An Animation Tool for Exploring Transactions in a DE

Sotiris Moschoyiannis, Member, IEEE, Paul J. Krause, Fellow, IMA, Pavlos Georgiou

Department of Computing, Surrey University, Guildford, GU2 7XH, UK
e-mail: (s.moschoyiannis, p.krause, p.georgiou)@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. In previous work we have described how the so-called *transaction languages* can express concurrent and distributed interactions between online services in a transactional environment. In this paper we outline how transaction languages capture the history of a long-running transaction and highlight the benefits of our *true-concurrent* approach in the context of DEs. This includes support for the recovery of a long-running transaction whenever some failure is encountered. We introduce an animation tool that has been developed to explore the behaviours of long-running transactions within our modelling environment. Further, we discuss how this work supports the declarative approach to the development of open distributed applications.**

Index Terms—**digital ecosystems, service composition, collaborative working, web transactions, digital economy.**

I. Introduction

The concept of a Digital (Business) Ecosystem has been in circulation since 2002 \cite{Nachira}. This was in response to the observation that although there were at the time over 19 million 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, 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" \cite{ChangDE}. 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 \cite{PJK09} 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 \cite{PJK09} that the nature of the latter, the digital infrastructure, can impact (undesirably to an extent 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. While the basic premise of SOC \cite{Pap03} is that business requests can be defined in terms of finer-grained subtasks that are available as loosely-coupled online services so that applications are built up as networks of collaborating applications distributed within *and* across organisational boundaries, its realisation through WS-* web services efforts \cite{WWWC}, also known as Big Web Services \cite{RichardsonRuby} are rooted in enterprise systems vendors and the history of that industry.

This has lead to two issues in particular that we believe have held back the evolution of the infrastructure or environment for DEs. Firstly, the remote procedure call style of SOAP, WSDL and UDDI leads to a tighter coupling between services and their respective host organisations than is appropriate for a DE. Secondly, the modelling of business transactions has focussed on the perspective of a single (coordinating) organisation.

We explore the first issue in []. In this paper we focus on the need to model interaction scenarios (including transactions) from a global perspective (service *choreography*) rather than from an individual participant's viewpoint (service *orchestration*). We argue that while the latter has received good attention from the formal modelling community, the former still has a number of open issues that need addressing. Specifically, there is an increasing consensus that this requires a formalism that handles true concurrency (rather than falling back on an interleaving semantics). This is something that is naturally provided by *vector languages*. True to our interest in open Digital Ecosystems, we illustrate our usage of vector languages to model service choreography with a modelling tool we have developed that we have made freely available as a web service.

In the next section we review the main approaches that have been taken to modelling service interactions, with a specific focus on long running transactions. After that we provide an overview of the design of our modelling tool. We then illustrate the use of vector languages for modelling service orchestration with that tool. Finally, we conclude with some pointers to the next steps we propose to take.

II. Modelling Approaches to Service Interaction

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. This is evident in mainstream transaction protocols for business activities such as the *Business Transaction Protocol (BTP)* \cite{BTP} and *Web Services Transaction (WS-Tx)* \cite{WS-tx} (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 \cite{Lit03}.

In addition, standardisation proposals that deal with the interplay between services and business processes, such as WS-BPEL \cite{WS-BPEL} (service *orchestration*) and WS-CDL \cite{WS-CDL} (*choreography*), use the concept of a *long-running* or *web* transaction and include modelling constructs for failure and compensations, but do not support a definitive mechanism for recovering a transaction; this is typically dealt with in ad-hoc ways by the application programmer. Furthermore, 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.

A. Concurrency models

The various approaches to concurrency proposed in the literature can be classified in two classes: those based on the assumption that concurrency can be reduced to its sequential non-deterministic simulation, the so-called interleaving semantics, see for example the influential contributions by Milner \cite{CCS} and Hoare \cite{CSP} and the rich literature they have given rise to; and those in which concurrency is considered as a primitive notion and is modelled explicitly by means of causal independence, the so-called *true concurrency* or *partial order semantics*, see for example the seminal work by Petri \cite{PetriNets} and Mazurkiewicz \cite{Maz88}, Winskel \cite{NPW81}, Shields \cite{Shi85}.

The basic difference between the two classes can perhaps be seen most clearly by means of a simple example. Let us consider a system concurrently performing actions a and b. From the point of view of any interleaving semantics, $a \parallel b$ and $a{:}b + b;a$ would be considered equivalent models of such a system (we use CSP-like expressions here, where '||' denotes parallel composition, '+' denotes a choice, and ';' denotes sequential composition). Even though one says that a and b happen concurrently and the other says that there is choice between doing a and then b or doing b and then a, these behaviours are identified as the same even by *bisimulation* \cite{CCS}, one of the most discriminating equivalence notions for concurrent systems.

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. In other words these are determined to be two different behaviours of the system. This can be achieved by means of a notion of *independence* (a symmetric, irreflexive relation) defined on actions that can give rise to an equivalence relation on sequences of actions that involve consecutive independent actions. In this way, concurrency is identified *explicitly* instead of it being reduced to the non-deterministic choice between the sequentialisations of the actions involved.

In a transactional environment, the choice of a true-concurrent vs interleaving semantics can have a significant role to play, because the way the forward actions are modelled (while the transaction is executing successfully) has a bearing on recovery management since the forward actions need to be compensated for, in the reverse order, whenever a failure makes this necessary.

B. Formal approaches to business transactions

There are common features in the approaches to describing the behaviour of a long-running 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* \cite{G-MS87}.

