OPAALS PROJECT

Contract n° IST-034824

WP3: Autopoietic P2P Networks

Del 3.10 – Integrated autopoietic DE architecture

Project funded by the European Community under the "Information Society Technology" Programme

Contract Number: IST-034824

Project Acronym: OPAALS

Deliverable N°: D3.10

Due date: 31/08/2009

Delivery Date: Sept 2009

Short Description:

This document is the last in a series of deliverables (D3.1, D3.2, D3.3, D3.6) that aim to frame the infrastructural specifications in Digital Ecosystems research. As it has been shown (D3.6) in OPAALS the technological and social concerns in providing the necessary digital infrastructure are not treated as distinct but as part of the same continuum. This deliverable sets out the key concepts and characteristics of the current integration framework and clarifies the future work plans. The key aspects of the core DE architecture, in terms of P2P and transaction support, as well as formal analysis and current implementation roadmaps, are described from a computer science viewpoint.

The process of putting the design models and the different point of views are discussed briefly. In this deliverable we show how this process has been set up and demonstrate the consensus reached on key aspects of the core DE architecture.

In order to provide a complete overview, this document contains several chapters (2-5) that contain material that has already presented in earlier deliverables, although subject to a number of refinements and corrections. New material is presented in Chapters 1, 6, 7 and 8.

Author: UniS

Partners contributed: IPTI, TI

Made available to: SUAS, IITK, LSE

VERSIONING		
VERSION	DATE	NAME, ORGANIZATION
0.1	7/2009	**A. RAZAVI (UNIS)**
0.2	8/2009	**A. RAZAVI (UNIS), P. KRAUSE (UNIS)**
1.0	9/2009	**A. RAZAVI (UNIS), P. KRAUSE (UNIS)**

Quality check

Internal Reviewers: Paul Malone (WIT), Ossi Nykänen (TUT)

Dependencies:

Work Packages	WP12: Task 12.8 – OS principles of communication and collaboration, Task 12.9, 12.10 – Business models in DEs

	WP11: Task 11.1 – collaboration and innovation in the Knowledge Economy, Task 11.4 – social innovation networks WP10: Task 10.10 – visualisation of P2P infrastructure WP5: Task 5.6 – adding e-business support in current DBE, Task 5.7 – integration of P2P network services into the DE
Partners	TI, BCU, UniKassel, IPTI, NUIM, UL, IITK, TUT, UNIVDUN
Domains	Computer Science domain: P2P networks, interactions, long-running transactions, distributed systems and networks, redundancy, diversity, consistency, concurrency, design for failure, formal semantics, behaviour patterns, lock mechanisms, formal analysis, distributed identity, distributed trust. Social science domain: participation, power, control, technology infrastructure for competitive advantage, monopoly, lock-in, proprietary software / platforms, knowledge, openness, reciprocity, context-dependent trust, identity. Natural Science domain: simple reference to key analogies with ecosystems in nature and their key features found mostly in studies of biodiversity.
Targets	Computer, social and natural science researchers, SMEs, business analysts. Computer science communities: database, transactions, P2P networking and applications, formal methods. Social science: language, socio-economics, governance, power and control, identity and trust.

TABLE OF CONTENTS

EXECUTIVE SUMMARY

One of the most important priorities for any healthy economy in general and European countries specifically is Small and Medium enterprises and their sustainability and growth:

> "*Micro, small and medium-sized enterprises (SMEs) play a central role in the European economy. They are a major source of entrepreneurial skills, innovation and employment. In the enlarged European Union of 25 countries, some 23 million SMEs provide around 75 million jobs and represent 99% of all enterprises.*" (European Commission, 2005)

The innovations, affiliations, competition and collaborations of these enterprises relies on several factors from a healthy economy and this is one of the reasons, despite their crucial roles in other aspects of society, for support and considerable investment, provided in a large scale;

> "*...support for SMEs is one of the European Commission's priorities for economic growth, job creation and economic and social cohesion.*" (European Commission, 2005)

Maintaining an optimised model for supporting these enterprise relationships needs a compound representation. One of the well-known metaphoric attempts for grasping this complexity is '*business ecosystem*' that describes the business environment as an economic community which "is supported by a foundation of interacting organizations and individuals--the organisms of the business world." (Moore, 1993).

With the introduction of the Internet and increased connectivity, "Digital Ecosystems" (Digital Business Ecosystems) have been introduced as an evolution of this model (Nachira, 2002a). In this context, Business has measured as "*An economic community supported by a foundation of interaction organisation and individuals – the organisms of business world*". (Nachira, Dini, & Nicolai, 2007).

Inspired by the natural Ecosystems, Digital Ecosystems are seen as having four properties: '***Interaction and engagement***', '***Balance***', '***Domain clustered and loosely coupled***' and '***Self-organisation***' (Chang & West, 2006a). As a result, a Digital Ecosystem needs a self-organising digital infrastructure aimed at creating a digital environment for networked organisations that supports a loosely coupled business interaction between them when stability and sustainability of this dynamic environment is maintained. In contrast with conventional models (client-server, Peer-to-Peer or centralised service oriented architecture), a Digital Ecosystem is an open community without any centralised control (Chang & West, 2006a), (Chang, West, & Hadzic, 2006). The loose coupling and local autonomy of participants, with the lack of centralised control, will result in distributed coordination which creates a fully distributed environment (Moschoyiannis & Darking, 2008). By supporting a loosely coupled transaction (interaction) model, fully distributed environment and avoid any centralised control or coordination, the European Commission has considered this conceptual model as the ideal framework for supporting Small and

Medium Enterprises (Nachira, 2002a), (Nachira et al., 2007).

As the distributed system is asynchronous, there is no fixed upper bound on how long it takes for a message to be delivered or how much time elapses between successive steps of a processor (Tanenbaum & Steen, 2008), (Attiya & Welch, 2004), (Kshemkalyani & Singhal, 2008). Therefore, modelling distributed algorithms is supposed to be independent of any particular timing parameter. Consequently for providing a loosely coupled business transaction (interaction) model, which aims to work in a sustainable network of participants, three challenges should be taken into account: '*Asynchrony*', '*Limited local knowledge*' and '*Failures*' (Tanenbaum & Steen, 2008), (Attiya & Welch, 2004).

This report provides a loosely coupled distributed business transaction (interaction) model. The proposed (self-) recovery mechanism for solving failures respects the local autonomy of participants and at the same time optimises the cost of recovery. The abstract agent model for participants facilitates an asynchronous interaction model. Meanwhile the cluster based stability model takes into account the limited local knowledge of each participant; the conceptual repositories use a customised replication mechanism for increasing their local knowledge in order to increase the local stability of environment. The planned virtual super peers offer a dynamic method for maximising the sustainability of the global network without relying on any centralised infrastructure provider or applying any control. One of the important characteristics of the projected model is avoidance of single points of failure, which shows the resistance of the system against crisis or smart hackers' attacks. The solution is fully distributed and avoids any level of centralised coordination or control. In terms of the main requirements, the participants of such a model are considered as Small and Medium Enterprises (SMEs), but we believe that Large Enterprises could also benefit from such a model.

This report is structured as follows. In **Chapter 1**, the history of '*Digital Ecosystems*' from introducing the conventional '*business ecosystem*' towards the necessity of defining a new conceptual framework as Digital Ecosystems, has been reviewed. The characteristics and requirements have been discussed in the last part of the chapter. As providing a distributed transaction model is the main part of requirements for a Digital Ecosystem, **Chapter 2** provides a broad review of existing transaction models and discuss their characteristics in terms of long-running business transactions required for Digital Ecosystems.

In **Chapter 3**, we outline the basic characteristics of Digital Ecosystems and their interaction model in terms of a loosely coupled service-oriented environment and then discuss transaction models that have been developed with conventional service-oriented architectures (web services) in mind. We argue that existing transaction models for web services are tailored to the needs of large-scale enterprises (LEs) and raise barriers to the adoption of such technology by SMEs. More specifically, such models rely on coordination frameworks that are not fully distributed and do not respect the loose-coupling of the underlying services.

The proposed distributed transaction model for Digital Ecosystems is presented in **Chapter 4**. At the heart of this is the coordination model which has been customised for SMEs in a loosely coupled, service oriented environment for Digital Ecosystems and relies on

the designed log mechanism, along with the optimised lock system. The proof of consistency and details of its infrastructure has been discussed at the last part of this chapter. **Chapter 5** provides the details of the recovery mechanism and the reaction of the model in opposition to failures. Meanwhile avoiding failures by using diversity in a Digital Ecosystem and designing a self-recovery mechanism for facing Network failures are the other issues that have been discussed.

In **Chapter 6**, we describe the formal underpinning to analyse the distributed behaviour of our proposed coordination model for the Digital Ecosystem transaction model. Naturally, the next step is to look for network architectures that can support this collaborative software environment of the Digital Ecosystem, where collaboration is conducted based on the transaction model. We are concerned with such aspects in **Chapter 7**, which in a certain important sense pave the way for future extensions for implementation. The projected virtualisation levels, by using VSPs (Virtual Super Peers), provides a dynamic and sustainable model, which avoids any centralised control and tries to adapt itself to the dynamic behaviour of the environment.

Chapter 8 provides a brief review about the implementation support for coordination aspects of the transaction model through the log mechanism that ensures local consistency and the sustainable environment for a Digital Ecosystem. This implementation is ongoing, with the involvement of various partners in the OPAALS project. The implementation has not been finalised at the time of writing. One may anticipate that by applying it in the actual business environment, various customisations may be applied and alter the final implementation. The Conclusion contains some concluding remarks and a brief discussion on future directions.

AIMS & OBJECTIVES OF THE REPORT

The main aim of this report is proposing a loosely coupled solution based on the context of digital ecosystems, and in particular the report concerns with services, transactions, and network support within the digital ecosystem initiative.

Such a solution should provide a distributed coordination model with full consistency and recoverability properties, which is based on loosely-coupled service composition. Meanwhile the solution must support a dynamic topology that continuously adapts to reflect the actual usage of the network in terms of business transactions.

The other important property is the resistance against fragmentation. Furthermore, the proposed solution ought to avoid a central point of command and control and shall not suffer a single point of failure.

SUPPORT PUBLICATIONS

The various parts of this report (and WP3) has been reflected in series of publications, here we mention some of them

BOOK CHAPTER:

- A. Razavi, P. Krause, S. Moschoyiannis, (2009), Digital Ecosystems: challenges and proposed solutions, in the processed to be published in Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies and Applications, IGI Global publication.

JOURNALS:

- A. Razavi, S. Moschoyiannis, P. Krause (2009). An open digital environment to support business ecosystems. Peer-to-Peer Networking and Applications Springer Journal.
- P. Dini, G. Lombardo, R. Mansell, A. Razavi, S. Moschoyiannis, P. Krause, A. Nicolai, L. Len (2008) Beyond interoperability to digital ecosystems: regional innovation and socio-economic development led by SMEs, International Journal of Technological Learning, Innovation and Development 2008 - Vol. 1, No.3 pp. 410 - 426.

PEER-REVIEWED CONFERENCES/WORKSHOPS:

- P. Krause, A. Razavi, A. Marinos, S. Moschoyiannis (2009) Stability and Complexity in Digital Ecosystems, 3rd IEEE International Conference on Digital Ecosystems and Technologies, IEEE Computer Society, Istanbul, Turkey (in press - best paper award).
- A. Razavi, A. Marinos, S. Moschoyiannis and P. Krause (2009) RESTful Transactions supported by the Isolation Theorems, The Ninth International Conference on Web Engineering (ICWE 2009), San Sebastian, Spain (in press).
- Marinos, A. Razavi, S. Moschoyiannis and P. Krause (2009) RETRO: A Consistent and Recoverable RESTful Transaction Model, IEEE 7th International Conference on Web Services (ICWS 2009), Los Angeles, CA, USA (in press).
- A. Razavi, A. Marinos, S. Moschoyiannis and P. Krause (2009), Recovery management in RESTful Interactions, 3rd IEEE International Conference on Digital Ecosystems and Technologies, IEEE Computer Society, Istanbul, Turkey (in press).
- A. Razavi, S. Moschoyiannis, and P. Krause (2008) A Self-Organising Environment for Evolving Business Activities, The First International Workshop on Computational P2P Networks: Theory & Practice (ICWMC 2008 // ICCGI 2008 // ComP2P), 2008, July 27 - August 1, 2008 - Athens, Greece.
- S. Moschoyiannis, A. Razavi, and P. Krause (2008) Transaction Scripts: making implicit scenarios explicit. In Proc. of ETAPS 2008 - Formal Foundations of Emebedded Software and Component-Based Software Architectures (FESCA'08), ENTCS, Elsevier

- A. Razavi, S. Moschoyiannis and P. Krause (2008) A Scale-free Business Network for Digital Ecosystems, In Proc. of IEEE Int'l Conf. on Digital Ecosystems and Technologies (IEEE-DEST 2008), IEEE Computer Society
- S. Moschoyiannis, A. Razavi, Y. Zheng and P. Krause (2008) Long-Running Transactions: semantics, schemas, implementation, In Proc. of IEEE Int'l Conf. on Digital Ecosystems and Technologies (IEEE-DEST 2008), IEEE Computer Society.
- A. Razavi, S. Moschoyiannis and P. Krause (2007) Concurrency Control and Recovery Management for Open e-Business Transactions. In Proc. of Communicating Process Architectures (CPA 2007).
- A. Razavi, S. Moschoyiannis and P. Krause (2007) A Coordination Model for Distributed Transactions in Digital Business Ecosystems. In Proc. of IEEE Int'l Conf. on Digital Ecosystems and Technologies (IEEE-DEST 2007), IEEE Computer Society
- A. Razavi, P. Malone, S. Moschoyiannis, B. Jennings and P. Krause (2007) A Distributed Transaction and Accounting Model for Digital Ecosystem Composed Services. In Proc. of IEEE Int'l Conf. on Digital Ecosystems and Technologies (IEEE-DEST 2007), IEEE Computer Society.

OPAALS WORKSHOPS:

- A. Strømmen-Bakhtiar, A. Razavi, Emerging Problems in the Digital Business Ecosystem. Tampere, Finland 2008.
- A. Razavi, S. Moschoyiannis and P. Krause, The Agent-based Digital Business Ecosystem: towards the Semantic Web, OPAALS Conference, ROME 2007.
- P. Dini, A. Razavi, A. Nicolai, G. Lombardo, S. Moschoyiannis, L. Rivera Leon, P. Krause, Beyond Interoperability to Digital Ecosystems, OPAALS Workshop, ROME 2007.
- S. Moschoyiannis, A. Razavi and P. Krause, Transaction Vectors; A Non-Interleaving Semantics for Long-Running Transactions, OPAALS Conference, ROME 2007.

1. DIGITAL ECOSYSTEM AND CHALLENGES

This chapter includes two sections. In the first section we review the history of the Ecosystem briefly as a metaphor for business and in that respect, the problems have been investigated (this material has been published in (Strømmen-Bakhtiar & Razavi, 2008). In the second section, the modern term of Digital (Business) Ecosystems and its aims have been expressed. At the end of the chapter, we have tried to revisit the requirements of such architecture.

1.1 BUSINESS ECOSYSTEMS & THE COMPLEXITIES OF A DIGITAL WORLD

In 1993 James F. Moore in the article "*Predators and Prey: The New Ecology of Competition*" compared the business environment to an ecological system. He used the metaphor "*business ecosystem*" to describe the business environment as an economic community which "is supported by a foundation of interacting organizations and individuals--the organisms of the business world." (Moore, 1993)

The resemblance between the two systems is of course not complete and there are certain recognisable differences between the two, such as self optimisation (Hannon, 1997), conscious decision making (Lewin, 1999) or the intelligence of actors in the ecosystem (Iansiti & Levien, 2004).

Nevertheless, this comparison of economy to biology has been seen as extremely relevant and useful, not only because this comparison improves our understanding of the roles and interconnectedness of various actors in the business ecosystem, but also because of increasing connectivity and complexity of these systems.

Of course, one can consider an economy to be a national business ecosystem (Rothschild, 1995) composed of many smaller systems, all of which are directly or indirectly interconnected. What many call an industry can now be considered to be either an ecosystem or a part of one. These business ecosystems are populated by (some loosely) interconnected organisms: businesses, consumers, the government and other stakeholders.

As far as the business population of each business ecosystem is concerned, the majority is composed of Small and Medium size businesses (SMEs) along with a few large ones, the so called *Keystones*. Marco Iansiti and Roy Levin compare the role of these keystone companies to those of keystone species in nature (Iansiti & Levien, 2004). They argue that we live in an interconnected world, the landscape of which is made of a network of networks, with keystones at the hubs and niche players surrounding the hubs.

Until recently the majority of attention of both academia and governments has been focused on these keystones, believing that what is good for the keystones is good for the business ecosystem and hence the nation. However, over time, it has become clear that,

depending on government regulations and policies, these keystones can play either a positive or a negative role in the business ecosystem.

1.1.1 KEYSTONE: THE PROBLEMS OF DEFINITION AND BEHAVIOUR

In nature a '*Keystone*' species is defined as "*a species whose effect is large and disproportionately large relative to its abundance*" (Power et al., 1996). It is also argued that Keystones help in determining or regulating the number and type of other species in their communities. It is generally believed that keystones play a positive and a necessary function in the ecosystem. This view is also shared by the majority of those who have rushed to adopt the concept for the study of business ecosystems.

But the keystone concept has been adopted in both computer and business literature without actually questioning the correctness of the analogy. Even if, for the sake of the argument, we accept the analogy, we cannot ignore the arguments of the critics of the concept in its original context, namely the definition and role of keystones in ecology. For instance Payton et al. (Payton, Fenner, & Lee, 2002) argue that:

> *The idea that some species may function as keystones has not been without its critics (Mills et al. 1993; Hurlbert 1997), who argue that while the concept has 'developed tremendous currency and fashionability' it has also become increasingly ill-defined, and now means little more than 'important for something' (Hurlbert 1997). In defence of this position, opponents cite the vagueness of keystone definitions and their inconsistent usage in the literature, and an implied dichotomy between keystone and non-keystone species that has not been demonstrated in nature. Hurlbert concedes that the notion was 'appealing and harmless', but as a well-defined concept or phenomenon he concludes it 'has had a stultifying effect on ecological thought and argument'*.

It is important to note that in nature a keystone species does not have the ability to freely relocate to another ecosystem, nor does it have the option of outsourcing parts of its activities. And most importantly, in nature a keystone species lacks the intelligence level of its namesake in business ecosystems.

1.1.2 KEYSTONES: FROM INDUSTRIAL AGE TO INFORMATION AGE

Technological innovations have always led to the creation of new companies by entrepreneurs who have tried to take advantage of those innovations to create a competitive advantage for themselves in the marketplace; which in turn necessitated the adaptation of those innovations by the older established companies.

Until relatively recently, the rate of diffusion and adaption of new technologies was rather slow. Lack of speedy communication was one of the reasons behind this glacial diffusion rate. For example, new machines were invented and used in one part of the world without it being introduced and used in other parts. A good example of this is the printing press, which was invented in China a few hundred years prior to its "re-invention" in Germany in 1439.

However, the Industrial revolution (1760-1840) (Toynbee & Jowett, 1887) reduced the distance between the continents. The improvements in steam engines (steam locomotive 1804), invention of the telegraph in 1830s and later the telephone in the 1860s effectively reduced distances between towns, countries and continents. This reduction in distance also opened new markets, allowing manufacturers and producers to increase production capacity, which in turn led to increasing size of these companies.

Indeed the origins of the modern diversified corporations can be traced back to the creation of large corporations at the beginning of the last century, when mass-production (brought about by innovations in work methods and mechanical automation) allowed many companies to grow rapidly and prosper at a rate that had never been seen before in history. Companies such as Ford, General Motors, Standard Oil Company and others grew from small businesses to large corporations (keystones), whose turnover matched the GNP of many small nations.

Until the early 1920s, these corporations were primarily single business units. Ford manufactured cars, while Standard Oil was concerned with oil exploration, extraction and refining. The driving force behind these corporations came from their owners who knew their businesses well and exercised total control. However changes in size and organisational forms required new strategies.

In this era competition, by and large, was seen as "dominate or absorb" with one exception: the creation of cartels. "Price competition among large-scale rivals proved mutually destructive to profits and after a brief period of cut-throat competition, business enterprise turned to cartels, trusts and other monopolistic forms of organisation designed to eliminate price competition" (Dillard, 1967). Here the theories of Cournot (Cournot, 1960) (monopoly, duopoly, and oligopoly), first published in 1838, were put to work. This allowed the owners, the so-called "robber barons"(Josephson, 1962), to concentrate on increasing internal efficiency of their organisations.

Part of this internal efficiency was achieved by focusing on economies of scale; i.e. to produce a product faster, better and cheaper than competitors, using mass-production (changes in the supply side). As the competition intensified and marketing and customer demand became more important, these companies began to change their focus to economies of scope (Chandler, 1990); that is, to producing "different products" faster, better and cheaper.

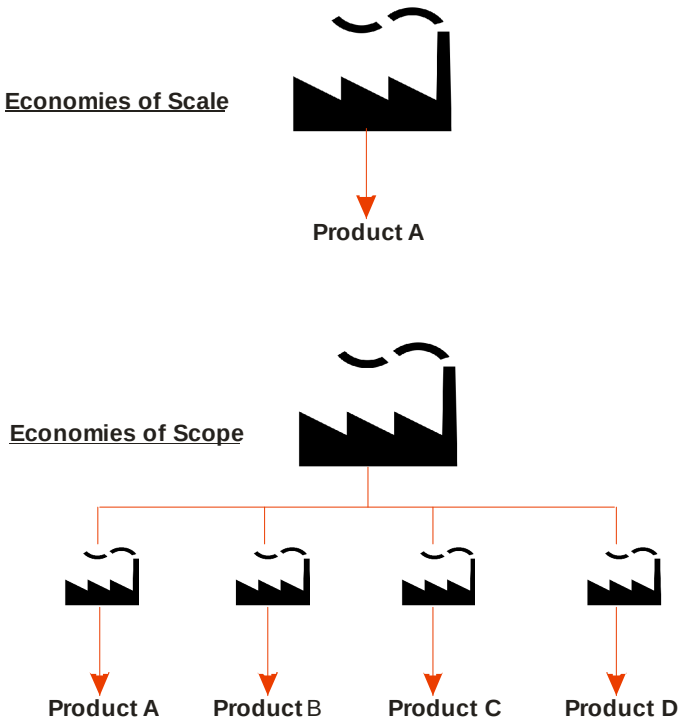

FIG. 1-1 ECONOMIES OF SCALE AND ECONOMIES OF SCOPE

As can be seen, there is a marked difference between the economies of scope and economies of scale. Economies of scale spread fixed costs over a large number of units of production of the same product or enterprise, while economies of scope involve spreading the cost of a set of resources or skills over two or more products or enterprises. One should also note that economies of size and scope are not mutually exclusive. While economies of scope allow costs to be spread over several enterprises, the size of each enterprise can of course also be increased to achieve economies of scale.

The improvement in internal efficiency was not limited to production technologies. The management practices, routines and business processes were also examined and improved. As businesses, especially manufacturers, sought ways to reduce their costs and improve their responsiveness, they adopted new concepts such as Just-In-Time (JIT) systems, which demanded a closer cooperation (i.e., timely access to information and products) between producer, suppliers and customers.

Intranet and extranets were the answer to many of these problems. Intranets allowed big firms to create their own private internet, sharing the organisation's information or operations with their employees. Extranets (which can also be an extension or part of the Intranet) in turn allowed these firms to connect to suppliers, vendors, partners, and customers.

Intranets and extranet were the result of the general work done in the late 60s and early 70s by the Advanced Research Projects Agency (ARPA) on the creation of the first internet connection (between UCLA and SRI International in Menlo Park, California); the start of the so called ARPANET. By mid 1980s the internet had changed from NetWare Core Protocol

(NCP) protocols to Transmission Control Protocol/ Internet Protocol (TCP/IP). In 1991, the European Organization for Nuclear Research (CERN) publicised its World Wide Web project, which laid the foundation for the following explosive growth of the internet.

The creation of the Internet along with the rise of the mini- and micro-computers in the late 1970s and Personal Computers (PCs) in the 80s facilitated the transition from the industrial era to the information era and the network economy.

1.1.3 CHANGING THE FACE OF COMPETITION

Intranets and extranets allowed companies to connect their offices, plants, suppliers and customers into closed networks. This allowed them a better overview of their operations while strengthening the coupling between themselves and their customers and suppliers. Indeed implementing JIT (Just-In-Time) without such networks would have been impossible. Similarly, outsourcing of many goods and services is extremely difficult without the existence of such networks.

Until a few decades ago, creating such networks required large amounts of capital, specialised software, hardware and expertise. This made it a perfect tool for erecting entry barriers around industries. In addition these new technologies gave large corporations geographical independence. It allowed these corporations to relocate parts of their operations to low-cost countries, effectively reducing their operations' costs. It started with manufacturers and was followed by service companies. Some may see the advent of these ICT technologies as one of the main facilitators of globalisation. Today a large company can, for example, have its administration in London, its production facilities in Shanghai, and its accounts and billings in New Delhi.

There are many reasons given for this relocation, chief among them being the pressures from global competition. This can partly be accepted for standardised low-value goods in the manufacturing industry. However it is rather difficult to accept this reason for service industries. For example the competition in utility such as gas or water is national or in some cases even regional. Moving British Gas customer services such as billings or customer enquiries to New Delhi does not look like the result of international competition in this sector in the UK. More plausibly, this is due to the result of increasing pressure from shareholders on the company's executives for higher returns.

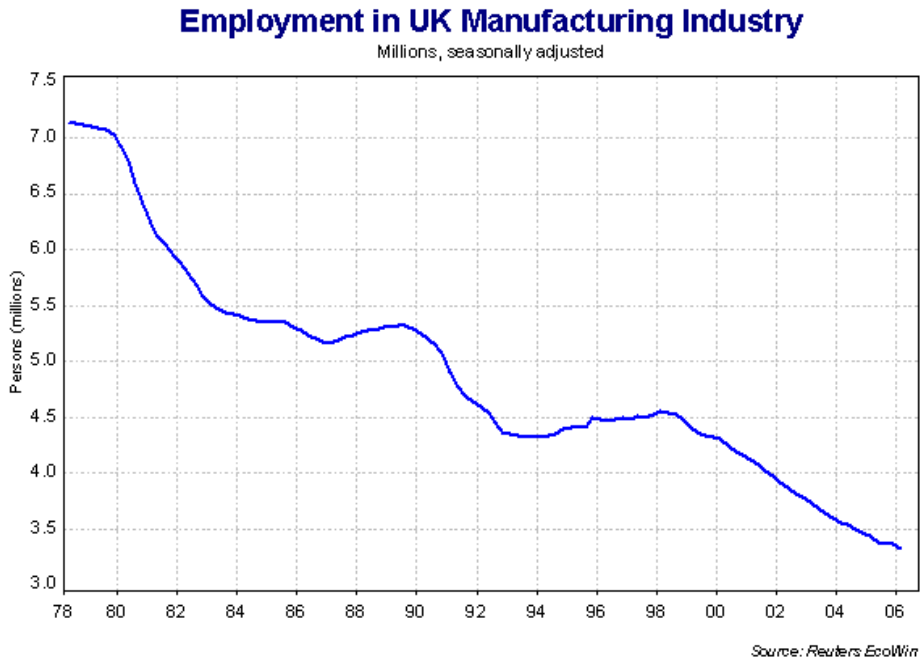

FIG. 1-2 EMPLOYMENT IN UK MANUFACTURING (APPLEGATE, AUSTIN, & MCFARLAN, 2002)

Whatever the case may be, the relocation or outsourcing of many services and manufacturing activities to low-cost countries has put a tremendous pressure on local SMEs, which traditionally have been the providers of many goods and services to the larger corporations.

In addition, those SMEs that were directly competing with the large corporations also felt the pressure of these relocations and outsourcings. Not only could they not compete with the low-cost countries, where past of services or products were being produced, but they also lacked the distribution channels of the large companies.

> "*Lacking the ability to share information with others, small suppliers, distributors, and consumers were disconnected, doing business on the margins in industries dominated by large players. Economists called these unconnected spaces in an organisation, market, or industry network holes. Those 'holes' became a prime target for Internet entrepreneurs in the late 1990s. While the key technological innovations of the Industrial Economy enabled economies of scale and scope in production, the technological innovations of the Network Economy are improving economies of distribution, especially in relation to coordination, communication, and information sharing.*" (Applegate et al., 2002)

The Internet was seen as a solution to many of these problems. First, the Internet itself was and is a dynamic and exciting new area (it was and is considered) where large companies do not enjoy their traditional advantage of size. Second, the Internet itself was

and is seen as a brand new market ripe for new products (software and hardware). And finally SMEs see in the Internet the perfect opportunity to find and establish new connections to other businesses and customers. But as we shall see, the Internet has its own problems.

1.1.4 First Mover Advantage and Scale-free Networks

After each technological innovation, a group of start-ups and SMEs move in to take advantage of the new opportunities. Technological evolutions usually give birth to new industries where start-ups and existing SMEs, because of their size (agility) enter first, becoming the first movers. First movers generally have the advantage of registering patents, establishing brand names, changing the economics of the market making it difficult for others to enter and compete and so on. In young industries, first movers have the ability to, in a very short time, become industry leaders or keystones. We have seen this with Microsoft and personal computer operating systems. Microsoft was one the few first movers in this segment. Microsoft managed very quickly to establish itself as the leading PC operating system provider and thereby create a quasi standard for PC operating systems, which was and is highly lucrative. To dominate and discourage others from entering the market, it bundled its operating system with the PCs, making it extremely difficult for others to compete. Even almost free operating systems such as Linux with compatible quality, and the same or better functionality, have had a hard time competing with the MS operating system. Using its operating system as the main platform (and a cash cow), Microsoft has tried to limit competition in other segments of the PC industry (e.g., office, internet browser, multimedia player etc.).

We have seen other first movers such as Yahoo and later Google (search engines), Adobe (readers) following the same strategy in other segments, trying hard to set standards or changing proposed standards to their advantage, although with less success than Microsoft. Nevertheless, the first mover advantage has been very rewarding for these companies, especially in the Internet. First movers have managed to become strong hubs with many links, becoming almost impervious to competition from smaller actors.

The Internet is a network of computers. This network has no determined structure and expands in a random fashion. New nodes (computers) constantly connect and disconnect themselves to the network via links (vertices) to other computers. Studies (Barabási & Albert, 1999), (Barabási, Albert, & Jeong, 2000) have shown that the topology of this network is governed by a power-law distribution. This means that often a few nodes evolve in such a way as to attract a large number of links while many nodes continue to exist with only a few links. This is especially true for World Wide Web (WWW), where page links act as links to other pages and hence internet sites. This gives those nodes (with large number of links) and companies that own them a disproportionate power in the network. First movers have managed early on to become large hubs. SMEs in this sector have very little chance of becoming hubs. The entry barriers in this sector are getting higher and higher. If by chance or design an SME manages to acquire a number of links (e.g., Alibaba.com in China), it is

bought (Alibaba was acquired by Yahoo) and integrated into the existing hub.

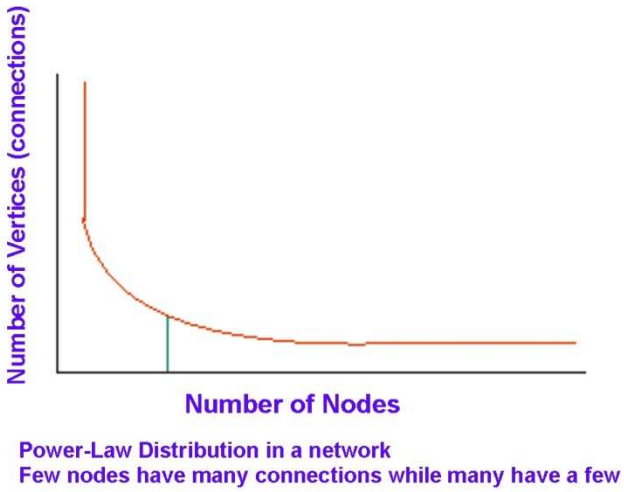

FIG. 1-3 DISTRIBUTION DEGREE IN SCALE-FREE NETWORK

The power of these hubs is so strong that even large companies with sufficient financial resources and know-how such as Microsoft have trouble competing effectively in these segments. Similarly, as Microsoft used its size and financial resources (derived from its operating system) to spread its wings and expand into different segments, these hubs are also trying to establish themselves as Goliaths in other areas.

As was mentioned earlier, first movers take advantage of their position and try to establish themselves as the industry leaders. As such, they constantly try to either set the industry standard or change the proposed standards to their own advantage. So here we see two distinct problems: the first being the structure of the networks where hubs try to dominate and the second being the question of standards being set for the creation of applications that will be running on those networks.

1.1.5 SCALE-FREE NETWORKS; A SOLUTION OR MAJOR PART OF PROBLEM

We have already mentioned that in scale–free networks, a few nodes become hubs, dominating the network. There are some arguments for the importance and thereby existence of hubs within a network, chief among them being that these hubs provide stable connections that are online all of the time.

> "*Hubs, the highly connected nodes at the tail of the power law degree distribution, are known to play a key role in keeping complex networks together, playing a crucial role from the robustness of the network to the spread of viruses in scale-free networks. Our measurements indicate that the clustering coefficient*

> *characterizing the hubs decreases linearly with the degree. This implies that while the small nodes are part of highly cohesive, densely interlinked clusters, the hubs are not, as their neighbours have a small chance of linking to each other. Therefore, the hubs play the important role of bridging the many small communities of clusters into a single, integrated network.*" (Ravasz & Barabási, 2003)

There are several problems with having these hubs. The first is that they are costly to maintain. The second is that of increasing traffic and traffic congestion during peak time. To answer the increasing traffic, one has to keep increasing the capacity of the hub, leaving substantial free capacity in the off-peak times, which results in an increase in the overhead costs. In addition these hubs present major targets for intentional or unintentional attacks. Any disruption at these hubs will result in major fragmentation of the network. Meanwhile there is another relevant discussion about the digital age and its interaction model, which comes back to coordination. These two cause an unbalanced environment which despite discussions about fairness, will not provide the necessary sustainability in its environment and reliability for its interaction model. In the next section, we review the role of coordination in an interaction model and the state of the art in that respect.

1.1.6 Problem of coordination and Service-Oriented Architecture (SOA)

Having addressed the problem of scale-free networks and the emergence and influence of major players, we shall turn our attention now to the problem of coordination of information systems of different enterprises over the net. The ideal solution, as has been proposed and pursued, is to establish mechanisms that allow different businesses with different practices, equipment and technologies to seamlessly and effectively communicate with each other. One of the solutions that has received strong support from major players and other organisations has been the Service Oriented Architecture (SOA).

We must remember that architecture is more of a mind-set or a development philosophy than anything else. As such, the concept has been around for a number of years and did not become interesting for major software developers and vendors until the emergence of Web Services (WS). A WS is defined as "a software system designed to support interoperable machine-to-machine interaction over a network". (Booth et al., 2004)

The whole concept of SOA revolves around services and loose coupling. Services can be described as special functions that are accessible over standard Internet protocols and are independent from platforms and programming languages. Here the focus is on functionality, and not on the programming languages or methods used. We can for example call the viewing of an online bank statement, a service. Services unlike the old procedures or functions have no calls to each other embedded in them. Instead protocols are defined which describe how one or more services can talk to each other. The applications are created by a business process expert or software engineer who links and sequences

services, in a process known as orchestration.

> "*The goal of SOA is to allow fairly large chunks of functionality to be strung together to form ad hoc applications which are built almost entirely from existing software services. The larger the chunks, the fewer the interface points required to implement any given set of functionality; however, very large chunks of functionality may not be granular enough to be easily reused. Each interface brings with it some amount of processing overhead, so there is a performance consideration in choosing the granularity of services. The great promise of SOA is that the marginal cost of creating the n-th application is zero, as all of the software required already exists to satisfy the requirements of other applications. Only orchestration is required to produce a new application.*" ("Service-oriented architecture - Wikipedia, the free encyclopedia," n.d.)

An application is the orchestration of many services that have different dependencies on each other. This is often referred to as a work flow or business process, which usually means a collection of transactions. Transactions require data consistency and recoverability. An example of data consistency would be that regardless of a transaction deployment with any rate of concurrency, the results should be consistent. By recoverability, we mean that regardless of any failure, one can roll-back the transaction to its initial state.(Singh & Huhns, 2005)

These two are the main responsibilities of the coordinator (Cabrera, Copeland, Feingold, R. Freund, T. Freund, Johnson, Joyce, Kaler, Klein, Langworthy, Little, et al., 2005). In theory, either party to the transaction can assume the role of coordinator. However the current systems place a heavy demand on computational complexity (Razavi et al., 2006), making it very difficult for the majority of SMEs to assume this role. Therefore, the current system requires that a third party with sufficient resources assumes this responsibly. Here we have to mention that the coordinator enjoys many privileges (Razavi et al., 2006) without the associated responsibilities (at least, the full responsibility of recovering a failure without involving participants).

The role and function of the coordinator is determined by the WS-coordination protocol as defined in Windows Communication Foundation (WCF) (Cabrera, Copeland, Johnson, & Langworthy, 2004). This protocol is used by the two existing and competing industry frameworks: Business Transaction Protocol (BTP : supported by Oracle, Sun Microsystems, Choreology Ltd, Hewlett-Packard Co., IPNet, SeeBeyond Inc. Sybase, Interwoven Inc., Systinet and BEA System in term of OASIS Business Transaction Protocol (Furnis et al., 2004)) and Web Services transaction (WS-AtomicTransactions (Cabrera, Copeland, Feingold, Freund, Freund, Johnson, Joyce, Kaler, Klein, & Langworthy, 2005) and WS-BusinessActivities (Cabrera, Copeland, Feingold, Freund, Freund, Joyce, et al., 2005): supported by Microsoft, Hitachi, IBM, IONA, Arjuna Technologies and BEA Systems).

