A Coordination Model for Distributed Transactions in Digital Business EcoSystems

Amir R Razavi, Sotiris K Moschoyiannis, Member, IEEE and Paul J Krause

Department of Computing, School of Electronics and Physical Sciences, University of Surrey, Guildford, Surrey, GU2 7XH, UK,
e-mail: (a.razavi, s.moschoyiannis, p.krause)@surrey.ac.uk

Abstract—In this paper we present a model for coordinating distributed long running and multi-service transactions in Digital Business EcoSystems. The model supports various forms of service composition, which are translated into a tuples-based behavioural description that allows to reason about the required behaviour in terms of ordering, dependencies and alternative execution. The compensation mechanism warrants consistency, including omitted results, without breaking local autonomy. The proposed model is considered at the deployment level of SOA, rather than the realisation level, and is targeted to business transactions between collaborating SMEs as it respects the loose-coupling of the underlying services.

Index Terms—multi-service transactions, choreography, loose-coupling, behaviour patterns

I. INTRODUCTION

The goal of Service-Oriented Computing (SOC) [15] is to enable applications from different providers to be offered as services that can be used, composed, and coordinated in a loosely coupled manner. Web services provide a realisation of SOC. However, we will argue in this paper that current Web Services technology needs further development before the goal of SOC can be fully realised. We will also show how some of these limitations can be overcome.

Our primary concern is with the support for long-term business transactions involving open communities of SMEs. A business transaction in this paradigm can be either a simple usage of a web service (rarely in B2B relationships) or a mixture of different levels of composition of several services from various service providers. We will argue that the current transaction and business coordination frameworks can lead to issues with tight coupling and violation of local autonomy for the participating SMEs.

The next section very briefly summarises our findings from a comprehensive review of current work on transaction frameworks. Following that, we introduce a new coordination model which supports loose coupling between coordinators and service providers. We provide a formalisation of this coordination model which is then used to support compensation and forward recovery mechanisms. Finally we outline the next steps in development of this model.

II. DISTRIBUTED TRANSACTIONS WITH SOA

Well-known transaction models for web-services include BTP [8] and WS-BusinessActivity [2], whose coordination mechanism is based on WS-Coordination [3]. A study of the underlying coordination framework, reported in [16], however, shows it to suffer from some critical decisions about the internal build-up of the communicating parties; a view also supported in [1].

The Coordinator and Initiator roles are tightly-coupled and the Participant contains both business and transaction logic. These presumptions are against the primary requirements of SOA for loose-coupling and local autonomy, and thus are not suitable for a Digital Business EcoSystem (DBE), especially when SMEs are involved.

A further concern has to do with the compensation mechanism. Behavioural patterns such as “validate-do” and “provisional-final” [9], [17] are not supported while the “do-compensate” pattern, which is supported, results in a violation of local autonomy, since access to the service realisation level is required (see [16] for further details). Prescribing internal behaviour at the realisation level raises barriers for SMEs as it inevitably leads to their tight-coupling with the coordinators.

The desirable element here is to analyse the transaction behaviour before run-time and adopt different behaviour patterns and compensation strategies to prevent unexpected behaviour, but without breaking local autonomy.

III. COORDINATION MODEL

A. Order of Execution in Service Composition

In the Digital EcoSystem paradigm, networked organisations engage in complex transactions involving the composition of a number of sub-transactions. Typically, a two phase commit (2PC) protocol is recommended for nested transactions [13]. The necessity of two phases is underlined by the long-running nature of business transactions. The first phase prepares the transaction and involves declaring dependencies, setting up the relationships, and indicating the boundaries and side-effects of updates (possibly using locks). The second phase is supposed to finalise or abort the transaction.

In the proposed model, we consider an additional intermediate phase which concerns potential failure of subtransactions (one or more coordinators or services). This adaptation of 2PC allows some leverage in attempting alternative subtransactions, or re-starting only the failed subtransaction, before re-starting the whole transaction. This will be further discussed in Section IV of the paper where we describe a forward recovery mechanism for our model.

In our approach, a transaction is represented by a tree
structure (see Fig.1). Each node is either a coordinator (a composition type) or a basic service (a leaf). Five different coordinator types are considered, drawing on [18], that allow for various forms of service composition to be expressed in our model.

1. **Sequential**: Where the execution of a service is dependent on the previous one. This coordinator can handle sequential process-oriented service composition with provision for both Sequential with Commit Dependency (SCD) and Sequential with Data Dependency (SDD).