The work on *compensating CSP* \cite{cCSP} is motivated by XLANG \cite{XLANG} (Microsoft's predecessor to WS-BPEL) and extends the trace semantics of CSP \cite{CSP} to model compensations in long-running transactions. Preceding this was work on the StAC language \cite{BuF04}, which was inspired by BPBeans, a framework for modelling business processes that was 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 \cite{BLZ03} uses π-calculus for modelling long-running transactions and the resulting *πt*-calculus targets BizTalk [], the visual environment of XLANG. \cite{BLM02} addresses short-lived transactions in BizTalk. A timed extension to π-calculus, *web*π, is considered in \cite{LaZ05}, which does not target a particular standardisation initiative but attempts to formalise key concepts in orchestration, like our approach does albeit for the choreography of the services in a long-running transaction. The interleaving semantics used for this results in the characterisation of timed bisimilarity, which is aimed to deal with latency in business activities. Other formal approaches to reasoning about the composition of web services and transaction mechanisms include [] which uses interface automata [] and LTL to describe and verify properties using SPIN, and \cite{GLM07} which proposes an event calculus extension which includes a semantic notion of structural congruence but is interpreted over labelled transition systems.

Of the formal approaches described above, cCSP \cite{cCSP} and *πt*-calculus \cite{BLZ03} are of particular interest here because, like our approach, they consider subcomponents of a transaction or subtranscations within a larger transaction. In concurrent execution, \cite{BLZ03} considers multiple copies of the same subtransaction executing concurrently inside a transaction (see Section 3.2 in \cite{BLZ03}). The transaction is allowed to complete even if some subtransaction fails. This may be suitable for certain business request 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 a travel arrangement transaction (this example is also considered in BTP \cite{Lit03}), if the subtransaction responsible for booking a flight fails, then there is no longer a need to book a hotel and schedule local transport at the destination.

In *compensating CSP* \cite{cCSP} 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 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 DE environment.

\cite{BMM05}, which also applies a CSS-style interleaving model to parallel *Sagas* \cite{parallelSagas} for orchestration, avoids this problem by including special 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 \cite{formal-choreo} describes a choreography language that is targeted at WS-CDL \cite{WS-CDL}. 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 \cite{BCPV04} which targets the Web Services Choreography Interface (WSCI) \cite{WSCI}, 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 \cite{GLM07} talks about choreography but the approach in fact formalises WS-BPEL, hence it targets orchestration.

It transpires that most works 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 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 subactivities in the collaboration required to meet a given business request.

It is also worth pointing out that an interleaving semantics, where concurrency between actions is reduced to a 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 where applications and business requests require a similar collaboration but across organisational boundaries and hence 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 \cite{MarKra09} 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 (and perhaps different ones even for each run of the corresponding transaction).

III. Modelling Transaction Behaviour

As mentioned before, our aim is to provide formal support for a long0running transaction in terms of the underlying service interactions (e.g. invocations of RESTful APIs of online resources). In this section we provide a brief outline of the key ideas behind the formal model, which are then demonstrated in the visualisation tool presented in Section IV and V. A formal account of modelling forward behavior in our approach has been given in \cite{IEEE-DEST08} while a comprehensive treatment of the formal semantics for both forward and compensating behavior is given in \cite{FIpaper – *in press*}.

The WS-CDL choreography standard \cite{WS-CDL} by W3C aims to provide a global view of the interactions between multiple online services in order to achieve a common goal. The standard 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 \cite{MarKra10} 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 \cite{PTD06} 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 \cite{DEST08-trans} we have provided a schema for describing *transaction contexts*. Fig. \ref{fig:transtree} shows a transaction tree with five basic operations on services or 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 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.

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 as shown for example in the scenario of Figure 1.

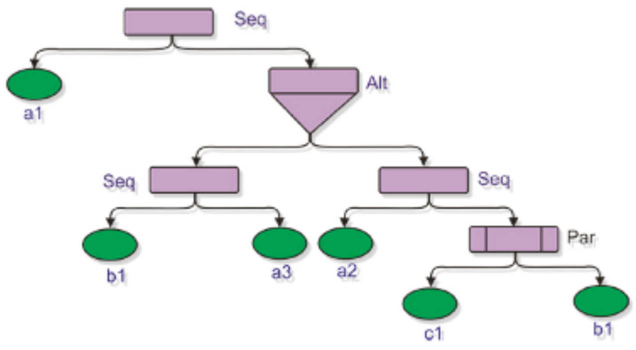


Figure 1. Transaction context drawn as Transaction Tree

Hence, a transaction is associated with a set of actions *M* and each subcomponent 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.

Each participant will have its own set of constraints that dictate the order in which the actions (in its own subset of *M*) 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 its corresponding actions have been performed. This describes the individual participant's viewpoint, which is of interest in service *orchestration*, e.g., see WS-BPEL \cite{WS-BPEL}, and can be described in terms of a sequence of action names. Such a sequence is recorded in our formal notation of transaction languages on a given coordinate of the transaction vectors. The set of all behaviours of participant *i* is given by the set of all string prefixes of its set of actions (with reference to \cite{FIpaper} these are elements of $\mu(i)^*$, where the sets $\mu(i)$ partition *M*).

The behaviour of the transaction as a whole, from a global perspective, which is of interest in *service choreography*, e.g., see WS-CDL \cite{WS-CDL}, can be described using a tuple of strings, one for each participant, recording the observable actions across the whole conversation involved in the transaction. We do not use one tuple, but rather a tuple is formed for each significant action (service interaction) occurring in the transaction. Thus, we end up with a set of such tuples, which is what constitutes the corresponding *transaction language*.

