqcup \underline{v}$ is understood as asserting that '$\sqcup$' is defined.

Discreteness imposes a finiteness constraint in the sense that it excludes infinite ascending or descending chains of actions with respect to time ordering. In fact, it ensures that situations like those resulting in Zeno-type paradoxes will never arise.

We further require that every occurrence of an action (e.g. service invocation) is recorded in the set of vectors associated with the transaction. This guarantees that any earlier part of behaviour is itself a behaviour and motivates the following definition.

Local left-closure. Let $V \subseteq V_T$, $i \in C$ and $x \in \mu(i)^*$. Then, V is *locally left-closed* iff, whenever $\underline{v} \in V$ and $\Lambda < x \leq \mu(i)$ then there exists $\underline{u} \in V$ such that $\underline{u} \leq \underline{v}$ and $\underline{u}(i) = x$.

The local left-closure property is intended to resolve ambiguities that may arise from not having enough vectors in the transaction language to describe the course of the behaviour in question; not the start or the end, but the 'gaps' in between. This requires that every occurrence of an event is 'recorded' in the language of the transaction. This implies the presence of a distinct *prime* element in V for each occurrence of an action. Primes play a central role in the more general theory of parallelism [16] and in particular with respect to associating vector languages with order-theoretic objects used to determine the temporal relation between occurrences of actions. For the purposes of the present paper, and the adaptation of this theory in deriving a formal model for long-running transactions, it suffices to understand that, in this context, the notion of prime refers to transaction vectors which have a unique other vector immediately beneath them. Such an ordering is determined by the *covers* relation among vectors, given earlier.

To establish some terminology for the sequel, we say that the set of vectors $V \subseteq V_T$ associated with 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. The idea is that in checking against these properties 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 still possible. This draws upon previous work on vector languages and UML sequence diagrams in [7], which is adapted to refining transaction contexts in the next section.

4 Elaborating behavioural scenarios

In the previous section, we have used vector languages to capture the coordination of the service interactions between coordinator components in a transaction. In Section 2 we used UML sequence diagrams to describe the order in which the underlying services need to be deployed. In this section, we show how transaction languages, and associated order structures, can be used to determine whether there are any discrepancies between the specified order of execution, given in the sequence diagram, and the actual order in which the services can be deployed.

First, we need to understand how to obtain a transaction language from the corresponding sequence diagram and then show how in checking against *normality* we can identify missing behaviours. In previous work [8] we have shown how vector languages can be obtained from sequence diagrams. Here we only outline the general idea.

There are graphical positions or *locations* along the lifelines of participating instances in a sequence diagram that are of particular significance, especially when the diagram is considered in a formal setting. Our approach draws upon the work in [5] where locations are treated formally to obtain the corresponding model. To obtain the corresponding vector language we associate vectors to the locations of the diagram and use the resulting language to determine the relations between those interactions in order to reflect the meaning of the diagram. Fig. 6 shows the relation between the locations in a simple sequence diagram and the corresponding vector language, which is depicted in a Hasse diagram. The XML description of the translation for the vector $(a1, b1)$ is given on the right, and is derived following the schema shown in Fig. 7.

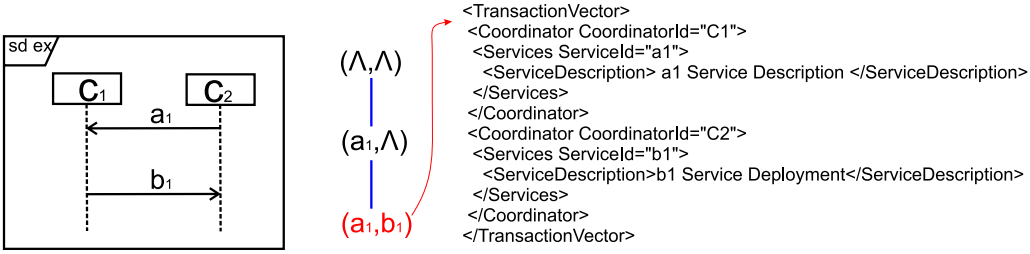

Fig. 6. A sequence diagram and its corresponding transaction language

The vectors associated with each location are obtained from the vectors of the immediately preceding location, by concatenating the action (the corresponding column vector) associated with the location being considered. By convention the initial location is mapped onto the empty vector $\underline{\Lambda}_T$. By moving down the diagram,

from one location to the next, whilst mapping each location to (a set of) vectors, the sequence diagram is translated into vectors.