2. **Parallel coordinator**: This coordinator handles parallel process-oriented service composition covering Parallel with Commit Dependency (PCD), Parallel with Data Dependency (PDD) and Parallel without Dependency (PND). Sequential Alternative: the services will be attempted in succession until one produces the desired outcome, as specified by some criterion (e.g. cost, time, etc). This coordinator is particularly useful for forward recovery as will be discussed in Section IV.

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

4. **Data-oriented**: this coordinator handles data-oriented service composition and specifically deals with released data items within a transaction (between its sub-transactions) or partial results released between different transactions.

5. **Delegation**: this coordinator allows the whole transaction or a sub-transaction to be delegated to another platform, e.g. as a means of overcoming traffic bottlenecks or low bandwidth connections.

Fig. 1 shows a transaction tree with four basic services whose order of execution is determined by the five coordinator types employed. We have adopted the notation of [14] extended with a symbol for the data-oriented coordinator.

![Fig. 1 Transactions in a tree structure](image)

The tree structure representation of a transaction allows us to exemplify the local coordination that is required for the services involved to be performed in unison in accomplishing the goal prescribed by the transaction.

The long-term nature of business transactions frames the concept of a transaction in Digital Business EcoSystems and makes defining a consistent transaction model even more challenging. The conventional view of a transaction [4] is based on the ACID (Atomicity, Consistency, Isolation, Durability) properties, which have been successfully applied to relational database management systems. However, in many new distributed applications such as CAD projects, e-commerce solutions and advanced simulators, these properties present unacceptable limitations. It can be argued that the conventional view of a transaction cannot capture the primary requirements of the era of DBEs.

From a business point of view, most usage scenarios in Digital EcoSystems involve long-term transactions and thus Atomicity is an unacceptable constraint. From a SOA for distributed transactions point of view, Isolation can lead to significant degradation of performance in the services offered (as critical data is locked until a transaction completes) or to increased probability of deadlock (as services may be locked into composite transactions that do not terminate). Additionally, in long-term transactions, partial results need to be shared between different transactions before their termination (commitment). This poses further challenges with regard to ensuring consistency of the underlying transaction model.

### B. Managing Service Dependencies

We have seen that in our approach transactions are understood as pertaining to SOC for B2B interactions. Hence, a transaction has structure, comprising a number of sub-transactions which need to be coordinated accordingly (and locally), and execution is long-term in nature.

In order to relax the ACID properties, particularly Atomicity and Isolation without compromising Consistency, we need to consider some additional structure that will warranty the consistency of the transaction model. Maintaining consistency is critically important within a highly dynamic and purely distributed environment of a Digital EcoSystem. But at the same time, the model should defer from any tight-coupling between initiator and coordinator or between initiator and participant, as is the case with WS-Trans [2]. In order to ensure that we can exploit the potential benefits of SOC, we want to respect its primary requirements such as loose-coupling. We therefore keep state information at the deployment level, rather than the realisation level, and defer from interfering with service execution as such as we wish not to break local autonomy of the realisation platform.

Note that ‘state’ here we refer to the activations within a transaction, in terms of its local coordinators and/or subtransactions at any given point, and not to state as in the execution of the invoked services. The latter notion of state is related to the realisation level and is to be dealt with by the local platform on which these services run.

In what follows, we introduce two directed graphs that capture the dependencies between sub-transactions (coordinators or services) of a single transaction or belonging to different transactions. Keeping track of such dependencies is essential if the underlying transaction model is to provide capabilities for reverse action (in case some subtransaction or even a whole transaction fails or is aborted), for applying deadlock control, for transparency during delegation. Furthermore these two graphs together with the transaction tree allow for releasing partial results between different transactions in our approach.

The Internal Dependency Graph (IDG) keeps logs of value dependencies within a transaction tree. Each node
represents a coordinator or service (sub-transaction) and the
direction of the arc between nodes indicates a dependency
of one node on another.

In the case of the sequential coordinator, for example, the
services will be executed serially and each can use the re-
results of the previous one. In the transaction tree of Fig.1, s3
and s4 are children of a sequential coordinator and hence s4
can use the results released by s3. This has as a consequence
that if s3 is aborted, then s4 must also be aborted. This de-
pendency between s3 and s4 is shown in the IDG of
Fig.2(i).