Each tuple or vector in the language can be seen as describing a *snapshot of behaviour*, in that it describes what actions have taken place and on which coordinator components of the transaction. The language as a whole provides a global view of the ordering conditions and constraints across all participants in the multi-party conversation involved in the transaction. This draws upon Shields' *vector languages* which provide a generic behavioural model for communicating systems and can express *true concurrency* in that independence on actions is lifted onto vectors, and the generated equivalence relation that identifies concurrent execution is in turn lifted onto the vector language level.

In what follows we briefly outline the way transaction vectors are obtained in modeling forward behavior (by coordinate-wise concatenation) and the way that compensating actions are handled in transaction recovery (by applying right-cancellation coordinate-wise) in our approach. A full account can be found in \cite{FIpaper- *in press*}.

For instance, the *transaction vector* (a_1a_2, Λ, c_1) provides a snapshot of the interaction within an online collaboration that involves three online resources. In particular, it describes that portion of the interaction in which participant CC_1 has received a request (e.g. a GET operation) a_1 followed by a_2 on the resource it provides and participant CC_3 has received c_1 on its own resource while no operation has been received yet on the resource provided by CC_2(Λ denotes the empty sequence). The schemas for generating serialisable representations (in Json and xml) of the dependencies and the required orderings are discussed in the context of the tool presented in Section V of this paper.

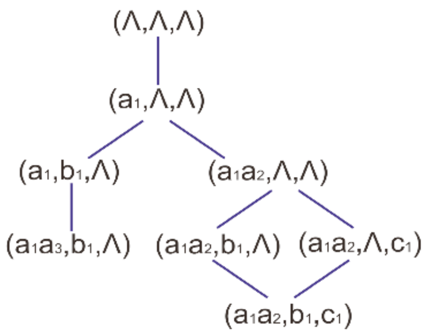


Figure 2. Pattern of interactions for transaction of Fig.1

In describing forward behaviour, the transaction vectors record the sequences of requests issued by the different participants of the transaction. This is done by lifting the well known operation of concatenation (denoted by '.') on sequences onto vectors, by performing it coordinate-wise. For example, the vector (a_1a_3, b_1, Λ) is obtained by concatenating the previous vector (a_1, b_1, Λ) with the vector (a_3, Λ, Λ) as follows:

$$(a_1, b_1, \Lambda). (a_3, \Lambda, \Lambda) = (a_1a_3, b_1, \Lambda)$$

The vector (a_3, Λ, Λ) is used to represent the action a_3 and we refer to such vectors as *action vectors*. Hence, the set of vectors has an additive structure and this infers a partial order on the set of such vectors, which comprise the transaction language for a given transaction.

In our example, the transaction vector (a_1, b_1, Λ) is smaller than the transaction vector (a_1a_3, b_1, Λ). It is the action vector that extends it to its larger one. This determines *causality* – a_1 and b_1 must happen before a_3 can. Some transaction vectors may be incomparable. For example, the transaction vectors (a_1, b_1, Λ) and $(a_1a_2, \Lambda, \Lambda)$ are incomparable and denote *conflict* - a choice between doing b_1 on CC_2 and doing a_2 on CC_1. Incomparable vectors also capture concurrency. For example, the vectors (a_1a_2, b_1, Λ) and

(a_1a_2, Λ, c_1) are incomparable and denote *concurrency* – b1 on CC2 and c1 on CC3 happen concurrently. The action vectors that lead to these two vectors, i.e., (Λ, b_1, Λ) and (Λ, Λ, c_1) are *independent* – the actions that appear on them concern distinct coordinates, and hence distinct participants. Whenever independent action vectors are both enabled after the same behaviour (after $(a_1a_2, \Lambda, \Lambda)$ in this case) and they occur consecutively, then the corresponding actions are concurrent.

As for compensation, this also draws upon a well known operation on sequences, namely cancellation '/'. If x and y are sequences given by $x = abc$ and $y=bc$, then $x / y = a$. This operation is also lifted onto vectors. In fact, we are interested in applying right-cancellation to transaction vectors with actions vectors. For example, if we apply '/' to the transaction vector (a_1a_3,b_1,Λ) with the action vector (a_3,Λ,Λ) we get the vector describing the immediately preceding behaviour (see Figure 2) as follows:

$$(a_1a_3, b_1, \Lambda) / (a_3, \Lambda, \Lambda) = (a_1, b_1 ,\Lambda)$$

A result in \cite{FIpaper-inpress} shows that this gives a transition structure to a transaction language which allows to obtain a state-based description of transaction behaviour in terms of finite state automaton.

A detailed treatment of true concurrency, as well as conflict and causality, in transaction languages can be found in \cite{Fesca08, IEEE-DEST08}. \cite{FIpaper-*inpress*} contains a more detailed account and includes the treatment of compensations in our model.

IV. A Tool for Visualising Transaction Models

We describe the development of the animation tool in this section. Although some of the details are not essential to the understanding of the tool, they do provide some insights into the state of the art in building interactive web-based applications.

The primary input into the tool is a transaction tree. As outlined in Section III, this is translated into a transaction language, effectively a set of transaction vectors. Hence, the transaction tree is converted into serialisable vector format and presented as an automaton graph. Nodes or states in the automaton represent the different vectors in the corresponding transaction language. The automaton is then animated, highlighting each state in turn both in forward execution (forward behaviour) and recovery (compensating behaviour) whenever the user introduces a failure during forward execution.