Despite all claims and advertisements, both coordination protocols for business

transactions violate loose coupling on one side and offer just one pattern of behaviour (clarifying the completion protocol in a transaction and determining the recovery method in respect to that protocol (Razavi, Moschoyiannis, & Krause, 2007c)) for participants of transactions on the other (Razavi, Moschoyiannis, & Krause, 2007d).

Violating loose coupling, is not only contrary to SOA, but also gives the coordinator the opportunity to estimate the local state of SMEs (in realistic terms, this tight coupling between coordinator and participants means the coordinator is aware of the local state of participants at any given time during or after the transaction). For instance in a business transaction between two businesses, where PayPal is the coordinator, PayPal can in-effect estimate during the transaction the income of both parties in the transaction.

This tight coupling results in the participants (e.g. SMEs) losing their local autonomy (their local states of businesses will be visible to the coordinator). At the same time the pattern of behaviour supported by the coordinator framework (do-compensate (Furnis & Green, 2005)), forces participants to apply specific methods of fault recovery during a transaction failure (Furnis & Green, 2005), (Vogt, Zambrovski, Gruschko, Furniss, & Green, 2005), (Razavi et al., 2007a). This not only enforces a specific problem solving method in event of a failure (which is known by the companies supporting the protocols) but also imposes the responsibility of sorting the failure out to participants.

By having an overview of the participants' recovery pattern behaviour and their local state, the coordinator can construct or simulate the participants' business models (Vogt et al., 2005), (Razavi et al., 2007a).

We have seen that whenever there is an innovation there are a few players that take advantage of that innovation and establish themselves as the main actors within that industry. This happened in operating systems, office software, database and other sectors of the IT industry. Now a new opportunity is presenting itself: the coordination role.

Many large companies are already jockeying for position to take advantage of this opportunity to establish themselves as the leading companies that provide coordination services. Considering the importance of coordination and the power that a major coordinator can gather onto itself, it would be wise to revisit and re-examine the coordination process. We can ill-afford the emergence of large monopolies or oligopolies here.

1.2 Paradigm shift and the Digital Business Ecosystems concept

Because of the necessity for providing a new model, concept and definition, the research community, different organisations and governments have started to propose a new conceptual framework for digital ecosystems. In 2002, the first intuition has been proposed by Nachira in the European Commission (Nachira, 2002b). This important discussion paper has been finalised in a new conceptual framework (Nachira, 2002a), called Digital Business Ecosystems:

> "*The synthesis of the concept of Digital Business Ecosystems emerged in 2002 by adding 'digital' in front of Moore's (1996) "business ecosystem" in the Unit ICT for business of Directorate General Information Society of the European Commission (Nachira 2002a).*" (Nachira, Nicolai, Dini, Le Louarn, & Leon, 2007)

In this conceptual framework, Business is considered as "*An economic community supported by a foundation of interaction organisation and individuals – the organisms of business world*". (Nachira et al., 2007). This economic community produces goods and services of value to customers, who themselves are members of the ecosystem". (Moore, 1993) A wealthy ecosystem sees a balance between cooperation and competition in a dynamic free market. (Nachira et al., 2007). The European Commission has specified the explicit aims of such a Digital Business Ecosystem:

> "*A Digital Business Ecosystem results from the structurally coupled and co-evolving digital ecosystem and business ecosystem. A network of digital ecosystems, will offer opportunities of participation in the global economy to SMEs and to less developed or remote areas. These new forms of dynamic business interactions and global co-operation among organisations and business communities, enabled by digital ecosystem technologies, are deemed to foster local economic growth. This will preserve local knowledge, culture and identity and contribute to overcome the digital divide.*" (European Commission, 2008)

This conceptual framework tries to solve the current challenges of businesses (recall 1.1.4 and 1.1.5). That is why the projects funded by the EC have focused to provide a modern dynamic network to support business interactions of small and medium enterprises without relying on a large organisation of any sort:

> "*The model of business ecosystem developed in Europe, [in contrast with Moore's conventional definition], is less structured and more dynamic; it is composed of mainly small and medium firms but can accommodate also large firms.... This model is particularly well-adapted for the service and the knowledge industries, where it is easier for small firms to reinvent themselves than, For instance, in the automotive industry.*" (Nachira et al., 2007)

Developing this conceptual framework has not been limited to Europe. The Digital Ecosystems and Business Intelligence Institute in Australia (DEBII, 2009) as a leading research institute on Digital Ecosystems studies, not only has developed and constituted Digital Ecosystems. One of the brightest descriptions of ecosystem in this term can be found

in Chang and West approach:

> "*There are four essences of ecosystems: (1) Interaction and engagement (2) Balance (3) Domain clustered and loosely coupled (4) Self-organisation*" (Chang & West, 2006a).

They describe these attributes as;

'*Interaction and engagement*' has been described as inter-disciplinary interaction, when species need to interact between each other for social well-being and to engage with each other to find interesting things, to share their resources or defend themselves.

"*Balance*" is assigned to the harmony, stability and sustainability within an ecosystem. If some species or parts of an ecosystem are getting disproportionally tensioned, dried, overheated, divided, the whole ecosystem may collapse.

"*Domain clustered and loosely coupled*" clarifies the freedom of species to join to an ecosystem and mentions they are loosely coupled, taxonomic groups of which the members have similar culture, social habit, interests and objectives. Each species preserves the environment and is proactive and responsive for its own benefit. They are able to live together and support each other for sustainability.

The "*Self-organisation*" points that each species is independent, self-empowered, self-prepared, undertakes self-defence, is self-surviving and undertakes self co-ordination through swarm intelligence. In case of natural disaster one cannot ask 'where is the president', 'what logistics systems are provided' and so forth.

1.2.1 Small and Medium Enterprises, local autonomy and loose coupling

As with the DEBII (Digital Ecosystems and Business Intelligence Institute) vision (Chang, Quaddus, & Ramaseshan, 2006), the European Commission's projects have focused on loose coupling between participants of a Digital Ecosystem (Dini et al., 2008). Furthermore, the local autonomy of small and medium enterprises for deploying transactions has been persisted. In this, any collaboration (in terms of business transactions) will be based on demand of participants rather than on any external decision. In addition, the preservation of local autonomy and avoidance of any external control or pressure has been considered within the definition of requirements for any business transaction (Moschoyiannis & Darking, 2008), (Dini et al., 2008). We can see the similarity of these important requirements in DEBII's definition, when it focuses on a loosely coupled, demand driven environment when each participant (here digital species) has a specific objective (benefit or profit):

> "*We define a Digital Ecosystem as a loosely coupled, demand driven collaborative environment where each digital species is proactive and responsive for its own benefit or profit*" (Chang et al., 2006)

Boley and Chang (Boley & Chang, 2007) has described loose coupling as a freely bound open relationship between participants, when the term is opposite to a tightly coupled relationship (where each party is heavily dependent on one another and the roles are predefined). In terms of 'Demand Driven', their definition illustrates the deference between the driving force coming from outside 'push-in' rather than 'pull-in' (Chang et al., 2006). And by example, they have explained the importance of the real motivation from participants rather than a force from outside. As with their definitions, local autonomy in transactions focuses on a similar concept and tries to avoid interferer control or force from outside in any transaction (from its initiation to the actual execution).

1.2.2 Digital Ecosystems and comparison with similar concepts

Conceptually the participants' interactions, their loosely coupled connection and local autonomy, sustainability of their environment and ability for self-organising are characteristics of a Digital Ecosystem (Moschoyiannis & Darking, 2008). This metaphoric similarity has been discussed in (Chang & West, 2006a) with "*Interaction and engagement*", "*Balance*", "*Domain clustered and loosely coupled*" and "*Self-organisation*". We have discussed this earlier in this section. In addition, one of the important clarifications of the Chang and West approach is when they determine the role of Agents (participants) and their digital environment (network) for presenting these characteristics;

> "*Digital ecosystem(DES)= Eco-environment + Eco-agents*" (Chang et al., 2006)

The loose coupling and local autonomy of participants, in the lack of centralised control, will result in distributed coordination, which creates a fully distributed environment (Moschoyiannis & Darking, 2008). As coordination of the interaction has been performed and controlled by each participant independently, the business transaction can be triggered, executed and committed by any participant/agent (Krause et al., 2008). These characteristics make Digital Ecosystems different to conventional architectures:

> "*Unlike the client-server architecture, where the communication is centralised and it is a command and control environment; Unlike the Peer-to-Peer architecture, where each agent has well defined roles, they can only be client or server, but not both; Unlike the Grid architecture, where it stitches partners together on resource sharing but cannot avoid counter-free riding; Unlike the Web*

> *service network, where brokers are centralised, service requesters and providers are distributed, and this hybrid architecture does not guarantee trust and QoS - the Digital Ecosystem is an open community, and there is no centralised control or fix roles.*" (Chang & West, 2006a)

1.2.3 DIGITAL ECOSYSTEMS AND TRANSACTIONS

As we have seen, the distributed interaction model without any external control is one of the important characteristics of a Digital Ecosystem. Whilst SOA was introduced with promises of loose coupling, the actual interaction model of business transactions has imposed tight coupling between participants and coordinator. In addition, by involving the local states of participants the recovery model for these transactions not only violates the local autonomy of participants but also enforces a particular standard in the local design of participants (details can be found in chapter 3). Digital Ecosystems try to solve these contradictions in the transactional level when the environment is fully distributed and does not suffer a centralised control (Moschoyiannis & Darking, 2008). We have conceptualised this interaction model by a virtual set of participants (agents) in each transaction (which can be found in chapter 4). Boley and Chang have used the metaphoric concept of a '*swarm*' for formalising the Virtual Private Transaction Network;

> "*A swarm is a set of agents which have common characteristics and are able to interact and engage directly or indirectly with each other.*" (Boley & Chang, 2007)

In (Chang & West, 2006a) the '*Swarm Intelligence*' concept offers a clearer specification for such agents and their interaction model; they identify two elements in Swarm Intelligent: '*Species or Agents*' and '*Leading Species or Agent*'.

'*Species or Agents*' are described as an individual or an organisation, which is equivalent to participants in (Krause et al., 2008). Each of them has (in a business world according to their business model) its own niche or role to play. Furthermore each one has dual functions or roles; they can be client, and they can also be a server when they can just use the result of a service deployment or deploy their own services. They can carry out bi-directional communication, not just one way.

A '*Leading Species or Agent*' is equivalent to the initiator of a transaction in (Razavi et al., 2006). It has the same features and functionality as any other agent, but simultaneously facilitates, leads and directs the collaborative swarms (as with the interaction model in a business environment relying on an initiator, an actual transaction is initiated by an initiator).

As there are many types of Interaction styles in Digital Ecosystems such as transaction based, social network based, state based, protocol based, computation/calculation based, swarm based etc., although the report is to address Transaction Based Interaction in Digital

Business Ecosystems. The main reason for this is the focus on businesses. As the business activities have been modelled on long-running business transactions (recall 3.4), proposing a sustainable transaction based model for digital ecosystem is one of the main objectives of business community (6th objective of OPAALS project[1]). In the business world, any participant can initiate a transaction (being a Leading Agent), but loose coupling, local autonomy and at the same time, the sustainability of the network (environment) should be maintained. Avoidance of centralised control, resistance against failures, provision of a self-organised model which can recover itself in case of failure, and provision of an environment which has resistance against single points of failure are the main characteristics which we would like to reach in a digital ecosystem. All of these requirements are founded in the primary needs of any business, which is '*Business Transactions*'. The main dispute in term of business activities is whether it is possible to enable '*Long-running Transactions*' answering to this challenge without violating the main principles of the Digital Ecosystems' concept. In order to address this issue, we first (in the next chapter) present a brief review of established transactional responses to business activities (including Long-running transaction) problems. Then we analyse the main requirements of such a model based on Digital Ecosystem characteristics (chapter 3) and will discuss the main proposed transaction models for service oriented environment. The rest of this document will focus on proposing a model for a Digital Business Ecosystem.

[1] http://www.opaals.org/opaals.php

2 INTRODUCTION TO TRANSACTION MODELS

This chapter demonstrates an introduction for transaction; these transaction models generally are optimised for different types of database applications. However, the fundamental difference between the traditional representation of Distributed Transactions for Databases and the representation of Distributed Transactions for Digital Ecosystems is the service-oriented aspect of Digital Ecosystems, specifically the loose coupling and local autonomy characteristics of it. We leave investigating these specifications and potential solutions for them to chapter 3, and will focus here on general transaction models.

As we have seen in the previous chapter, businesses in digital ecosystems need to have an interaction model for their business activities. These business transactions should follow the digital ecosystem principles (such as loose coupling, distributed coordination, etc see chapter 1 for more detail). The conventional definition of a transaction (Date, 2003), (Gray & Reuter, 1993)is based on ACID properties (Atomicity, Consistency, Isolation and Durability, refer to 2.1). While ACID properties provide a good foundation for conventional database applications, they can present unacceptable limitations and reduce performance in advanced applications.

These limitations have been categorised to three specific problems with conventional transaction models (Moss, 1985), (Kakeshita & Xu, 1992), (Haghjoo, Papazoglou, & Schmidt, 1993), (Elmagarmid, 1992): *Long-lived transactions*; *Lack of partial results*; and *Omitted results*.

The necessity of change is derived from the nature of modern applications (such as business applications, CAD projects, etc) (Elmagarmid, 1992). For example, the specification of a transaction may allow it to be completed over a period of hours or even days (a “long-lived transaction”). In addition, the obligation for cooperation between transactions can be specified in a business process rule (a requirement for availability of “partial results”). Finally, the instability of the Internet environment can define a new requirement for keeping important results even when the connection between two platforms is lost (“omitted results”). In this chapter, the basic problems and general transaction models have been introduced, but a more detailed discussion which is focused on requirements for digital ecosystems and current practice in transactional models will be continued in chapter 3.

2.1 TRANSACTION; DEFINITION AND CONSTRAINTS

The general term ‘Transaction’ has been introduced by Gray (Gray, 1992) and is defined by the four properties contained in the ACID acronym. A transaction that is started when a system is in a consistent state may make the state temporarily inconsistent, but it must terminate by producing a new consistent state - *Consistency* is the C in ACID. This temporary inconsistency may not affect other concurrent transactions. This maintains the illusion that

each transaction runs in *Isolation* - the I in ACID. This means that the inputs and consequent behaviour of some parts of the transaction processing system may be inconsistent, even though each transaction executed in isolation would be correct. It follows that concurrent execution should not cause application programs to malfunction, which is the first law of concurrency control. Equally, if some operation within a transaction should fail, it should automatically undo all previous actions and return to the original consistent state. This property is *Atomicity*, the A in ACID. Also, none of the updates or messages of committed transactions should be lost - *Durability* is the D in ACID.

Shared objects between concurrent transactions are the main challenge for a transactional environment. On one hand the concurrency control should support the concurrent activities. However, update of an object which it is accessed by another transaction can violate the consistency of the environment. Therefore concurrency control should be able to provide consistency and an acceptable level of isolation. Furthermore, recoverability of a transaction relies on these limitations too. When transactions are prevented from accessing inconsistent objects of other transactions, each transaction will be recoverable. Through many different approaches to supporting concurrent execution of transactions, a consensus on the use of locks as a preferred solution has emerged (Gray & Reuter, 1993), (Bernstein, Hadzilacos, & Goodman, 1987), (Ramakrishnan & Gehrke, 2003), (Gray, 1992). When implementing locking, we should take into account that concurrent execution should not have lower throughput or much higher response times than serial execution. The second law of concurrency control (Gray & Reuter, 1993) is to avoid high computational overheads. Use of the isolation theorem is the conventional way to apply the locking mechanism with acceptable performance. We will discuss this in the next section.

2.1.1 ISOLATION THEOREMS

A number of Isolation theorems are used to show the correctness of a transactional system by applying certain constraints (Gray & Reuter, 1993). The constraints will be explained in this section and their primary presentation will be shown; we will reference the proof of correctness of constraints, in the classical isolation theorems. Similar to most theorems in computer science, the formal presentation of isolation theorems have relied on sequences as their primary building block where $S \parallel S'$ indicates concatenation of Sequences S and S', the *i*th element of a sequence is represented by S[i] and S' as a subsequence of sequence S is symbolized in a manner similar to set notation with $S' = \langle S[i] | predicate(S[i]) \rangle$ (shows *S'* is subsequence of S).

In order to present a theoretical aspect of the model, we show the formal vocabulary that is larger than the standard read and write operations. A similar approach will be used in terms of long-running transactions in chapter 4 (by using the sub-transaction notation).

The important point in conventional isolation theorems is applying suitable locks before accessing the objects (all possible resources); for avoiding violating consistent access, in terms of READ and WRITE objects, the SLOCK and XLOCK should be applied on the objects and these locks should be released when the dependency on the objects expires (SLOCK or

shared lock will be used in term of reading objects and XLOCK or exclusive lock is used for writing on objects). Therefore, the model supports the major *actions* of READ, WRITE, SLOCK, XLOCK, UNLOCK on the objects, as well as generic actions BEGIN, COMMIT, ROLLBACK. READ and WRITE have the usual meaning: READ returns the named object's value to the user (program), while WRITE alters the named object's state. A *transaction* is any sequence of actions starting with a BEGIN action, ending with a COMMIT or ROLLBACK action, and containing any other BEGIN, COMMIT, or ROLLBACK actions. Fig 2-1demonstrates an example in terms of a conceptual transactional access to objects R1 and R2.

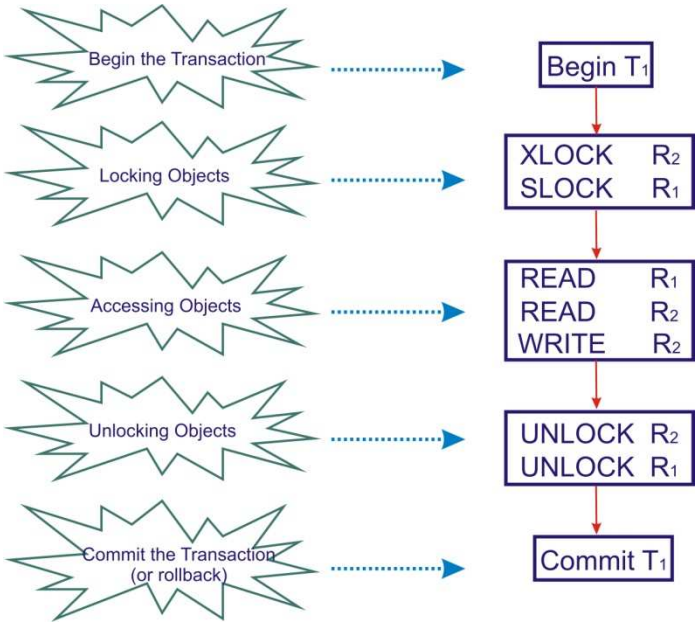

FIG. 2-1 A SAMPLE TRANSACTION

Transactions are characterized symbolically by a sequence such as$\langle\langle t, a_i, r_i\rangle | i = 1, \dots, n\rangle$. This means that the i^{th} step of transaction t preformed action a_i on objectr_i. To simplify the transaction model, the actions will be defined in simpler manner. BEGIN, COMMIT, and ROLLBACK are defined in terms of other actions, so that only READ, WRITE, LOCK (SLOCK and XLOCK), and UNLOCK actions remain.

A *simple transaction* is composed of READ, WRITE, XLOCK, SLOCK, and UNLOCK actions. Every transaction, *T*, can be translated into an *equivalent simple transaction* as follows (Gray & Reuter, 1993):

(1) Discard the BEGIN action.

(2) If the transaction ends with a COMMIT action, replace that action with the following sequence of UNLOCKS:

<UNLOCK A | if SLOCK A or XLOCK A appears in *T* for any object/resource A>.

(3) If the transaction ends with a ROLLBACK statement, replace that action with the following sequence of PUTs and then UNLOCKs:

<WRITE A | if WRITE A appears in *T* for any object/resource A> ||<UNLOCK A | if SLOCK A or XLOCK A appears in *T* for any resource A>.

The idea here is that the COMMIT action simply releases Locks, while the ROLLBACK action must first undo all changes to the objects the transaction wrote and then issue the objects the transaction wrote and then issue the unlock statements. If the transaction has no LOCK statements, then neither COMMIT nor ROLLBACK will issue any UNLOCK statements, as that would risk violating isolation.

2.1.2 Well-formed and Two Phased transactions

A transaction is said to be *well-formed* if all its READ, WRITE, and UNLOCK actions are covered by locks, and if each lock action is eventually followed by a corresponding UNLOCK action (Gray & Reuter, 1993), (Date, 2003).

A transaction is defined as *two-phase* if all its LOCK actions precede all its UNLOCK actions. A two-phase transaction *T* has a growing phase, *T[1], …,T[j]*, during which it acquires locks, and a shrinking phase, *T[j+1], …,T[n]*, during which it releases locks (Gray, 1992). The simplified Fig 2-1 (focusing on the formal locks), has been shown in Fig 2-2 and the concept of well-formed and two phase is indicated.

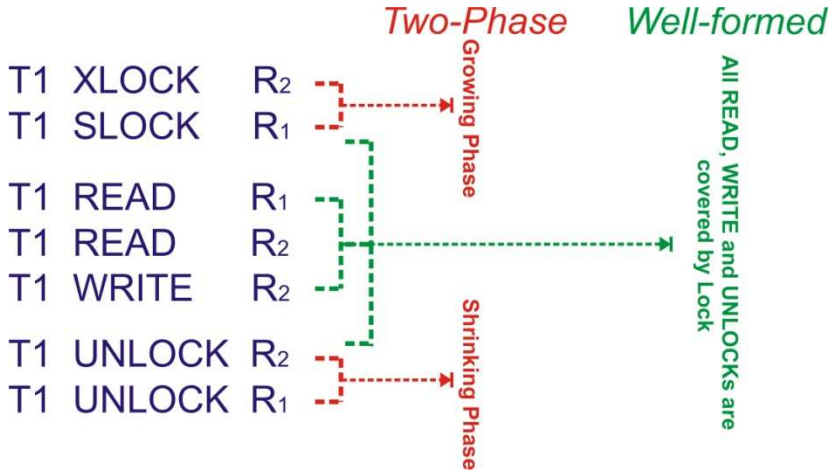

FIG. 2-2 TWO PHASE AND WELL-FORMED LOCKING

Now we can provide the definition of serial histories as one of the important concepts in the isolation theorems. First, a *history* is any sequence-preserving merge of the actions of a set of transactions into a single sequence for the set of transactions and is denoted$H = \langle \langle t, a, r \rangle_i | i = 1, \dots, n \rangle$. Each step of the history $\langle t, a, r \rangle$ is an action a by transaction t on object r (any resources). A history for the set of transactions $\{T_j\}$ is a sequence, each containing transaction T_j as a subsequence and containing nothing else. A history lists the order in which actions were successfully completed. *Serial histories* are “one-transaction-at-a-time” histories. In serial histories, as no concurrency is induced there is not any inconsistency and no problem with viewing “dirty” data by other transactions.

2.1.3 Legal history and Lock compatibility

As expected, a history should not complete a lock action on an object when that object is locked by another transaction. But if two or more transactions want to just read the content of an object, they do not change the object version (state). This may not cause any conflict or access to dirty data (data/object which has been written by another transaction) but the transaction has not committed and may change the version of the object again. The table 1 shows the lock compatibility. The locking compatibility rules constrain the set of allowed histories.

TABLE 1. LOCK COMPATIBILITY

Compatibility		Mode of Existing Lock	
		Share	*Exclusive*
Mode of Requested Lock	Share	Yes	No
	Exclusive	No	No

Legal history: Histories that obey the locking constraints are called *legal*. In Fig 2-3, three histories are shown, where History 1 and 2 are legal and History 3 is not.

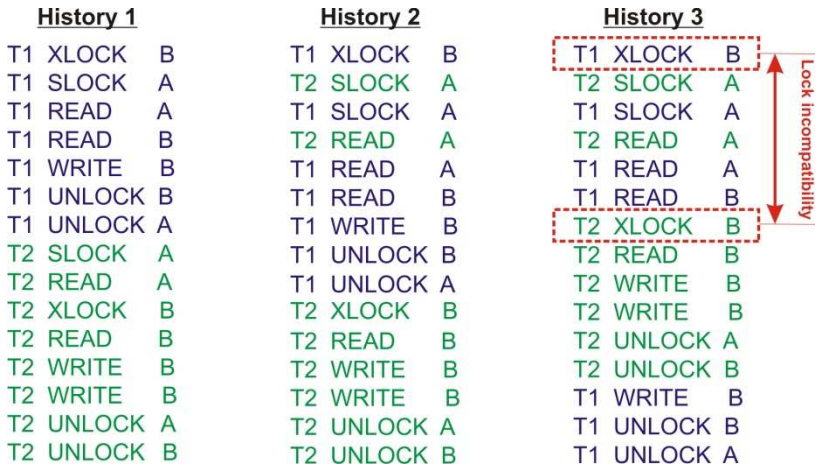

FIG. 2-3 LEGAL HISTORIES AND ANOMALY

In Fig 2-3, history 1 is a serial history. It is obviously legal, as each transaction will be run in sequence and no locks will conflict. History 2 is a non-serial legal history. There are no incompatible locks between T1 and T2 as T2 applies an XLOCK on object B only when T1 has performed an UNLOCK. Finally, history 3 is a non-serial and not legal history, as object B has an XLOCK by T1 but T2 applies an XLOCK on the same object, which is illegal according to Table 1. As a consequence, we can see that T1 then performs a WRITE based on its earlier READ and overwrites T2's WRITE, which is the case of 'Lost Updates' (this is one of the classic anomaly cases which is discussed in (Gray, 1992) and chapter 4).

When we are able to use the non-serial and legal (consistent) histories, it is possible to

increase the concurrency. As far as not conflicting in XLOCKs, the concurrency can be maximised. Fig. 2-4 shows a comparison between two histories of transactions T1 and T2. When in history 1, two transactions have to be executed one at a time, in history 2 until point *c*, two transactions can be concurrent (it is equivalent with history 2 in Fig 2-4).

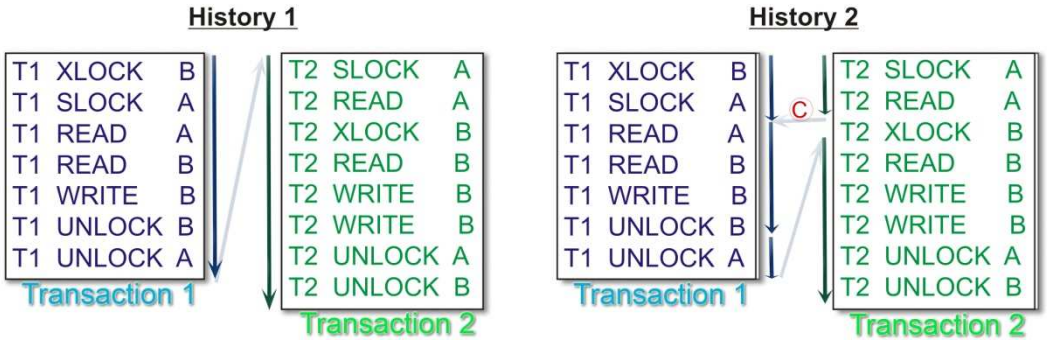

FIG. 2-4 SERIAL AND CONCURRENT HISTORIES

Based on Isolation theorems (Gray, 1992), (Date, 2003), any well-formed and two phase legal history is isolated and can fulfil the consistency of the transaction model.

2.2 LONG-RUNNING TRANSACTION CHALLENGES AND GENERAL SOLUTIONS

While the conventional transaction model can provide solutions for a huge range of applications (such as databases and ACID transaction based application), it has some limitations for advanced applications (Elmagarmid, 1992). CAD applications, business to business interactions, distributed computing and most of enterprise applications deal with transactions which have long durations. Based on the conventional model, these transactions will lock resources and limit access to them. Therefore other transactions cannot access to them during the life of the transaction. This waiting time will reduce the overall performance of the system dramatically, increase the probability for deadlock and violate the concurrency laws (the overall performance of concurrent transactions should not be less than a serial execution of them) (Gray & Reuter, 1993). This transactional problem is addressed by '*Long-lived transactions*'.

Eliot Moss introduced the first revolutionary answer to the Long-lived transactions problem in 1985 with the title of "Nested Transactions" (Moss, 1985). These Nested Transactions broke the atomicity of conventional transactions, but required the definition of new integrity rules instead. However, many problems remained unsolved. Different non-conventional transaction models were subsequently derived from the Nested-Transactions model, which changed the face of the transactional world.

In the Nested transaction model, each transaction can have a tree structure (including many sub-transactions) and each node can share its results to the others in the same transaction (Fig 2-5). However, one Nested transaction cannot share its results with any of

the other nested transactions. Therefore each nested transaction still was atomic and isolated as far as other nested transactions were concerned and hence the ACID properties were still applied, although at a different level.

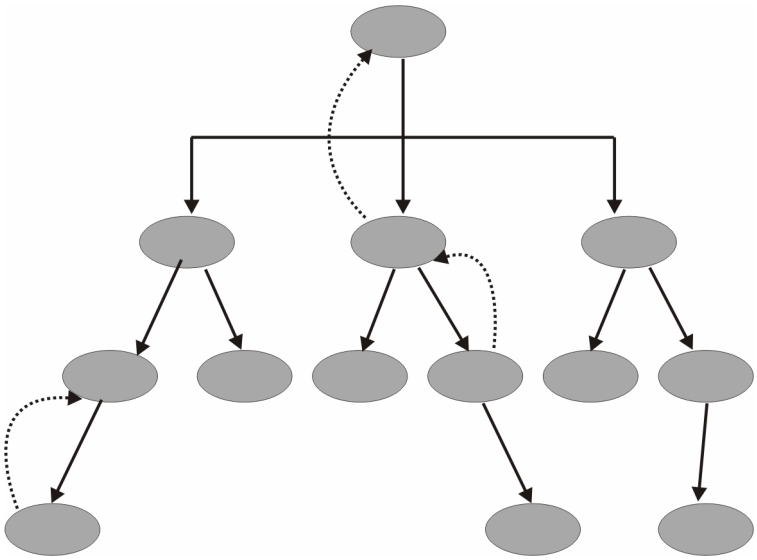

FIG. 2-5 CONVENTIONAL NESTED TRANSACTION

One of the important requirements in the distributed computing domain is the ability to access to information before transaction commit. In the conventional Nested Transaction model (Moss model), in order to maintain transaction consistency, an internal limitation has been applied and the interaction (sharing the results) is available only internally within a transaction. This means only the parent of a sub-transaction can receive the results of its child and all other transactions have to wait for commitment of the main transaction. In some applications, this can create a bottleneck, when other transactions need the result of that particular transaction. This will be more dramatic when we are dealing with several concurrent transactions and they need some results from each other. In these cases, the logic of application may need access to the results of uncommitted transactions but the transaction constraints do not allow such access. This problem arises through unavailability of '*Partial Results*' (Kakeshita & Xu, 1992), (Haghjoo & Papazoglou, 1992), (Elmagarmid, 1992).

In addition, based on conventional nested transactions, if a Transaction is aborted (for any reason) all of its sub-transactions must also be aborted. This means all of the intermediate results produced up that point must be rolled back to the initial values. In most cases, the transaction abortion is followed up with a restart (that means the philosophy for abortion was another try to execute the transaction). But based on this model, there is not any differentiation between full abortion and restart, and in both cases, all sub-transactions will be rolled back and will be redone; this will cause a huge overhead and waste the resources of the environment (where successful operations could be kept until the transaction restart). This problem is known as '*Omitted Results*' (Haghjoo et al., 1993), (Kakeshita & Xu, 1992).

2.3 Different Answers to the problem

The Partial Result problem is perhaps one of the most complicated problems in the distributed computing and applications which are involved with long-lived transaction. In one hand, maintaining the consistency and integrity during sharing these partial results is a challenging task and on the other hand recovering a failed transaction can cause considerable complexity. Any solution can apply some extra limitations and need to provide addition infrastructure in architectural level. In general, two different approaches have been purposed to answer this problem:

- **Accepting Nested Transactions rules**: In this category, the infra-structure of Nested Transactions has been accepted and consistency of transactions is maintained by applying the integrity rules of a Nested Transaction. In term of recoverability, the conventional recoverability is used and the log-based method is applied (Moss, 1987). For solving partial results, each transaction should include all relevant transactions as sub-transactions. In this way, the partial results can be shared between these sub-transactions (rather than outside transactions). The best-known model in this approach area is Multidatabase (Nodine, 1993), (Nodine & Zdonik, 1994), (Xiao Weijun, Zhengding, Bing, & Sarem, 2001).

- **Open Nested Transaction**: Open Nested transactional models use the main principle of conventional nested transaction (using the tree structure and nesting sub-transactions) but the constraints have been relaxed and by introducing new consistency rules some level of the partial results is allowed. Then based on the transaction model and its requirements, transactions are able to release some of their uncommitted results. In this case, the Integrity rules need to be completely different (as well as some additional structure for their models). Some of the well-known transaction models in this category are: Cooperative Transaction model ((Ramampiaro & Nygård, 1999), (Haghjoo et al., 1993)); Multilevel Transaction Model (Deacon, Schek, & Weikum, 1994); Coordinating Transaction Model (Li, Xu, & Liu, 2002); Extended-Saga Transaction Model (Garcia-Molina, Gawlick, Klein, Kleissner, & Salem, 1991); Mega Transaction Model (Haghjoo, 1996); DOM Transaction Model (Elmagarmid, 1992).

In the rest of this section, briefly we will review these models but before reviewing these models, we start with Saga as one inspirational model which by introducing a compensation mechanism, has effected most of transactional models (including conventional nested transactions).

2.3.1 SAGAS (AND EXTENDED SAGAS) TRANSACTION MODEL

In the primary Sagas transaction model (Elmagarmid, 1992), (Garcia-Molina et al., 1991) the structure of a transaction is based on a linear, sequence of serial sub-transactions that are run one after the other. The important properties of these sub-transactions, is compensability.

Let transaction S (as a Saga transaction) include sub-transactions T1, T2, T3, …, Tn. Each sub-transaction Ti, as an atomic unit, has a corresponding compensation transaction, Ci. If any fault arises, the Compensating transactions will be run in the opposite sequential order (Figure 2-2).

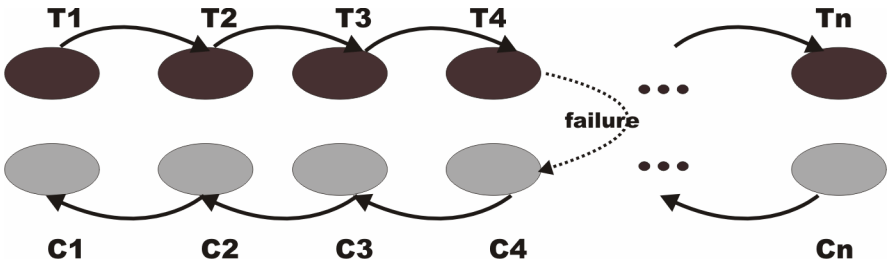

FIG. 2-6 SIMPLE SAGA TRANSACTION MODEL

The main ideas behind Saga have been reused in several models. Firstly, any Saga transaction can be imagined as an independent compensatable unit (although this assumption is not actually used in the Saga transaction model). This idea is used in Mega transaction (Haghjoo, 1996) and in recovery management of the Co-operative Model (Ramampiaro & Nygård, 1999). Secondly, the linear structure of a Saga transaction model is used in the Coordination model (Li et al., 2002) and in the Mega transaction model (Haghjoo, 1996), (Razavi, 1999) as a scheduler (Serial scheduler).

The structure of Saga (linear sub-transactions) can cover many different transactions in a distributed environment. However, in reality we cannot model every transaction as a series of compensatable sub-transactions (as the nature of some transactions is not compensatable) further more the linear structure does not allow different types of business activities to be modelled as transaction (Garcia-Molina et al., 1991). The extended-saga transaction model tries to solve these problems. By introducing a tree structure, extended-saga improves the model in several areas (mostly extended-saga is called ACTA; means actions in Latin), in Workflow technology (Eder & Liebhart, 1994), whereas distributed transactional component based applications in Enterprise JavaBeans use ACTA (M. Prochazka, 1999), (Marek Prochazka, 2001)). But the isolation of a Saga transaction is still is a major inhibitor to releasing partial results during the lifetime of a Saga transaction. Meanwhile concurrency inside of a transaction is limited too. Even in extended Saga, parallel sub-transactions cannot share their results and in an actual scenario, if there is a shared resource, they have to execute one after the other (serial/sequential execution).

2.3.2 MULTI-LEVEL TRANSACTION MODEL

The Multi-level Transaction model (Deacon et al., 1994) has the same structure as the Nested transaction model, but with an improvement for allowing some level of the partial result problem. A sub-transaction of a Multi-level transaction can release results to the same level in the other Multi-level transactions (Fig 2-7).