When value dependencies exist between children of a
parallel coordinator, then if one of the children is aborted,
the rest of the children must also be aborted. The structure
of the corresponding IDG, see Fig.2(ii), shows such de-
pendencies and in this sense facilitates recovery manage-
ment to rollback the system.

In a highly dynamic and purely distributed environment
such as a Digital Business EcoSystem, it is often the case
that a sub-transaction requires access to a data item released
(possibly as a partial result) by a sub-transaction belonging
to a different transaction.

In other words, dependencies may exist not only within a
transaction but also between transactions (which, may take
place on different platforms). For example, consider the case
of (compensable) subtransactions that release partial
results in a conditional commit state [14].

To capture such dependencies we introduce the External
Dependency Graph (EDG). This keeps track of dependen-
cies between (services or coordinators of) different trans-
cactions. The log structure it provides can be used in recovery
routines for running a compensating procedure.

Fig.3 shows part of the EDG for the transaction trees T1
(of Fig.1) and T2. The data-oriented coordinators d1 and d2
of T1 release partial results that are required by d3 of T2.

Now, if for some reason d1 (or any other sub-
transactions on which d1 depends, for that matter) was
aborted, then d3 should also be aborted along with any sub-
transactions of T2 which depend on it. Based on the log in-
formation provided by the EDG and the corresponding
transaction trees, we would like to recalculate d3 based on
the data items released by d2 and defer from aborting (at
least part of) transaction T2.

The IDG and EDG provide a means of recording impor-
tant system logs which can be stored locally, on the corre-
sponding local coordinator, but their effect is both local, in
terms of local faults, forward recovery and contingency
plans, and global, in terms of abortion, restarting, recalculat-
ing, and alternative execution.

C. Formal Reasoning on Service Dependencies

We have seen that a transaction can be represented by a
tree structure, showing the coordination of the leaves in per-
foming the corresponding transaction. In our behavioural
model of a transaction it suffices to use formal notation for
the leaves only. The aggregation coordinators (nodes) are
manifested in the structure of the resulting formal construc-
tion, and there is no need for additional notation. A transac-
tion T, then, is associated with a set of leaves L which con-
sists of a set of basic services S, a set of data-oriented coordi-
nators D and a set of delegation coordinators Dlg. Thus, L
= S U D U Dlg.

A transaction is also associated with a finite set of events
that may occur (on its sub-transactions) upon activation, e.g.
re service invocation, initialisation, commitment, service re-
turn, release result (return), termination, abort, etc. We de-
note this set by M. These events take place on the leaves
and therefore it seems appropriate to say that each leaf is in
turn associated with a set of events that may occur on that
leaf, depending on its nature. We denote this set by
\( \mu (l) \),
\( l \in L \), and require that \( \bigcup_{l \in L} \mu (l) \subseteq M \).

In any behaviour of a transaction T, each subtransaction
on the leaves will be activated and experience a sequence of
events formed over the corresponding set \( \mu (l) \), \( l \in L \). We
may thus describe the behaviour of the transaction by as-
signing such sequences to each of its leaves.

Definition 1. Let T be a transaction. We define V_T to be
the set of all functions \( \nu : L \rightarrow M^* \) such that \( \nu(l) \in \mu (l)^* \).
We refer to elements of \( V_T \) as transaction vectors.

\( \mu (l)^* \) denotes the set of finite sequences over \( \mu (l) \).

Mathematically, the set \( V_T \) is the Cartesian product of the
sets \( \mu (l)^* \), for each l. Effectively, transaction vectors are n-
tuples of sequences where each coordinate corresponds to a
leaf in the transaction tree (hence, n is the number of leaves)
and contains a finite sequence of events that have occurred
on that leaf. When an event occurs on a leaf, it appears on a
new transaction vector at the appropriate coordinate.

In what follows we describe the basic order-theoretic
properties of transaction vectors and show how the order
structure of sets of such vectors expresses ordering con-
straints on the activation of sub-transactions. We have seen
that transaction vectors are essentially tuples of sequences.
This can be exploited in defining operations on the vectors
in terms of well-known operations on sequences.