A significant feature of the architecture of the tool is that by using the client for the core processing we remove the need of data transfer and the risk of losing responses. The main challenges faced in the implementation process involved algorithm design. The drawing and animation algorithms were the hardest ones to build.

For drawing the transaction tree and the corresponding automaton graph, the difficulty lied with the positioning of nodes. Both graphs could have multiple levels and multiple nodes at each level (in the case of the automaton, these are nodes corresponding to incomparable vectors). This meant that the algorithm needed to be generic. The more nodes each level had, the more likely it was for some to collide. The main idea of the solution was to take the level that had the maximum nodes in it and start drawing both the levels above and the levels below. At each level collisions were checked and nodes were moved to avoid them. Offset was calculated when node positions exceeded the graph size. The graph would increase its size and using the offset, nodes would be drawn inside it.

The animation algorithm was even more challenging than the drawing one. The individual parts that made this algorithm difficult were the following. Firstly, the timing events, when the execution reached a node it should be coloured showing that it is processed. Secondly, the underlying formal model is a model of *true concurrency* and this meant that concurrent nodes should be animated at the same time (in parallel to each other). This was to impress the point that there was no valid behaviour in the transaction in which a concurrent action could be seen as happening on its own. Thirdly, since the formal model can also capture *conflict* (choice), all different paths that could be followed had to be calculated in advance but only one should be animated in each run of the transaction in hand. This was to impress that the choice was resolved in one way or another in each run, but not both actions could happen in a single run, Recursion was the key here as well. The algorithm would start at a header node by animating it and would use two arrays to store the nodes needed to be visited for the current forward execution of the run and the nodes that needed to be visited in recovery (compensating execution) whenever a failure was introduced. Parallel nodes would be animated in sync with the header. This information was already there, so it was not necessary to calculate this while animating.

The core technologies used to make this project possible are shown in Figure 3.

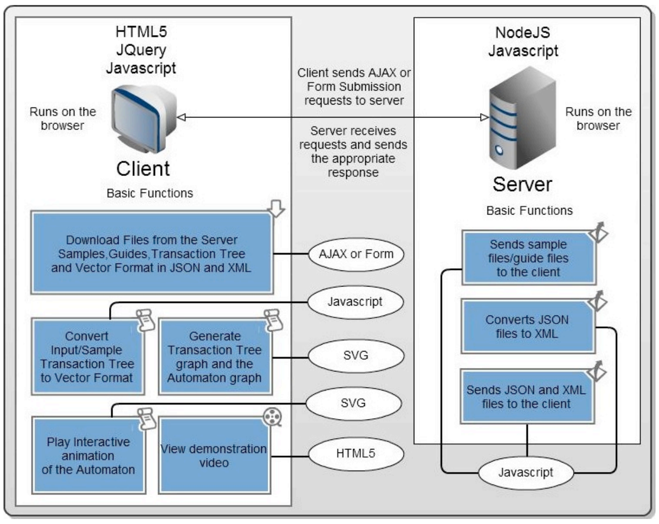

HTML5
JQuery
Javascript
Runs on the browser
Client
Basic Functions
Client sends AJAX or Form Submission requests to server
Server receives requests and sends the appropriate response
NodeJS
Javascript
Runs on the browser
Server
Basic Functions
Download Files from the Server Samples,Guides,Transaction Tree and Vector Format in JSON and XML
AJAX or Form
Sends sample files/guide files to the client
Javascript
Convert Input/Sample Transaction Tree to Vector Format
Generate Transaction Tree graph and the Automaton graph
SVG
Converts JSON files to XML
Sends JSON and XML files to the client
SVG
Play Interactive animation of the Automaton
View demonstration video
HTML5
Javascript

Figure 3: Overall architecture of the animation tool.

A. Server

The implementation of the server was done using NodeJS. This technology allows a JavaScript server implementation that runs on the browser. It makes communications with the client interface very easy. More specifically ExpressJS was used, which is a nodeJS web development framework and provides a simple and effective way of building web services.

B. Client

The client interface was developed using HTML5 and JavaScript. Having an interface that was as user friendly as possible was very important in building the visualisation tool. The *JQuery* JavaScript library was used. Some of tis functions were also helpful for the development of the processing algorithms and the communication with the server. The presentation of both the transaction tree and the automaton is done with SVG (Scalable Vector Graphics). This technology allows 2D graphics and animations alongside HTML.

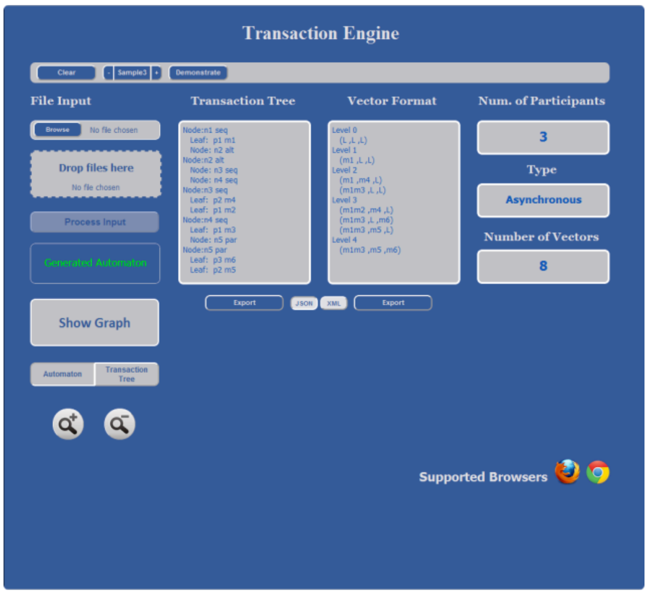
Both client and server JavaScript files were compiled from Coffeescript. This language is written a bit differently to Javascript. It helps the developer save time using shorter expressions to do the things JavaScript does – essentially it exposes all the carefully designed features of JavaScript, but with a much simpler and cleaner syntax.