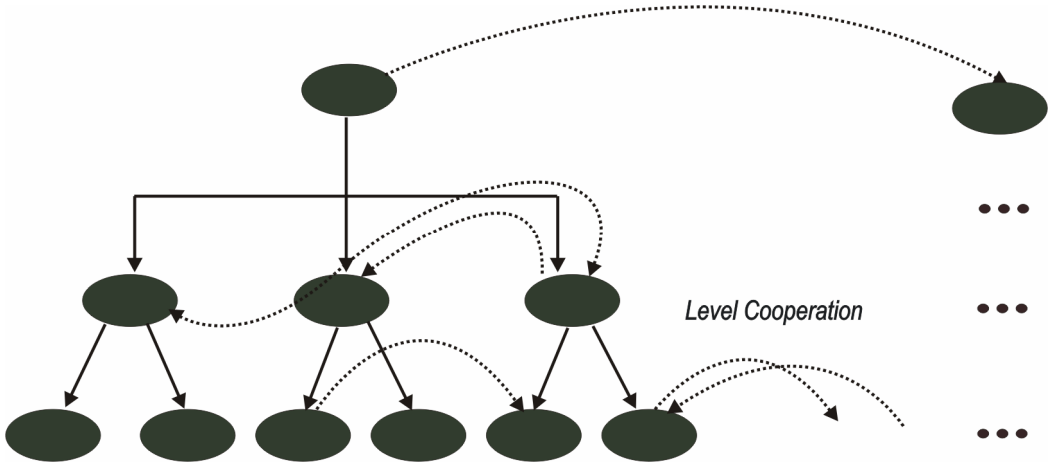

FIG. 2-7 MULTI-LEVEL TRANSACTIONS

Therefore there is a virtual network (a graph structure) for the same level of children (sub-transactions). These important graphs try to give an "info-structure" to compensate for the lack of isolation and atomicity (compared with a conventional transaction model). The Multi-level transaction model has been used in several applications.

In general, the nature of an application must be compatible with the Multi-level definition. Unfortunately, in several cases this limitation reduces usability of the model (or developers have to follow complicated instructions for creation of the necessary compatibilities and modifying their main application). Not only is the model dealing with considerably high overhead structures (global Multi-level graphs) even when the construction of parallel graphs may not be used at all, but also the need for deadlock control will cause another extra overhead in any practical situation (depending on the nature of the application and the value of releasing data in the same level).

2.3.3 CONTRACT TRANSACTION MODEL

The Contract Transaction Model (Elmagarmid, 1992), (Verharen & Papazoglou, 1998) is designed for controlling complicated and long calculations in specific environments such as CAD and Office automation. The Contract Transaction Model is one of the best-known

models for several motivational ideas (even though it is little used now in practical situations). For example, one innovative feature is that when a failure occurs, Contract transaction tries to continue its running by finding alternative ways to proceed; 'Forward Recoverable behaviours'.

The Contract model gives permission for commitment of partial results before final commit of a Contract transaction (releasing partial results). Also it tries to warranty consistency and integrity with forward recovery. There are two types of sub-transactions:

Compensating transactions: These transactions try to undo the effect of unsuccessful/unnecessary sub-transactions.

Contingency transactions: When a sub-transaction fails, this alternative can be run in an attempt to avoid an abort (forward recovery).

The Contract model introduces contingency and Vital / Non-Vital transactions (with alternative sub-transactions) as two important ideas. Unfortunately, in a particular environment, structural limitations and huge overheads are the main problems for implementation of this model (even some sources categorize this model as just an extension to the conventional model that adds some control mechanism to several ACID transactions in the Contract of a bigger transaction) (Date, 2003), (Limthanmaphon & Zhang, 2004).

2.3.4 Split Transaction Model

This model was introduced for supporting 'Open–ended applications' like VLSI designs, CAD/CAM projects and so on. Any application with the following three properties is defined as an 'Open–ended application' (and theoretically can be modelled by the Split transaction model):

1. **Uncertain Duration:**The transaction life time can be hours or months; you cannot determine any specific time limitation on the run time.

2. **Uncertain Development:**There is no prediction about operations and limitations on them.

3. **Interaction with other Concurrent Activities:**Concurrent operations can affect each other and share their results at any time (not just after commit, which will not be at a specific time, based on property 1).

The manager of the Split Transaction Model splits a transaction into two ordered transactions and divides the resources between them (Elmagarmid, 1992). To split transaction T into transactions A and B, where A is ordered before B, the following properties must be satisfied (for splitting resources):

$$AwriteSet \cap BwriteSet \subseteq BwriteLast$$

$$AreadSet \cap BwriteSet = \varnothing$$

$$BreadSet \cap AwriteSet = ShareSet$$

The first property says if A and B write on an object, B must write after A. In other words A cannot write on B's output. The second property says A cannot see B's results. The third property says B may see some of A's results. These properties guarantee A is ordered before B. If ShareSet and BwriteLast are nil (in this sense, A and B are independent) A and B can be ordered anyway or they can be run completely individually.

If ShareSet or BwriteLast are not nil (this is called a serialized situation), A can be ordered before B. In this order, objects in ShareSet must be unchanged (by A). On the other hand, B with using uncommitted A's data can commit. If A aborts, B must be aborted too (because B can read A's written data). The main aim of Split transactions is commitment of one of the Split transactions (A in this example) and releasing useful results for the main transaction which can continue without interruption.

Split transaction is criticized for applying limited consistency rules and it may not be applicable in business activities. However, it introduces several important ideas that are used by the other models;

1. Adaptive recovery: Tasks (application) will be committed by strings of (sub) transactions that will not be disaffected by some failures.
2. Reduce Isolation: Releasing resources (data items) by committed parts of a transaction (partial commitment).
3. Ordered access to resources.

2.3.5 FLEX TRANSACTION MODEL

A Flex Transaction is a composition of a task set. Each task is equivalent to a series of sub-transactions and it can commit successfully just when one of these sub-transactions is committed (O. Kuhn, Elmagarmid, & E. Kuhn, 1993). A Flex transaction will commit when all of its tasks finish successfully. In the Flex model, the main transaction can use a description of the dependencies between sub-transactions (task's sub-transactions).

Three important dependencies in the Flex model are: '*Failure Dependency*'; '*Success Dependency*'; and, '*External Dependency*'. Failure and Success dependencies determine the order of running sub-transactions in different situations (failure and successful finishing). External dependency shows dependencies of sub-transactions to transactions that do not belong to the main transaction. These dependencies make the creation of a compensating mechanism possible.

The Flex model was implemented in VPL (Vienna Parallel Logic) for the first time (Elmagarmid, 1992).

2.3.6 Cooperative Transaction Model

The Cooperative transaction model (Haghjoo et al., 1993), (Ramampiaro & Nygård, 1999) is used for several different structure / mechanisms. A Cooperative transaction model is a hierarchy of sub-transactions (similar to nested transaction) where there is a possibility for releasing data-items in sub-transactions before commitment of the main transaction (for reaching the result, cooperation between transactions is necessary and this cooperation is in effect the sharing of uncommitted data-items).

The common architecture of Cooperative transactions enables release of partial results for other transactions (Fig 2-8). However, how they do this, what rules must be applied, which structure design is considered for concurrency control and recovery management and so forth, can be completely different in each customized model.

Different Answers to the problem

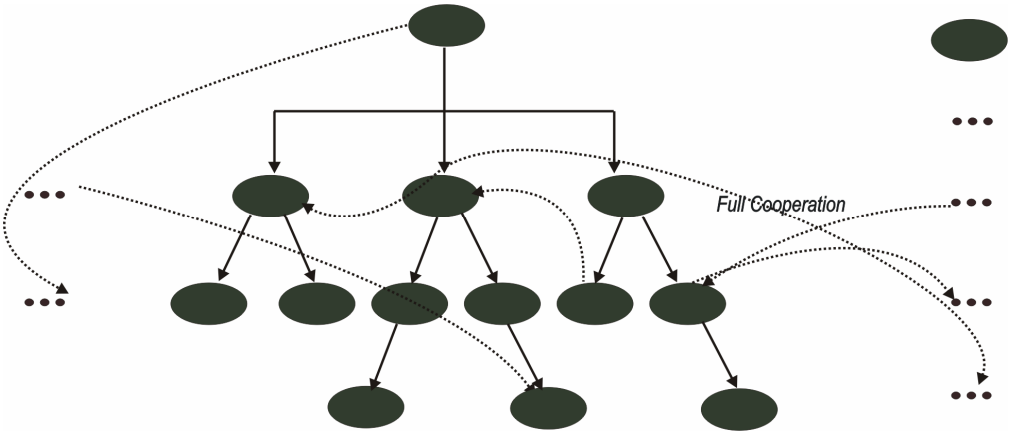

FIG. 2-8 COOPERATIVE TRANSACTION MODEL

In real situations, for implementation of Cooperative model, different structures are selected. The limitation for applying the customized model (suitable for a specific application environment) for saving database consistency and integrity must be considered (Li et al., 2002). The important points of each Cooperative transaction models are the structure of:

- Logs (data structure) for chains of dependent sub-transactions (those releasing data-items and/or using the other's released data items).
- Recovery Management implementation; heavy and centralised recovery management which guarantee atomicity of transactions
- Concurrency control structure; a server manage the access right for shared data-items
- Optimizing overheads and logs; the model has minimised the overhead (through centralised control)
- Mechanisms for breaking loops/ deadlocks and any other illegal events; recording logs help the model for preventing deadlocks and resolving failures

Despite the advantages of the cooperative transaction model, dependency to a centralised control is its bottleneck and it cannot be applied in a fully distributed

environment (when avoiding single point of failure is the priority, as in a Digital Ecosystem).

2.3.7 Coordinating Transaction Model

The idea of categorising sub-transactions in the Cooperative Transaction Model (Li et al., 2002) inspired a new type of model, called the Coordinating transaction model. In the coordinating transaction model, sub-transactions are divided to two categories; Control nodes (sub-transaction) and executive nodes (sub-transactions). Control nodes can determine the order of running/termination of their children and/or how (validation/mechanism) they could release their results to their siblings or even to other transactions (partial results) (Dalal et al., 2003), (Li et al., 2002).

Coordinating transactions work based on grouping transactions with three types of management (in the commitment condition). Database consistency is achieved through the concept of scheduling transactions (grouping). There are two different notifications for 'writing' (modification/creating). The written data items can be shared between group members, or can be private to modifiers (owners). Based on the idea of grouping transactions, different implementation models have been derived from the Coordinating transaction model, which can provide some level of distribution (flexibility for extending the distribution between coordinators is the major difference between coordinating transaction model and cooperative transaction model).

2.3.8 Mega Transaction Model

The Mega Transaction Model (Haghjoo, 1996) is an open-nested coordinating model that was introduced for special distributed environments (e.g. aircraft design projects, Open-ended projects). The structure of the model is a tree that consists of compositions of schedulers in the root and middle nodes, and executive / delegations (agents) in the leaves. Fig 2-9 shows an example of such a model; symbols are derived from chains of researches by (Haghjoo et al., 1993), (Haghjoo, 1996), (Verharen & Papazoglou, 1998), (Papazoglou, Dells, Bouguettaya, & Haghjoo, 1997), (Papazoglou, Delis, Haghjoo, & Bouguettaya, 1996).

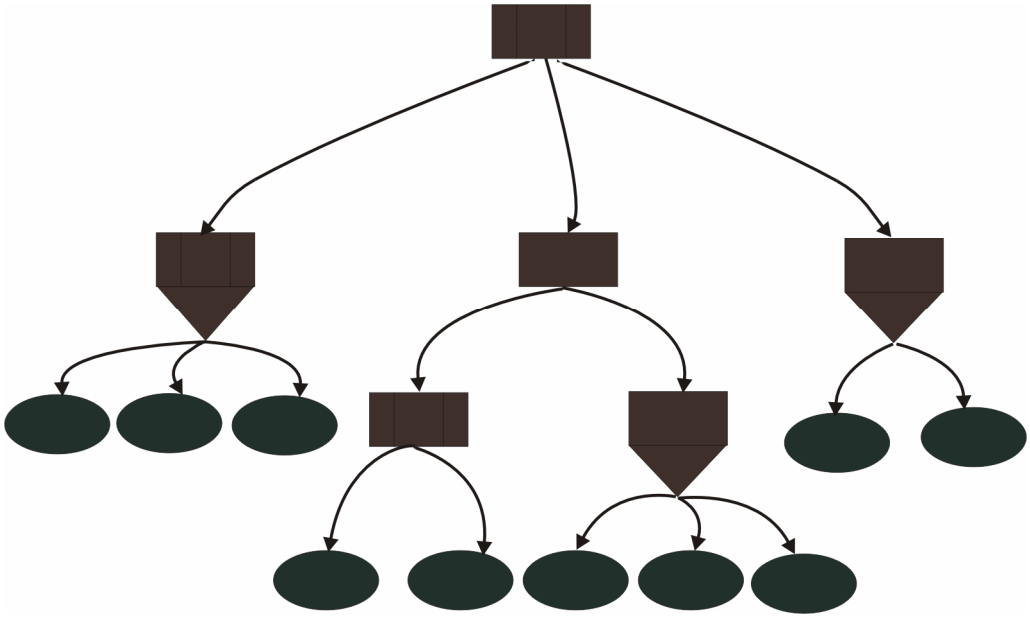

FIG. 2-9 MEGA TRANSACTION MODEL

The possibility for using any composition of schedulers gives a powerful advantage to this model for covering a huge range of applications. Different derivations of this model have been implemented and customised for different requirements. Mega Transaction uses compensation as a part of Recovery Management (Haghjoo, 1996), (Razavi, 1999), and cooperation between different transactions is available in any level (by considering the integrated recovery manager). Delegation is considered as a blanket mechanism for providing heterogeneous facilities. The restriction of dependencies to the scheduler gives the opportunity for optimization (Papazoglou et al., 1997), (Haghjoo, 1996) but this causes some side effects for the coverage of real-time applications. Mega transaction could be considered as a wise counsel for designing customised Coordinating transaction models in different environments (which can be optimised based on requirements).

2.3.9 DOM Transaction Model

This model is an improved Cooperative transaction model. It was designed in GTE (formerly General Telephone & Electronics Corporation) laboratories for the management of distributed object projects in a heterogeneous environment (Elmagarmid, 1992). Theoretically DOM (Distributed Object Management) transaction model can have different heterogeneous components such as different types of Database structures (Relational and Object-Oriented data sets) and even non-Database parts (like different file/record based systems).

DOM includes different structure blocks that can combine for creating more complex

structures. Depending on requirements; these blocks can act like the conventional transaction model, a nested transaction model or even an open-nested transaction model. DOM provides these facilities through a structure that includes five different levels transaction:

1. **Top transactions:** A Top transaction is a root of the (closed) nested transaction model. It is visible to the system where it is running, 'Top transactions' can be defined by the user or be the result of other transactions. They can be mixed by Multi transactions but they cannot release partial result by themselves (this can be done by Multi transaction).

2. **Multi transactions:** These are for long-lived transactions and the accessing of information from external databases (over which DOM cannot have any control). Sub-transactions within a Multi transaction can potentially release their results outside of the Multi transaction (to the other transactions outside of the main transaction). Transactions in a Multi transaction can in turn be a Multi transaction or a Top transaction.

3. **Compensating transactions:** There is a compensating transaction for each Top transaction in a Multi transaction. In fact the compensating transaction is a Top transaction that can undo the effect of other Top transactions. When a Multi transaction aborts, a compensating transaction of each committed Top transaction must be run.

4. **Vital and Non-vital transactions:** Multi transaction components (sub transactions) can be Vital or Non-vital transactions. If a Vital transaction aborts the main Multi transaction must be aborted, but if a Non-vital transaction aborts, the main Multi transaction can continue.

5. **Contingent transactions:** Contingent transactions can be alternatives for some main transactions in a Multi transaction. If a transaction aborts and has a Contingent transaction, the Contingent transaction will be run automatically.

2.4 Analysis Parameters

There are three different parameters in structuring the design of any transaction model: what is the main structure of model; which type of transactions are supported in the model; and, how it supports/optimises distribution. In this section we try to provide a brief analysis on them.

2.4.1 Model Structure

At the first level, advanced transaction models are categorised into two different structural classes (Elmagarmid, 1992):

1. Those models that try to solve the long-life problem by making a hierarchical structure of tasks (sub-transactions); The Cooperative model, Mega transaction, DOM are some models that are using a hierarchical structure, whereas Saga transaction, and Extended Saga use the second category.
2. Those models thatfollow a linear structure of tasks (sub-transactions) .

The main challenge in hierarchical structure are the implementation overheads which are conventionally controlled by a centralised server. Alternatively coordination models try to solve the problem by spreading coordination and avoiding a centralised control but generally their solution is involved with local states of participants (which cause huge complexity and it is against of some disciplines such as service oriented computing).

On the other hand, the linear-based transactions (such as Saga) have less design complexity but they reduce transaction concurrency dramatically and as result the overall performance of the environment will be low, the probability for deadlock stays high and in most cases they may not be considered as a feasible solution for long-running transactions.

Another categorization for model structure is the ability for supporting dynamic decisions in the model. This was the main reason for defining the Split model. Dynamic structure is one of the important factors for supporting heterogeneous environments and optimizing the network cost (if suitable structure can be defined as ADT/Abstract Data Type that could have possible implementations).

2.4.2 Transaction Types Support

The variety of sub-transactions is one of the most important items, not only for model generalization, but also for the performance/flexibility of models in different environments. Some of these sub-transactions are '*contingent transactions*', '*compensating transactions*', '*Vital/Non-Vital transactions*'.

Contingency helps transaction models to avoid some failures by selecting alternative sub-transactions or improving their performance by the abortion of slow transactions. The efficient usage of these sub-transactions is in coordinating models when there is a control node (Elmagarmid, 1992).

A compensating mechanism, not only may give the possibility of safe release of partial results to a transaction model, but also can help recovery management when failure occurs. In some models, the entire recovery procedure may rely on such transactions (as in the Saga model or DOM transaction model).

Vital/Non-Vital transactions can give some type of priority to sub-transactions and improve the performance of a transaction model. This also improves robustness as abortion of the main transaction will not happen when a low priority (Non-Vital) sub-transaction aborts.

2.4.3 DISTRIBUTION

Distribution can be considered as an important measurement for transaction models. When a model is able to provide a fully distributed consistency and recovery model, the traffic complexity will reduce and at the same time, the necessary consistency processing will be spread among participants. Furthermore, the entire model does not rely on single point of failure. But this may cause additional complexity and needs for accessing the local state of each participant. The degree of distribution and level of delegation of tasks to participants can affect this complexity

2.4.4 COMPARISONS

This chapter has surveyed the major advanced transaction models, which try to solve the problems inherent in long-lived transactions. Not all of them can be directly applied in service oriented computing but they do provide basic definitions for them (in the next chapter we will focus on transaction models on SOC). On the other hand, as we have seen in chapter 1, the Digital Ecosystems requirements are not only for a service oriented environment, which should warranty the local autonomy of participants, but also by providing support for business activities should be able to provide a solution for long-life transactions, with the corresponding needs for releasing partial results, and cashing potentially useful results on failure of a transaction.

We include a summary of their main properties in Table 2.1.

Analysis Parameters

Transaction Model	Structure	Sub-transactions	Distribution
Conventional	-	-	-
Nested Transaction	Hierarchical Static	Contingency Non-Vital	Central control
Sagas Transaction	Linear Static	Compensating	Locality
Multi-level	Limited Hierarchical Static	Contingency	Central Control
ConTract Model	Linear Static	Contingency Non-Vital Compensating	Not any specific method
Split	Linear Dynamic	-	NA
Flex	Hierarchical Dynamic	Contingency Non-Vital Compensating	NA
Cooperative	Hierarchical Static	Contingency Non-Vital Compensating	Mobility (migration)
Coordinating	Hierarchical Dynamic	Contingency Non-Vital Compensating	NA
Mega Transaction	Hierarchical Dynamic	Contingency Non-Vital Compensating	Delegation
DOM	Hierarchical Dynamic	Contingency Non-Vital	Open to design

		Compensating	

3 BUSINESS TRANSACTIONS IN A DIGITAL ECOSYSTEM

The purpose of any business network, including Digital Business Ecosystems, is to enable networked organisations to engage in business transactions that realise their core business activities. Specifically digital ecosystems insist on distributed coordination of such transactions and support for loose coupling and local autonomy in the environment (Chang et al., 2006), (Razavi, Moschoyiannis, & Krause, 2008a), (Moschoyiannis, Razavi, Zheng, & Krause, 2008). If such a network is to support B2B (Business to Business) interactions between SMEs, in a digital ecosystem, it should be fully distributed (no central point of control for performing a transaction or network operation) and should also offer a consistency model for performing transactions. This means it should be highly resistant to *failures* – a stable environment where business transactions can be executed.

Each business transaction is the result of peer-to-peer interactions, between several nodes (SMEs) of this network for reaching a specific target. These nodes are called *participants*; their logical components for involving in a transaction are their services. The context of a transaction clarifies the orchestration of services. The locality of services clarify the necessary interactions between participants for satisfying the business logic of the transaction. Therefore we are dealing with a service-oriented environment, which includes several overlapping networks of transactions.

3.1 SERVICE-ORIENTED ENVIRONMENT AND THE DEPLOYMENT LAYER

In simple terms Service-Oriented Computing (SOC) (Dillon, Wu & Chang 2007a), (Dillon, Wu & Chang 2007b), (Papazoglou, Traverso, Dustdar, Leymann, & Kramer, 2006), (Singh & Huhns, 2005), (Chang & West, 2006a), aims to enable applications from different providers to be offered as services that can be used, composed, and coordinated in a *loosely coupled* manner. In this paradigm, services are fundamental elements for developing solutions. They are platform-agnostic computational elements that support rapid, low-cost composition of distributed applications. Services perform functions, which can be anything from simple requests to complicated business processes. The actual architectural approach of SOC is called SOA (Service-Oriented Architecture) and is particularly applicable when multiple applications running on varied technologies and platforms need to communicate with each other. In this way, enterprises can mix and match services to perform business transactions with minimal programming effort. SOA is a way of reorganizing software applications and support infrastructure into an interconnected set of services, each accessible through standard interfaces and messaging protocols. In this way, enterprises use composition of services to perform business transactions with minimal programming effort and provide a consistent environment.

However, service composition has several distinct characteristics which distinguish it from classical workflow integration, conventional transactional implementation and software component integration. As SOA is a loosely coupled environment, service composition relies

on parameters for invocation of a service (also called access interfaces), rather than working with the local state of execution of the respective services. Additional complexity results from the fact that a variety of services, from different platforms, need to pass their results in a loosely coupled manner.

At the same time, the complexity of different types of service composition requires a consistency model at the deployment level. Meanwhile since the locality of services is distributed (in different participants / service providers or SMEs), the quality of service composition in a business transaction depends on the degree of connectivity that the underlying network exhibits.

3.1.1 LOOSE COUPLING AND SERVICE REALIZATION

In SOA services are considered as atomic units whose local structure, or run-time state, cannot be forced to be made visible or explored by other parts of the architecture. The use of loosely-coupled services is a basic premise in the service-oriented computing paradigm, which distinguishes between two broad aspects of services (Papazoglou, 2003), (Singh & Huhns, 2005) as shown in Fig. 3-1: *service deployment*, which is subjected to our transactional service composition, as opposed to *service realization*.

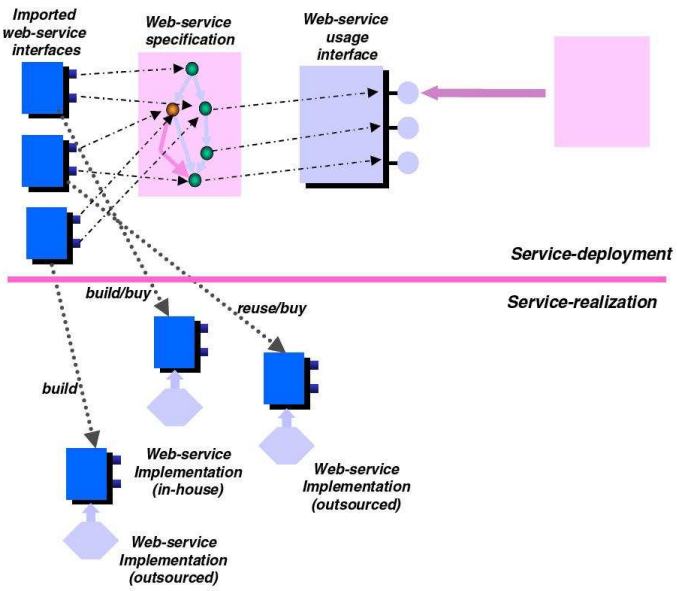

FIG. 3-1 REALIZATION AND DEPLOYMENT LEVELS IN SOC (PAPAZOGLOU, 2005)

The service realization strategy involves choosing from an increasing diversity of different options for services, which may be mixed in various combinations. Our approach abstracts away from the service realisation level but at the logical level, what we consider is '*there is a*

business function implemented in software somehow and this is the interface to it'(Papazoglou, 2003).

A service in SOA is designed in such a way that it can be invoked by various service clients and is logically decoupled from any service caller (loose coupling). This means there are no assumptions of any kind in the service as to what kind of service consumer is using it and for what purpose and in what context. In turn, the service callers are coupled with the service in as much as they know what the services are, how they can be used, and what they can accomplish.

Therefore, at the service deployment level the interfaces of the services will be used to compose different services and accomplish the required behaviours and results in terms of business transactions.

3.1.2 SERVICE COMPOSITIONS AND LONG-RUNNING TRANSACTIONS

Based on the specification advocated in (Yang, Papazoglou, & van den Heuvel, 2002),(Singh & Huhns, 2005),(Papazoglou et al., 2006)service composition can be considered along the following dimensions: data, process, security, protocol. In this chapter we are concerned with providing P2P network support for distributed transactions and hence we will be concerned with the aspects of data and process composition. In general, security and protocol compositions are usually addressed on top of the transactional layer.

In particular, process-oriented service composition is concerned with the following aspects:

Order: indicates whether the composition of services is serial or parallel.

Dependency: indicates whether there is any data or function dependency among the composed services.

Alternative service execution: indicates whether there is any alternative service in the service composition that can be invoked - alternative services can be tried in either a sequential or a parallel manner.

Following (Yang et al., 2002) these aspects can be seen within different types of service composition as follows.

Data-oriented service composition: The data generated at the service realisation level are released in terms of different data-objects. In this service composition, these data can be shared and manipulated between participants of a single transaction or, where partial results are concerned, be shared by participants of other transactions.

Sequential process-oriented service composition: This type of service composition invokes services sequentially. The execution of a component service is dependent on its previous service. These sequential dependencies can be based on commitment in which case we talk about *Sequential with commitment dependency (SCD)* where one cannot begin unless the previous service commits, or dependency on data which in this case we talk

about *Sequential with data dependency (SDD)* where one service relies on other service's outputs as its inputs.

Parallel process-oriented service composition: In this service composition, all the services can be executed in parallel. There may be data dependencies between them in which case we talk about *Parallel with data dependency* (PDD) or there may be differences in how and when the services can be committed (depending on some condition) in which case we talk about *Parallel with commit dependency* (PCD). When there are no dependencies between the parallel services we talk about *Parallel without dependency* (PND).

Alternative service composition*:* This type of service composition indicates that there are alternative services to be deployed and one of them is necessary. They are categorised to two different types: *Sequential alternative composition (SAt)* where there is an ordering for deployment of these services, and *Parallel alternative composition (PAt)*, where there is no ordering (preference) between them and deployment of either service can satisfy the composition.

Generally, one or more service compositions can satisfy the user request. It can be seen that due to order and data in service composition, there can be increased complexity in composing services especially when transactions require a number of different services from different networked organisations. This means that there is a need for a context and data consistency model (at the deployment level) which can provide the correctness of the results.

A digital ecosystem for business can be modelled as a network of business transactions between the various participating organisations (e.g., SMEs) (Chang et al., 2006). Fig. 3-2 shows the network structure resulting from a number of service compositions which need to be deployed for performing business transactions. Furthermore, in terms of deployment of services and sharing their data (data dependency aspect of service compositions), the peer-to-peer interactions between the participants of each transaction should be supported by a reliable infrastructure which reduces the possibility of failure and provides effective recoverability for the transaction taking place.

3.2 Network of business transactions

The aggregations of the business activities taking place between the different partners create several virtual business networks. When these business activities are conducted by means of long-term transactions which involve the execution of services from different service providers, these result in the creation of temporary networks interconnecting the participating organisations. These are typically separate disconnected networks resulting from transactions between participants, but overlaps may exist due to some participants being involved in more than one transaction in the same or different business domain of the digital ecosystem.

Fig. 3-2 shows the conceptual unstructured network of a digital ecosystem. It can be seen

in the figure that transactions can have overlapping sets of participants.

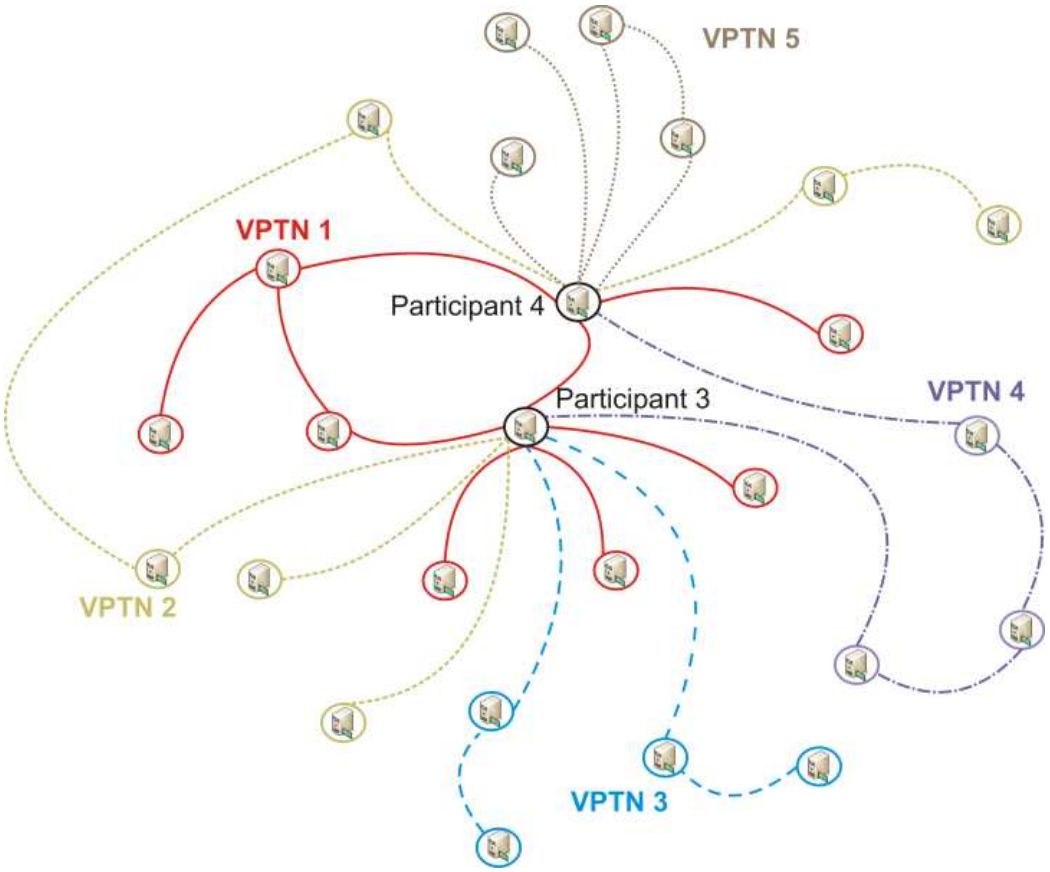

FIG. 3-2 A DIGITAL ECOSYSTEM

In a digital ecosystem, each transaction and its participants, during the execution of the transaction, creates a temporary network. This network is supposed to achieve a specific goal (transaction goal) and provide some results. During the execution of the corresponding transaction, several services will be deployed. Based on the transaction logic, the underlying services will be deployed in a specific order and their results will be combined (according to the specific service compositions used in the transaction). All of these will be private to the participants of the transaction in question. The resulting network is called '*Virtual Private Transaction Network*' or in short form '*VPTN*'. Fig 3-2 shows a digital ecosystem which includes five VPTNs.

The idea is to re-use the (possibly disconnected) VPTNs in providing a connected network for conducting business activities. In a digital business ecosystem, the main motivation of the participants (SMEs) is their business activities (transactions), and thus these sub-networks (VPTNs) are the major part of the system and their success or failure is a major factor for the usability of the ecosystem. For this reason, and before describing the specifics of our approach towards the P2P architecture that supports distributed business transactions in chapter 4, we describe our distributed transaction model focusing on the aspects that are relevant to the underlying supporting network.

3.3 CHALLENGES

As each VPTN is formed by the execution of a transaction amongst the various participants, the very existence of the VPTN relies on the success of the transaction which in turn relies largely on the accessibility of its participants' services. The general term 'Transaction' has been introduced by Gray (Gray, 1992), (Gray & Reuter, 1993) and as we have discussed, is defined by the four properties contained in the ACID acronym. In a transactional environment, consistency should be preserved even when a failure is encountered and the aborted transactions can be rerun. This main principle applies equally to *long-running* business transactions, whose execution is long-term in nature – anything from minutes to hours to days – and whose specification involves the deployment of a number of services from different service providers. For example, a transaction can include several service compositions and after deploying the data at the deployment level, the data can be accessed several times during the course of execution of the transaction and until it *commits*. Furthermore, the data may be required to be shared with another transaction before the first one commits, in what is often referred to as a case of *partial results*. The conventional ACID properties for transactions may not be applicable in such cases because the transaction life-cycle usually can be longer than that of conventional transactions. Also, partial results can violate the conventional isolation expectation of ACID transactions(Gray & Reuter, 1993), (Haghjoo, 1996), (Date, 2003)**.**It transpires that adhering to ACID properties may be rather restrictive in a business context as a number of B2B (Business to Business) scenarios would for instance require the realisation of partial results.

In what follows, we briefly outline current approaches towards relaxing ACID properties and highlight the potential weaknesses with respect to three main criteria: concurrency control (this can cover Consistency and Isolation), compensation and recovery (this covers Atomicity) and replication (for satisfying Durability).

A consortium of companies came together under the umbrella of the Organization for Advance Structured Information Systems (OASIS) and developed the *Business Transaction Protocol* (BTP), which was aimed at B2B transactions in loosely-coupled domains such as Web services (Furnis et al., 2004). At the same time, others in the industry released other specifications: Web Services Coordination (WS-Coordination) (Cabrera, Copeland, Feingold, R. Freund, T. Freund, Johnson, Joyce, Kaler, Klein, Langworthy, Little, et al., 2005) and Web Services Transactions (WS-AtomicTransactions and WS-BusinessActivity) (Cabrera, Copeland, Feingold, R. Freund, T. Freund, Johnson, Joyce, Kaler, Klein, & Langworthy, 2005), (Cabrera, Copeland, Feingold, R. Freund, T. Freund, Joyce, et al., 2005). Recently, Choreology Ltd. has started to make a joint protocol which tries to cover both models and this effort has highlighted the caveats of each as mentioned in (Vogt et al., 2005), (Furnis & Green, 2005), (Razavi et al., 2006).

In what follows we discuss certain important aspects in transactional models and highlight potential pitfalls of current transaction frameworks.

3.3.1 CONCURRENCY CONTROL

For providing a consistent environment, during concurrent actions (service deployment and compositions), WS-* (WS-AtomicTransactions and WS-BusinessActivity) and BTP, are using the two-phase commit (2PC) protocol, which requires synchronisation for the phases. This is applied through a centralised coordination framework, based on WS-Coordination (Cabrera et al., 2004). Fig. 3-3 shows a simple example of the use of WS-Coordination for executing a transaction where the Initiator creates a coordination context and the Participants, based on their registered services, deploy their respective services. The synchronisation for concurrency control is done in a centralised manner. This causes a single point of failure as well as a single dependency on the provider(s) of the centralised coordinator framework.

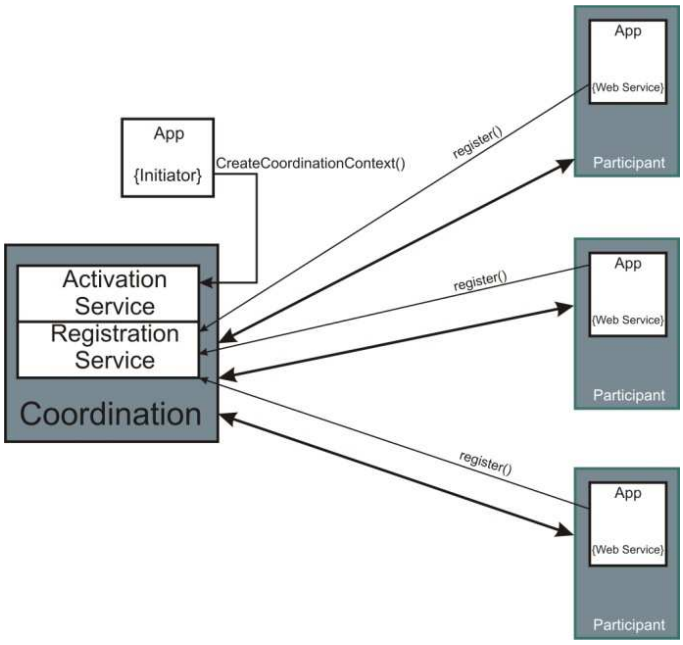

FIG. 3-3 CENTRALISED COORDINATION

In addition, a more careful study of this coordination framework, such as that reported in (Furnis & Green, 2005), shows it to suffer from some critical decisions about the internal build-up of the communicating parties - a view also supported in (Vogt et al., 2005). 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, particularly loose-coupling of services and local autonomy, and thus are not suitable for a digital business ecosystem, especially when SMEs are involved. This is because smaller organisations tend to be more sensitive in revealing their local design and implementation precisely because this is often where their model lies(Razavi et al., 2007a), (Strømmen-Bakhtiar & Razavi, 2008), (Singh & Huhns, 2005).