There are some cases however in which this rationale does not apply. In particular, locations within different operands of an **alt** or **par** need to be treated differently. This is because we have to take into account the various execution sequences that are possible when encountering these interaction fragments. Note that a location is also used to mark the beginning and the end of interaction fragments superimposed on the diagram. The first location of each operand in an **alt** or **par** fragment is considered in relation to the start location of the fragment rather than its immediately preceding location. The vectors of the end location of an **alt** fragment with k operands are considered in relation to the last location of each operand - to reflect the fact there are k alternative scenarios. The vectors of the end location of a **par** fragment are carefully obtained to reflect the fact the actions appearing within are effectively unordered. Full details of the formal construction behind the translation can be found in [8].

The schema for the formal translation given in Fig. 7 is used for deriving XML representations of the corresponding transaction language that reflects its order structure. As we will see, these *transaction scripts* can be used to identify implicit behavioural scenarios in the corresponding transaction context (Fig. 3 and 4).

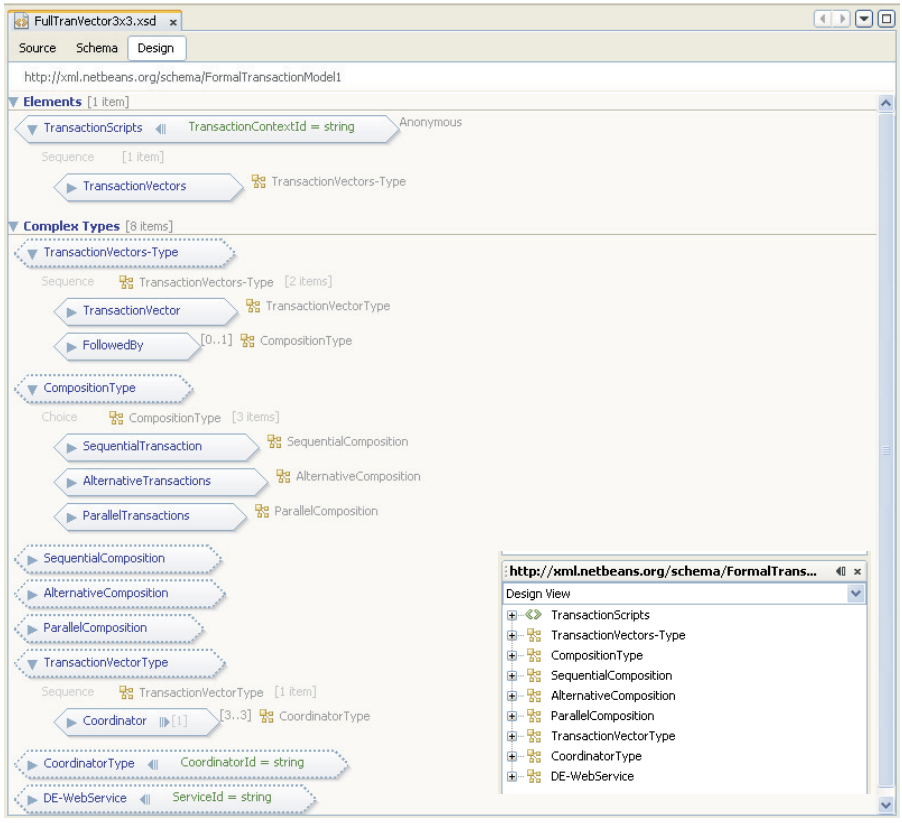

Fig. 7. XML schema for the formal translation of behavioural scenarios

Our approach can handle both synchronous and asynchronous communication. The synchronous case is captured by a shared action, e.g. it would be represented by a column vector such as $\underline{\alpha} = (a1, a1, \Lambda)$ which denotes the simultaneous occurrence of sending and receiving $a1$. Asynchronous communication is captured by distinct column vectors, e.g. $\underline{\alpha}_1 = (\Lambda, a1, \Lambda)$ and $\underline{\alpha}_2 = (a1, \Lambda, \Lambda)$ describing the consecutive actions of sending and receiving $a1$, which would result after concatenation with the corresponding transaction vectors into vectors related by '$\lhd$' which infers immediate causality.

Since we have not explicitly discussed simultaneity within the formal framework in this paper, we will assume asynchronous communications only and hence use $\underline{\alpha}_1 = (\Lambda, a1, \Lambda)$ and $\underline{\alpha}_2 = (a1, \Lambda, \Lambda)$, and in particular, we will be concerned with $(a1, \Lambda, \Lambda)$ as it is receiving $a1$ that corresponds to the actual service deployment which is of primary interest in a transactional setting.

Recall the sequence diagram of Fig. 4 describing the interactions between services of the coordinator components. The transaction language that models the behaviour represented in the sequence diagram is given in Fig. 8. The corresponding XML description can be found online following [1].