C. Hosting

To deploy the web application Heroku was used. This is an easy way of deploying Web applications on the cloud. It supports six languages so far. These are Ruby, Java, NodeJS, Scala, Clojure and Python.

V. THE VISUALISATION TOOL IN ACTION

The home page of the application is shown in Figure 4. The top three buttons are *Clear*, *Sample* and *Demonstrate*. *Sample* is used to request a sample input from the server to be processed. Using *Clear* the user can clear the page of any input given and messages printed. *Demonstrate* shows a demonstration video which serves as help guide but also shows the functionalities of the application.



```
Node:n1 seq
  Leaf: p1 m1
  Node: n2 alt
Node:n2 alt
  Node: n3 seq
  Node: n4 seq
Node:n3 seq
  Leaf: p2 m4
  Leaf: p1 m2
Node:n4 seq
  Leaf: p1 m3
  Node: n5 par
Node:n5 par
  Leaf: p3 m6
  Leaf: p2 m5
```

```
Level 0
  (⊥ ,⊥ ,⊥)
Level 1
  (m1 ,⊥ ,⊥)
Level 2
  (m1 ,m4 ,⊥)
  (m1m3 ,⊥ ,⊥)
Level 3
  (m1m2 ,m4 ,⊥)
  (m1m3 ,⊥ ,m6)
  (m1m3 ,m5 ,⊥)
Level 4
  (m1m3 ,m5 ,m6)
```

Figure 4: The Visualisation Tool's Home Page

Then on the top of the left side panel, there is a *Browse* button, to browse for an input file and a box where input files can be dragged and dropped. Below is the *Process* button, which is enabled when input is fed to the application. It is used to process the input and produce the vector format and the graphs for both the transaction tree and the automaton. After that there is a box that prints out helpful messages for the user. Next is a button for showing the selected graph. This can be either the automaton or the transaction tree graph. Switching between the two is possible at any stage after they have been drawn, using the two buttons below that one. On the bottom left are the *zoom in* and *zoom out* buttons, which are useful when the user needs a closer or more high-level look of the application and the graphs.

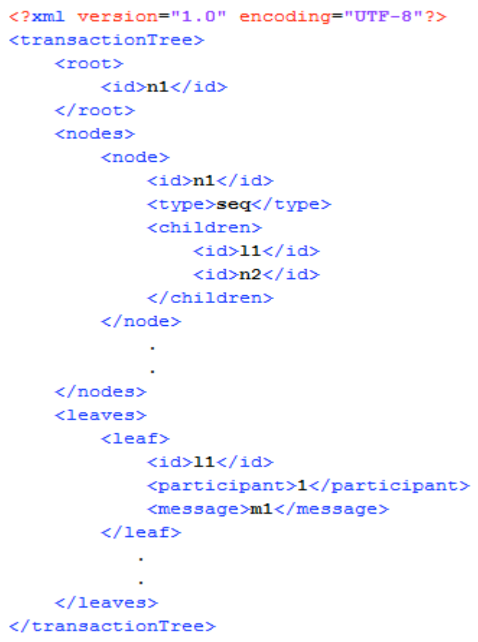
Next to this panel are two large boxes. The left one contains a formatted text representation of the transaction tree. The right one is used for the transaction vector format text representation, and vectors are grouped in levels. Below these, there are the export buttons, for exporting either of them. There is a selection between exporting XML or Json. The three smaller boxes on the right provide further details on the vector format: the number of participants; its type (synchronous, asynchronous, or both); and, the number of vectors it has. The bottom right of the page informs the user of the browsers supported – namely Firefox and Chrome.

A. Transaction Tree and Vector Format

In \cite{MarKraDEST09} 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. The application takes the transaction tree as input in both XML and Json formats. Figure 5 shows the Json representation for the transaction tree of Figure 1.

```
{
 "root":{"id":"n1"}
 ,
 "nodes":[
    {"id":"n1","type":"seq","children":["l1","n2"]},
    {"id":"n2","type":"alt","children":["n3","n4"]},
    {"id":"n3","type":"seq","children":["l2","l3"]},
    {"id":"n4","type":"seq","children":["l4","n5"]},
    {"id":"n5","type":"par","children":["l5","l6"]}],
 "leaves":[
    {"id":"l1","participant":"1","message":"m1"},
    {"id":"l2","participant":"2","message":"m4"},
    {"id":"l3","participant":"1","message":"m2"},
    {"id":"l4","participant":"1","message":"m3"},
    {"id":"l5","participant":"3","message":"m6"},
    {"id":"l6","participant":"2","message":"m5"}]
}
```
Figure 5: A Json Representation of a Transaction Tree

Figure 6 shows the corresponding XML represetantion of the same transaction tree. The root element has the id of the root node in the transaction tree. Following is the list of all the nodes in the tree. Each node can be of type *seq* for sequential, *alt* for alternative or *par* for parallel. The type shows the way that the children of that node are executed. Each node has one or two children that can be of type *node* or *leaf.* These are added to the list of children and are identified by their id.