3.3.2 COMPENSATION AND RECOVERY

In a highly dynamic environment of multiple services from different service providers there need to be procedures in place that allow cope with failure. When a failure occurs before a long-running transaction terminates its execution (before transaction *commit*) there is a serious risk of inconsistency due to the fact that released results of the transaction have not been finalised. 'Recovery' is the procedure for addressing this problem. The mechanism, which has been used by WS-* and BTP is *compensation*. As the coordination protocol for long-running transactions only uses BACC (Business Activity with Coordinator Completion) protocol, in terms of success or failure, it is the Coordinator who can send the completed message after which the transaction transitions to the second phase (commit or abort) of the 2PC protocol.

Behavioural patterns such as "validate-do" and "provisional-final" (Vogt et al., 2005), (Razavi et al., 2007c), (Furnis & Green, 2005), (Razavi et al., 2006) 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.

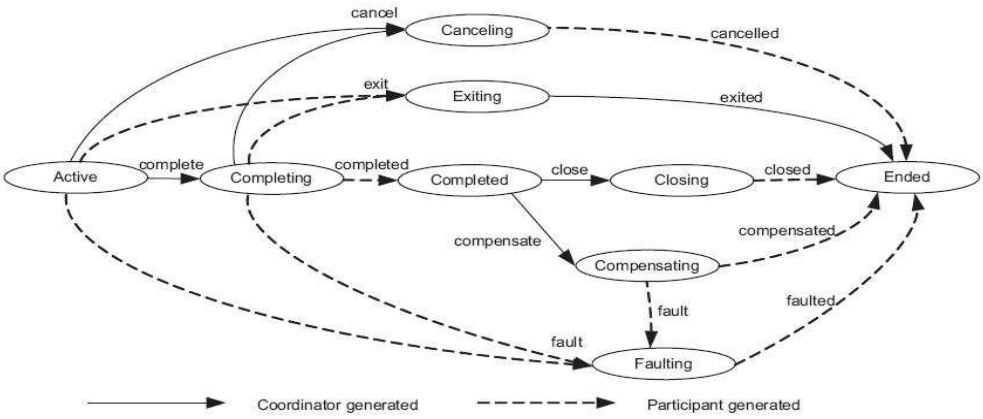

FIG. 3-4 BACC PROTOCOL

As a result of only supporting BACC and hence applying the "do-compensate" behaviour pattern, when a fault occurs, the transaction transitions to 'Faulting' state (see BACC in Fig. 3-4), where only the Participant is involved. Since the control is with the Coordinator in this case (BACC), the Coordinator needs to have visibility of the Participant's states. Notice that after 'Completed', the 'Closing' and 'Compensating' states are controlled by the Coordinator. This has the implication of the Coordinator needing all details of the Participant to perform the compensating procedures. This limitation results in breaking the autonomy of the local platform as it forces to prescribe the internal behaviour of the realisation level of services. Prescribing internal behaviour at the realisation level raises a barrier for SMEs as it inevitably leads to their tight-coupling with the Coordinator.

3.3.3 Replication and durability

As synchronization is synchronised, the consistency log and recoverability (compensation) information, has to be stored in centralised coordinator. In addition, as there is not any infrastructure for checking other participants stability, and issuing the privacy of each transaction, the replication and archiving of data are done by a centralised backup in the coordinator (even when the archiving the information of other participants' transaction can violate the privacy and their local autonomy, this seems like the only option in the current protocols).

It can be seen that there are a number of challenging aspects in providing support for long-running transactions so that the corresponding VPTNs can be used faithfully in determining the connected network supporting a digital ecosystem. In the next section, we describe a distributed transaction model that is designed to address such aspects and paves the way for using the emerging VPTNs as the main building block for the underlying P2P network for digital business ecosystems.

3.4 Service Oriented transactional models

In this section, we provide a brief review of the transaction models which are currently used for service oriented environments. In 2001, a consortium of companies including Oracle, Sun Microsystems, Choreology Ltd, Hewlett-Packard Co., IPNet, SeeBeyond Inc. Sybase, Interwoven Inc., Systinet and BEA System, began work on the Organization for Advance Structured Information Systems (OASIS) Business Transaction Protocol (BTP), which was aimed at business-to-business transactions in loosely-coupled domains such as Web services. By April 2002 it had reached the point of a committee specification, (Ceponkus et al., 2002)and (Furnis et al., 2004).

At the same time, others in the industry, including Microsoft, Hitachi, IBM, IONA, Arjuna Technologies and BEA Systems, released their own specifications: Web Services Coordination (WS- Coordination) and Web Services Transactions (WS-AtomicTransactions and WS-BusinessActivities) (Cabrera et al., 2004). Recently Choreology Ltd. has started to make a joint protocol which tries to cover both models and by mentioning their problems in several detailed reports, tries to solve them (Furnis & Green, 2005).

3.4.1 WS-Protocols (WS-x)

WS-Protocols set consists of the three protocols;

- ✓ WS-Coordination (Cabrera, Copeland, Feingold, Freund, Freund, Johnson, Joyce, Kaler, Klein, Langworthy, Little, et al., 2005), the most important of WS-protocols

which it defined as an extensible coordination framework and its coordinating (sub) transactions in any levels.

- ✓ WS-AtomicTransactions (Cabrera, Copeland, Feingold, Freund, Freund, Johnson, Joyce, Kaler, Klein, & Langworthy, 2005), leveraging WS-Coordination(WS-C) for use with systems aware of ACID properties. Actually it covers atomic transactions which follow a conventional commit protocol.
- ✓ WS-BusinessActivities (Cabrera, Copeland, Feingold, Freund, Freund, Joyce, et al., 2005), designed for support of long-lived activities. It tries to provide enough foundations for applying long-term business activities in the transactional manner (which it is called WS-BusinessActivities).

WS-AtomicTransactions and WS-BusinessActivities use WS-Coordination framework as the coordinator protocol.

WS-COORDINATION SPECIFICATION

In WS-Coordination specifications, three roles are described for communicating parties: Initiator, Participant and Coordinator. The entity aiming for a consensus among multiple Web Services is playing the Initiator role, the entity offering some service that needs to be coordinated during the interaction has Participant role and the Coordinator is an entity coordinating the communicating parties to achieve the consensus (Fig 3-5 shows a simple example of such a coordinator framework).

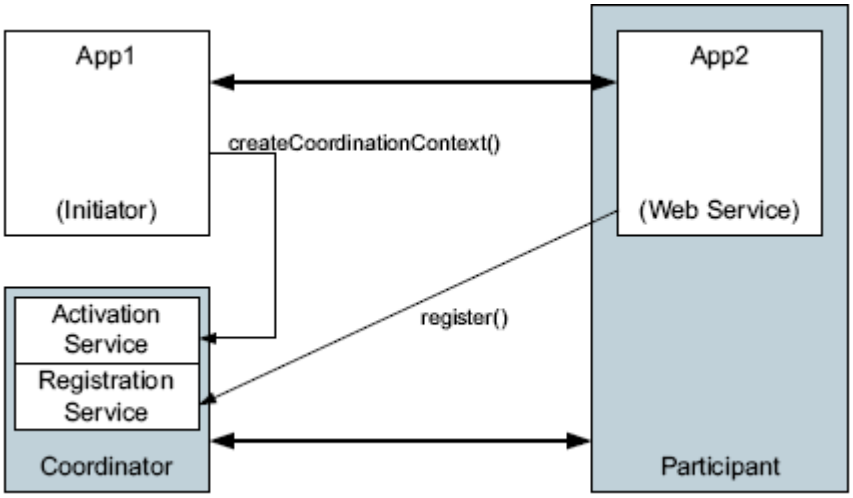

FIG. 3-5 COORDINATION ROLES AND MESSAGE EXCHANGES

The message exchanges have to be introduced (in the specification body), which requires the '*Activation*' and '*Registration*' of participants. The CoordinationContext is acquired from the Coordinator's Activation Service for the '*Activation*' phase. The CoordinationContext is

attached to business messages being exchanged between the communicating parties, embedded in a SOAP[2] header.

The aim in the registration phase is to form a logical connection between Coordinator and Participant which will be done by negotiation and exchanging endpoint addresses of the Coordinator's and Participant's protocol services (in this sense, the message flow over this logical connection depends on the coordination protocol being used and is not part of WS-Coordination specification.).

WS-AtomicTransactions and WS-BusinessActivity specification

The WS-AtomicTransactions specification uses a conventional two phase commitment protocol for coordination (Bernstein & Newcomer, 1997). In contrast, the WS-BusinessActivity specification defines two different coordination protocols; the Business Activity with Coordinator Completion (BACC), and Business Activity with Participant Completion (BAPC). Fig 3-6 shows these possible protocols for coordinating WS-BusinessActivity.

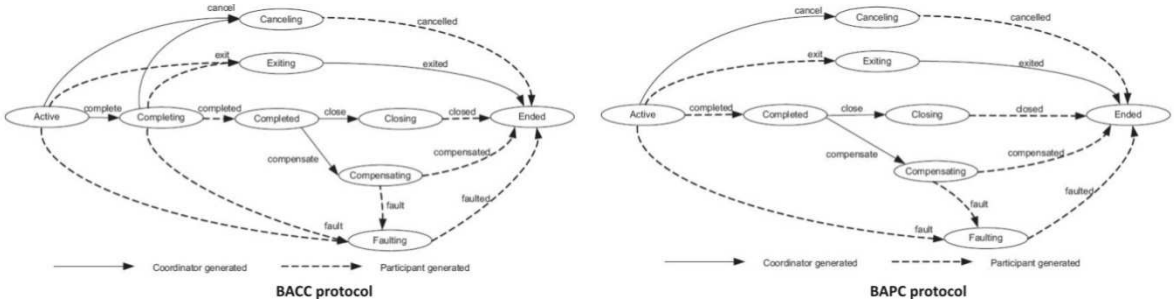

FIG. 3-6 THE POTENTIAL WS-BUSINESSACTIVITY COORDINATION PROTOCOLS

The differences between conventional 2PC and BAPC/BACC come back to the manner of their commitment; the first phase of BACC and BAPC, is used for exchange of business messages between the parties, then in case of BAPC, the end of the first phase occurs when the Completed message is sent from '*Participant*' to '*Coordinator*', indicating that the '*Participant*' has completed processing and stored all data persistently. The second phase is

[2] SOAP, originally defined as Simple Object Access Protocol, is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks. It relies on Extensible Markup Language (XML) as its message format, and usually relies on other Application Layer protocols (most notably Remote Procedure Call (RPC) and HTTP) for message negotiation and transmission. SOAP can form the foundation layer of a web services protocol stack, providing a basic messaging framework upon which web services can be built.

used for confirmation or negation of results achieved during the first phase.

The difference between BAPC and BACC is the procedures for business activities; the BAPC is designed for activities in which the decision about a transition from the first to the second phase can be made by the '*Participant*', but the BACC is designed for activities in which this decision is made by the Coordinator.

Based on analysis of Vogt and Zambrovski (Vogt et al., 2005), WS-BusinessActivity can only support "*do-compensate*" behaviour patterns (Furnis & Green, 2005) for the Participants, because it supports BACC coordination protocol (See fig 3-6 and recall 3.3.2). In the "*do-compensate*", the '*confirm state*' has priority over the temporary states. But in the other behaviour patterns (such as "*provisional-final*" and "*validate-do*" (Furnis & Green, 2005)), the participant establishes a (temporary) "*complete*" state of the application data that will be changed to the confirmed state if and when the '*Close*' message is received.

As a result of applying "*do-compensate*", WS-BusinessActivity is not able to transit to not '*Faulting*' state from '*Close*' state which means any action cannot be done on data when the system is in the close state! Forcing this limitation, by breaking the autonomy of local platforms and prescribing the internal behaviour to realisation level of services [see section 3.1.1] has despoiled one of the primary foundations of the SOA paradigm.

On the other hand, the protocol authors made some decisions about the internal build-up of communication parties as described in (Cabrera et al., 2004). The tightly-coupled Coordinator and Initiator as well as Participant encapsulating both business and transactional logic are examples of that. These presumptions are against of SOA characteristics (loose coupled and highly dynamic (Papazoglou & Georgakopoulos, 2003), (Yang et al., 2002)) particularly the web-services' nature, especially when SMEs are involved in a transaction.

The other side effect of the identical roles of '*Close*' in "*do-compensate*" pattern behaviour, is the wide limitations on alternative scenarios, which not only keep a tight rein on the coverage of business requirements, but also, by considering the highly dynamic nature of SMEs, they cannot be a candidate for the coordinator. By bearing in mind the designer assumption (tightly-coupled), necessity for stability, on one hand will force the system for choosing a few highly stable nodes as the coordinators which in the best case will produce a decentralised system (not fully distributed as the designers were promised (Cabrera et al., 2004)). On the other hand, SMEs' local autonomy has been violated and they cannot be candidates for Initiator as well as Participant roles.

3.4.2 OASIS & BTP

The Organization for the Advancement of Structured Information Standards (OASIS) has developed the Business Transaction Protocol (BTP) to provide a model for reaching the requirements of SOA design with emphasis on long-running collaborative business

applications. BTP is designed to support long-term (running) transactions and a transaction concept that goes beyond data-centric transactions (Ceponkus et al., 2002).

BTP STRUCTURE:

By using a nested transaction model (recall chapter 2) as its infrastructure, BTP has a transaction tree. In contrast to most of the transaction models, BTP uses an open-topcommit (Dalal et al., 2003) protocol (sometimes called a three phase protocol) which lets the application use an extra phase (as an intermediate phase) between phases. This ability is applied by expanding the range of available verbs "*prepare*", "*confirm*", "*cancel*", etc to include explicit control over both phases.

BTP TRANSACTIONS TYPES:

By using a tree structure BTP can mix different types of (sub) transactions. To address the specific needs of business transactions, BTP makes a separation in the categorization of (sub) transactions by introducing two types of extended transactions:

Atom: The atom transaction type uses a classic two-phase commit protocol and the transaction is considered as the "*logical unit of work*" which achieves a consistent result. In this context, a unit of work can be a number of business actions which is measured as an ACID (recall chapter 2) transaction (as an important emphasised feature in BTP (Furnis et al., 2004) should either be completed together or reversed to a state as if not executed). The all-or-nothing semantics associated with the atom support precisely this style of interaction. Where a single party cannot make good on a business level agreement, or there is a technical failure that prevents that party from providing its service, the atom abstraction ensures that all parties involved in the interaction are freed from their obligations (Dalal et al., 2003).

Cohesion: By relaxing Atomicity (in ACID properties), a '*Cohesion*' transaction tries to overcome the long-term (long-running) transaction problem. It permits the terminator of transaction to confirm or cancel the selection of work based on the business model (high-level business rules). Actually, cohesion architecture tries to provide an abstraction for multiparty B2B interactions and it has a high reliance on business model ontology. Based on this logic, cohesion (in contrast with conventional transaction model) according to business model, can provide participants with different outcomes which some might confirm while others cancel. Cohesion's customised version of two-phase protocol, from the high level specifies (in intermediate level) precisely which participants to prepare and which to cancel. As during transaction life time, it might meet conditions that allow it to cancel or prepare some of its units, although the cohesion might take several hours or days to arrive at its confirm-set (the set of vital participants that must confirm to successfully termination of the transaction). When the confirm-set is determined, the cohesion collapses down to an atom,

and this final phase all members of the confirm-set see the same outcome. In this way, the action of Cohesion is categorised as conventional nested transactional model [recall 2.2] with slightly flexible for commitment protocol but same view for the long-term transaction tree.

BTP ROLES, ACTORS AND CAST

A BTP role is specific to a particular relationship between software agents (known as BTP actors) participating in a business transaction. Two types of relationships exist between different roles in the BTP model; *inferior* and *superior*. Inferior (applicable on Atom/Cohesion) is responsible for reporting to the superior that it is prepared for the outcome, regardless of whether the associated operations' provisional effects can be confirmed or cancelled. Superior (applicable on coordinator/composer) is the party that coordinates the transaction. A superior gathers reports from its inferiors and must determine which to cancel or confirm. If the transaction is an atom, BTP's predefined rules require that every inferior be confirmed unless at least one has signalled that it wishes not to, in which case all will be cancelled. If the transaction is '*cohesion*', it will not rely on just the decision made by inferiors and it has to be a combination of the decisions made by inferiors and its own business rules which is highly dependent on the business model.

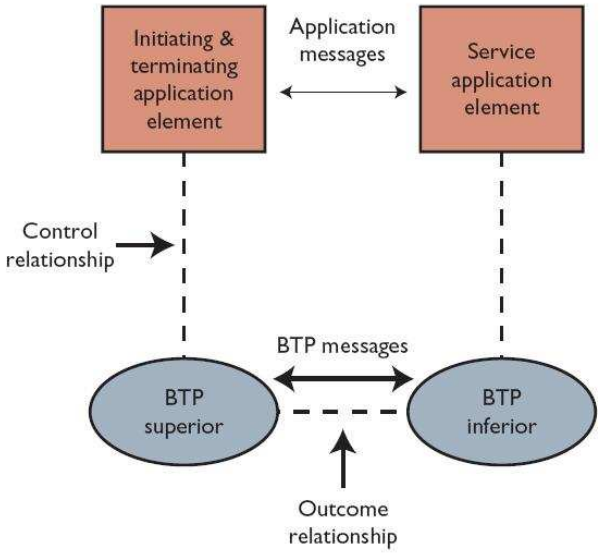

FIG. 3-7 BUSINESS TRANSACTION PROTOCOL GENERAL SCHEMA

Therefore BTP as a nested transaction model, stresses each inferior has only one superior, but a superior can have multiple inferiors. The tree of such relationships could thus be wide, deep, or both. According to transaction's relative position in the transaction tree (a superior coordinates an inferior). This is important because an actor's role could change during a

transaction and defer different behaviour for the same logical entity.

Several other actors are introduced on BTP for handling more specific roles. One of the most important actors as is superior for an atom is called a coordinator; it is responsible for supervision of the atomic two-phase completion protocol. For cohesion, the most significant superior is a composer; (a sub-coordinator (atom) or sub-composer (cohesion)). The other important actors are 'transaction factory' (which is typically a Web service that creates the context for a business transaction and manages the coordinator associated with it), 'initiator' (which starts a business transaction at an application element's request), 'participant' (which is the entity that eventually does the real transaction work on behalf of the Web service), and 'terminator' (is an application element that inculcates the coordinator of a business transaction to either confirm or cancel the transaction). In the next part, a classic simple scenario, which it is mentioned in several references, will show the main structure of the model.

SIMPLE SCENARIO:

- **Beginning a transaction**: by sending a request (for a new business transaction) from initiator to a factory (transaction factory) a transaction begins, two different situations can arise:

  - If it is a new top-level transaction, the composer or coordinator will be the decider (the term given to the most senior coordinating entity for a transaction).
  - If the new business transaction is made the direct inferior of an existing transaction (by specifying a superior transaction as its parent in the consequent begin message), the request creates a sub-composer or sub-coordinator that is ultimately controlled by its parent composer or coordinator.

The factory sends a begun message to the initiator in reply to the begin request, indicating the related context for the new business transaction.

- **Enrolling a participant in a transaction**: For interaction in the transaction, the new transaction has to register what it is done by sending an enrol request to a superior to enrol an inferior and the superior answers with an enrolled message to point out that the inferior has been successfully enrolled in the transaction.

- **Confirming a transaction-phase one**: when association phase is finished, the first phase of 2PC confirmation protocol can be initiated by the terminator. A preparation message will be sent by the superior to its enrolled inferiors that have sent neither a cancelled nor a resign message;

  - If a coordinator or composer that has been requested to confirm has only one remaining inferior in the confirm-set, it can delegate the confirm-or-cancel decision to that inferior by sending a confirm_one_phase message, which avoids performing the two-phase exchange.

- If the inferior is able to proceed with the work asked of it, it sends a prepared message to its superior — either unsolicited or in response to the superior's prepare message — but only once the inferior has determined that the operations associated with it can be confirmed or cancelled as instructed by the superior.

- **Confirming a transaction – phase two**: If the first phase of the completion is successful (that is, either all participants agree to confirm in the case of an atom, or the participant decisions plus business logic still allow confirmation for a cohesion), the terminator sends a confirm_transaction message to a decider to request confirmation of the business transaction, which then causes the confirm message to be propagated to each enrolled participant in the case of an atom, or to a subset of the enrolled participants (specified by the inferiors- list parameter of the confirm_transaction message) in the case of a cohesion. On receipt of a confirm message, an inferior sends a confirmed message once it applies the confirmation — whether in reply to confirm, after making an autonomous confirm decision, or in response to a confirm_one_phase. A decider sends a transaction_confirmed message to a terminator in reply to confirm_transaction if all the inferiors in the confirm-set are able to complete their associated work (and, for a cohesion, all other inferiors cancel).

3.4.3 BTP UNSOLVED CHALLENGES:

The first version of BTP was not specifically about transactions for web services (the intention was it can be also used in other environments) (Ceponkus et al., 2002). Actually BTP defines the transactional XML protocol and must specify all of the service dependencies within the specification. Recently Choreology Ltd. has tried to point out and modify the model and consider possible connection between BTP and WS-Tx for customising the model for web services (Furnis & Green, 2005). The reflection of these works can be seen in the latest version of BTP too (Furnis et al., 2004).

As mentioned in 3.4.2.2 BTP is a conventional nested transaction model, which means transactions can be nested and internally they can share uncommitted results but they are isolated for the transactions on the environment. That means the partial results cannot be covered in this model.

The other important point is the relationship between superior (coordinator/composer) and inferior. As there is not any specific mechanism for alternative service composition in (specifically) composers and (with less side effect) coordinators, the relationship should be tight-coupled (as it is not mentioned directly in system specification) or the whole transaction will face a regular cascading roll back danger (as a very simple example, we can check the nature of SMEs which it is dynamic with loosely coupled binding which means accessibility and characteristics of provided services can be changed regularly, which means composer/coordinator had to be roll back and in the case of composer because it is not isolated from other composers of the same transaction, it might shared some its results with them and all dependent composers had to be roll back/restart too).

Another side effect of the last problem is the probability for dead-lock. As the reaction of

the transaction to dead-lock was not mentioned (and even worse, the algorithm for detecting deadlock is not clear), the lack of consideration of this issues can be consider as an important weakness of the model. Even worse than that is that even if there was an internal structure for detection or prevention of deadlock with possibilities for regular abortions (rollback/restart) the probability for deadlock is high.

There are more issues such as recovery management algorithm, fully distributed concurrency control and so forth which are unanswered in BTP.

In the next chapter, we try to propose a model which follows the digital ecosystem principles and avoids the similar weakness of BTP and WS-x transactional model.

4 CONSISTENCY OF DISTRIBUTED TRANSACTIONS (VPTNS)

As discussed in the previous chapter, current protocols in transaction frameworks targeted at supporting business activities between networked organisations provide a centralised solution, which not only violates the primary concept of SOC (loose-coupling) but also does not cover all aspects of their business activities and can not be accepted in Digital Ecosystems environment which is loosely coupled and focuses on local autonomy of participants. The current models enforce involvement of local states of participants in their recovery model and violate their local autonomy and they create tight dependencies which are susceptible to the risk that comes with a single point of failure in the framework. In contrast, our proposed model, from the very beginning and early prototypes, advocates a fully distributed solution and relies on the P2P interactions between the platforms (here, participants) (Razavi et al., 2006), (Razavi et al., 2007a).

In this chapter, first we explore the conceptual design of such a Digital Ecosystem platform, which independently can provide SOA biding in a distributed manner, then will focus on the transaction specification of such a platform. The formal analysis of concurrency is the next section. Section 4 of this chapter provides details of necessary constraints and their proof. In the rest of the chapter we explain the provided structure for doing so.

4.1 GENERAL INFRASTRUCTURE OF EACH PLATFORM

As the kernel of each platform, we have designed a software agent which is responsible for coordinating the participant's business activities (transactions). This local agent also archives the information related to these activities (corresponding VPTNs) and improves the general connectivity of the network (its digital ecosystem), and in doing so it contributes to the so-called *network growth*(Razavi et al., 2007a). This is an important aspect when it comes to sustainability, especially in a fully distributed solution. This leads up to the primary requirement of a digital ecosystem (recall chapter 3) which is represented in Fig. 3-2 highlighting the fact that there is no centralised point of command and control in a digital ecosystem.

4.1.1 SOA AND DIGITAL ECOSYSTEM PLATFORM

As we discussed (recall chapter 3), SOA is the tangible architectural approach of SOC and is mainly applicable when multiple applications running on varied technologies and platforms need to communicate with each other. In this manner, enterprises can mix and match services to perform business transactions with minimal programming effort. SOA is a way of reorganizing software applications and support infrastructure into an interconnected set of services, each accessible through standard interfaces and messaging protocols. The basic SOA is not only about architecture of services, it is a relationship of three kinds of

participants: the *service provider*, the *service discoveror*, and the *service requestor* (*client*). The interactions involve '*publish'*, '*find'* and '*bind'* operations.(Papazoglou, 2003).

In a typical service-based scenario a service provider hosts a network-accessible software module (an implementation of a given service). The service provider defines a service description of the service and publishes it to a client or service discovery agency through which a service description is published and made discoverable. The service requestor uses a find operation to retrieve the service description typically from a discovery agency, i.e., a registry or repository like UDDI[3], and uses the service description to bind with the service provider and invoke the service or interact with service implementation (Papazoglou et al., 2006).

As Digital Ecosystems insist on local autonomy of each participant, the theoretical design of each platform should rely on an independent platform. Conceptually we consider this platform, as a local agent (Razavi et al., 2007b). The primary needs of this agent are the find, publish and bind operations. For this reason, it requires a '*Global repository*' (which keeps the information about other participants' web services), '*Web service promoter*' for publicising its web services for other participants (it can be kept in their Global repository), '*Web Service Information Investor*' (for updating its Global Repository, according to the other Participants' local web services). Fig 4-1 shows such a model.

[3] Universal Description, Discovery and Integration (UDDI) is a platform-independent, Extensible Markup Language (XML)-based registry for businesses worldwide to list themselves on the Internet.

General infrastructure of each platform

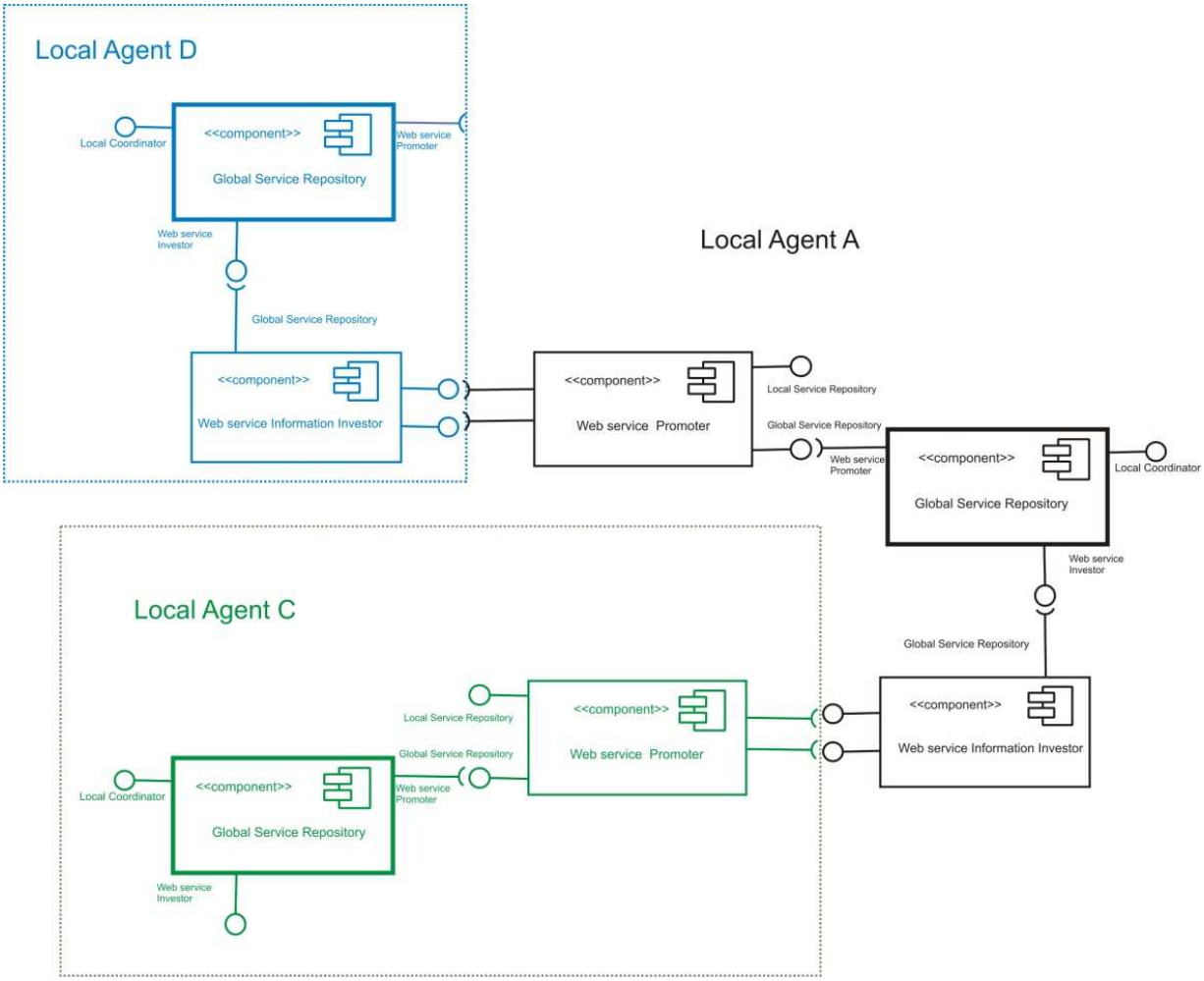

FIG. 4-1 REPOSITORY IN DIGITAL ECOSYSTEM

LOCAL WEB SERVICES INFORMER

Digital Ecosystems are inherently operated in a heterogeneous environment which is service-oriented. On one hand, this environment is supposed to support any type of web service, with any protocol in a loosely-coupled manner (local autonomy for SMEs), and on the other hand, it should support a proper commit protocol for long-running transactions (business activities as discussed in chapter 3 and (Razavi et al., 2007a)). In order to provide the Initiator of a transaction with the basic view about the web service which is to be used on its transaction, as well as the limitations / restrictions of the particular web service, we need to gather information about the web service. This information can be provided by each web service after its creation in some description language such as WSDL and/or can be provided manually by the SME which provides the web service.

Furthermore SMEs may change their web service protocol, parameters, etc, regularly and therefore the possibility for updating this information is necessary too. As a result, we need to provide two interfaces for keeping this information in the local agent as the component also requires an interface to the local repository (Figure 4-2 shows this component of our

local agent).

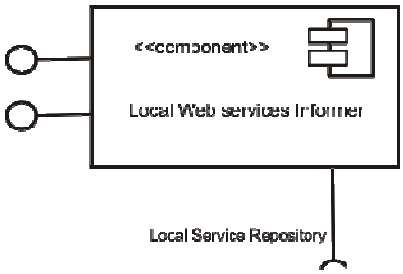

FIG. 4-2 LOCAL WEB SERVICES INFORMER

LOCAL SERVICE REPOSITORY

The Local Service Repository keeps information about each local web services in the platform (SME). This information is some description of each web service (for example it can be a SDL or WSDL document) and any extra information such as availability, last updates and etc (which may help other SMEs to have a clearer picture of that particular web service), can be included too. In the first place (as a component-based approach), the Local Service Repository should provide an interface to the Local Web Services Informer, e.g. for accessing the local web service description records. The next interface provided by the Local Service Repository gives access to the Local coordinator to use the web service descriptions for creating and running a transaction. Any updates, modifications or even the creation of web services should be promoted (at least for other partners with whom they are collaborating while running a transaction). That is the main reason the Local Service Repository requires an interface to another component, namely the Web Service Promoter, whose purpose is to promote the web services for other agents (Fig 4-3).

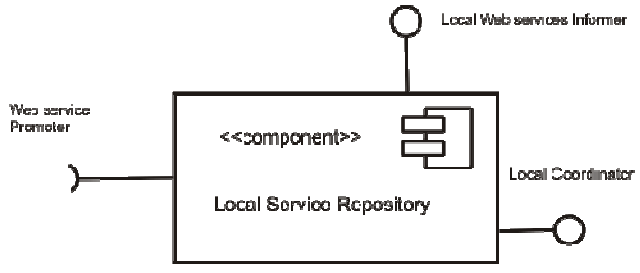

FIG. 4-3 LOCAL SERVICE REPOSITORY

WEB SERVICE INFORMATION INVESTOR

The structure of the local agent we considered in Figure 4.1 looks symmetric for both the local and the global view towards web services. Therefore the Web Service Information Investor, as a symmetric component for the Web Service Informer, does a similar job but this time for global web services. It *provides* two interfaces for creating a new web services record and updating the current web services. In addition, it *requires* an interface to the Global Service Repository (the symmetric component for Local Service repository (Fig 4-4)).

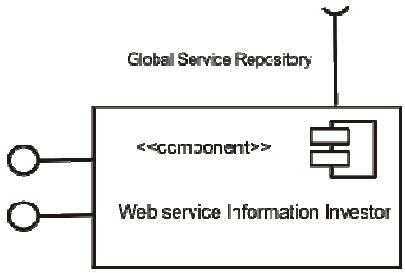

FIG. 4-4 WEB SERVICE INFORMATION INVESTOR

GLOBAL WEB SERVICE REPOSITORY

As with the Local Web service Repository, the Global service repository provides two interfaces: one for the local coordinator to access the web services record descriptions and the other one for the web services Investor to make changes on the Global Service Repository. The first interface plays a critical role for the local coordinator in making the decision about the protocol and the method for applying it on the transaction model. At the other side of the local agent is another SME which may change its web services descriptions regularly and even service availability can be an issue too. The second interface is important too, as updating the Global Service Repository is crucial.

On the other hand, the Global Service Repository should be able to inform the Web service promoter, as soon as any changes happen for its records. That is why it requires an interface to the Web service Promoter for doing that. Fig 4-5 shows this component.

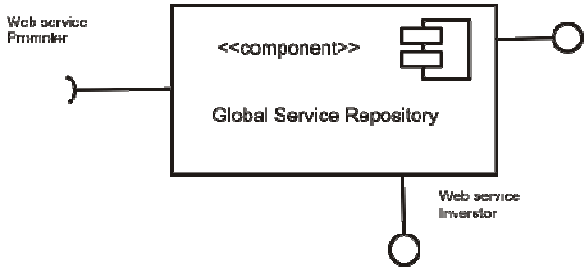

FIG. 4-5 GLOBAL SERVICE REPOSITORY

WEB SERVICES PROMOTER

The Web service Promoter is an important part of the local agent, as it reflects the situation of the web services of a local agent and the web services of any other connected agents to that particular agent. This can be done by using two interfaces which are provided for Local Service Repository and Global Service Repository respectively. Meanwhile, the Web service Promoter requires two interfaces from the other agent to inform the latest situation of its local web services and any other web services which are communicating with it (Figure 4-6).

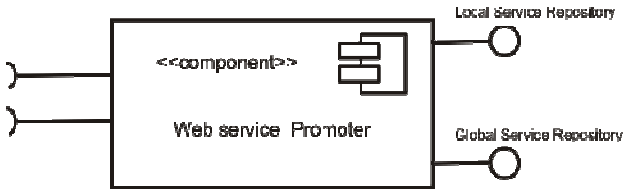

FIG. 4-6 WEB SERVICE PROMOTER

In fact two interfaces for this component should be provided by the Web service Information investor of the other agent. Figure 4-7 shows this connection between Local agent A and Local agent B. When any changes happen for some records of the local or the global service repository, they use the Web service Promoter's interfaces. The Web service Promoter in turn can use the interfaces provided by the Web service Information Investor in Local agent B, and the Web service Information Investor at agent B can update its Global Service Repository if needed (because in some cases it could be done already). As a result, the Global Service Repository of agent B will use the same interfaces for the Web service Promoter at agent B and this will be done for any connected agent to Local agent B. In this way, any changes on connected agents can be updated quickly.

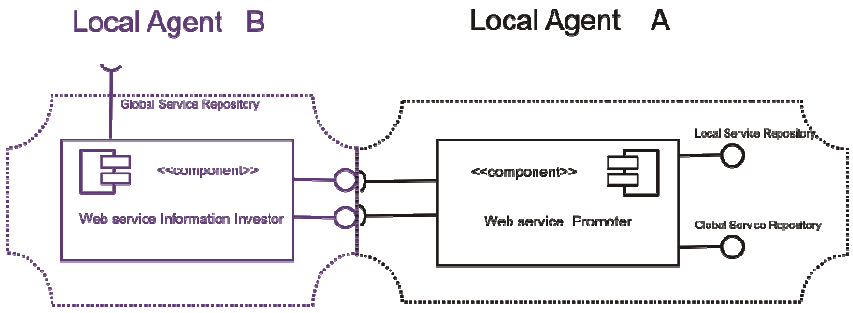

FIG. 4-7 WEB SERVICE PROMOTER ROLE BETWEEN TWO DIFFERENT AGENTS

LOCAL COORDINATOR

The kernel of the local agent is the local coordinator. Other components provide information for a local coordinator (on the local machine or even for a remote agent). The local coordinator facilitates our transaction model to be applied for complicated business activities (long-running transactions) as well as simple transactions. Generally, the Local coordinator requires an interface from the Local Service Repository for gathering the information about local web services which enables it to provide a recoverable and consistent mechanism for deploying transactions. This normally can be handled by a transaction context in response to a transaction request. Later in this chapter we will cover this (4-2).