Definition 2. For \( \nu, \pi \in V_T \), we define
\( \nu \circ \pi \) to be the unique vector \( \upsilon \) such that \( \upsilon(l) = \nu(l) \cdot \pi(l) \),
for each $l \in \mathbb{L}$ (concatenation)
- $\underline{u} \leq \underline{v}$ if $\underline{u}(l) \leq \underline{v}(l)$, for each $l \in \mathbb{L}$ (prefix ordering)
- $\text{lub}(\underline{u}, \underline{v})$ to be the vector $\underline{w}$ such that $\underline{w}(l) = \min(\underline{u}(l), \underline{v}(l))$, for each $l \in \mathbb{L}$
- $\text{lub}(\underline{u}, \underline{v})$ (if it exists) to be the vector $\underline{w}$ such that $\underline{w}(l) = \max(\underline{u}(l), \underline{v}(l))$, for each $l \in \mathbb{L}$
- if $\underline{u} \leq \underline{v}$, then we define $\underline{v} / \underline{u}$ to be the unique element $\underline{z} \in \mathbb{V}_T$ such that $\underline{z} \leq \underline{v}$ and $\underline{z} \geq \underline{u}$ (right-cancellation)

It can be shown (by arguing coordinate-wise) that a set of transaction vectors equipped with the operations of concatenation and prefix ordering forms a monoid and a partial order. $\Delta_T$ is used to denote the empty vector which has the empty sequence on each of its coordinates.

**Proposition 1.** A set of transaction vectors $V_T$ is
- a monoid under $\cdot$ and identity $\Delta_T$,
- a partial order under $\leq$ and bottom element $\underline{1}$.

The operations $\text{glb}(\ )$ and $\text{lub}(\ )$ of Definition 2 give the greatest lower bound and the least upper bound, respectively, of $\underline{u}, \underline{v} \in \mathbb{V}_T$, in the usual sense of lattices and domain theory [5]. 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 $\underline{v} / \underline{u} \cdot \underline{u}$ is the ‘continuation’ of $\underline{u}$ that extends it to $\underline{v}$.

Transaction vectors can be seen to be built up from the empty vector by a series of concatenations with a specific kind of vector, the so-called column vectors [Mos05], each of whose coordinates is either empty or contains a single event. For example, the column vector $\underline{e} = (s_1, \Lambda, \Lambda)$ represents the activation of the leaf corresponding to the first coordinate. If $s_1$ is intended to occur only after both $s_3$ and $s_4$ have, then this is described in the transaction vector $\underline{v} = (s_1, s_3, s_4)$ which is obtained as $\underline{v} = (\Lambda, s_3, s_4)(s_1, \Lambda, \Lambda) = (s_1, s_3, s_4) = \underline{v}$.

The study of the order-theoretic properties of such vectors in [12] shows that it is possible to express sequential, parallel and alternative behaviour. For the purpose of the present paper it suffices to understand that the ordering relation between different vectors of a transaction reflects the orderings between activations of its subtransactions.

For instance, $s_1$ and $s_2$ are sequential ($s_2$ can only be activated after $s_1$) in Fig.4(i) while they are mutually exclusive (alternative) in Fig.4(ii) and they are concurrent in Fig.4(iii).

![Fig.4 Order structure of transaction vectors](image)

Notice that the set of vectors in (i) does not include $(\Lambda, s_2, \Lambda)$, in (ii) does not include $(s_1, s_2, \Lambda)$ while in (iii) it includes all four vectors.

The transaction tree shown in Fig.1 has 6 leaves. The services $s_1$ and $s_2$ are to be executed in parallel (concurrently) followed by the data-oriented coordinator $d_1$. If the partial result released by $d_1$ (see Fig.3) does not meet the desired outcome, then $s_3$ and $s_4$ are executed in succession (sequentially) followed by $d_2$.

To model the behaviour of the transaction in our formalism, we assign each leaf to a vector coordinate (from left to right here). This results in the set of 6-tuples shown in the Hasse diagrams of Fig.5, which describe all possible series of sub-transaction activations in performing the transaction $T_1$ given in Fig.1. In Fig.5 there is a choice between the behaviour described in the diagram on the left and that on the right, and this reflects the sequential alternative scenarios of transaction $T_1$. This choice is deterministic and will be resolved on the basis of whether $d_1$ satisfies the desired outcome. Furthermore, in case some subtransaction fails, the vector-based description is used in providing compensating transactions, taking up on the “do-compensate” and “validate-do” behaviour patterns.