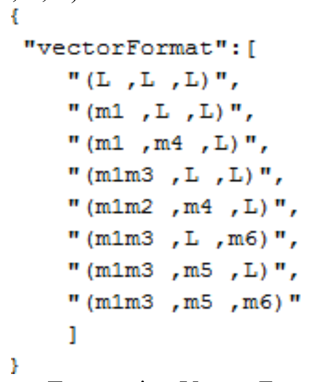


```
<?xml version="1.0" encoding="UTF-8"?>
<transactionTree>
    <root>
        <id>n1</id>
    </root>
    <nodes>
        <node>
            <id>n1</id>
            <type>seq</type>
            <children>
                <id>l1</id>
                <id>n2</id>
            </children>
        </node>
            .
            .
    </nodes>
    <leaves>
        <leaf>
            <id>l1</id>
            <participant>1</participant>
            <message>m1</message>
        </leaf>
            .
            .
    </leaves>
</transactionTree>
```

Figure 6: XML Representation of a Transaction Tree

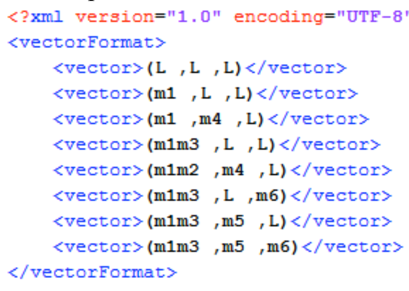
Leaves are the messages sent or the actions (invocations). Each leaf has an *id*, a *participant*, the person or object that is sending the message or invokes a service, and the message/service itself.

The output vector format representation in both Json and XML is shown in Figures 7 and 8, respectively. Once the transaction vectors are produced the corresponding order structure (recall Figure 2 in Section III) needs to be calculated along with the orderings if the respective actions; these either take place in a sequential (*causality*), alternative (*conflict*) or parallel manner (*concurrency*). To make the way this is done clearer the following sample vectors in this order will be used: (L, m2m4, L), (m1, L, L), (L, m2, L), (L, L, L), (m1m3, L, L)



```
{
 "vectorFormat":[
    "(L  ,L  ,L)",
    "(m1  ,L  ,L)",
    "(m1  ,m4  ,L)",
    "(m1m3  ,L  ,L)",
    "(m1m2  ,m4  ,L)",
    "(m1m3  ,L  ,m6)",
    "(m1m3  ,m5  ,L)",
    "(m1m3  ,m5  ,m6)"
    ]
}
```

Figure 7: Output Transaction Vector Format in Json



```
<?xml version="1.0" encoding="UTF-8"?>
<vectorFormat>
    <vector>(L  ,L  ,L)</vector>
    <vector>(m1  ,L  ,L)</vector>
    <vector>(m1  ,m4  ,L)</vector>
    <vector>(m1m3  ,L  ,L)</vector>
    <vector>(m1m2  ,m4  ,L)</vector>
    <vector>(m1m3  ,L  ,m6)</vector>
    <vector>(m1m3  ,m5  ,L)</vector>
    <vector>(m1m3  ,m5  ,m6)</vector>
</vectorFormat>
```

Figure 8: Output Transaction Vector Format in XML

The first thing done is to identify and store the events that occurred (messages or service invocations) in obtaining each vector. This is done using regular expressions; anything that matches the pattern m1…n is returned. The size of events is assigned as the vector's level. The capital letter L denotes no events (the empty sequence denoted by Λ in the formal model, recall Section III) whereas m1...n stands for an occurring event.

For example:

(m1, L, L) has events m1, so it will be on level 1.

(m1m3, L, L) has events m1, m3 so it will be on level 2.

The levels now provide the initial ordering of vectors where vectors are grouped in levels with the leftmost vector being the first one assigned to that level (lowest index in the list). Also the list of vector objects is assembled with each object having a level and its events. Using this information a parent child relationship can be formed between vectors and the ordering of them at each level can be found. The initial ordering of the sample vectors would be:

Level 0: (L, L, L)

Level 1: (m1, L, L), (L, m2, L),

Level 2: (L, m2m4, L), (m1m3, L, L)

To form the relationships a check is made between the vectors of the next level with the vectors of the previous one. The id for each vector is assigned to be its index in the list of vector objects assembled previously. The conditional statement made in Coffeescript is this:

```
if this.containsAll(events2, events1) && events2.length
==(events1.length+1)
```

where events2 are the events done by the vector in the next level and events1 the events done by the previous. So if the lower level vector has all events of the upper level vector and has one event more, then this vector is a child of the upper level vector.

Another point worth mentioning here is the way vectors in a transaction language are processed. Vectors representing sequential behaviours (*causality*) occupy a level. Vectors representing alternative choice (*conflict*) have the same parent, and same number of actions but with one event being different. For instance, two vectors that would be alternative are (m1,L,L), (L,m2,L) and their parent is (L,L,L)

Vectors that represent concurrency are treated similarly to alternatives but with the important difference that there also exists a vector that is obtained by the combination of the two action vectors representing concurrent actions. For instance, having these four vectors

(L,L,L), (m1,L,L), (L,m2,L), (m1,m2,L)

determines concurrency – m1 and m2 are concurrent. The first vector is the common parent of the second and third vector. The last vector is the combination of the second and third vectors. Finally sorting the vectors at each level is done using the parents that each vector has.