For communicating with another agent (its local coordinator), the local coordinator as well as providing an interface, requires an interface from the remote agent too. The Local coordinator also requires an interface from the Global Service Repository, especially when it acts as an Initiator of the transaction. This makes it possible to create the transaction script based on the knowledge of the other agents' web services. In the end, it requires the interface from its local web services to able to invoke them (see Fig 4-8).

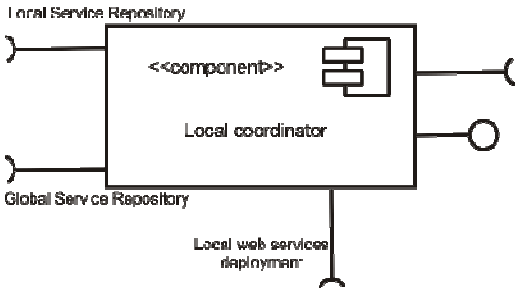

FIG. 4-8 LOCAL COORDINATOR

4.1.2 TRANSACTIONS AND COORDINATION IN DIGITAL ECOSYSTEM

Fig. 4-9 shows the conceptual structure of the local software agent of each participant. The '*local coordinator*' component supposes to coordinate the service requests to and from the local platform. In other words, it deploys services of the platform, coordinates the transactions and archives their information in a repository, which can be considered as the '*local service repository*'. In this way, all participants of a transaction will keep the archived information of the transaction.

General infrastructure of each platform

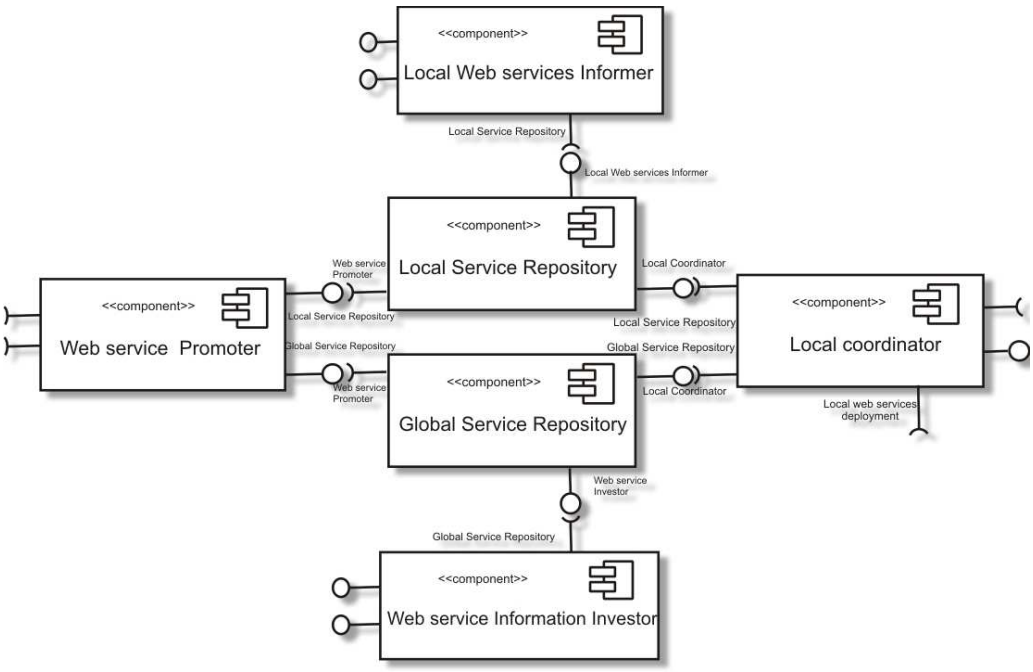

FIG. 4-9 THE STRUCTURE OF PARTICIPANT'S SOFTWARE AGENT

The '*local web service informer*' component updates any changes of local services in the '*local service repository*' and relevant participants can be notified of the changes through the '*web service promoter*'. The links to other participants will be kept in the '*global service repository*'. Note that at this stage, participants of different VPTNs are connected to each other (in (Razavi et al., 2007a) this is called the *birth stage* of the underlying network). In order to reduce the possibility of failures (discussed in further detail in chapter 5) and increase the network stability (chapter 7), the network connectivity, i.e. the number of links to other participants, may change. These changes will be done by two components; the '*web service information investor*', for updating new links to the global repository and the '*web service promoter*', for promoting new links to other participants (in (Razavi et al., 2007b), (Razavi et al., 2007a) this is referred to as the *growth stage*).

By using two repositories (Local and Global service repositories) the local agent tries to provide detailed information about local web services, but also general information about other web services. This enables the Local coordinator to invoke its local web services based on different protocols and on the other hand, by using general information about other (remote) web services, in some sort of XML description, such as WSDL or SDL, it can create the transaction context (will be discussed in the next section). The Local Service Repository should be updated by the Local Web Services Informer (any changes or updates can be effected on Local Service Repository). Meanwhile the Local Service Repository can promote its services to other agents through the Web Service Promoter.

The Global Service Repository can be updated by the Web service Information Investor and at the same time, can promote these web services (which are stored in Global Service Repository) to the other agents (any changes will be promoted too, and in this way other

agents can update their Global Service Repository). Fig 4-10 shows how communications between components of agents can improve the performance, can keep all agents' repositories updated and can provide enough information for the Local Coordinator of agents to run transactions.

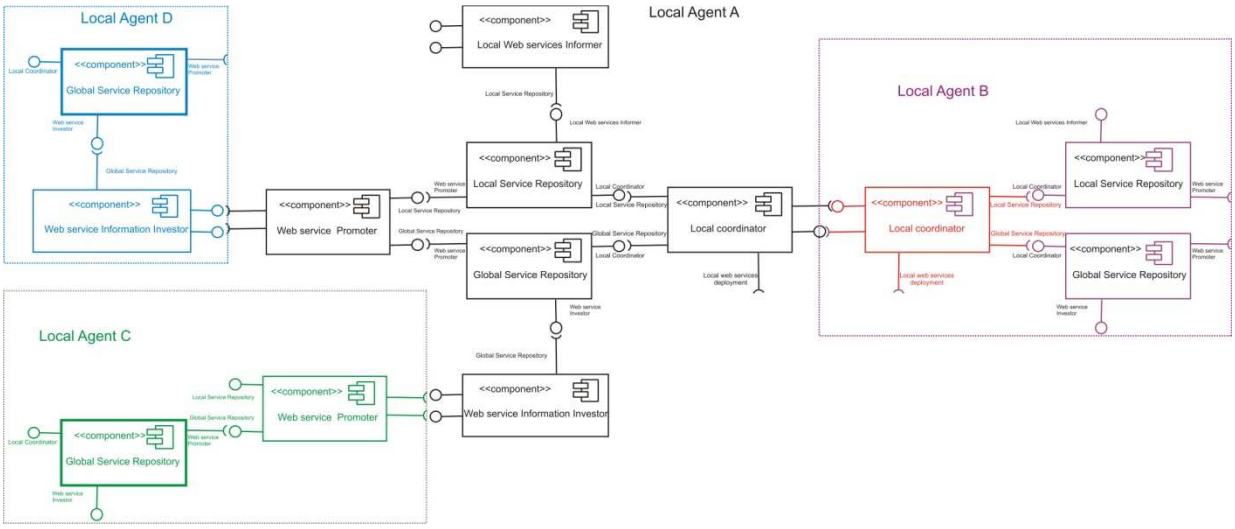

FIG. 4-10 OVERVIEW OF MULTI-AGENT ENVIRONMENT[4]

4.2 TRANSACTION CONTEXT

A transaction can be modelled as a nested structure of sub-transactions (Moss, 1985). These can be composite based on data or order (each sub-transaction acts like a service in the service composition framework discussion (Chapter 3, section 1; 3.1.1) and more details can be found in (Yang et al., 2002)). In that way, the lowest part will be services in each participant (the local coordinator shown in Fig. 4-8, can deploy them through one of its interfaces). Fig 4-11 shows such a structure where we have five services, which have been combined by different service compositions. The notation symbols used here are based on (Haghjoo & Papazoglou, 1992) and '*Seq*' is for Sequential, '*Par*' for Parallel and '*Alt*' for alternative service compositions.

This is called the *transaction context* and it is used to clarify the semantics of a transaction. We have described this semantics in (Razavi et al., 2007b) and (Moschoyiannis et al., 2008), also advocated a methodology for optimising this, in terms of behavioural properties of the interactions involved, in our previous works (Moschoyiannis, Razavi, & Krause, 2008). Meanwhile the details of this context, including the xml presentation of this example (given in Fig. 4-2), the xml schema for creating the tree and other infrastructural schemas relevant to this paper can be downloaded from (Razavi, 2009). Furthermore, the

[4] The image can be found in (Razavi et al., 2007c) and the main agent diagram is in Fig 4-9

Java parser for the model components is also available through the same link.

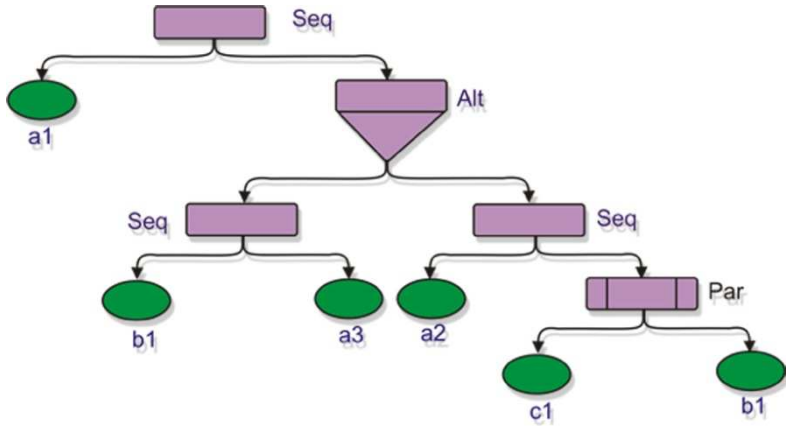

FIG. 4-11 TRANSACTION TREE

As we have seen in chapter 3, in terms of service composition, it is possible to have different levels of data-dependency between sub-transactions of a transaction – what is referred to as a *long-running* transaction (Moss, 1985), (Singh & Huhns, 2005). In terms of concurrent execution of sub-transactions of a transaction, this may cause data-inconsistency which is considered as a violation of one of the very first requirements of a transaction (recall chapter 2 in ACID, which has been described in section 1). This is the first challenge for a digital ecosystem, which will be addressed after explaining the transaction context by an example. The remaining two challenges, Recoverability and Durability will be addressed in chapter 5 and chapter 7, analysing the distributed pattern behaviour is explained in chapter 6.

As the first step, we look at a transaction (to be precise, a long-running transaction) as a set of sub-transactions, which may use each other's objects (data-items):

$$T = \{ST_1, ST_2, \cdots, ST_n\}$$

4.2.1 SIMPLE SEQUENTIAL SCENARIO

We start with a simple scenario and explore the structure with more details; the transaction context has been defined, as a nested transaction model, which has been presented as a recursive complex type in the xml schema. Figure 4-12 shows a simple sequential travel scenario where a travel agent tries to book the flight with an airline (here BA) and book a hotel room.

Transaction context

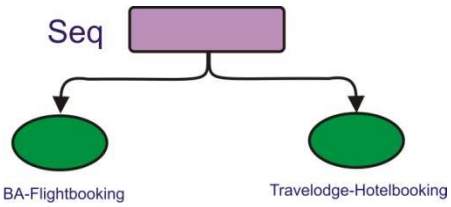

FIG. 4-12 SIMPLE SEQUENTIAL SCENARIO (TRANSACTION TREE)

Fig 4-13 shows the transaction context xml of the transaction tree of the example in Fig 4-12. The transaction 0001, includes three sub-transactions, where the main sub-transaction (010) is a sequential service composition of sub-transaction 011 and sub-transaction 012.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Document   : Transaction Context of a Simple Travel scenario
    Created on : 16 Feb 2009, 05:07
    Author     : A. Razavi
    Description: Sample scenario
-->
<TransactionTree TransactionId="0001"
xmlns="http://xml.netbeans.org/schema/TrnTree"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xml.netbeans.org/schema/TrnTree TranTree.xsd">
<SubTransaction SubTransactionId="010">
    <Composition CompositionType="Sequential">
        <SubTransaction SubTransactionId="011">
            <WebService ServiceId="FlightBooking">
                <ServiceDescription>
                    BA FlightBooking Service Description
                </ServiceDescription>
            </WebService>
        </SubTransaction>
        <SubTransaction SubTransactionId="012">
            <WebService ServiceId="HotelBooking">
                <ServiceDescription>
                    Travelodge HotelBooking Service Description
                </ServiceDescription>
            </WebService>
        </SubTransaction>
    </Composition>
</SubTransaction>
</TransactionTree>
```

Sub Transaction 011

Sub Transaction 012

Sub Transaction 010

FIG. 4-13 TRANSACTION CONTEXT FOR FIG 4-3

The actual execution of the transaction relies on the order of execution and this should clarify the distributed coordination of the execution. Fig 4-14 shows the sequence diagram of the transaction tree presented in Fig 4-12.

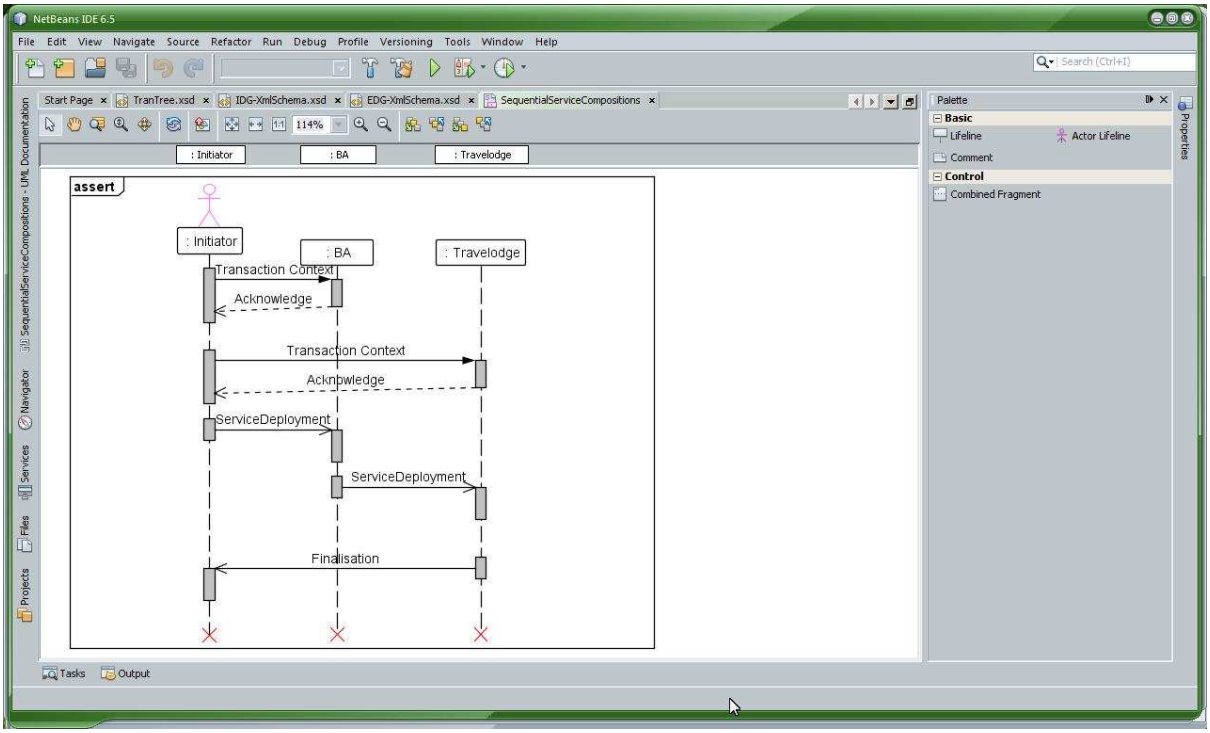

FIG. 4-14 SEQUENTIAL SERVICE COORDINATION WITH TWO PARTICIPANTS

Conceptually, the transaction initiator should have the address and other specification (such as their service descriptions, etc) of other participants in its repositories (Global and local service repository in Fig 4-9). Meanwhile the initiator's coordinator has interfaces for connecting to the other participants' coordinators (Fig 4-15).

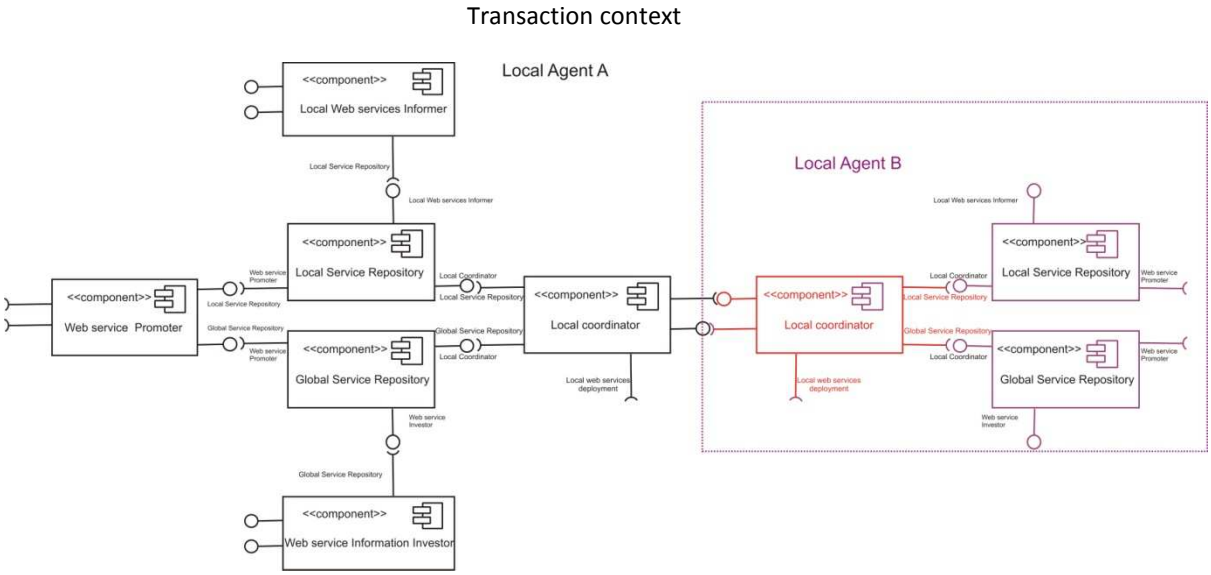

FIG. 4-15 LOCAL COORDINATOR INTERFACES WITH OTHER PARTICIPANTS COORDINATORS[5]

In Fig 4-16 the Travel Agency (Initiator), sends the transaction context to the other participants of transaction (BA and Travelodge), their response to the initiator is the main trigger of the transaction, and the initiator can send the deployment service of the first service through the first participant's coordinator which it is BA coordinator. The BA coordinator will deploy the flight service and will get the result (Fig 4-16).

[5] The image can be found in (Razavi et al., 2007c) and the main agent diagram is in Fig 4-9

Transaction context

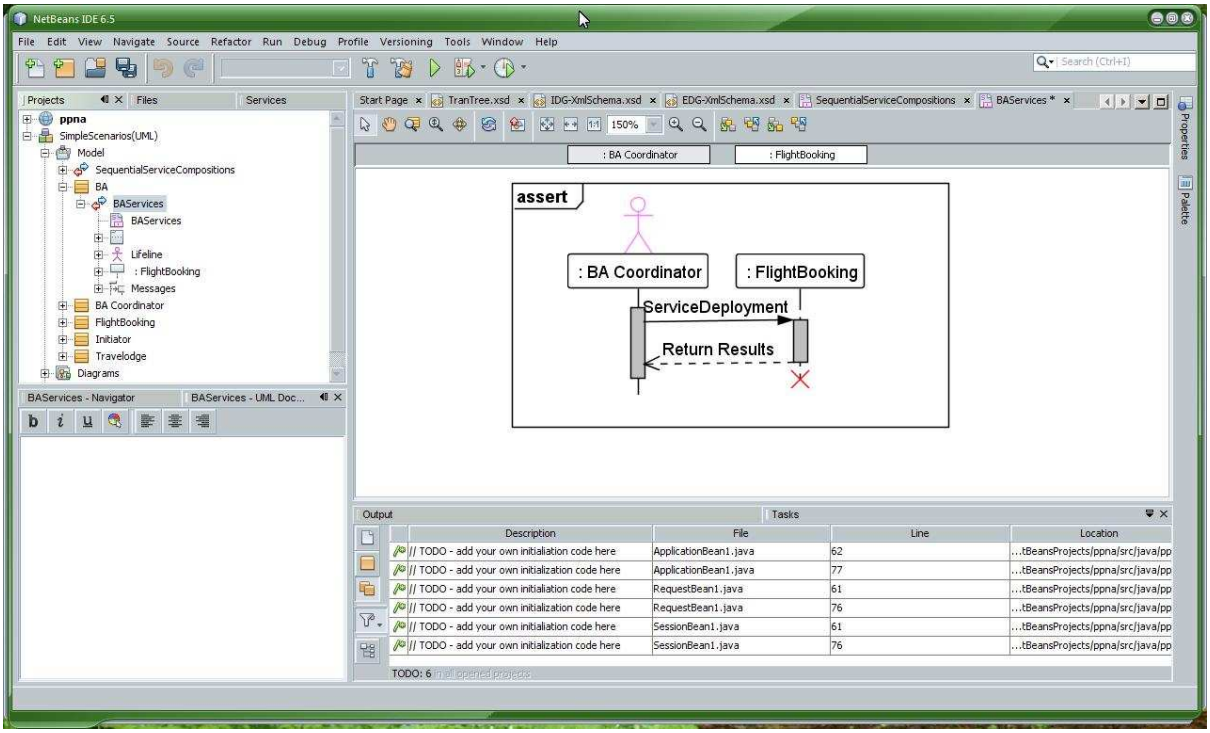

FIG. 4-16 BA DEPLOYS THE FLIGHTBOOKING SERVICE

After BA deployed the FlightBooking service, based on the transaction context (Fig 4-11), it sends the service deployment message (with results and proper specifications) to the next participants' coordinator (Travelodge in this example). Fig 4-14, shows this message exchange. When the Travelodge coordinator receives the deployment message, it will deploy the hotel booking service and will get the result (Fig. 4-17).

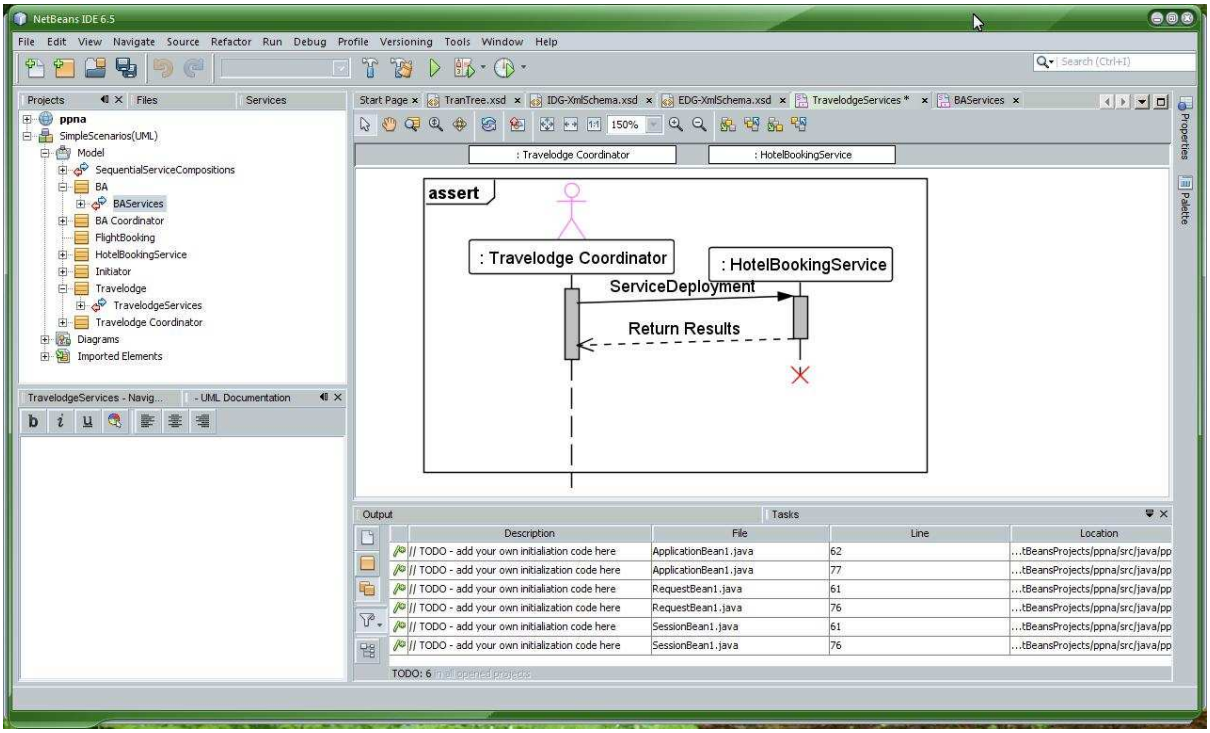

FIG. 4-17 TRAVELODGE DEPLOYS THE HOTELBOOKINGSERVICE

4.2.2 THE CONTEXT SCHEMA

The transaction context derives from an xml '*transaction context schema*'. Fig 4-18 shows the context schema. Each transaction has a unique id as assigned by the transaction initiator. The transaction context schema presents a transaction tree where the root contains the main sub-transaction. A sub-transaction can be a simple web-service, delegation type, composition or data-composition.

Transaction context

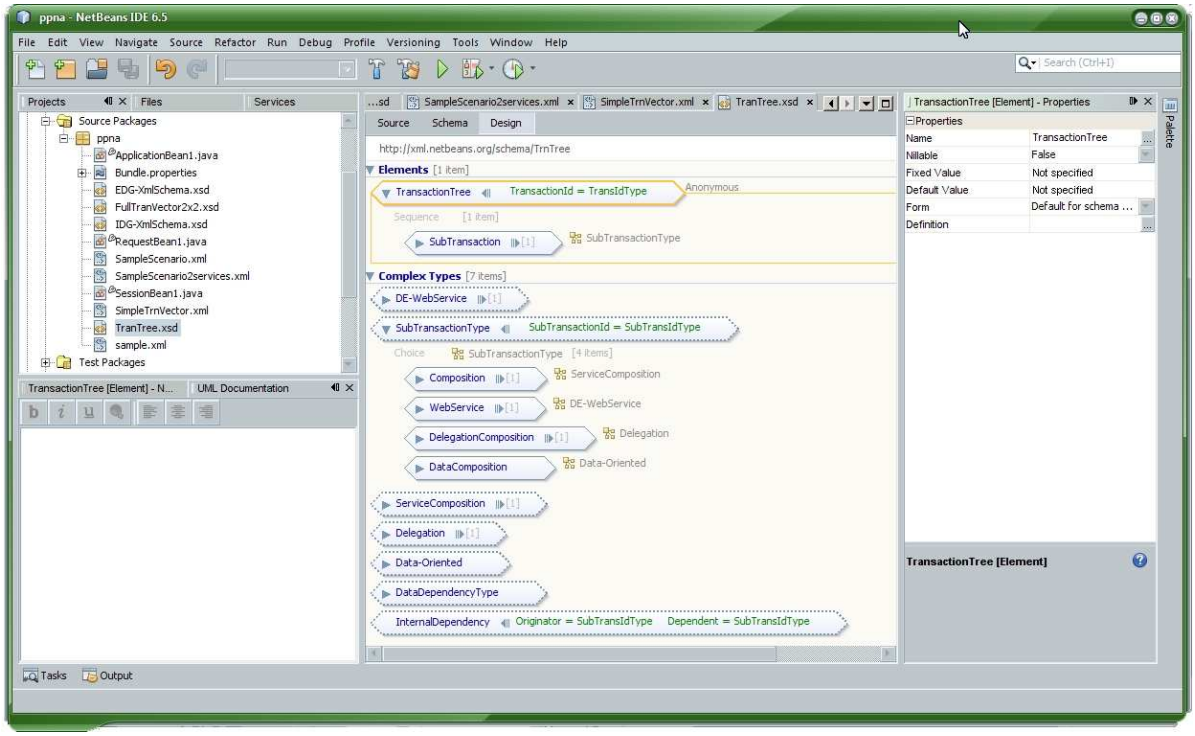

FIG. 4-18 TRANSACTION CONTEXT SCHEMA

For presenting a nested structure of sub-transactions, we use a recursive structure which relies on a complex type that is called '*SubTransactionType*'. Fig 4-19 shows the structure of SubTransactionType. As has been shown in the figure, a SubTransactionType includes a choice between four different elements '*Composition*', '*WebService*', '*DelegationComposition*' and '*DataComposition*'. Composition type is '*ServiceComposition*' which clarifies different compositions between services and sub-transactions of a transaction. WebService type is '*DE-WebService*' that shows the structure of services in a digital ecosystem. The DelegationComposition type is '*Delegation*' which can include any standard transaction structure, without any full check (this can leave us open for designing a converter between our transaction models and other transaction models standard in SOA). DataComposition type is '*data-oriented*' and determines different types of data dependencies in a transaction.

SubTransactionType has an attribute for identifying it inside the transaction, which means this id is unique inside of the transaction (each sub-transaction will have a unique identity between other sub-transactions of the same transaction). As Delegation and DE-WebService type should have a straightforward structure, we focus on different compositions and the possible complexity of data dependencies.

Transaction context

```
<xsd:complexType name="SubTransactionType">
    <xsd:choice>
        <xsd:element name="Composition" type="tns:ServiceComposition"/>
        <xsd:element name="WebService" type="tns:DE-WebService"/>
        <xsd:element name="DelegationComposition" type="tns:Delegation"/>
        <xsd:element name="DataComposition" type="tns:Data-Oriented"/>
    </xsd:choice>
    <xsd:attribute name="SubTransactionId"
    type="tns:SubTransIdType" use="required"/>
</xsd:complexType>
```

FIG. 4-19 SUB-TRANSACTION TYPE

SERVICE COMPOSITION

Fig 4-20 shows the structure of the '*ServiceComposition*' type. It includes two elements and an attribute. The '*SubTransaction*' element provides the recursive structure which can offer composition between any number of sub-transactions. The data dependency between sub-transactions is clarified by the '*DataDependency*' element and composition type has been shown by '*CompositionTypes*' element.

```
<xsd:complexType name="ServiceComposition">
    <xsd:sequence>
        <xsd:element name="SubTransaction" maxOccurs="unbounded" type="tns:SubTransactionType" minOccurs="2"/>
        <xsd:element name="DataDependencies" type="tns:DataDependencyType" maxOccurs="unbounded" minOccurs="0"></xsd:element>
    </xsd:sequence>
    <xsd:attribute name="CompositionType"
    type="tns:CompositionTypes" use="required"/>
</xsd:complexType>
```

FIG. 4-20 SERVICE COMPOSITION

Fig 4-21 shows the structure of '*CompositionTypes*' which clarifies the possible composition types. There are three different composition types; '*Sequential*', '*Parallel*' and '*Alternative*'. This covers the requirements of order-based service composition in the service oriented computing paradigm (3.1.2). In Sequential composition, sub-transactions are deployed in a sequential order (one after the other). In Parallel composition, sub-transactions are executed concurrently. And in term of alternative composition, just one transaction between all sub-transactions will be executed and the rest will wait for the result. If that sub-transaction is not successful, the others may have a chance for execution. When there is data dependency between sub-transactions of these simple service compositions, the complexity of the transaction can increase dramatically.

Transaction context