Notice the lozenge formed by $s_1$ and $s_2$ which execute in parallel (in both cases). Also, notice that the Hasse diagram on the left implies that $(s_1, s_2, d_1, s_3, \Lambda, \Lambda) \leq (s_1, s_2, d_1, s_3, s_4, \Lambda)$ which means that $s_4$ can only happen after $s_3$ has (sequentially).

![Fig.5: Transaction vectors for $T_1$](image)

Therefore, in our approach given the tree structure of a transaction we may derive a formal description of its intended behaviour, in terms of activations of its sub-transactions and the coordination between them. The resulting behavioural patterns (see Fig.5) can be analysed before run-time as a means of preventing certain anomalies (such as race conditions) which could result in unexpected behaviour when the transaction actually takes place [12].

### IV. Compensability and Forward Recovery

Within a Digital Business EcoSystem, a number of long running and multi-service transactions take place, each comprising an aggregation of sub-transactions. There is an increased likelihood that at some point a subtransaction might fail. This may be due to a platform failure or its coordinator not responding or, simply, because it is a child of a Parallel Alternative coordinator and some alternative sub-transaction has already met the pre-set condition. It transpires that there must be a way to compensate for such occasions and defer from aborting or even restarting the whole transaction.

Compensability is an intrinsic aspect of a transaction model, especially within a business environment. Particular thought has been given in setting up our model for provid-
ing a compensation mechanism that warrants consistency. When a subtransaction fails, typically its coordinator has to be aborted along with its children.

We have seen (Section III) that two types of information are released before commitment, in 2PC, that give rise to dependencies between subtransactions: (i) released results between subtransactions of a transaction, for which we use an internal log with the structure of the IDG that shows the internal routine of recovery; (ii) partial results between different transactions before their commitment, for which we use another log with the structure of the EDG that shows the external dependencies.

To ensure consistency at all times, such dependencies need to be taken into account so that dependent subtransactions are also aborted. At the same time, the remaining (safe) subtransactions that may have provided valuable results should not be aborted or restarted, what is often referred to as omitted results.

In the remainder of this section, we outline the mechanism for compensation in our approach and describe the use of forward recovery to address omitted results. Our aim in designing recovery management for our model has been to (a) isolate the failed subtransactions (those using its partial results either directly or indirectly) and (b) rollback the system to a consistent checkpoint (normally, the start point of the failed subtransaction).

The part of the system that is affected by a failed subtransaction is determined using the corresponding IDG and EDG. More specifically, we create a tree, the so-called compensation tree, in a similar fashion to the directed graphs of Section III. The failed subtransaction, or coordinator becomes the root of the tree and its children are all its dependent subtransactions. We illustrate this process of localising the ‘damaged’ part of the system with the example of the previous section.

Consider the transaction trees shown in Fig. 3. If the subtransaction involving the execution of service s2 fails, for example, then any subtransactions that are dependent on s2 should be aborted. These are identified in the corresponding IDG (Fig. 2) for dependent subtransactions within T1, and EDG (Fig. 3) for dependent subtransaction in T2. The resulting dependencies in this case are shown in Fig. 6 on the left, which says that service s1 must be aborted along with any result released by the data coordinator d1 as well as any result used by d3 in T2.

Having identified the affected subtransactions, we may proceed to rollback the system. Rollback of services s2 and s1 is based on the semantics of the service description. The rollback procedure for the data-oriented coordinators is more involved as we need to ensure consistency of partial results. For this purpose we apply a locking mechanism to isolate the affected part: an internal lock, for any data items used internally (based on IDG), and an external lock, for data items available to other transactions (based on EDG). By traversing these graphs, the locks are converted to a recovery lock that allows access only to the recovery routine, which eventually creates the compensating transactions. Due to space limitations we do not discuss the locking mechanism in more detail.

Hence, the abrupt termination of s2 does not necessarily cause the whole transaction T1 to be aborted. It is left in an intermediate state (subtransaction failed) where we apply forward recovery and allow T1 to eventually transition to a Confirm state, essentially following the “do-compensate” behaviour pattern. In fact, execution in this case will continue with s3 (this is where forward recovery comes into play) and the remaining subtransactions in that part of the tree. This is because the root of T1 is a sequential alternative coordinator which defines alternative execution scenarios. These were formally described in terms of the corresponding transaction vectors in Fig. 5, where the two Hasse diagrams describe the alternative ways of executing transaction T1. In the occasion that s2 fails, the diagram on the right is no longer possible but execution will continue following the path described in the diagram on the left.