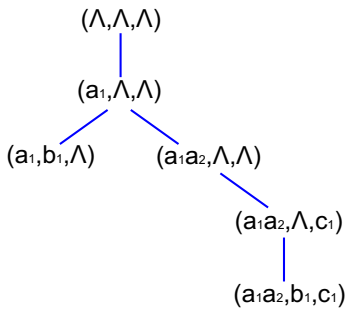

Fig. 8. Transaction language for the interaction of Fig. 4

In Section 3 we argued about *normality* in transaction languages and identified properties, discreteness and local left-closure, that ensure the well-formedness of this tuples-based description of behaviour. By careful examination of the transaction language in Fig. 8, it can be seen that while it is locally left-closed, the discreteness property is violated. Indeed, the vectors $\underline{u} = (a1a2, \Lambda, \Lambda)$ and $\underline{v} = (a1, b1, \Lambda)$ are both smaller than vector $\underline{w} = (a1a2, b1, c1)$ and their greatest lower bound $((a1, \Lambda, \Lambda))$ is in the set, but their least upper bound, given by $(a1, b1, \Lambda) \sqcup (a1a2, b1, \Lambda)$, is not.

According to our mathematical framework, this vector should be added to make V_T discrete and thus also normal. The effect of adding in the missing behaviour, as shown in Fig. 9, is that there is now potential concurrency between $a1$ and $b1$ (first occurrence of) and $c1$ and $b1$. So there are two additional scenarios of execution, on top of the two scenarios described explicitly in the sequence diagram of Fig. 4. The component developers can now determine whether these scenarios describe desirable behaviour or not.

The service calls $c1$ and $b1$ can take place concurrently since, although they

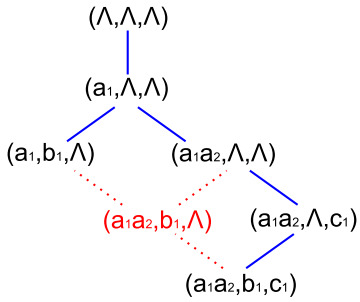

Fig. 9. Discrete transaction language for the interaction of Fig 4

were initially designed to occur sequentially (recall the sequential composer in the transaction tree of Fig. 3), there is a dependency between $a2$ and $b1$, and between $a2$ and $c1$, but there is no dependency between $c1$ and $b1$, and hence they are not necessarily related by immediate causality (as the sequence diagram of Fig. 4 would indicate). This is reflected in the refined sequence diagram of Fig. 10 where the implicit scenarios have been made explicit. Upon receiving a call for service deployment $a2$ the component $CC1$ can proceed to do $c1$ and $b1$ in any order, including at the same time.

We note that the formal model also indicates potential concurrency between $a1$ and $b1$ (the first occurrence of $b1$). This is a situation known as *asymmetric confusion* in Net theory [12] - a situation where the choice between an action happening on its own and concurrently with some other action is never actually resolved. Within reason, we would expect the application programmer (of the coordinator component $CC1$) to prohibit concurrency in this case at the implementation level. At the design level, this can be done by adding a type of acknowledgement message $a3$ that will infer immediate causality, as shown in Fig. 10 (right). The corresponding XML representations can be found online following [1].

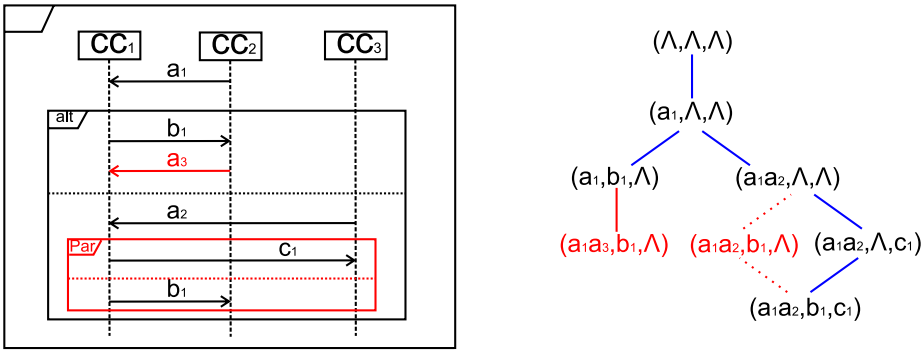

Fig. 10. Transaction language of the elaborated behavioural scenarios

Fig. 11 shows the corresponding transaction tree which is now optimised to reflect the complete set of scenarios of execution.