```
<xsd:simpleType name="CompositionTypes">
    <xsd:restriction>
        <xsd:simpleType>
            <xsd:restriction xmlns:xsd="http://www.w3.org/2001/XMLSchema" base="xsd:string">
                <xsd:enumeration value="Sequential">
                    <xsd:annotation>
                        <xsd:documentation>Sequential Service Composition</xsd:documentation>
                    </xsd:annotation>
                </xsd:enumeration>
                <xsd:enumeration value="Parallel">
                    <xsd:annotation>
                        <xsd:documentation>Parallel Service Composition</xsd:documentation>
                    </xsd:annotation>
                </xsd:enumeration>
                <xsd:enumeration value="Alternative">
                    <xsd:annotation>
                        <xsd:documentation>Alternative Service Composition</xsd:documentation>
                    </xsd:annotation>
                </xsd:enumeration>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:restriction>
</xsd:simpleType>
```

FIG. 4-21 TYPE OF COMPOSITION

DATA ORIENTED COMPOSITION

In terms of data dependency between sub-transactions, we have to deal with two different dependencies. Firstly, between sub-transactions of the same transaction and the second category is about passing data item(s) between sub-transactions of different transactions. When a sub-transaction shares a data-item with another sub-transaction of the same transaction; this dependency is called '*Internal Dependency*'. And if two sub-transactions from different transactions share a data-item, we call the dependency '*External Dependency*'.

Fig 4-22 shows the structure of 'Data-Oriented' Complex type, which clarifies dependencies between sub-transactions of a composition. The first element has been used for simple description of the dependency and the second element is a '*DataDependencyType*', which can be either an internal dependency between sub-transactions or an external dependency.

In the case of internal dependency, for identifying the involved parties, the id of the originator and dependent sub-transactions is sufficient. But for external dependency, transactions' and sub-transactions' id of originator and dependent data items is needed.

FIG. 4-22 DATA ORIENTED AND DEPENDENCIES

4.3 CONCURRENCY AND DATA INCONSISTENCY

One of the most important challenges is keeping the data consistency between sub-transactions (ST_i) when there is data-dependency. As we have shown in the previous section (4.2.2), in a digital ecosystem (as a service-oriented architecture), sharing data-items between sub-transactions is possible and can be done in terms of data dependency. This will bring a huge complexity which may not be solvable by using conventional transactional models (recall chapter 3) and may need some dramatic changes and application of the conventional theorems in a different way and inside of a long-running transaction. In this section we focus on a theoretical explanation of the problem.

4.3.1 ISOLATION; STATIC AND DYNAMIC ALLOCATION

The classic view of isolation considers the transaction in terms of inputs and outputs (Bernstein et al., 1987), (Gray & Reuter, 1993). In our approach, this means that sub-transactions have read (input) and write (output) operations. Write operations are understood as operations that affect the state of resources. This means it appropriate to consider read operations as inputs for sub-transactions and write operations as their outputs. Then isolation between two sub-transactions can be expressed as:

$$\boldsymbol{O_i} \cap \left(\boldsymbol{I_j} \cup \boldsymbol{O_j}\right) = \text{for all} \boldsymbol{i} \neq \boldsymbol{j} \quad \text{EQ. 1}$$

Let I_i be the set of objects read by sub-transactionST_i (its inputs), and O_i be the set of objects written upon by the sub-transaction ST_i (its outputs). Based on EQ.1, the set of sub-transactions $\{ST_i\}$, for all *i*, when their outputs are disjoint from one another's inputs and outputs, they can run in parallel with no concurrency anomalies. Hence, by applying EQ.1 any sub-transaction scheduler can work.

Conventionally, for applying EQ.1 each sub-transaction should declare its input-output set and then a scheduler is able to compare the new sub-transaction's need to all running sub-transactions and in case of a conflict, the initiation of the new sub-transaction would be

delayed until the conflicting sub-transactions complete. This approach is called '*Static allocation*'. It has been argued, however, that the computational complexity of analysing the inputs and outputs before running transactions can cause a bottleneck on scalability (Elmagarmid, 1992), (Gray & Reuter, 1993).

The '*Dynamic allocation*' scheme has been introduced as an alternative approach. Under the prism of dynamic allocations sub-transactions can be viewed as sequences of operations on deployed objects. A particular object is subject to one operation at a time. Each operation of a sub-transaction is either *read* (using the object as the input for service deployment/composition) or *write* (over-writing/updating the object as the output of a service deployment/composition).

4.3.2 CONCURRENCY CHALLENGES

Objects go through a sequence of versions as they are updated by *write* operations. In contrast, *read* operations do not change the object version. If a sub-transaction *read*s an object, the sub-transaction depends on that object version. If the sub-transaction *write*s an object, the resulting object version depends on that sub-transaction. When a sub-transaction aborts and goes through the undo logic, all its *write* operations must be undone. This results in the object getting a new version, thus the 'undo' looks like an ordinary new update.

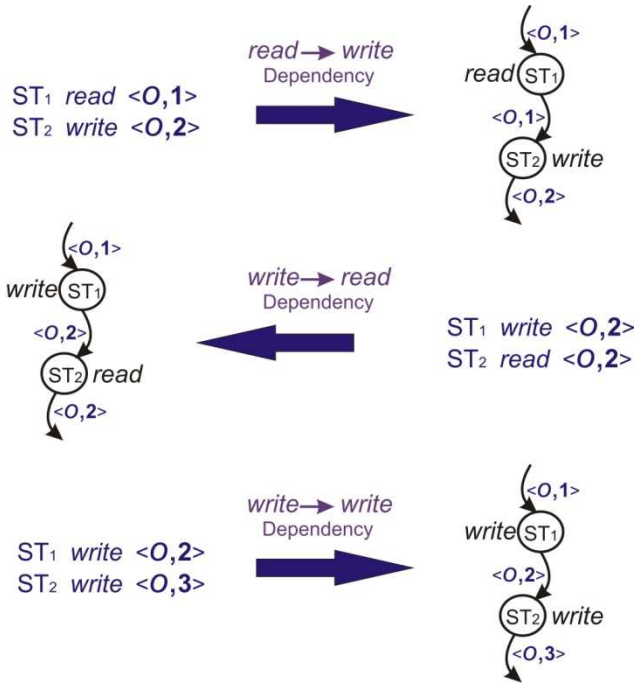

FIG. 4-23 DEPENDENCY GRAPH

Theoretically, a dependency graph can be read as a time sequence. In Fig 4-23, an edge from sub-transaction ST_1 to ST_2 indicates that ST_1 accesses an object later accessed by ST_2,

and at least one of these access operations created a new version. In that sense, ST_1 ran before ST_2. In a purely sequential execution of the transaction - running ST_1 to completion, and only then running ST_2 to completion - all dependency arrows will point from ST_1 to ST_2. However, as the execution of the transaction depends on the semantics (transaction context), there can be different composition types; in parallel execution, the dependency arrows can form an arbitrary graph. This brings about the issue of *cyclic dependencies*, which should be avoided as they can give rise to concurrency anomalies.

The main conclusion of applying the transactional properties is that any dependency graph without cycles implies an isolated execution of the corresponding sub-transactions. So if the dependency graph does have cycles, then the sub-transactions were not executed in isolation. If the dependency graph has no cycles, then the sub-transactions can be topologically sorted to make an equivalent execution history in which each sub-transaction can be ran serially, one completing before the next began. This result in conventional transactions has been given in (Elmagarmid, 1992), (Gray & Reuter, 1993) and implies that each sub-transaction ran in isolation, as if there was no concurrency. It also implies that there were no concurrency anomalies.

It is not difficult to see that violation of the isolation property is related to the various dependency cycles. Similarly to conventional transactions, cyclic dependencies in long-running transactions are categorised into three generic forms:

Lost updates: The first sub-transaction's *write* (deploying the data) is overwritten by the second sub-transaction which uses *write* based on the initial value of the object.

In Fig 4-24, we show the conflict of these *write*s. The sub-transaction ST_2 tries to update the object *o*, based on the previous version of the object (denoted by 1 in the figure), while sub-transaction ST_1 is updating the object based on the same object version (1). This means that ST_2 may update the object without considering the sub-transaction ST_1. One of the updates will be overwritten without being taken into account. This is referred to as *lost update*.

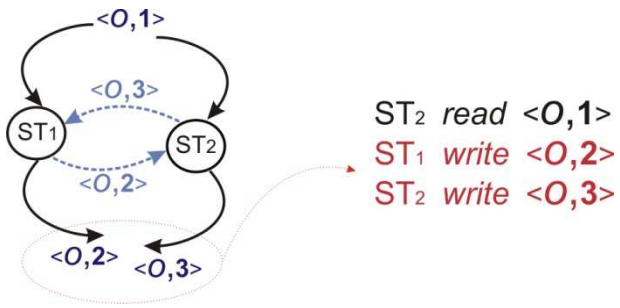

FIG. 4-24 LOST UPDATE

Since ST_1 and ST_2 are both using the object o, each write creates a dependency on the other sub-transaction, denoted by the light-blue dashes in Figure 4-24 These two *writes*

create a cycle of dependencies which results in an inconsistent state for the long-running transaction as a whole.

Dirty read: A sub-transaction *read*s an object which has been *written* before by another sub-transaction which also *writes* to it after the *read* action. This means the first sub-transaction may find inconsistency in the object. This is equivalent with the Phantom Problem in DBMS (Date, 2003), (Bernstein et al., 1987) – in short, this refers to the case where a transaction (T) can read changes made to an object during an ongoing transaction, so the object can be changed further while the transaction (T) is in progress, and as a result the transaction (T) is vulnerable to be accessing inconsistent data.

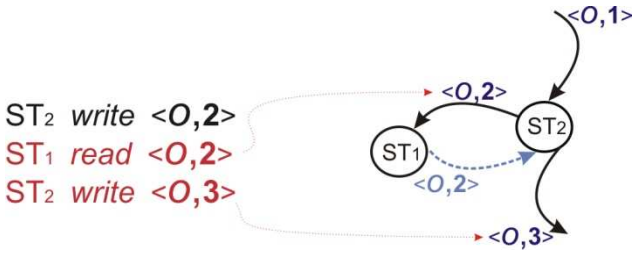

FIG. 4-25 DIRTY READ

Figure 4-25 shows sub-transaction ST_1 accessing the object *o*, which is under ongoing changes by sub-transaction ST_2. Therefore ST_1 may *read* inconsistent data from the object *o*, which in this example is the temporary version of the object (version 2), which is supposed to be finalised to version 3 only before sub-transaction ST_2 commits (or aborts). In term of dependencies, when ST_1*read*s the version 2 of object o it creates a dependency with ST_2, denoted by the dashed line in Figure 4-25, but when ST_2 writes on the object the opposite dependency is created, denoted by the solid line. As discussed earlier, it has been shown that this cycle of dependency leads the long-running transaction to an inconsistent state, meaning that in their life-cycle the two sub-transactions are working on two different values for the same object.

Unrepeatable read: In this case, a sub-transaction *read*s an object twice, once before another sub-transaction's *write* action and for a second time after the *write* action (the second sub-transaction may *write* a new version and commit). This means that a sub-transaction changes the object (*write*) when another sub-transaction had ongoing access (*read*) to it and has not yet finalised its access.

Concurrency and Data Inconsistency

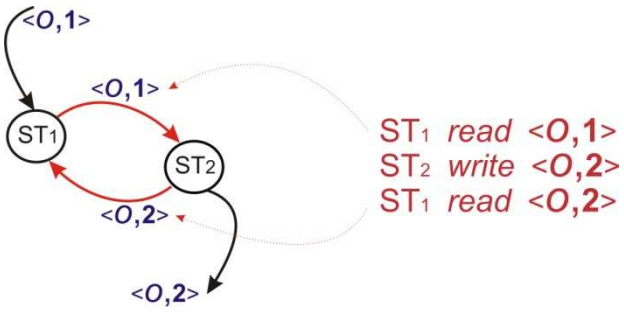

FIG. 4-26 UNREPEATABLE READ

Figure 4-26 shows a simple scenario of *unrepeatable read* where ST_1 access to an object *o* and retrieves the version 1 of the object, but if there is a second attempt by ST_1 the result of *read* is different because during the execution of ST_1 the sub-transaction ST_2 had access to the object (*write*) and changed it to a new version (here version 2). In terms of dependency, *write* on object *o* by ST_2 after the first *read* of the object by ST_1 creates the *read* to *write* dependency, and the next read operation creates a *write* to *read* dependency between the same sub-transactions. This results in inconsistency on object o in the life-cycle of the two sub-transactions which belong to the same long-running transaction.

These simple examples can be extended to more complicated scenarios when considering a few intermediate sub-transaction dependencies where the final dependency returns to the first sub-transaction. These cycles in dependency graphs are called *wormholes*(Gray & Reuter, 1993), (Elmagarmid, 1992).

The *isolation theorems* are the classic method for showing the correctness of the transactional environment (Gray & Reuter, 1993), (Elmagarmid, 1992), (Moss, 1985). The main result of the study in isolation can be summarised as follows: a serial execution of transactions is always correct, when each transaction follows the commit or full rollback of the other one. Therefore, if we show a concurrent execution of transactions is equivalent to a serial execution, we may use the isolation theorem to deduce correctness of the transaction execution.

The conventional isolation theorems use centralised synchronisation for applying the two-phase locking (2PL) scheme (Bernstein et al., 1987), (Date, 2003). 2PL and lock compatibility guarantee the environment is free of any wormholes (Gray & Reuter, 1993). It also shows that an environment without wormholes is isolated, i.e. the transactions' execution is equivalent to a serial execution of them, and thus the system is consistent.

Our approach advocates a fully distributed solution and hence for avoiding the centralised synchronisation we do not use 2PL. The implications of *write* operations are handled instead using dependency graphs. For any *write* operation in the semantics of a transaction (in service compositions with data dependency, section II, A, refer to PDD and SDD (J. Yang et al., 2002), (Singh & Huhns, 2005)) it the dependency graphs determines the access rights to the corresponding object. The necessary graphs can be propagated in VPTN. In short, by using the dependency graphs we avoid the wormholes.

Before introducing the dependency graphs, we give the definition of a wormhole, its theorem in our long-running transactional model and the proof for correctness of a wormhole-free long-running transaction.

4.3.3 FORMAL DEFINITIONS

For defining an execution of a long-running transaction we use the standard term *history*. A *history* of a long-running transaction is any sequence-preserving merge of the actions of a set of sub-transactions into a single sequence for the set of sub-transactions and is denoted by$H = \langle\langle st, a, o\rangle_i | i = 1, \dots, n\rangle$. Each step of the history $\langle st, a, o\rangle$ is an action *a* by sub-transaction *st* on object *o*. A history for the set of sub-transactions $\{ST_j\}$ is a sequence, each containing transaction ST_j as a subsequence and containing nothing else.

In effect, a history lists the order in which actions were successfully completed. *Serial histories* are one-subtransaction-at-a-time histories. In serial histories as there is no concurrency, there is not any inconsistency and no problem with viewing inconsistent data by other transactions.

As each action in the history changes the version of the object, we need to formalise the versioning definition before defining dependencies between sub-transactions in the history. The *version* of an object *o* at step *k* of a history is an integer and is denoted *V(o,k)*. In the beginning, each object has version zero (*V(o,*0*)*=0). At step *k* of history *H*, object *o* has a version equal to the number of *write*s of that object before this step. Formally, this means:

$$V(o,k) = \left|\left\{\langle st_j, a_j, o_j\rangle \in H \middle| j < k and a_j = WRITE and o_j = o\right\}\right|.$$

Note we will use capitalisation for operation from now on in order to make the notation more clear.

Now we are able to define dependency in a history. Each history *H* for a set of sub-transactions $\{ST_i\}$ defines a threefold *dependency relation* DEP(*H*), defined as follows.

Let *ST*1 and *ST*2 be any two distinct sub-transactions, let *o* be any object, and let *i*, *j* be any two steps of *H* with $i < j$. Suppose step $H[i]$ involves action *a*1 of *ST*1 on object *o*, step $H[j]$ involves *a*2 of *ST*2 on *o*, and suppose there is no *write* of *o* by any transaction between these steps (there is no $\langle ST', WRITE, o\rangle$ in $H[i+1], \dots, H[j-1]$). Then DEP(*H*) is defined as:

$\langle ST, \langle o, V(o,j)\rangle, ST'\rangle \in \mathrm{DEP}(H)$ if

*a*1 is a WRITE and *a*2 is a WRITE

or

*a*1 is a WRITE and *a*2 is a READ

or

*a*1 is a READ and *a*2 is a WRITE.

This classic definition captures all dependencies (WRITE→WRITE, WRITE→READ and READ→WRITE).

The dependency relation for a history of a long-running transaction defines a directed *dependency graph*. Sub-transactions are the nodes of the graph, and object versions label the edges. This means that if $\langle ST, \langle o, j\rangle, ST'\rangle \in \mathrm{DEP}(H)$, then the graph has an edge from node *ST* to node *ST'* labelled by $\langle o, j\rangle$. It follows that two histories are equivalent, if they have the same dependency relation.

4.4 WORMHOLE THEOREM FOR DE LONG-RUNNING TRANSACTIONS

This theorem is a well-known theorem in isolation theorems. In what follows we describe how it can be adapted to determine wormhole-free transactions of the kind considered in our approach, i.e. long-running transactions. Hence, before using the classic theorem and its proof (Gray & Reuter, 1993) we introduce the equivalent concept and notation in long-running transactions, and then adapt the proof to long-running transaction.

The dependencies in the history of a long-running transaction can define a time ordering of the sub-transactions. Conventionally this ordering is signified by $<<<_H$, (or simply by $<<<$ when the history is clear from context), and it is the *transitive closure* of $<<<$. It is the smallest relation satisfying the equation $ST <<<_H ST'$:

if $\langle ST, o, ST'\rangle \in \mathrm{DEP}[H]$ for some object version *o*, **or**

($ST <<<_H ST''$ and $\langle ST'', o, ST'\rangle \in \mathrm{DEP}[H]$ for some sub-transactions ST'', andsome object *o*).

In terms of the dependency graph, we can say that $ST <<< ST'$ if there is a path in the dependency graph from sub-transaction *ST* to sub-transaction ST'.

The $<<<$ ordering defines the set of all sub-transactions that run before or after *ST*;

$$\mathrm{BEFORE}(ST) = \{ST' | ST' <<< ST\}$$

$$\mathrm{AFTER}(ST) = \{ST' | ST <<< ST'\}$$

If *ST* runs fully isolated (i.e., it is the only sub-transaction, or it *read* and *write* objects not accessed by any other sub-transactions), then its BEFORE and AFTER sets are empty, and it

can be scheduled in any way. When a sub-transaction is both after and before another distinct sub-transaction (*ST* here), it is called *wormhole transaction* (ST' here):

$$ST' \in \text{BEFORE}(ST) \cap \text{AFTER}(ST)$$

It is not hard to see that serial histories do not have wormholes - in a serial history, all the actions of one transaction precede the actions of another, i.e. the first cannot depend on the outputs of the second.

Based on the wormhole theorem, a history is isolated if, and only if, it has no wormhole sub-transactions. On the other hand, the isolated histories have the *unique* property of having no wormholes. The theorem dictates that a history that is not isolated has at least one wormhole; $ST <<< ST' <<< ST$.

In graphical terms we can say that if the dependency graph has a cycle in it, then the history is not equivalent to any serial history because some sub-transaction is both before and after another sub-transaction. Fig 4-24, 4-25 and 4-26 demonstrate simple cases of such cycles. A wormhole in a particular history is a sub-transaction pair in which *ST* ran before ST' which ran before *ST*. A history is said to be isolated if it is equivalent to a serial history.

4.4.1 ISOLATED HISTORY HAS NO WORMHOLE

As the first part of the proof of this concept, the classical testimony of the wormhole theorem has been adopted to long-running transactions and the proof is an adaptation of that given by Gray in (Gray & Reuter, 1993).

Theorem. An isolated history has no wormholes.

This proof is done by contradiction. Suppose that in a long-running transaction, *H* is an isolated history of the execution of the set of sub-transactions $\{ST_i | i = 1, \dots, n\}$. By definition, *H* is equivalent to some serial execution history of the same long-running transaction, denoted by *SH* (obviously for that same set of sub-transactions). Without loss of generality, assume that the sub-transactions are numbered so that $SH = ST_1 \| ST_2 \| \dots \| ST_n$. This means *SH* is equivalent to starting with the execution of all actions in sub-transaction ST_1 (followed by) concatenated to the execution of all actions in sub-transaction ST_2 (followed by) concatenated to ... execution of all actions in sub-transaction ST_n.

Now suppose that *H* has a wormhole. We will show that it is impossible for it to be isolated. Having a wormhole means that there is some sequence of sub-transactions $ST, ST', ST'', \dots, ST'''$ in *H* such that each is BEFORE the other (i.e., $ST <<<_H ST$), and the last is BEFORE the first (i.e., $ST''' <<<_H ST$). Let i be the minimum sub-transaction index such that ST_i is in this wormhole, and let ST_j be its predecessor in the wormhole (i.e., $ST_j <<<_H ST_i$). Since *i* is minimum, ST_j comes completely AFTER ST_i in the execution history *SH*, so that $ST_j <<<_{SH} ST_i$ is impossible (recall that *SH* is a serial history). But since *H* and *SH* are equivalent, $<<<_H = <<<_{SH}$; therefore, $ST_j <<<_{SH} ST_i$ is also impossible. This contradiction proves that if *H* is isolated, it has no wormholes, or, as we say, is a *wormhole-free* history.

4.4.2 WORMHOLE-FREE HISTORY IS AN ISOLATED HISTORY

Now for the second part of the theorem we still need to show that a history without wormholes is isolated. In what follows we adapt the classic Wormhole theorem proof (Gray & Reuter, 1993) but for a long-running wormhole-free history.

If a long-running transaction has *n* sub-transactions (the number of sub-transactions is n), then they appear in the history *H* of the long-running transaction. The induction hypothesis is that any n sub-transactions history *H* which is wormhole-free is isolated (this means *H* is equivalent to some serial history *SH* for that set of sub-transactions).

If $n < 2$, then any history of the long-running transaction is a serial history, since only zero or one sub-transaction appears in the history. In addition, we have already seen that any serial history is an isolated history. The basis of the induction, then, is trivially true.

Suppose the induction hypothesis is true for $n-1$ sub-transactions, and consider some history *H* of n sub-transactions that has no wormholes. Pick any sub-transaction ST, then pick any other sub-transaction ST', such that $ST <<< ST'$, and continue this construction as long as possible, building the sequence $Q = \langle ST, ST', \ldots \rangle$. Either Q is infinite, or it is not. If Q is infinite, then some sub-transaction ST'' must appear in it twice. This, in turn, implies $ST'' <<< ST''$, and hence ST'' is a wormhole of *H*. But since *H* has no wormholes, Q cannot be infinite. The last transaction in Q, and let us call it ST^*, has the property $\mathrm{AFTER}(T^*) =$, since the sequence cannot be continued past ST^*.

Consider the history $\mathrm{H}' = \langle \langle st_i, a_i, o_i \rangle \in H | st_i \neq ST^* \rangle$. This says that H' is the history *H* with all the formal actions of transaction ST^* removed. By the choice of ST^*,

$$\mathrm{DEP}(\boldsymbol{H'}) = \{\langle \boldsymbol{ST}, \langle \boldsymbol{o}, \boldsymbol{i} \rangle, \boldsymbol{ST'} \rangle \in \mathrm{DEP}(\boldsymbol{H}) \big| \boldsymbol{ST'} \neq \boldsymbol{ST^*}\} \qquad \text{EQ. 2}$$

H' has no wormholes (since *H* has no wormholes, and $\mathrm{DEP}(H) \supseteq \mathrm{DEP}(H')$). The induction hypothesis, then, applies to H'. Hence, H' is isolated and has an equivalent serial history $SH' = ST_1 \| ST_2 \| \ldots \| ST_{n-1}$ for some numbering of the other sub-transactions.

The serial history $SH = SH' \| ST_n = ST_1 \| ST_2 \| \ldots \| ST_{n-1} \| ST^*$ is equivalent to *H*. To prove this, we need to show that $\mathrm{DEP}(SH) = \mathrm{DEP}(H)$. By construction,

$$\mathrm{DEP}(\boldsymbol{SH}) = \mathrm{DEP}(\boldsymbol{SH'} \| \boldsymbol{ST_n}) = \mathrm{DEP}(\boldsymbol{SH'}) \cup \{\langle \boldsymbol{ST'}, \langle \boldsymbol{o}, \boldsymbol{i} \rangle, \boldsymbol{ST^*} \rangle \in \mathrm{DEP}(\boldsymbol{H})\} \qquad \text{EQ. 3}$$

Also, by definition, we have $\mathrm{DEP}(SH') = \mathrm{DEP}(H')$. Using this to substitute equation EQ. 2 into equation EQ. 3 gives:

$$\mathrm{DEP}(\boldsymbol{SH}) =$$

$$\{\langle \boldsymbol{ST}, \langle \boldsymbol{o}, \boldsymbol{i} \rangle, \boldsymbol{ST'} \rangle \in \mathrm{DEP}(\boldsymbol{H}) \big| \boldsymbol{ST'} \neq \boldsymbol{ST^*}\}$$

$$\cup \{\langle \boldsymbol{ST'}, \langle \boldsymbol{o}, \boldsymbol{i} \rangle, \boldsymbol{ST^*} \rangle \in \mathrm{DEP}(\boldsymbol{H})\}$$

$$= \mathrm{DEP}(\boldsymbol{H})$$

Thus, the identity $\mathrm{DEP}(SH) = \mathrm{DEP}(H)$ is established, and the induction step is proven.

The wormhole theorem is the basic result from which all the others follow. It essentially says “cycles are bad”. Wormhole is just another name for cycle. The wormhole theorem can be stated in many different ways. One typical statement is called the *Serializability Theorem*: A history *H* is isolated (also called a *serializable schedule* or a *consistent schedule*) if, and only if, $<<<_H$ implies a partial order of the transactions. (Alternatively: if and only if it defines an acyclic graph, or implies a partially ordered set (Gray & Reuter, 1993)). This is the basis for our log system to create dependency graphs and avoid the bad cycle.

4.5 INTERNAL DEPENDENCY GRAPH

We have shown that by avoiding wormholes we can release results between sub-transactions of a long-running transaction. In doing so, we use the dependency graph to trace released data items (objects) between each participant. This graph is updated regularly and the cycle (wormhole) can be detected in each step. As this graph captures dependencies between sub-transactions of a transaction we call it *Internal Dependency Graph* (IDG).

For clarifying the access-rights, inside of each participant we use a simple lock mechanism which is compatible with the conventional S/X Lock. The only difference with S/X Lock is the UN-Lock mechanism. Since participants are executing a sub-transaction and the result can only be visible in that particular long-running transaction, instead of unlocking the data we introduce an *internal lock* I-Lock which unlocks the data items in the context of a particular transaction. This means the data item will be available for other sub-transactions of the transaction, which are executing in other participants. As mentioned in the beginning of the section, the execution of a transaction will be done by the ‘*local coordinator*’ of the participants. Fig 4-27 shows a simple example.

External dependency graph

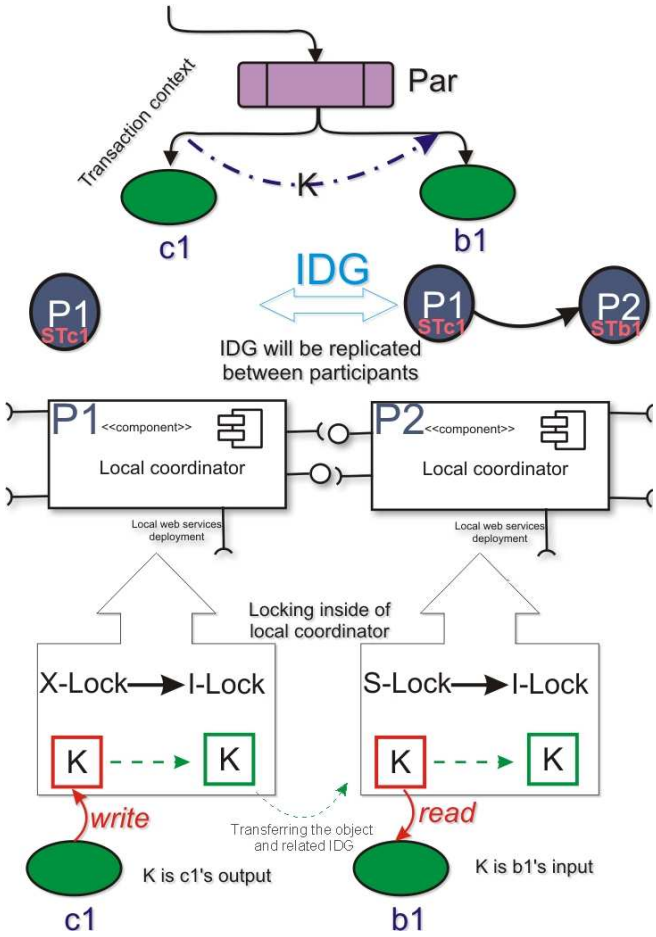

FIG. 4-27 LOCK AND IDG

Fig. 4-27 shows part of a long-running transaction, where a parallel composition between service c1 and b1 has a data-dependency (recall the composition type PDD in Section II, A). We have assumed that service c1 is a service offered by participant P1 and service b1 is of participant P2. Based on the transaction context described in the beginning of this section (Section III, A), c1 and b1 can be deployed in the context of two sub-transactions (STc_1 and STb_1). When participant P1, deploys the service c1 the service creates a data-item K as the output (notice that P1 uses X-Lock for writing/creating the object) and sub-transaction STc1 commits and releases the result for other sub-transactions. To do this P1 uses I-Lock and pre-request for the object while the IDG can be created for the object. Our example illustrates the case where participant P2 needs the output of c1 (STc1). It can use the result which has been released by I-Lock and if there is a dependency between the two sub-transactions present in the corresponding IDG (STc1 <<< STb1), also shown in the figure, P2 uses S-Lock for reading K and proceeds to use it as the input in service b1. Then P2 can again release it by using I-Lock on the data-item. Any subsequent usage of the data-item will be done by checking and updating the IDG. In this way, local coordinators can avoid any cycle (e.g., STc1 <<< STb1<<<...<<< STc1).

4.6 EXTERNAL DEPENDENCY GRAPH

As the life-cycle of long-running transactions is long, occasionally, releasing results between these transactions before their termination (commit/rollback) can be valuable for a digital ecosystem (Moschoyiannis et al., 2008), (Razavi et al., 2006) in covering a range of B2B scenarios. However, these 'partial results' can be costly - in case of abortion of the first long-running transaction we may face cascading abortion (Moschoyiannis et al., 2008), (Razavi et al., 2007d). This is why they should be used when it is necessary and there is possibility for *forward recovery* in case of abortion of the first transaction - this is further discussed in the next section. As the partial results are released before the actual commit of a long-running transaction, the mechanism for releasing them is called *Conditional Commit*.

For conditional commit again we use a dependency graph in combination with the wormhole theorem. It is important to note that:

- in the first place two long-running transactions had full invisibility towards each other, therefore the released data-item from the first long-transaction has to be read by the second transaction
- as all the data-items are in the deployment level (recall discussion in chapter 3, section 1) they will be created by the transaction in the first place, that means the first operation on a data-item for any long-running transaction will be a *write* (in fact, it can take place in one of its sub-transactions but this is the primary assumption of SOA).

Therefore in any conditional commit between transaction T1 and T2 there is a *write→read* dependency and as the first transaction is not fully committed, any *write* operation can create a wormhole (T1<<<T2<<<T1 in term of *write→read→write*). That is why after releasing the partial result the data-item will be read-only and this cannot change until the first transaction commits. Note also that the second transaction cannot commit before the first one does and as a result it will have a commit dependency.

For addressing this limitation, we define a C-Lock for the conditional commit of partial results and the dependency graph for releasing these data-items is called *External Dependency Graph* (EDG). In addition to capturing the dependencies on particular data-items released between transactions, this graph also captures the commit dependency. By using the EDG the second transaction can not commit unless it receives a confirmation from the first transaction that it has committed. Figure 4-28 shows an example of using conditional commit.

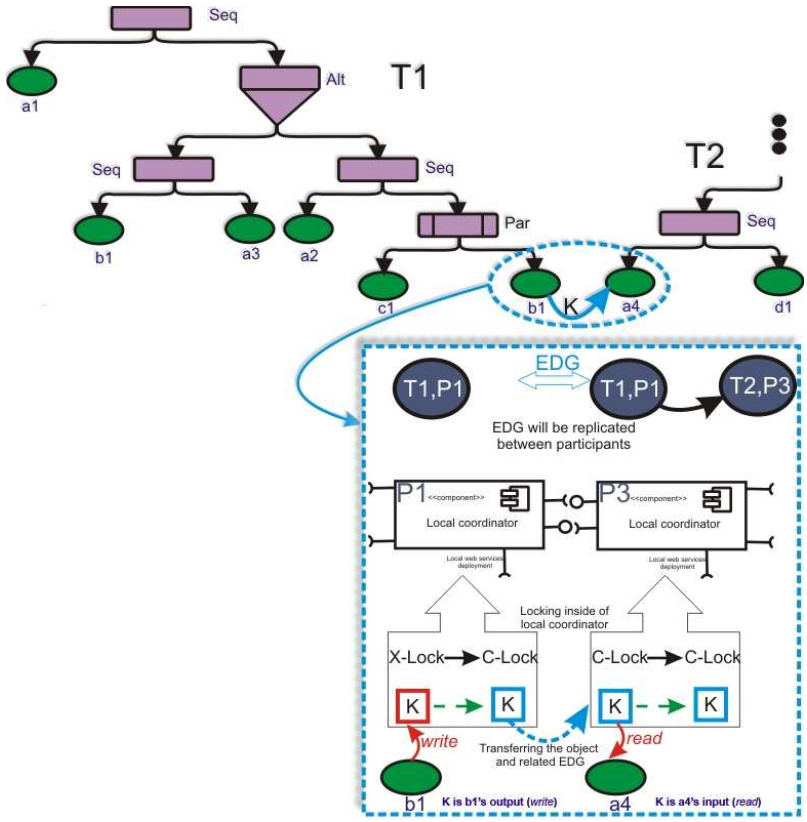

FIG. 4-28 EDG AND C-LOCK

In Figure 4-28 transaction T1 releases the partial result (data-item K) to transaction T2. As a result, the related External Dependency Graph (EDG) will be created by coordination of Participant P1 (one of T1's participants which has done the last action on data-item K), and the graph is shared by P3 (one of T2's participants which is going to do the first *read* operation on data-item K). The important point is that the C-Lock has been used for data-item K and the lock will be inherited by any participants which are going to use the data-item. In this way, the data-item will remain 'Locked' until the transaction, the one which has created the data-item (*originator transaction*; here is T1), commits. Meanwhile, in terms of abortion in the originator transaction, all dependent transactions should be rolled back. In the next section (subsection B and C) we describe the recovery procedure that needs to take place.

4.7 XML REPRESENTATIONS AND SCHEMAS

As the transaction context, relies on xml presentation (recall 4.2), for applying the theorem and the primary designing of logs (IDG and EDG), we use xml too. The rest of this chapter introduce the schema for such a presentation. These schemas can be downloaded from (Razavi, 2009).

4.7.1 INTERNAL DEPENDENCY GRAPH SCHEMA

Based on the theorem, for avoiding wormholes the first necessity for internal dependency logs is the possibility for tracing dependencies. Fig 4-29 shows the root of internal dependency log, when the first element identifies the data-items which should be shared and the second element shows the chains of dependencies.

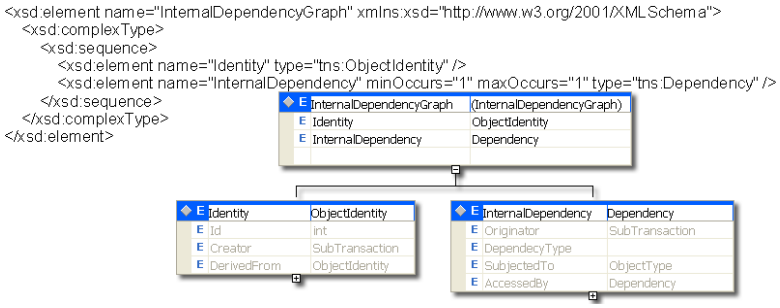

FIG. 4-29 TRACING INTERNAL DEPENDENCY

For identifying the data item, we use '*ObjectIdentity*' type (see Fig 4-30), which includes three elements; '*Id*' (a unique identity in scope of a transaction), '*Creator*' (to identify the sub-transaction which has created the data-item) and '*DerivedFrom*' (shows the relationship between the data-item and other data items in scope of a transaction).

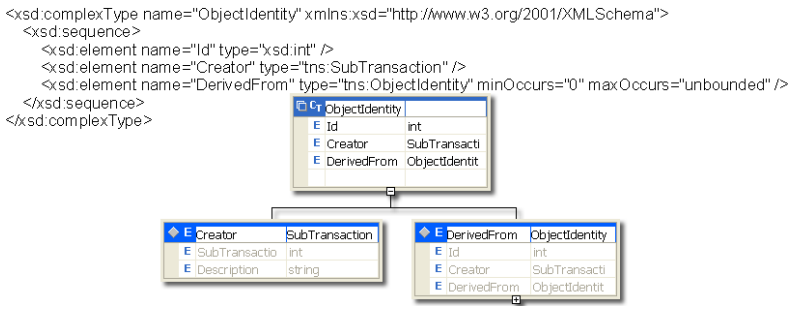

FIG. 4-30 OBJECT IDENTITY AND RELATED SCHEDULE HISTORIES

Fig 4-31 shows the structure of '*Dependency*' complex type. By using the recursive structure of this type, we can create the graph and trace the chains of dependencies for a data-item, in this way we can avoid any cycle in the graph. The dependency type has three elements; '*Originator*' (sub-transaction that share the data item), '*SubjectedTo*' (the modifications on the data-item) and '*AccessedBy*' (shows the next level of dependency on the object).

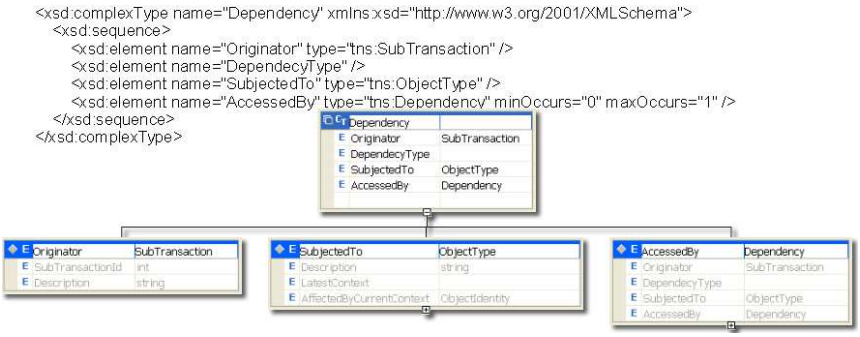

FIG. 4-31 RECURSIVE STRUCTURE AND CREATING IDG

In fact the dependency on the object can grow by using '*AccessedBy*' element. In this way, not only the dependencies on the data-item is traceable and we can avoid wormhole, but also the chains of modifications are accessable (through '*SubjectedTo*' element which is an *ObjectType*) and in term of failure the content of data item is recoverable step by step (failure and recovery is analysed in chapter 6). Fig 4-32 shows the structure 'ObjectType'.

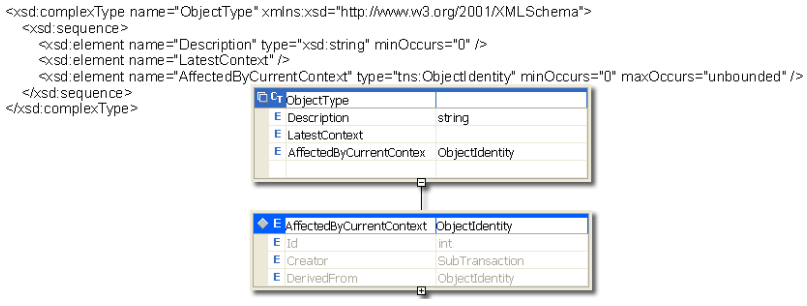

FIG. 4-32 RECORD OF CHANGES AND AFFECTING OTHER OBJECTS

Fig 4-33 shows the full structure of the internal dependency schema in Visual Studio .NET (2003 version). In the implementation chapter, we use the relevant class diagram to show

the structure of the parser and implemented module for it.

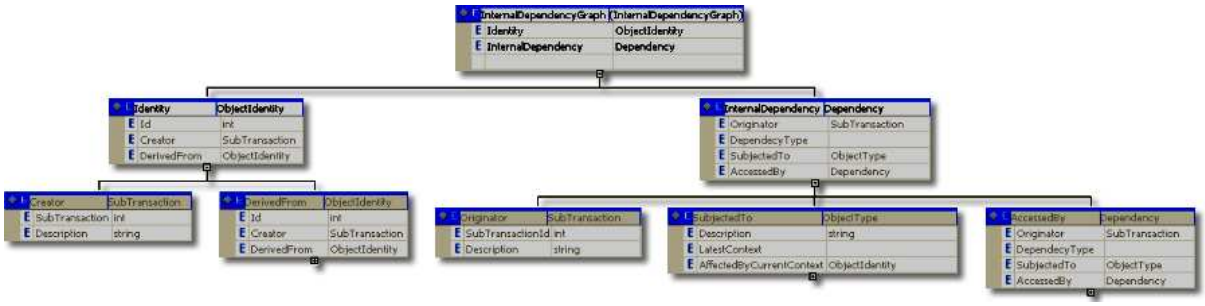

FIG. 4-33 INTERNAL DEPENDENCY GRAPH STRUCTURE

4.7.2 External Dependency Graph Schema

As with the Internal Dependency Graph Schema, the External Dependency graph should enable us to trace dependencies. But in this log mechanism, dependencies are between different transactions. Fig 4-35 shows the root of the external dependency log, when the first element identifies the originator of dependency, next element is a data-item which needs to be shared and the third element shows the dependencies to this data-item.

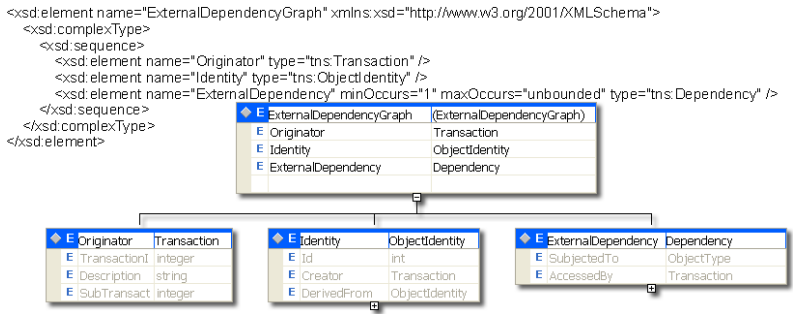

FIG. 4-34 TRACING EXTERNAL DEPENDENCY

As we have seen in the theorem, an external dependency does not create the same read and write chains and the actual shared data-item will be read only. This means that the dependency will be between the originator of the data-item and other transactions which accessed to that data item. Fig 4-36 shows '*Dependency*' complex type, when '*SubjectedTo*' shows the data item content and its relationships with other data items, '*AccessedBy*' clarifies transactions which has accessed (read) the data-item.

XML Representations and Schemas

```
<xsd:complexType name="Dependency" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:sequence>
    <xsd:element name="SubjectedTo" type="tns:ObjectType" />
    <xsd:element name="AccessedBy" type="tns:Transaction" minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>
```

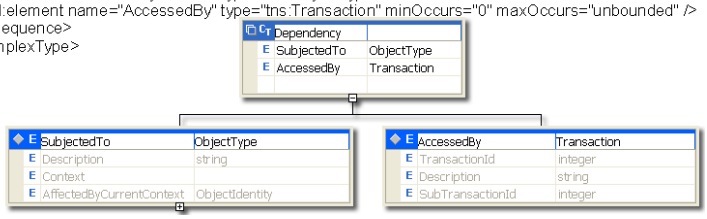

FIG. 4-35 CREATING EXTERNAL DEPENDENCY GRAPH

The other element of the external dependency graph (Fig 4-35) is the identity of the data item. Because we are dealing with a distributed system, it is not feasible to consider a unique identity for each data item but we can identify the data item through the transaction which created it. Fig 4-37 shows '*ObjectIdentity*' complex type, where the object identity has been clarified by three elements: an Id which will be unique for a transaction; the (transaction) creator of the data item; and the optional element is data item(s) which are related to this data item (if this data item is derived from them).