In this way we have implemented the coordinate-wise concatenations with action vectors that give rise to vectors describing transaction behaviour.

B. *Automaton and Transaction Tree Graphs*

Using the transaction tree input, its graph is produced. This is shown in Figure 9.

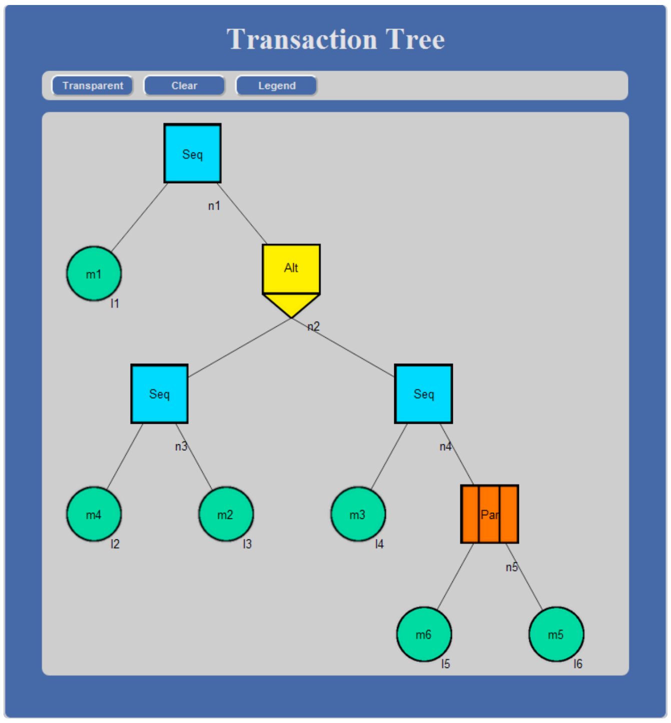


Figure 9: Drawing the transaction tree (of Figure 1)

This follows the notation found in SOC literature, \cite{SOC}, but we have added colouring to impress the different composition types. The cyan nodes denote sequential execution, the yellow nodes alternative execution and the orange nodes parallel execution. The green nodes are the leaves representing messages being sent or the service invocations (the actions). This graph has three buttons. The *Transparent* button makes it transparent to show the home page outputs. The *Clear* button clears the graph that can then be redrawn. Finally the *Legend* button shows a legend with information for each of the shapes.

Using the transaction vector format order structure, which is obtained as described in Section V -A), the automaton can be produced. The automaton can be animated showing both the forward and the recovery execution. The user can interact with the animation to introduce failures to the execution, forcing it to enter the recovery mode where all nodes that were highlighted as green before (in going forward) are effectively undone and highlighted back to white (in compensating) in the reverse order.

This can be seen in Figure 10, which shows the automaton generated for the transaction language given earlier in Figure 2 (of Section III).

The forward execution has reached the failure node and recovery execution has started. The failure node is about to become white, indicating that it is being undone. The rest of the animation will undo all green nodes unless the user selects a processed node as the point where forward execution should be retried. Here six buttons can be found. The *Run* button is used for running the animation. Surrounding it are a Plus and a Minus button for selecting a different path of

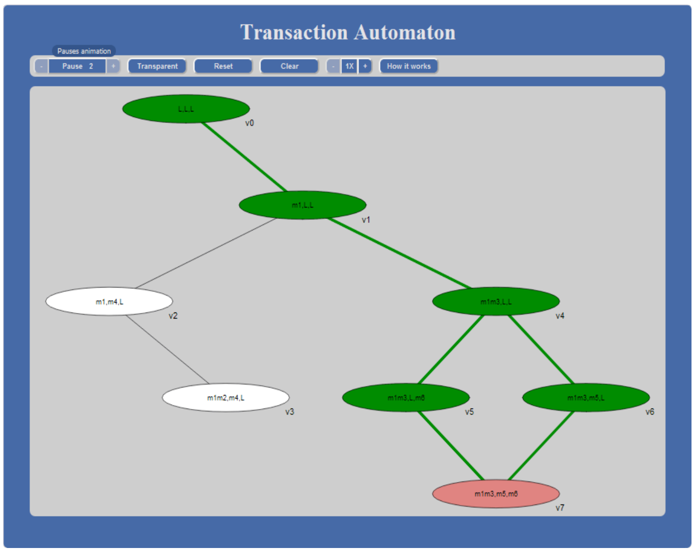


Figure 10: Animating the corresponding automaton (for the transaction language of Figure 2)

execution. This means changing the way the choices at the alternative nodes are resolved. As soon as the animation starts, Run becomes Pause for pausing and if paused, becomes the Resume button for resuming the animation. Transparent again allows a view of the home page. The Reset button resets the automaton to its initial state. Clear is for clearing and redrawing the graph. A selection of the animation speed can be made using the small plus and minus buttons surrounding the speed output. The last button, called "How it works" shows a menu of information on how the automaton animation works, e.g. how to introduce errors, how to attempt forward recovery.

VI. Conclusions

The formal model discussed in this paper is flexible enough to address a range of scenarios in environments where there is no single point of command and control. The notation afforded by *vector languages* allows us to capture what is happening on each subtransaction, and at each point during execution, across the whole transaction. Concurrency is modelled explicitly using the notion of causal independence in the model. It is essentially a partial order model of concurrency - the construction is lifted onto tuples of sequences (one for each participant of the transaction) rather than events, as in the event structures model \cite{NPW81}, or individual sequences, as in Mazurkiewicz *trace languages* \cite{Maz88}. 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. In the event of a failure, the compensating actions are performed in the reverse order of the corresponding forward actions (as in *Sagas* \cite{G-MS87}) while concurrent forward actions are compensated concurrently.