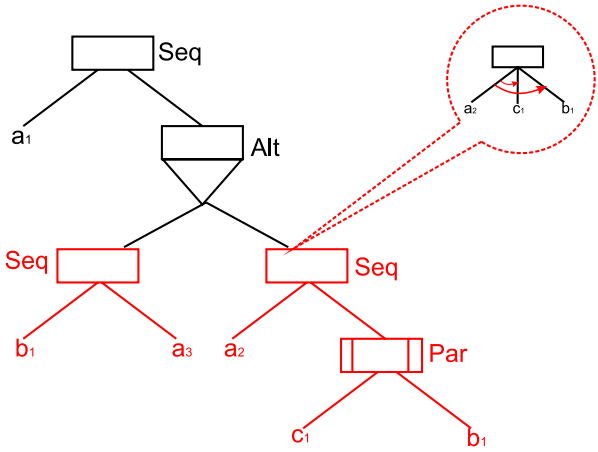

Fig. 11. Optimised transaction tree

5 Conclusions and Related Work

We have described a forma framework for the coordination of concurrent and distributed service invocations between coordinator components involved in a long-running transaction. In particular, we have used transaction trees (structure) and UML sequence diagrams (interactions) to specify a transaction. We then showed how a formal description of this initial transaction context can be used to reason about the corresponding behavioural scenarios and identify implicit scenarios of interaction which may indicate emergent behaviour. We have seen that our formal framework can determine the complete set of behaviours and make all possible scenarios explicit in the corresponding scenario-based specification.

Schemas for deriving XML representations of both a transaction context and the corresponding transaction scripts (reflecting transaction vectors) needed to refine the behavioural scenarios of the initial transaction context, were also given. The complete source files can be found online following [1].

In [13] we have been concerned with dependencies due to data sharing and have presented an extended lock mechanism that ensures consistency and drives the rollback procedure if some failure later in the transaction makes recovery necessary. Work is in progress on integrating the transaction language model presented here with the lock mechanism so that the orderings between transaction vectors trigger the appropriate lock scheme whenever necessary.

The approach described in this paper has focused on dependencies between (services of) coordinator components within a transaction. Dependencies may also exist across transactions due to the need for releasing some results of a transaction - often referred to as *partial results* - to another transaction before it commits. An extension to address partial results and compensating actions (using the right-cancellation operator '/') is currently under investigation.

References

[1] www.computing.surrey.ac.uk/personal/st/S.Moschoyiannis/trnscripts.

[2] F. L. Cabrera, G. Copeland, J. Johnson and D. Langworthy. *Coordinating Web Services Activities with WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity.* http://msdn.micorsoft.com/webservices/default.aspx, January 2004.

[3] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge Mathematical Textbooks, Cambridge University Press, 1990.

[4] P. Furnis, S. Dalal, T. Fletcher, A. Green, A. Ceponkus, and B. Pope. *Business Transaction Protocol, version 1.1.0.* available from www.oasis-open.org/committees/download.php/9836, November 2004.

[5] J. Küster-Filipe. Modelling Concurrent Interactions. *Theoretical Computer Science*, 351(2):203–220, 2006.

[6] A. Mazurkiewicz. Basic Notions of Trace Theory. In *Linear Time, Branching Time and Partial Orders in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 285–363. Springer Verlag, 1988.

[7] S. Moschoyiannis. *Specification and Analysis of Component-Based Software in a Concurrent Setting.* PhD thesis, University of Surrey, 2005.

[8] S. Moschoyiannis, P. J. Krause, and M. W. Shields. A True-Concurrent Interpretation of Behavioural Scenarios. In *ETAPS 2007 - FESCA'07*, ENTCS. Elsevier, 2007. To appear.

[9] S. Moschoyiannis, M. W. Shields, and P. J. Krause. Modelling Component Behaviour Using Concurrent Automata. In *ETAPS 2005 - FESCA'05*, volume 141 of *ENTCS*, pages 199–220. Elsevier, 2005.

[10] OMG. *Unified Modeling Language: Superstructure, version 2.0.* OMG document formal/05-07-04, available from http://www.omg.org, August 2005.

[11] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Kramer. Service-Oriented Computung Research Roadmap. In *Dagstuhl Seminar Proc. 05462, Service-Oriented Computing (SOC)*, pages 1–29, 2006.

[12] C. A. Petri. Introduction to General Net Theory. In *Proceedings of Advanced Course on General Net Theory of Processes and Systems: Net Theory and Applications*, volume 84 of *LNCS*, pages 1–21. Springer-Verlag, 1979.

[13] A. Razavi, S. Moschoyiannis, and P. Krause. Concurrency Control and Recovery Management for Open e-Business Transactions. In *Communicating Process Architectures (CPA 2007)*, pages 267–285. IOS Press, 2007.

[14] A. Razavi, S. Moschoyiannis, and P. Krause. A Coordination Model for Distributed Transactions in Digital Business Ecosystems. In *IEEE Digital Ecosystems and Technologies (DEST 2007).* IEEE Computer Society, 2007.

[15] M. W. Shields. Concurrent Machines. *Computer Journal*, 28:449–465, 1985.

[16] M. W. Shields. *Semantics of Parallelism.* Springer-Verlag London, 1997.