```
<xsd:complexType name="ObjectIdentity" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:sequence>
    <xsd:element name="Id" type="xsd:int" />
    <xsd:element name="Creator" type="tns:Transaction" />
    <xsd:element name="DerivedFrom" type="tns:ObjectIdentity" minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>
```

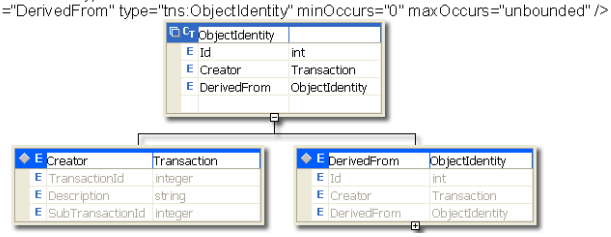

FIG. 4-36 IDENTIFYING OBJECT AND ITS AFFECTING OTHER OBJECTS

The '*Transaction*' complex type is shown in Fig 4-36, which includes the transaction identity and its (involved) sub-transaction. The '*Description*' can provide extra information about the involvement of the transaction (which we have left for additional information about the parsing and possible compatibility issues).

```
<xsd:complexType name="Transaction" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:sequence>
    <xsd:element name="TransactionId" type="xsd:integer" />
    <xsd:element name="Description" type="xsd:string" />
    <xsd:element name="SubTransactionId" type="xsd:integer" />
  </xsd:sequence>
</xsd:complexType>
```

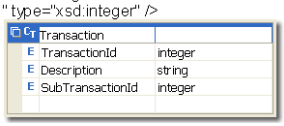

FIG. 4-37 IDENTIFYING EFFECTIVE TRANSACTIONS AND SUB-TRANSACTIONS IN EDG

Fig 4-40 shows the full structure of the external dependency schema in Visual Studio .NET (2003 version). In the implementation chapter, we use the relevant class diagram to show the structure of the parser and the implemented module for it.

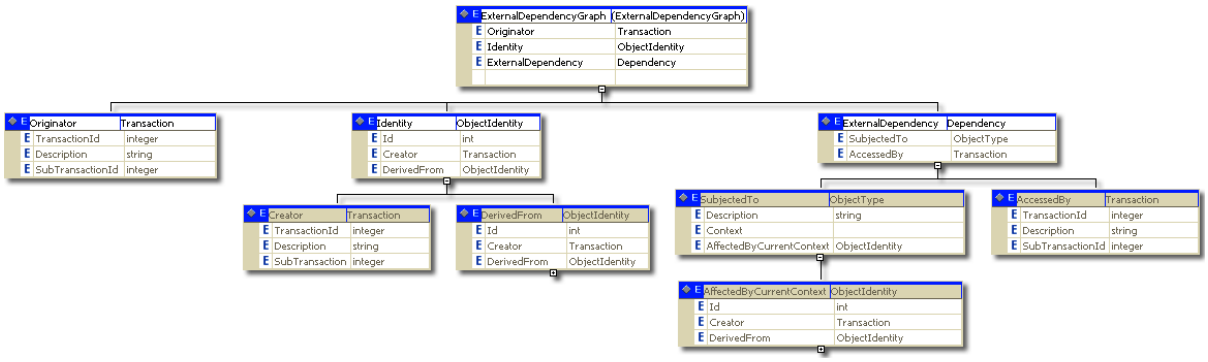

FIG. 4-38 EXTERNAL DEPENDENCY GRAPH STRUCTURE

5 FAILURES AND RECOVERABILITY

Up to this point we have been concerned with correctness of our lock scheme used in long-running transactions for a Digital Ecosystem. Locks are typically used in transactional models for assurance of data consistency and integrity in a concurrent environment. The other important issue related to the self-organising ability of a Digital Ecosystem (Chang & West, 2006a), is the self-recovery (or self-healing (Krause et al., 2008)) in terms of failure.

Recovery Management is used to preserve atomicity in transaction models. Unfortunately, conventional lock mechanisms severely (and intentionally) limit concurrency in a transactional environment. Such lock mechanisms also limit recovery capabilities. The conventional recovery mechanism is managed by centralised servers or a third party coordinator in terms of a compensation mechanism (Cabrera, Copeland, Feingold, R. Freund, T. Freund, Joyce, et al., 2005). Finally, existing recovery mechanisms themselves afford a considerable overhead to concurrency (for these issues recall chapter 2 and 3). This chapter provides an integrated solution for recovery management and concurrency control. These are considered as necessary features for the management of long-term transactions within "*digital ecosystems*" of small to medium enterprises (Razavi et al., 2007a).

First we turn our attention to failures that may occur at both the transactional and the network levels, and then show how these can be addressed in recovering the system to a consistent state. By introducing the forward recovery mechanism, we try to reduce the cost of recovery and in the last section will focus on the relationship between connectivity and recoverability (this chapter is optimised version of our work at Communicating Process Architectures / CPA; (Razavi et al., 2007d)).

5.1 NETWORK FAILURE AND TRANSACTION FAILURES

We have seen in Chapter 3, section 2 (3.2), that the network supporting a digital ecosystem can be conceptualised as the result of several business transactions where each transaction creates a private network, the so-called VPTN.

Conceptually, we consider D_U as all Digital Ecosystems, where T_U is all of the possible transactions in these Ecosystems and P_U all possible participants of the Ecosystem. Each transaction is the result of compositions of services from several participants. This can be described by the pair (t, P_t), where P_t is the set of participants which are involved in a transaction t.

The universe of digital ecosystems comprises all of the possible transactions and their participants and can be defined as:

$$D_U = \{(t, P_t) | t \in T_U \wedge P_t \subset P_U\}$$

We define a Digital Ecosystem *DE*, as a subset of D_U, where all of its participants, by

engaging in its transactions, are connected.

Each VPTN can be recognised by the transaction of its participants

$$(t, P_t)$$

where

$$P_t = \{Participantsinvolvedin Transaction t\}$$

In this paradigm, the maximum number of links which a participant may have is given by:

$$|P_t| - 1$$

In our transaction model, coordination of the underlying services is distributed and addresses both the order and the data dependencies, and hence the actual number of links is always less than this. It is given by:

$$1 \leq Nooflinksforanodein(t, P_t) \leq |P_t| - 1$$

EQ. 4

But based on the definition of a digital ecosystem, VPTNs can have overlaps which make a connected network. Therefore, nodes can be involved in several VPTNs and as a result, they will have additional links through participants of different transactions. Thus, a participant $R \in P_{t_i}, i = m..n$, which is involved in transactions:

$$\{t_m, \cdots, t_n\}$$

will have links between:

$$m + n \leq Nooflinksof R \leq \sum_{i=m}^{n} |P_{t_i}| - (m + n)$$

EQ. 5

If *R* participates in different transactions it is through different participants. Studies show most business networks follow the power-law distribution degree [16], which means that a very small number of nodes are involved in the majority of the transactions (EQ 5), and even in each transaction they will have the maximum numbers of links (EQ. 4).

Figure 13 shows a simple digital ecosystem, where '*Participant 4*' and '*Participant 3*', are involved in all of the VPTNs, and in each VPTN they get the majority of links.

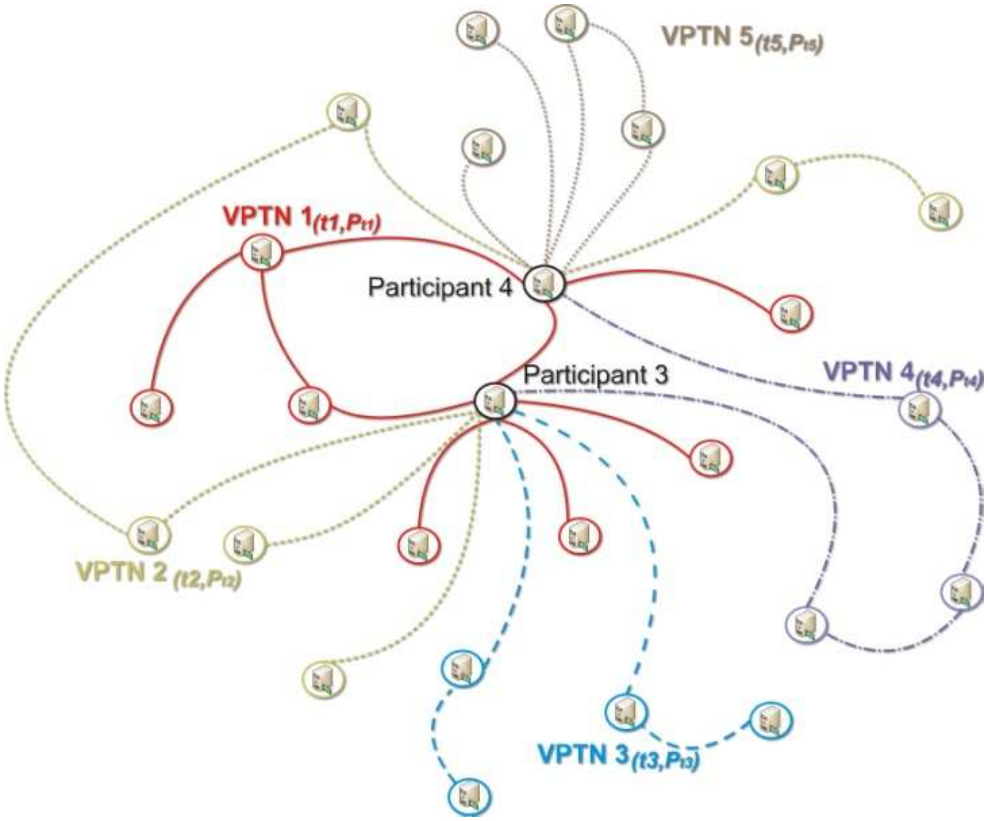

FIG. 5-1 DIGITAL ECOSYSTEM OF CONNECTED VPTNS

This has as a drawback that any problem in either '*Participant 4*' or '*Participant 3*' (or both) can cause serious disruptions in all VPTNs. Meanwhile a simple failure on '*Participant 4*' or '*Participant 3*' can fragment the network, which means that even if the involvement of '*Participant 4*' or '*Participant 3*' was restricted to alternative service composition, still the transactions may not be executed. Moreover, since '*Participant 4*' and '*Participant 3*' have obtained this important role based solely on their business transactions they may not be the best candidate for providing connectivity for the network. In other words, their emergence as a highly connected node has been driven by the volume of transactions they take part in and no other factor. This raises the question of whether it is desirable for a digital business ecosystem to rely on very few nodes in general.

Before addressing such aspects, we show how each VPTN will react to a failure and how it can be recovered, how the cost of failure can be reduced, and how full abortion of the transaction can be avoided. Then, in section 5.4 we examine how the possibility for failures can be reduced all together.

5.2 Recovery Procedure

We start with the well-known 'Rollback theorem' and build our recovery procedure around the concepts of degenerating the transactions and, of course, avoiding wormholes.

Rollback Theorem: A transaction that unlocks an exclusive lock and then does a '*Rollback*' is not well formed and, is a potential wormhole, unless the transaction is degenerate.

As the theorem is well known, we refer the interested reader to (Gray & Reuter, 1993) for the actual proof. The important point of the theorem is that we have to degenerate the transaction to effect rollback. For this purpose we can use the logs provided by the dependency graphs described in Section III and trace them. The only caveat is that the digital ecosystem network (of VPTNs) is distributed and therefore there is no centralised synchronisation. This entails that there is a risk for wormholes.

5.2.1 TWO PHASE RECOVERY

For avoiding wormholes, we have designed the recovery procedure in line with our consistency model (logs/locks) for concurrency control. Overall, Recovery Management in combination with the concurrency control procedure runs in two phases:

1. *Preparation phase:* consists of sending a message (abort/restart) to the participants of all sub-transactions that puts them (and their data) into an isolated mode (preparing for recovery). This helps avoid any propagation of inconsistent data and possibility for creating wormhole during the actual rollback.
2. *Atomic Recovery Transaction routine:* the recovery routine will be run as an atomic procedure that can rollback and cancel deployed services of sub-transactions by using correct data-items.

Both phases in recovery management rely strongly on tracing the corresponding dependency graphs. This is where the necessary information is recorded for finding the changes on data-items, in different participants, and undoing them to bring the system back to a consistent state. Fig. 5-2 shows a sample scenario that extends that presented earlier in Fig. 4-11 and can be downloaded from (Razavi, 2009).

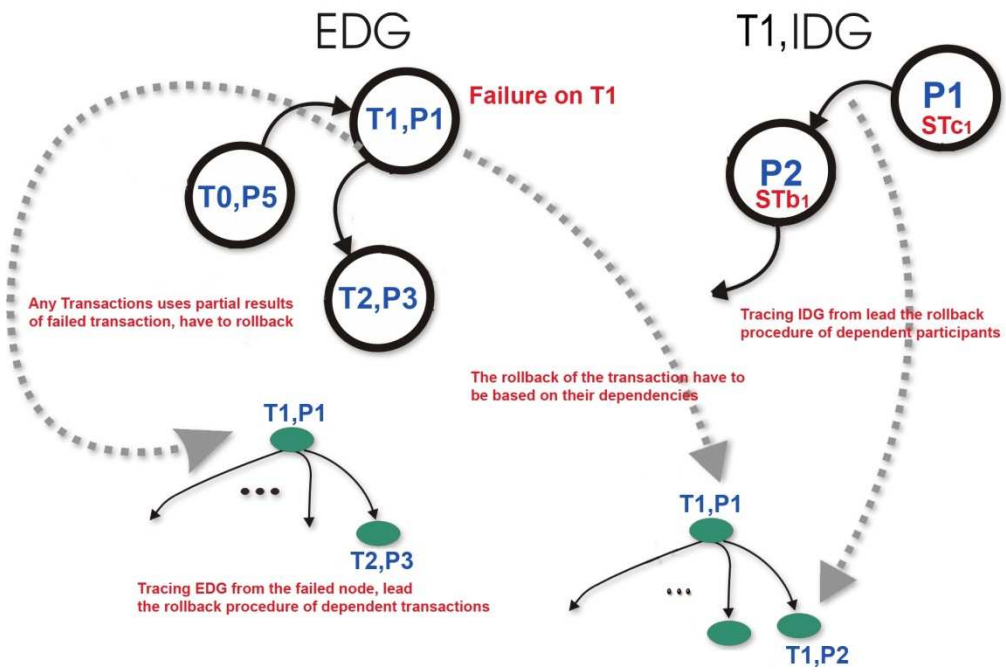

FIG. 5-2 TRACING DEPENDENCY GRAPHS FOR RECOVERY

According to Fig 5-12 a failure happens for *T1* while participant *P1* was trying to execute the transaction. The participant P1 has to stop any further progress on T1 and uses its EDG and IDG for informing about the failure on T1. As shown in the figure, participant P3 uses some results from P1 for transaction T2, which means P3 has to start a similar procedure for transaction T2. This is the external dependency. Participant P3 now uses its IDG, which indicates that the participant P2 needs to be informed for stopping the potential execution of transaction T1 (in this case, it will the sub-transaction STb1 of transaction T1). In fact, it has to inform any dependent transaction or sub-transactions by checking its related graphs to cater for all internal dependencies. Now for stopping the transaction progress (isolation of T1 affection) upon failure, we need an internal structure inside of the local coordinator.

5.2.2 ISOLATED RECOVERY

We have seen that the first phase of Recovery Management tries to just isolate the damaged (or failed) part of the system by distributing a message that can isolate all worked data-items of transactions of sub-transactions. We have also seen that in the transaction model, the I-Lock and C-Lock are locks which release data-items which can be the most problematic part of transactions in recovery. Tracing the relevant IDG and EDG can reveal these vulnerable data-items.

We introduce R-Lock as a fully isolated lock, which can be used just for rollback purposes. As part of tracing and stopping the progress of a failed transaction (and any affected transaction by the failed results of the transaction, in terms of partial results) we convert the data-items locks to R-Lock. Figure 5-3 shows the complementary routines, which in

combination with Figure 14 can show the first phase of the recovery.

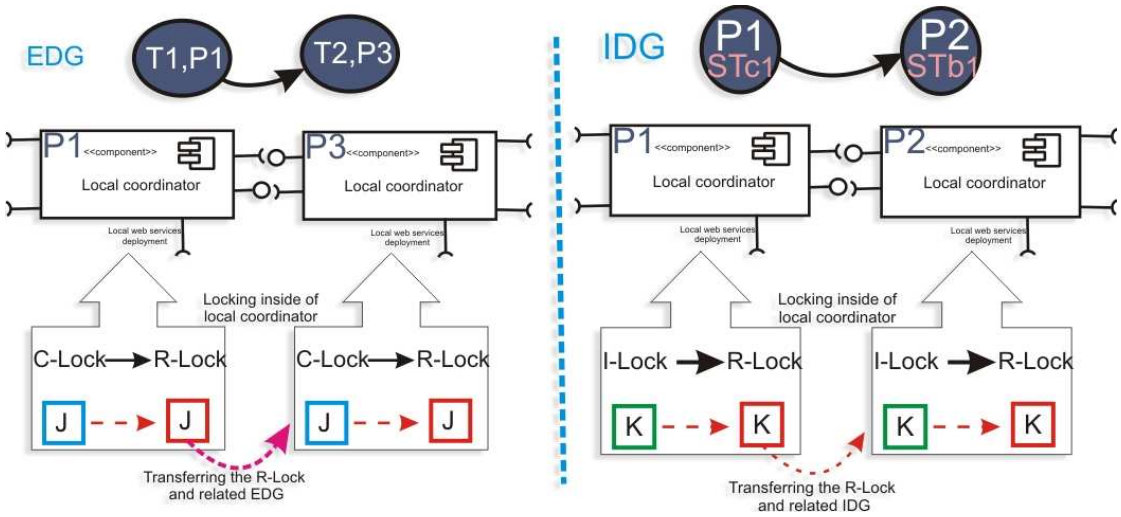

FIG. 5-3 R-LOCK AND ISOLATED RECOVERY

After using R-Lock (Fig. 5-3) the second phase of the recovery routine can be seen in the context of conventional shadow-based recovery. As the local agent of each participant keeps transaction information, such as changes and updates (even the committed transactions will have been archived) in its local repository (recall 4.1.1.2 and Fig. 4-3 and Fig. 4-9), the previous content of data-items can be retrieved and the deployed services can be cancelled. It is important to notice that for doing this the participant does not need any external help of other participants or synchroniser, and theoretically the second phase can follow the first phase without waiting for confirmation from all other participants.

However, a practical consideration has to do with what will happen if the failed transaction could not reach some of relevant participants. More generally, how can the possibility for full recovery be reduced, since it can be quite costly for the digital ecosystem? We attempt to address the second concern first.

5.3 FORWARD RECOVERY, FOR REDUCING THE RECOVERY COST

Full recovery can be costly in terms of resources, delays, business relations and so on. Further, we have seen that in a digital ecosystem dependencies may also exist across transactions so the effect of a recovered transaction may be magnified. For this reason it is desirable to avoid full recovery wherever possible. One way to do this is to design transactions with a number of alternative scenarios of execution. For doing so, we introduce '*Forward Recovery*' which is a mechanism for avoiding full recovery (Razavi et al., 2007d). The aim is during recovery failure to explore whether there is any possibility for successfully terminating the transaction following a different execution path to the one originally deployed, instead of rollback of the whole execution tree.

Forward recovery can be an option when there are alternative service compositions in the transaction (recall 3.1.2). By failing one sub-transaction of an alternative coordinator, that specific sub-transaction should be fully rolled back (but until the point of an alternative service composition) and then the alternative scenario can be tried by the participant's coordinator to commit the transaction with its other sub-transaction(s).

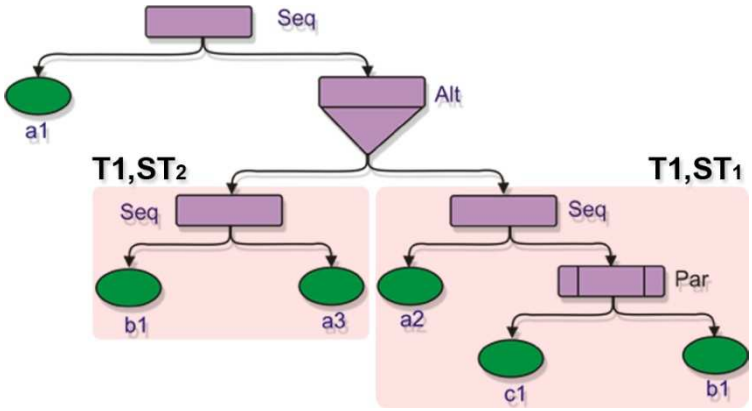

FIG. 5-4 ALTERNATIVE COMPOSITION AND FORWARD RECOVERY

Fig 5-4 shows an example in which transaction *T1* is using an alternative service composition (*Alt*), where there are two alternative paths; one following sub-transaction *ST1* and one following *ST2*. So far, we used the bottom part of *ST1* in Figure 11 and 12 to illustrate the release of a data-item inside of the transaction, and in Fig 5-3 and 5-4 for demonstrating the recovery procedure. If we assume *ST1* is attempted to execute, if a failure happens (for example in Fig 5-3 the relevant participant could not fulfil the execution of *STc1*), still we have the alternative path which is following sub-transaction *ST2* (in the left of the alternative service composition type-Alt).

For doing so, *ST1* must rollback. As was shown in the Fig 5-3 and 5-4, by using IDG and EDG, the internal and partial result should be traced and rolled back. After this, instead of continuing the recovery procedure and aborting the Transaction T1 completely, the second path of alternative service composition can be executed (*ST2*).

5.4 Network Connectivity and transactions

As the digital ecosystem is a distributed environment, where each participant can work in a loosely-coupled manner, the probability for failure in the network is very high. These failures are directly related to network connectivity - when the network connectivity is low, the probability for disconnection will be increased. We provide two mechanisms in our approach for addressing this challenging problem:

- Reducing the probability for network failure

- Providing the automatic procedure, in cases of unexpected network failures

In the beginning of this section we have shown a simple comparison for the best and worst case of connectivity for each participant. For reducing the probability for network failure, we try to provide a growth model to increase the connectivity. This is discussed in further detail in Section V and VI. Here, we introduce a mechanism for unexpected network failures.

When a participant involved in a transaction does not receive an expected response from another participant, it should be able to make decision despite that response. As the first assumption, each participant, in terms of the transaction context, considers an '*Expected Response Time*'. In each step of the long-running transaction execution, if a participant has not received any response during this time period, it automatically freezes the data-items related to the transaction. In this way, until the status of the transaction is clarified, the unnecessary execution of it will be suspended, the network resources will be saved and if there is failure, inconsistent data-items will not be spread in the VPTN and possibly the whole ecosystem.

For this temporary freezing of the data-items we introduce T-Lock ('*Time-out Lock*'). The T-Lock is rather like giving a time-out before rollback of a data item. The access to the data item will be limited until a deadline (time-out). If during this deadline the other participant responds, the original lock will be restored, i.e. T-Lock will be converted to the original lock. Otherwise, after the time-out elapses the recovery procedure will be started. The '*Expected Response Time*' for participants also depends on the network parameters and can be fixed statically based on the digital ecosystem's network characteristics. The '*Time-out*' of the T-Lock is related to the transaction expected life time and it can be varied according to the transaction context.

6 BEHAVIOUR OF COORDINATION

As discussed in the chapter 3, current protocols in transaction frameworks targeted at supporting business activities between networked organizations provide a specific pattern of behaviour, which not only violates the primary concept of SOC and Digital Ecosystems but also does not provide a truly distributed coordination model.

In this chapter, a formal foundation is provided for analysing the behaviour of the proposed model of long-running transactions and the distributed orchestration of the underlying service compositions. The formal semantics of behaviour of long-running transactions is aimed at describing the behavioural patterns services should follow in order to guarantee successful commitment or compensation within the transaction flow manager.

The proposed formal model for transactions, is based on Moschoyiannis, work and has been published at (Moschoyiannis, Razavi, & Krause, 2007), (Moschoyiannis et al., 2008) and (Moschoyiannis et al., 2008). It uses ideas taken from a variety of theories for describing the behaviour of communicating systems, from Shields' vector languages (Shields, 1979), (Shields, 1985), (Shields, 1997) to Mazurkiewicz traces (Mazurkiewicz, 1977), (Mazurkiewicz, 1988)to event structures (Nielsen, Plotkin, & Winskel, 1981) to process algebras (Milner, 1980), (Hoare, 1985). It draws upon a vector language-based description of behaviour, which allows monitoring or recording a number of communicating entities at the same time (groups of subtransactions), and has most recently been applied to modelling interactions between components of a (distributed) system in (Moschoyiannis, Shields, & Krause, 2005) and Moschoyiannis has explored the mathematic properties of the vector model (Moschoyiannis, 2005), (Moschoyiannis, 2004).

As the result of collaborative work with Moschoyiannis, this theory is adopted here to underpin the local coordination required for long-running multi-service transactions in a digital ecosystem(Razavi et al., 2007c), (Razavi et al., 2007b), (Razavi, Malone, Moschoyiannis, Jennings, & Krause, 2007), (Moschoyiannis et al., 2007), (Moschoyiannis et al., 2008) and (Moschoyiannis et al., 2008). This chapter is focused on usage of the model for analysis of the pattern behaviour of transactional model (Appendix 1, provides the details of the formal model).

6.1 DIGITAL ECOSYSTEMS: DISTRIBUTED AGENTS

As the kernel of each platform, we have considered a software agent which is responsible for coordinating the participant's business activities (transactions). As we have seen this local agent also archives the information related to these activities (corresponding VPTNs) and improves the general connectivity of the network (its digital ecosystem), and in doing so it contributes to the so-called *network growth*(Razavi et al., 2007a). This is an important aspect when it comes to sustainability, especially in a fully distributed solution. This leads up to the main definition of a digital ecosystem (recall chapter 3) which is represented in Fig 6-

1, highlighting the fact that there should be no centralized point of command and control in a digital ecosystem.

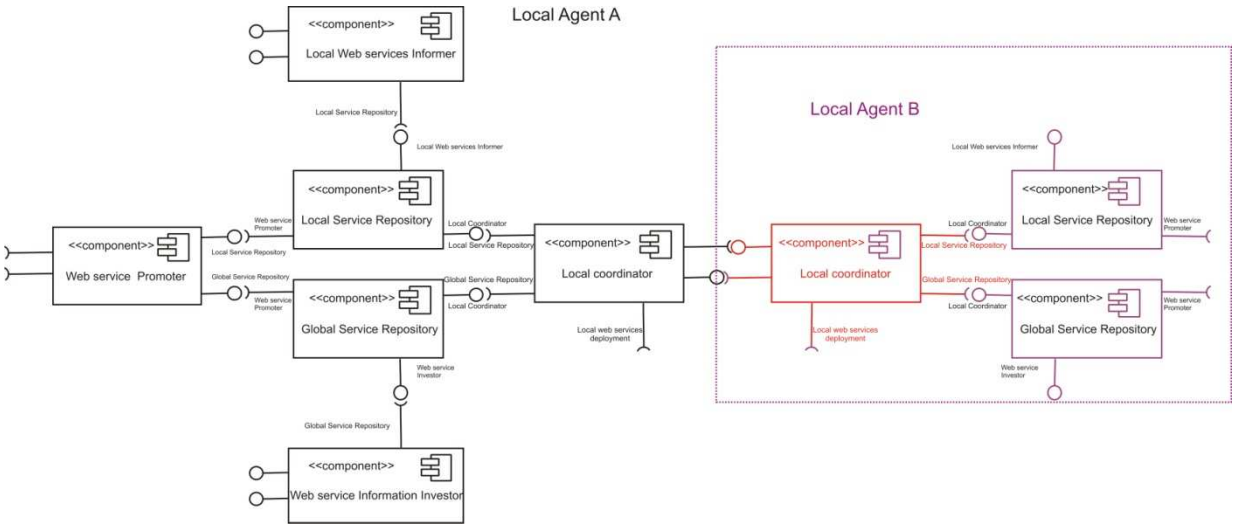

FIG. 6-1 LOCAL AGENT AND COORDINATION[6]

Fig. 6-1 shows the structure of the local software agent of each participant. The '*local coordinator*' component coordinates the service requests to and from the local platform. In other words, it deploys services of the platform, coordinates the transactions and archives their information in the '*local service repository*'. In this way, all participants of a transaction will keep the archived information of the transaction. The '*local web service informer*' component updates any changes of local services in the '*local service repository*' and relevant participants can be notified of the changes through the '*web service promoter*'. The links to other participants will be kept in the '*global service repository*'. Note that at this stage, participants of different VPTNs are connected to each other [in (Razavi et al., 2007a) this is called the *birth stage* of the underlying network]. For reducing the possibility of failures and increasing the network stability (as will be explained at chapter 7), the network connectivity, i.e. the number of links to other participants, may change. These changes will be done by two components; the '*web service information investor*', for updating new links to the global repository and the '*web service promoter*', for promoting new links to other participants [in (Razavi et al., 2007a) this is referred to as the *growth stage*]. Fig 6-2 shows a diagram, where four agents are communicating; Agent A and B are Participating in a Transaction (recall chapter 4) and Agent D and C are involved in the network growth for increasing the connectivity (will be explain chapter 7).

[6] The image can be found in (Razavi et al., 2007c) and the main agent diagram is in Fig 4-9

Transaction Context

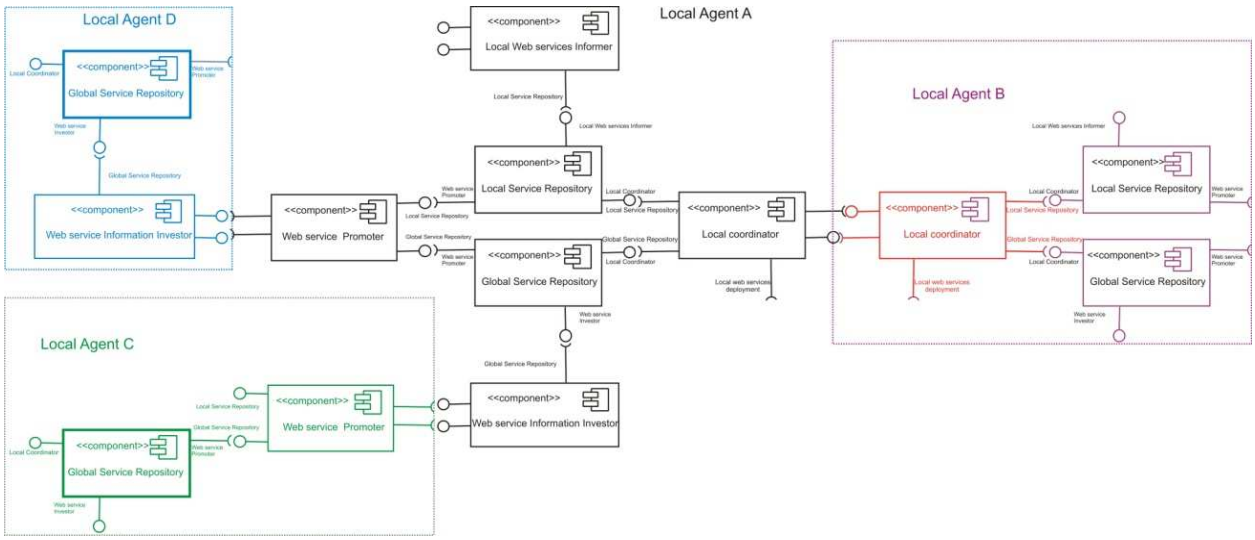

FIG. 6-2 LOCAL AGENTS COMMUNICATION DIAGRAM[7]

6.2 TRANSACTION CONTEXT

In previous chapters (Razavi et al., 2007c) we have described the use of a tree structure to represent transactions that involve the execution of services. This allows us to capture nested sub-transactions, the internal actions that need to take place in the course of execution of the transaction. To respect the loose-coupling of the underlying services, each participant provides its services and requests services of others through a coordinator component. Its purpose is to manage the communication between the different participants' platforms and the deployment of the corresponding services.

Drawing upon the latest work on the SOC computing paradigm (Papazoglou et al., 2006), we have considered different composition types which allow for various modes of service interaction in our model (recall chapter 3 and consistency model has been analysed in chapter 4). Fig. 6-3 shows a transaction tree with five basic services - a1, a2 and a3 of a local platform with coordinator component CC1, b1 of CC2, and c1 of CC3 - whose order of execution is determined by the corresponding composition types [transaction context symbols are based on the notation that first appeared in (Haghjoo & Papazoglou, 1992)].

[7] The image can be found in (Razavi et al., 2007c) and the main agent diagram is in Fig 4-9

Transaction Context

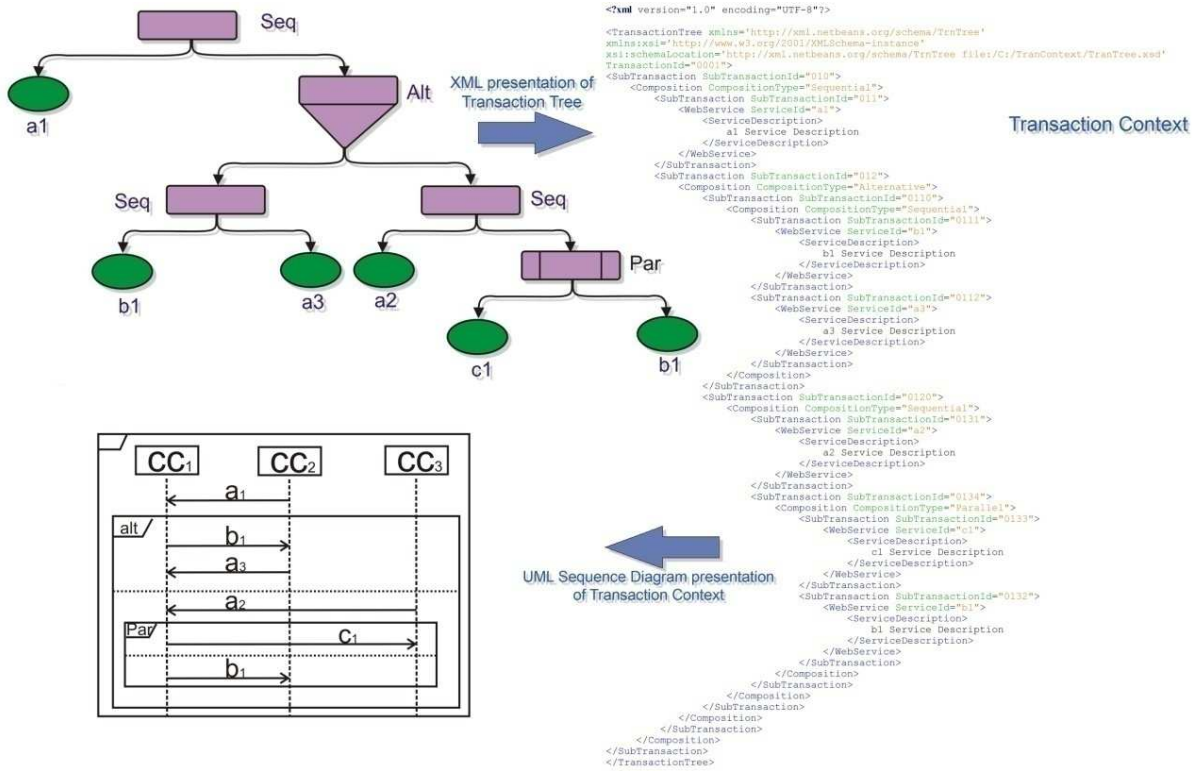

FIG. 6-3 TRANSACTION CONTEXT[8]

We note that the example transaction given in Fig. 6-3 is the same example which can be found in previous chapters [recall chapter 4 & 5, similarly this is explored in (Moschoyiannis et al., 2008)]. Primarily we have simplified the communications between coordinators and the abstraction of service deployment without considering acknowledgement or logs. This will be extended in this chapter to illustrate the key ideas as well as to show how these build on previous work in (Moschoyiannis et al., 2008), (Moschoyiannis et al., 2008).

A transaction tree determines the Participants and the respective services required for performing a business activity. In this sense, it sets the context of the conversation to follow and is issued by the Initiator of the transaction. In previous chapters [and (Razavi, 2009)] we have provided a schema for describing transaction contexts. The derived XML description of the transaction tree of our example (given in Fig. 6-3) can be found and downloaded following (Razavi & Moschoyiannis, 2008). The service interactions implied by a transaction tree can be modelled using a UML interaction diagram. The sequence diagram in Fig. 6-3 shows the three coordinator components of the participants in the transaction and the required service invocations between them. It can be seen that the behavioural scenarios, as given by the corresponding sequence diagram, determine the order of execution of the participating components' services.

As mentioned earlier, in a transactional setting we also need to deal with faults that may

[8] The XML schema is symbolic and can be checked from chapter 4 (section 4.2) or downloaded from http://personal.cs.surrey.ac.uk/personal/pg/A.Razavi/ppna/

arise at any stage during execution. These may be due to some service being unavailable (service failure or traffic bottleneck on the local platform) or some participant being temporarily disconnected (recall chapter 5) due to network failure. We have seen that a long-running transaction should either complete successfully or not take place at all. So as it has been shown, in the event of a failure, previous parts of the transaction that have already taken place should be 'undone' or be compensated for. We will analyse this behaviour, after explaining the execution script of the transaction.

6.3 Modelling the service deployment of a transaction in a distributed coordination orchestration

Based on definition, a distributed system is a collection of individual computing devices that can communicate with each other (Attiya & Welch, 2004). Conventionally in a distributed environment, each processor has its own semi-independent agenda, but for various reasons, including sharing of resources, availability, and fault tolerance, processors need to coordinate their actions. This may cause some algorithmic complexity which has been considered as the main difference with the parallel computing (Kshemkalyani & Singhal, 2008) and (Attiya & Welch, 2004).

Digital Ecosystems can be considered as a paradigm for distributing computing, as it promises on one side loose coupling between processors/participants (Chang & West, 2006b) which warranties the local autonomy (Razavi, Moschoyiannis, & Krause, 2009), and in the other side sustainability and balance (Nachira et al., 2007). (Boley & Chang, 2007) offers the usability of the system in the real world. Generally three challenges have been discussed for distributed systems: '*Asynchrony*', '*Limited local knowledge*' and '*Failures*' (Tanenbaum & Steen, 2008), (Attiya & Welch, 2004).

'*Asynchrony*'; The absolute and even relative times at which events take place cannot always be known precisely. This is the main reason for the current complexity in our interaction model (chapter 4). In terms of consistency of a transaction, the only measurement for success of a transaction is based on modelling the chains of events and sharing of resources based on access rights in a canonical order and avoidance of behaviours which violate this (recall 4.3 and 4.4). In contrast with conventional models we could not consider a global synchroniser (which needs centralised synchronisation).

In terms of '*Limited local knowledge*', as each computing entity can only be aware of information that is acquires, it has only a local view of the global situation. For obtaining this concept and facilitating the balance of the environment, we have considered two repositories (chapter 3) and designed a mechanism for link replication to increase the sustainability (chapter 8 and chapter 5). For solving unexpected '*Failures*' in a digital ecosystem, we have designed a distributed recovery model (chapter 5), which even tries to cover unpredicted failures (recall 5.4) and provides a mechanism for reducing the cost of recovery (recall 5.3).

In the rest of this chapter, we provide an analysis of the patterns of behaviour of such a

model and show its behaviours in different situations; from general service deployment, to reaction to failures and forward recovery.

6.3.1 DISTRIBUTED EVENT MODELLING

As the distributed system is asynchronous, there is no fixed upper bound on how long it takes for a message to be delivered or how much time elapses between successive steps of a processor (Tanenbaum & Steen, 2008), (Attiya & Welch, 2004), (Kshemkalyani & Singhal, 2008). As a result of this assumption, the modelling of the distributed algorithm should be independent of any particular timing parameter.

Formally a system (or algorithm) consists of *n* processors p_0, ..., p_{n-1}; *i* is the *index* of processor p_i. Each *processor*p_i is modelled as a state machine with state set Q_i. The processor is equivalent to a digital ecosystem participant and identified with a particular node in the topology graph. Furthermore each processor p_i can send message or receive message by doing so it can affect or being effected by other processors. Based on this a *configuration* has been defined; a *configuration* is a vector $C = (q_0, \dots, q_{n-1})$ where q_i is a state of p_i. The states of processor can change this can be effect the transmission channel (send messages to other processor) or being affected by the transmission channel (receiving messages from other processor). An *initial configuration* is a vector $(q_0, \dots, q_{n-1})$ such that where q_i is an initial state of p_i; which means each processor is in an initial state. Occurrences in a system are modelled as events, when events can be *computational event* (representing a computational step of processor p_i, when the transitional function is applied to its current accessible state), or delivery event (representing a delivery message from processor p_i to processor p_j).

A well-known concept in such modelling is that of '*execution segment*' (Attiya & Welch, 2004); an execution segment $\propto$ of a asynchronous message-passing system is a sequence of the following form:

$$C_{0,1}, C_{1,2}, C_{2,3}, \dots$$

Where each C_k is a configuration and each $_k$ is an event. If $\propto$ is finite then it must end in a configuration. An execution is an execution segment$C_{0,1}, C_{1,2}, C_{2,3}, \dots$, where C_0 is an initial configuration. A *schedule* (or *history*) has been associated to each execution which is the sequence of events in the execution, that is, $_{1,2,3}, \dots$. Conceptually the ideal situation is to have an *admissible schedule*; that is the result of an *admissible execution.* The admissible execution happens when none of the processors fail, they have an infinite number of computational events and the transmission channel does not fail to send any messages.

As a Digital Ecosystem is a service oriented environment and insist on loosely coupled binding between participants, the computational events are deployment of services and for reaching an admissible execution, we have focused on our log-based mechanism (recall chapter 4). For detecting, avoiding and recovering failures without being involved in an infinite computational model, the lock mechanism has been combined with the consistency model to design a recovery mechanism (recall chapter 5). For analysing such environment,

we use a customised subset of the conventional distributed event modelling, introduced by Moschoyiannis (Moschoyiannis et al., 2008).

6.3.2 TRANSACTION SCRIPT

In Appendix 1, we describe a formal language for pattern behaviour of our long-running transactions that allows to determine the patterns of interaction the underlying service invocations should follow in order to guarantee a successful outcome.

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. 6-4. 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 $\mu : C \rightarrow \phi(M)$ and require that $\bigcup_{i \in C} \mu(i) \subseteq M$.

As can be seen in Fig. 6-3, 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 (Shields, 1997) 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 $\underline{v}: C \rightarrow M^*$ 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, transaction vectors are *n*-tuples of sequences where each coordinate corresponds to a coordinator component in the transaction (hence, *n* is the number of leaves) and contains a finite sequence of actions that have occurred on (coordinator of) 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.

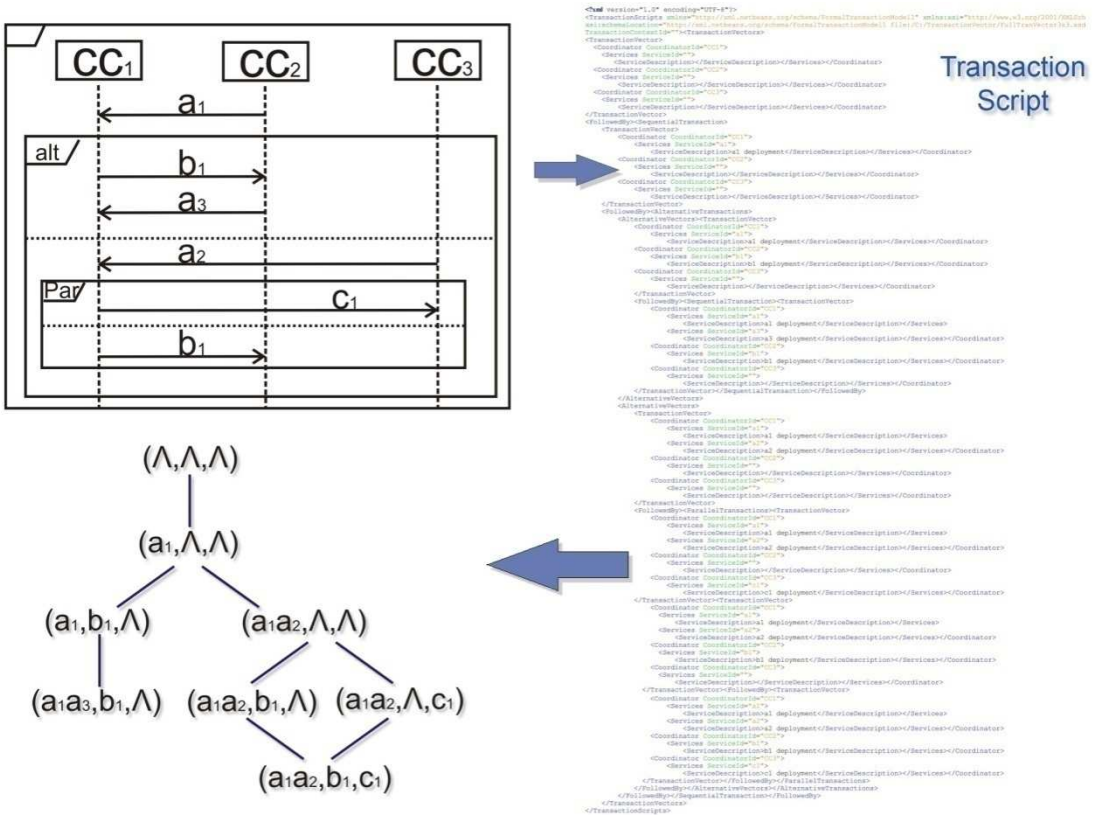

FIG. 6-4 TRANSACTION SCRIPT[9]

The idea is that the particular subset of transaction vectors, for a given transaction, expresses the ordering constraints necessary in the corresponding orchestration of the underlying services. For instance, in the transaction of Fig. 6-4, following action a1, there is a choice between b1 on the coordinator component CC2 and a2 on CC1. Whenever b1 happens, it is followed by a3 on component CC1 whereas a2 is followed by c1 and b1 which happen concurrently on CC3 and CC2, respectively. Notice that the order structure in case of concurrent actions (as in b1 and c1 here) exhibits the characteristic structure of a finite lattice.This means that after the behaviour described by vector $\underline{u}$ = (a1a2, Λ, Λ) both independent action vectors α1 and α2 take place, resulting in the behaviour described by vector v = (a1a2, b1, c1), which is their least upper bound.

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

$$A_T = \{\underline{a} \in V_T \setminus \{\underline{\Lambda}_T\} : l \in L \Rightarrow |\underline{a}(l)| \leq 1\}$$

where | x | denotes the length of sequence x. We refer to elements of A_T as *column vectors*.

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

[9] The XML schema is symbolic and can be checked from Fig 6-5 or downloaded from http://personal.cs.surrey.ac.uk/personal/pg/A.Razavi/ppna/

example, the vector (*a1*, Λ, Λ) represents the occurrence of an action *a1* on the service 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. Let us establish some notation. If x and z are sequences, we write *x*.*z* for the concatenation of *x* and *z*. As is well known this operation on sequences is associative with identity Λ, where Λ denotes the empty sequence. We also have a partial order on sequences given by $x \leq z$ if and only if there exists a sequence *y* such that *x*.*y* = *z*, and this partial order has a bottom element Λ.

We may now lift these well-known operations on sequences onto transaction vectors. This is done formally in the following definition.

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

- $\underline{u}.\underline{v}$ to be the unique vector $\underline{w}$ such that $\underline{w}(l) = \underline{u}(l).\underline{v}(l)$, for each $l \in L$ (*concatenation*)
- $\underline{u} \leq \underline{v}$ iff $\underline{u}(l) \leq \underline{v}(l)$, for each $l \in L$ (*prefix ordering*)
- *glb*($\underline{u},\underline{v}$) to be the vector $\underline{w}$ such that $\underline{w}(l)=\min(\underline{u}(l),\underline{v}(l))$, for each $l \in L$
- *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 $l \in L$
- 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,

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

The ordering amongst vectors is defined in terms of the usual prefix ordering operation on sequences appearing on their coordinates. For example, (*a1*, *b2*, Λ) ≤ (*a1a3*, *b2*, Λ) since *a1* ≤ *a1a3* and *b2* ≤ *b2* and Λ ≤ Λ. 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 *glb()* and *lub()* give the greatest lower bound and the least upper bound, respectively of $\underline{u}, \underline{v} \in V_T$, in the usual sense of lattices and domain theory (Davey & Priestley, 1990). As we will see, these operations are central to the treatment of concurrency in our approach.

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.

In terms of our notation it is appropriate to say that the independence relation on the set of actions *A* of a transaction equates all, and only those, sequences over $\mu(l)$, for each $l \in L$, which differ in the order of adjacent and independent actions. Note that when the independence relation is empty in the sets $\mu(l)$, for each $l \in L$, no actions can be concurrent in the corresponding sequences $\mu(l)^*$, for each $l \in L$, which amounts to our understanding of sequential transaction processing systems.

Drawing upon the extension of the independence relation ι to *behaviour vectors* in (Shields, 1997), 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

$$\underline{u} \, ind \, \underline{v} \Leftrightarrow \forall\, l \in L : \underline{u}(l) > \Lambda \Rightarrow \underline{v}(l) = \Lambda$$

This definition says that two transaction vectors are independent if the behaviours they describe concern distinct services (correspond to activation on different leaves of the corresponding transaction tree). This means that the behaviours described by u and v may occur independently.

The additional structure provided by the notion of independence allows to go round the lozenge once and always end up with the behaviour in which both actions happened concurrently (this can be seen in (Moschoyiannis et al., 2008).As we will see in the next section, this also applies to going backwards and allows compensating concurrently for concurrent forward actions.

The order structure of the transaction language determines the pattern the underlying service interactions between the coordinator components in the transaction should follow. Starting with the empty vector, and following the pattern of Fig. 6-4, each subsequent action (or concurrent actions) take place in going forward until the transaction as a whole terminates successfully. In (Moschoyiannis et al., 2008), we have provided a schema for deriving the XML description of the order structure of a transaction language (Fig 6-5). These so-called transaction scriptsdescribe the interactions between different coordinator components (recall Fig. 6-2) and determine the order in which the underlying services need to be deployed. Fig. 6-4 shows the transaction script generated from the vector-based behavioural description of the transaction in our approach and its XML schema has been shown in Fig 6-5.

Source Schema Design

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Document   : Transaction Script xml Schema 3 coordinators vector
    Created on : 06 March 2009, 16:07
    Author     : A. Razavi
    Description: Prepared for PhD Thesis
-->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://xml.netbeans.org/schema/FormalTransactionModel1"
            xmlns:tns="http://xml.netbeans.org/schema/FormalTransactionModel1"
            elementFormDefault="qualified">
    <xsd:complexType name="Vectors">
        <xsd:sequence>
            <xsd:element name="TransactionVector" type="tns:TransactionVectorType"/>
            <xsd:element name="FollowedBy" minOccurs="0" type="tns:CompositionType"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="CompositionType">
        <xsd:choice>
            <xsd:element name="SequentialTransaction" type="tns:SequentialComposition"/>
            <xsd:element name="AlternativeTransactions" type="tns:AlternativeComposition"/>
            <xsd:element name="ParallelTransactions" type="tns:ParallelComposition"/>
        </xsd:choice>
    </xsd:complexType>
    <xsd:complexType name="SequentialComposition">
        <xsd:sequence>
            <xsd:element name="TransactionVector" type="tns:TransactionVectorType"/>
            <xsd:element name="FollowedBy" minOccurs="0" type="tns:CompositionType"/>
        </xsd:sequence>
    </xsd:complexType>

     <xsd:complexType name="AlternativeComposition">
        <xsd:sequence>
            <xsd:element name="TransactionVector" type="tns:Vectors" minOccurs="2" maxOccurs="3"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="ParallelComposition">
        <xsd:sequence>
            <xsd:element name="TransactionVector" type="tns:TransactionVectorType" minOccurs="2" maxOccurs="3"/>
            <xsd:element name="FollowedBy" type="tns:Vectors"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="TransactionVectorType">
        <xsd:sequence>
            <xsd:element name="Coordinator" type="tns:AccessPoint" maxOccurs="3" minOccurs="3"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="AccessPoint">
        <xsd:sequence>
            <xsd:element name="DeServices" type="tns:Actions" maxOccurs="unbounded" minOccurs="0"/>
            <xsd:element name="Id" type="tns:AcessPointId"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Actions">
        <xsd:sequence>
            <xsd:element name="ServiceDescription" type="xsd:string"/>
        </xsd:sequence>
        <xsd:attribute name="Id" type="xsd:string" use="required"/>
    </xsd:complexType>
    <xsd:complexType name="TransactionsVectorId">
        <xsd:sequence>
            <xsd:element name="InitiatorId" type="tns:AcessPointId"/>
            <xsd:element name="LocalId" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="TransactionScripts">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="TransactionVectors" type="tns:Vectors"/>
                <xsd:element name="TransactionId" type="tns:TransactionsVectorId"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:simpleType name="AcessPointId">
        <xsd:restriction base="xsd:string"/>
    </xsd:simpleType>
</xsd:schema>
```