The Animation tool is currently supported in Firefox and Chrome, and is available at:
http://pavlos-engine-v5.herokuapp.com/

It is free to use and provides a first step in developing tool support for service interactions in a Digital Ecosystem that supports true concurrency. The underlying formalism integrates well with our work on SBVR and RESTful web services and we believe this is a valuable step forward in building a native Web environment that truly reflects the spirit of Digital Ecosystems.

VII. References

[1] O.R.P. Bininda-Edmonds et al, "The delayed rise of present-day mammals," *Nature,* vol.446, 2007, 507-512.

[2] H. Boley and E. Chang, "Digital Ecosystems: Principles and Semantics", in *Proceedings of the 2007 Inaugural IEEE Conference on Digital Ecosystems and Technologies*, 2007, 1-6.

[3] J.W. Brown, R.B. Payne and D.P.Mindell, "Nuclear DNA does not reconcile 'rocks' and 'clocks' in Neoaves", *Biol. Lett.*, 2007.

[4] P. Braun, E-Commerce and Small Tourism Firms, In: Marshall S., Taylor, W. and Yu X. (eds), *Encyclopedia of Developing Regional Communities with Information and Communication Technology*, Idea Group, 2006.

[5] W.A. Brock, K. Mäler and C. Perrings, "Resilience and Sustainability: The Economic Analysis of Nonlinear Dynamic Systems", in: L.H.S. Gunderson and C.S. Holling, q.v., 2002, 261-289.

[6] D. Buhalis, "Information Technology for Small and Medium-Sized Tourims Enterprises: Adaptation and Benefits," *Information Technology and Tourism*, **2**, 79-95, 1999.

[7] P. Dini, G. Lombardo, R. Mansell, A. Razavi, S. Moschoyiannis, P. Krause, A. Nicolai and L. Leon, Beyond interoperability to digital ecosystems: regional innovation and socio-economic development led by SMEs, *International Journal of Technological Learning, Innovation and Development,* **1:3**, 410 – 426, 2008.

[8] Eurostat, *Community Survey on ICT Usage and E-commerce in Enterprises*, Brussels, 2007.

[9] European Commission, *The European e-Business Report*, e-Business Watch, Luxembourg, 2007.

[10] G. Evans, J. Bohrer and G. Richards, "Small is Beautiful? ICT and Tourism SMEs – A comparative European Study", *Information Technology and Tourism*, **3**, 139-153, 2001.

[11] L.H.S. Gunderson, C.S. Holling and S.S. Light (eds), *Barriers and Bridges in the Renewal of Ecosystems and Institutions*, Columbia University Press, New York, 1995.

[12] L.H.S. Gunderson and C.S. Holling, *Panarchy: Understanding transformations in human and natural systems*, Island Press, 2002.

[13] C.S. Holling, D.W. Schindler, B.W. Walker and J. Roughgarden, "Biodiversity in the functioning of ecosystems", in C.A. Perrings *et al.*, *Rights to Nature,* 1995, 44-83.

[14] J. Marceau and M. Dodgson, *Systems of Innovation*, Proc. National Innovation Summit, Melbourne, 2000.

[15] R.M. May, *Stability and Complexity in Model Ecosystems,* Princeton University Press, Princeton and Oxford: 2001.

[16] M. Mazzanti and R. Zoboli, "Economic instruments and induced innovation: The European policies on end-of-life vehicles", *Ecological Economics*, vol.58, 2006**,** 318-337.

[17] S. Moschoyiannis, A. Razavi and P. Krause, "Transaction Scripts: making implicit scenarios explicit," in *Proc. ETAPS 2008 – FESCA'08*, ENTCS, Elsevier, 2009. *To appear*

[18] A. Ordanini, "Infomediation and Competitive Advantage in B2B Digital Marketplace", *European Management Journal*, vol.19, 2001, 276-285.

[19] C. Perrings, "Ecological resilience in the sustainability of economic development," *Economie Appliqueé*, vol.48, 1995, 121-142.

[20] A. Razavi, S. Moschoyiannis and P. Krause, "A Coordination Model for Distributed Transactions in Digital Ecosystems," in *IEEE Digital Ecosystems and Technologies* (*IEEE-DEST'07*), 2007.

[21] A. Razavi, S. Moschoyiannis and P. Krause, "A Scale-free Business Network for Digital Ecosystems," in *IEEE Digital Ecosystems and Technologies* (*IEEE-DEST'08*), 2008.

[22] A. Razavi, S. Moschoyiannis and P. Krause, "A Self-Organising Environment for Evolving Business Activities," in *Computing in the Global Information Technology, ICCGI'08.* 2008, 277-283.

[23] M. Scheffer, F. Westley, W.A. Brock and M. Holmgren, "Dynamic Interaction of Societies and Ecosystems," in: L.H.S. Gunderson and C.S. Holling, q.v., 2002, 195-239.

[24] A. G. Tansley, "The use and abuse of vegetational concepts and terms," *Ecology,* vol 16, 1935, 284-307.

[25] B. Walker and D. Salt, *Resilience Thinking*, Island Press, 2006.

[26] World Commission on Environment and Development, *Our Common Future*, Cambridge University Press, UK, 1987.