The failure of service s2 has a knock on effect on T1. This can be captured in our formal description by eliminating/hiding the coordinates corresponding to failed services and any other subtransactions dependent on them. The following definition introduces notation for the projection of transaction vectors onto vectors of ‘live’ subtransactions; live, in the sense that they have not failed or been aborted.

**Definition 3.** For \( \forall \in V_r \), we define \( \mathbb{V}_l = \mathbb{V}_{l:A} \) where \( s \in S \) is the subtransaction that failed and \( A \) is the set of all aborted subtransactions due to \( s \), and including \( s \).

This construction is reminiscent of the projections used to give the trace semantics of parallel composition in CSP [10]. In the case of T1, whose transaction vectors were given in Fig. 5, by applying Definition 3 we get the set of vectors depicted in the Hasse diagram of Fig. 7.

![Fig. 7 Forward recovery scenario for T1](image)

It can be seen that the execution vectors of T1 can still take place following the alternative scenario (initially described in Fig. 5 on the left) that leads T1 to a Confirm state, following the “do-compensate” behaviour pattern.

We note that the ordering relation ‘\( \preceq \)’ in \( V_r \) is transitive (and also reflexive and antisymmetric) by Proposition 1, and thus, the ordering between the vectors in Fig. 5 is preserved in their restricted form of Fig. 7. We omit further details.

The simple example described here demonstrates the
need for covering alternative service execution, in a principled manner, within a transaction model. Apart from describing the pathway of forward recovery (given within T1 here, as we have not provided the vectors for T2), the transaction vectors in our behavioural description are used for verifying the proper check point for rollback (see Fig. 7); in this case, it is the point in which nothing has happened yet, i.e. the starting point (root) of transaction (tree) T1.

Now, if service s3 also fails, then s4 must also be aborted, d2 will not be able to provide a consistent result and d3 will not be able to use any result from d2. These dependencies are depicted in Fig. 6 on the right. In this case, T1 now follows the “validate-do” [17] pattern so that potential effects on other transactions, i.e. T2, are undone.

In this case, the whole transaction T1 has failed (since no alternative scenario was successful). This does not necessarily mean that the whole of transaction T2 also has to be aborted. In fact, we apply the “do-compensate” behavioural pattern for T2 in this case. Based on the corresponding IDG and EDG, we may infer that d3 is the only part of T2 that needs to be aborted, along with any subtransactions that depend on it. Further, we have that d3, s5 and s6 are children of a sequential alternative coordinator (see Fig.3), which means that execution of T2 can continue with s5.

Therefore, T2 follows the “do-compensate” behavioural pattern once s2 fails, while T1 follows the “compensate” pattern until, and if, s3 also fails, in which case it turns to “validate-do”.

In a certain important sense, our coordination model allows for localising the points of failure, identifies the affected parts of the transactions involved and insulates the rest of the transactions from the localised failures.

V. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a model for coordinating distributed long-term transactions in Digital Business EcoSystems. Various forms of service composition have been considered in order to provide a closer representation of business transactions within a service-oriented architecture. We have given a formal description of the behaviour of multi-service transactions, which can be used to determine whether the services are correctly orchestrated and hence reason about the applicability of a solution. Our model is considered at the deployment level of SOA and we have presented a compensation mechanism that ensures consistency while respecting the loose-coupling of the underlying services. Further, it addresses omitted results, through forward recovery, in a way that does not break local autonomy.

Another strength of our approach is that it supports both the “do-compensate” and “validate-do” behaviour patterns without breaking the autonomy of the local platforms or making any assumptions at the realisation level. This is particularly important in a business environment, especially for SME’s. Further, these behaviour patterns can be applied dynamically on transactions in our approach, as was partly demonstrated in our example.

Recently, the formal verification of service composition in terms of the order of service invocation has received increasing interest (e.g. see [7]). Drawing upon earlier work on a formal notion of composition [11] within the vector-based behavioural description, which note can be obtained directly from scenarios [12], we could complement model-based verification techniques with respect to true-concurrency. This is something we are keen to explore further as it lays the groundwork for a coordinated and collaborative service invocation specification to support long-term and multi-service transactions in Digital Business EcoSystems.

ACKNOWLEDGEMENTS

This work was supported by the EU FP6 funded Projects DBE and OPAALS.

REFERENCES