FIG. 6-5 TRANSACTION SCRIPT SCHEMA

Transaction scripts reflect the corresponding transaction languages and hence describe

the dependencies between services of different participants' coordinator components, in terms of the orderings of the underlying service invocations. This means that when the scripts are parsed they provide the full *transaction history*, resulting from the actual deployment of the transaction, but based on the associated formal semantics of the transaction. The XML schemas and further details on transaction scripts can be found (downloaded) following (Razavi & Moschoyiannis, 2008).

6.4 Failures, Recovery and Pattern behaviours

So far we have described the use of vector languages (full details in appendix 1) in determining the patterns the underlying service compositions should follow in the course of a long-running transaction. In particular, by exploiting the order-theoretic structure of transaction vectors we have shown how all possible interaction scenarios can be captured, and analysed prior to deployment, in determining the set of allowed sequences of actions. In transactional terms, this allows to determine the history of the transaction based on the transaction semantics (Gray & Reuter, 1993). In practical scenarios, applications in a transactional environment should offer recoverability and consistency. These two central aspects are addressed in the following sections.

6.4.1 Execution History

In our approach, the transaction history is captured in the order structure of the corresponding transaction language which is used to express forward behaviour and is reflected in the derived transaction scripts. As shown in Fig. 6-6 a transaction language which includes alternative actions will have different allowed *execution paths*. These start from the empty vector and lead to (one of) the largest vectors in the language. The largest vectors describe maximal behaviour of the transaction, in the sense that they do not describe an earlier part of behaviour than any other vector does. In the transaction language of Fig. 6-4there are two maximal vectors, namely $\underline{v_1}$= (a1a3, b1, Λ) and $\underline{v_2}$= (a1a2, b1, c1). This means that one allowed sequence of execution is a1 →b1 →a3 and the other is a1 →a2 →(b1 concurrently with c1). Note that both allowed sequences end up in a maximal vector which corresponds to the behaviour exhibited when the transaction executes successfully until it terminates. Thus, in both cases the transaction produces a consistent state. In other words, transaction consistency is attained by reaching a maximal vector in the transaction language.

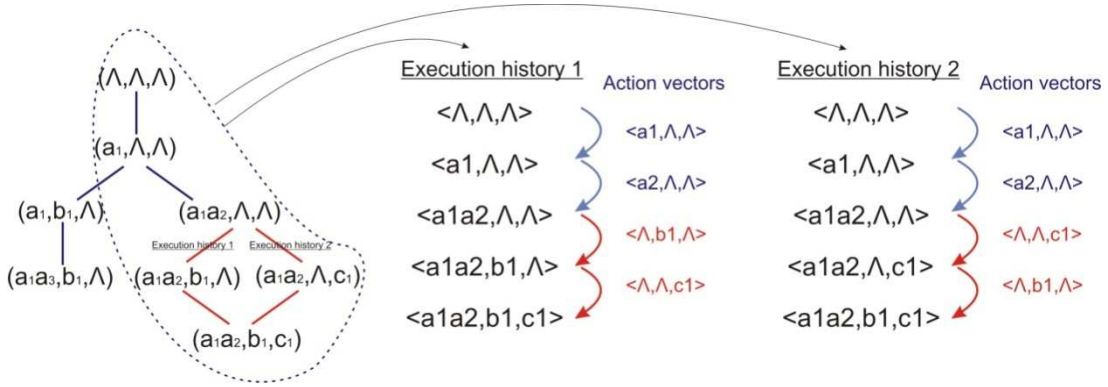

FIG. 6-6 EXECUTION HISTORY

When the transaction is actually deployed, then only one of the allowed sequences of actions can occur. In our example, this would depend on how the choice between a2 and b1, after a1 has occurred, is resolved. Fig. 6-6 shows the case where a2 occurred after a1, and hence the right branch of the Hasse diagram with the transaction language was actually deployed. It can be seen that each vector in turn is obtained by coordinate-wise concatenation with the appropriate action vector, according to the allowed sequence of actions along the particular execution path.

This distinction between the set of all allowed sequences of actions and the allowed sequence of actions that actually takes place when the transaction is deployed is important when considering compensations. Naturally, when a failure makes further progress impossible, we would only want to compensate for actions that actually happened up to that point and not for every possible action in the transaction. We will refer to the actual path of execution during a run of the transaction as the *execution history* of the particular deployment of the transaction. In Fig. 6-6 it appears there are two execution histories but these are the result of concurrency (between b1 and c1) and therefore, they are equivalent. This is because the series of concatenations with action vectors differ only in the order of the independent action vectors $\alpha 1 = (\Lambda, b1, \Lambda)$ and $\alpha 2 = (\Lambda, \Lambda, c1)$, and consequently they correspond to the same allowed sequence of actions.

Hence, the notion of independence is what allows us to identify equivalent behaviours in the presence of concurrency, something that is not possible when adopting an interleaving model, as done in (Butler, Hoare, & Ferreira, 2005). The benefit of being able to identify equivalent execution histories can perhaps be seen most clearly when it comes to compensation in transaction recovery, where as we will see it is only required to compensate once for all equivalent execution histories.

6.4.2 RECOVERY AND ROLLBACK

In this section we examine the formal approach to coordinating long-running transactions to handle compensations. This is to address occasions where some failure happens mid-way through execution in which case we need to have a way to express compensating sequences

of actions that need to take place in going 'backwards' while effectively undoing all previously successful forward actions.

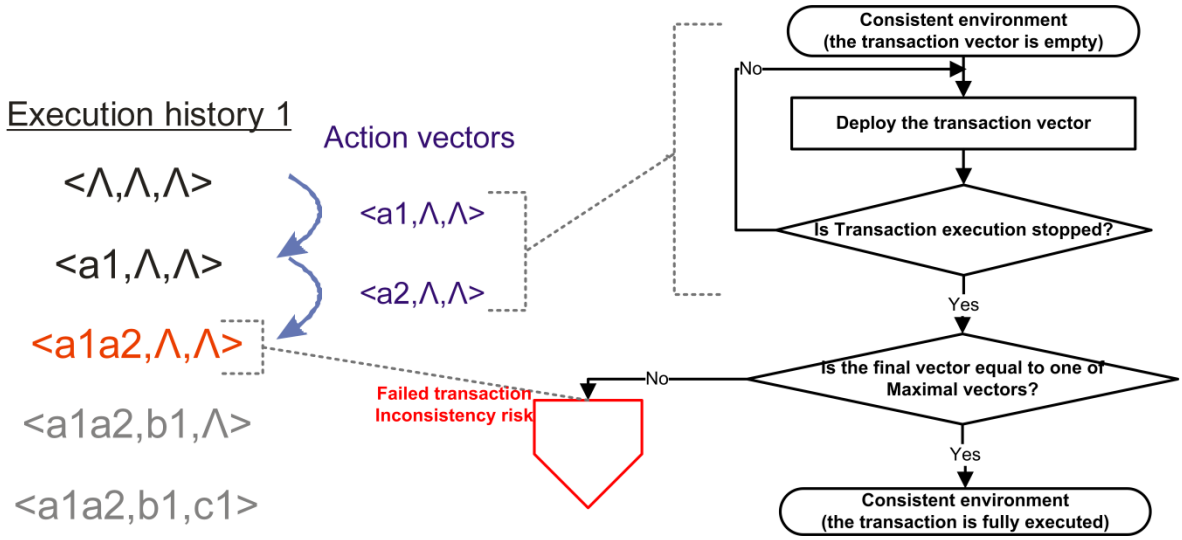

FIG. 6-7 FAILURE IN A TRANSACTION

When the deployment of a transaction stops before it reaches the maximal vector, there is a failure (transaction can not reach the commit and deploys all of its services), this has been shown in Fig 6-7. Based on atomicity, the transaction should be aborted. A cancellation operation on vectors is central to the handling of abortion (compensations) in the vector model. This is done by considering the usual operation of right-cancellation on sequences and then lifting it onto vectors (by applying it coordinate-wise) as follows.

Right-cancellation. Let $\underline{u}$, $\underline{v} \in V$. If $\underline{u} \leq \underline{v}$, then we define $\underline{u}/\underline{v}$ to be the unique element z such than $\underline{u}.\underline{z}=\underline{v}$.

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 dictates that what takes a transaction vector and extends it to its successors is an action vector (e.g. service invocation between different participants) of the transaction. It means that vectors are built by a series of concatenations with action vectors, and this is used to model forward actions, and it also means that successive applications of right-cancellations can be used to express the series of compensating actions required whenever some failure occurs during execution.

In other words, our approach uses (coordinate-wise) concatenation of vectors to model the occurrence of an action by

$$\underline{u}.\underline{\alpha}=\underline{v}$$

and, instead of introducing a separate notation and associated semantics for cancelling forward actions, we use right-cancellation to express the compensating action needed at each point in returning to the initial configuration.

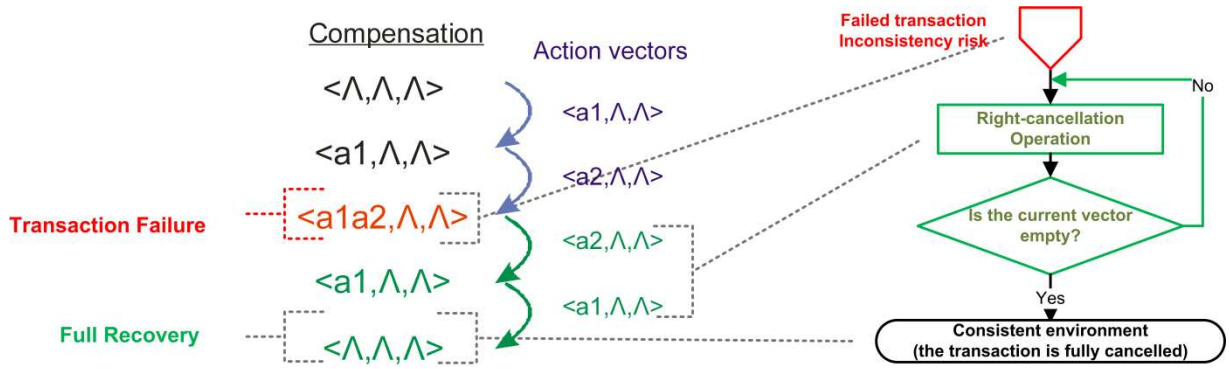

FIG. 6-8 TRANSACTION RECOVERY

Fig. 6-8 shows the case where a failure occurs after service a2 of component CC2 has been invoked. This means that it is no longer possible for the transaction to produce a consistent state by reaching the maximal vector (a1a2, b1, c1), as dictated by the allowed sequence of actions in its execution history. Consequently, the transaction needs to be recovered and this implies returning to the (consistent) state the system was in before the transaction started. This is represented by the empty vector. In returning to empty vector, all previously successful forward actions need to be compensated for. This is done by successive applications of the '/' operation given earlier.

The first application of right-cancellation moves the transaction back to (a1,Λ, Λ) and the next to (Λ, Λ, Λ). Hence, the system is returned to the empty vector which is now the (only) consistent state. In this case we talk about full recovery of the transaction. It might be worth noting that in the case where failure happened after (a1a2, b1, c1) our apporach would allow for the concurrent forward actions b1 and c1 to be compensated concurrently. This is because the vector (a1a2, b1, c1) has two immediate predecesors, and hence the independent action vectors α1 = (Λ, b1, Λ) and α2 = (Λ, Λ, c1) that extended them to (a1a2, b1, c1) would be called to be compensated for concurrently. In other words,

$$(\underline{v}/\ \underline{\alpha}_1)/\ \underline{\alpha}_2 = (a1a2,\Lambda,c1)/\underline{\alpha}_2 = (a1a2, \Lambda, \Lambda)$$

And also

$$(\underline{v}/\ \underline{\alpha}_2)/\ \underline{\alpha}_1 = (a1a2,b1,\Lambda)/\underline{\alpha}_1 = (a1a2, \Lambda, \Lambda)$$

6.4.3 FORWARD RECOVERY

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

One way to do this is to design transactions with a number of alternative scenarios of execution; in other words, allow for multiple execution histories in the corresponding

transaction script. In such cases, our approach comes with the provision for forward recovery which is a mechanism for avoiding full recovery. The aim is during compensation to explore whether there is any possibility for successfully terminating the transaction following a different execution history to the one originally deployed, instead of compensating for the whole execution history.

This is possible in our approach because all the execution histories are captured in the transaction language. Hence, in recovering a given execution history which failed, after the next forward action(s) is compensated for we check whether the resulting vector is part of a different execution history. If this is the case, then we attempt to go forward by performing the concatenations in accordance with the allowed sequence of actions in that execution history, as described in Section 4.2.

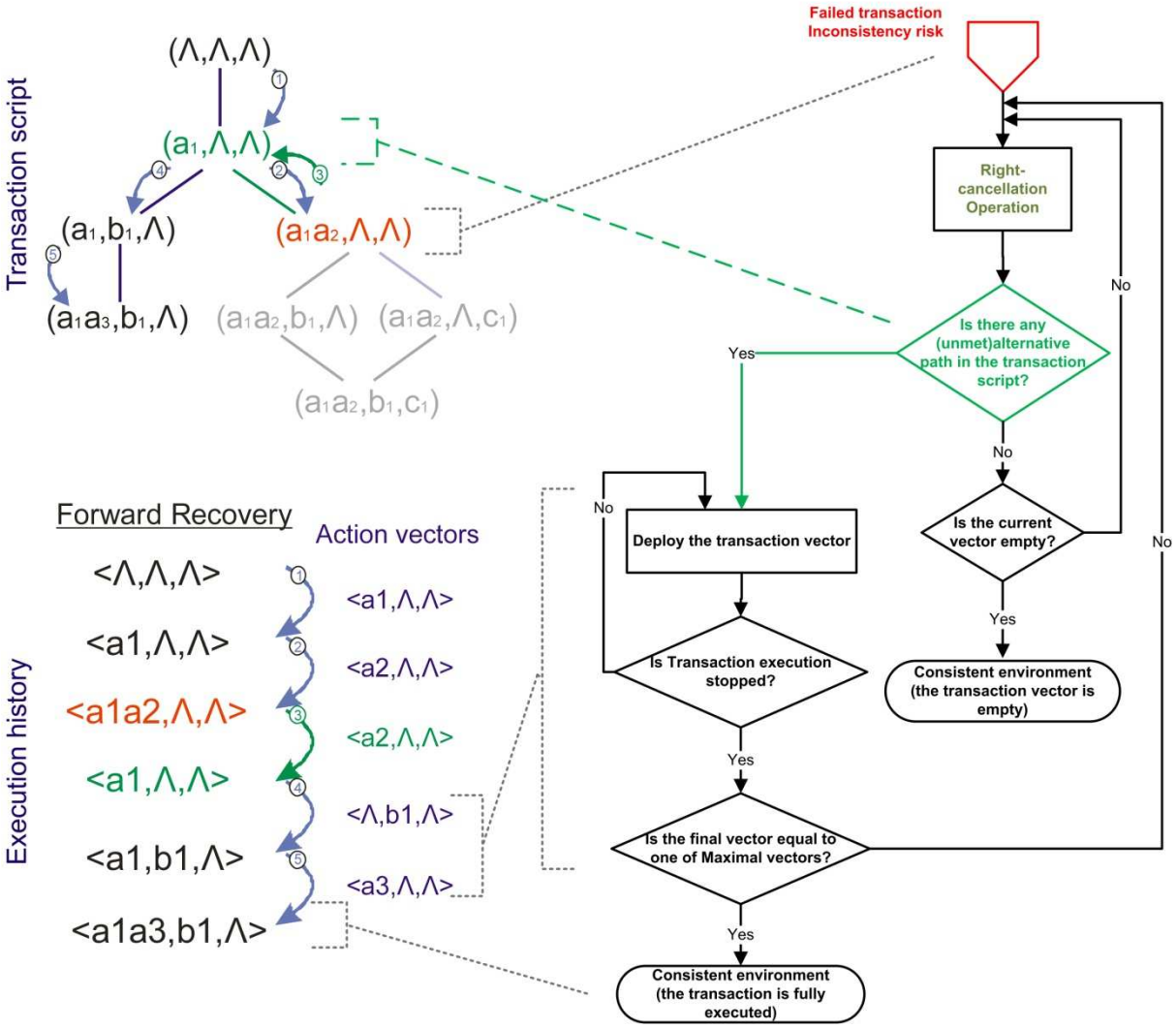

FIG. 6-9 FORWARD RECOVERY

As shown in Fig. 6-9, in going backwards, and while cancelling out one action (or a set of concurrent actions) at a time, we look each time whether there is an alternative path from the vector we arrived on (after applying the compensating action(s)) leading to a maximal vector.

6.5 COORDINATION AND TRANSACTION

In this part we extend our approach to coordinating long-running transactions to handle compensations. This is to address occasions where some failure happens mid-way through execution in which case we need to have a way to express compensating sequences of actions that need to take place in going '*backwards*' while effectively undoing all previously successful forward actions the operation per coordinator is called *right-cancellation,* which it is equivalent to call a cancelation for a service [formal definition of this can be found in (Moschoyiannis et al., 2008) and (Razavi et al., 2007b)]. Fig. 6 shows the case where a failure occurs after service a2 of component CC2 has been invoked. This means that it is no longer possible for the transaction to produce a consistent state by reaching the maximal vector (a1a2, b1, c1), as dictated by the allowed sequence of actions in its execution history. Consequently, the transaction needs to be recovered and this implies returning to the (consistent) state the system was in before the transaction started.

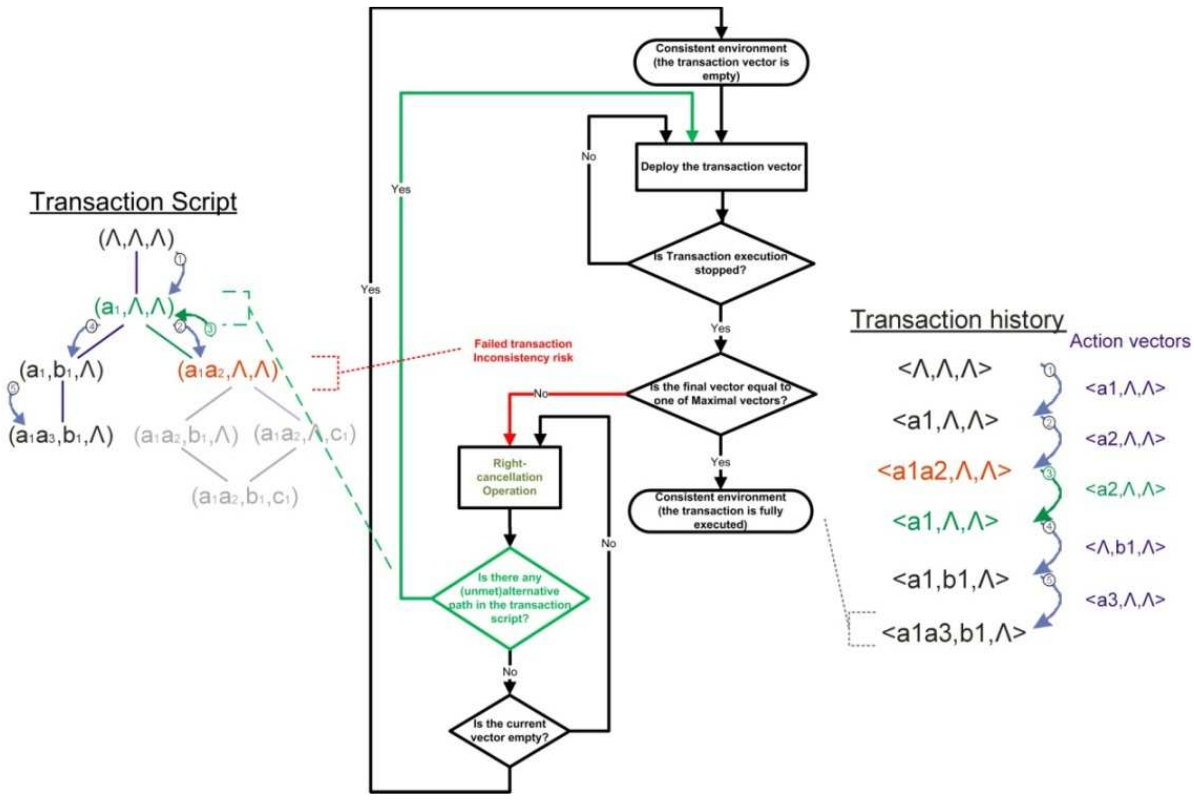

FIG. 6-10 TRANSACTION LIFE-CYCLE

We have seen how right-cancellation can be applied to generate compensating sequences of actions for the full recovery of a transaction. Full recovery however, can be costly in terms of resources, delays, business relations and so on. Additionally, in a highly transactional environment dependencies may also exist across transactions so the effect of

a recovered transaction may be magnified. For this reason it is desirable to avoid full recovery wherever possible. One way to do this is to design transactions with a number of alternative scenarios of execution; in other words, allow for multiple execution histories in the corresponding transaction script. In such cases, our approach comes with the provision for forward recovery which is a mechanism for avoiding full recovery. The aim is during compensation to explore whether there is any possibility for successfully terminating the transaction following a different execution history to the one originally deployed, instead of compensating for the whole execution history.

This is possible in our approach because all the execution histories are captured in the transaction language. Hence, in recovering a given execution history which failed, after the next forward action(s) is compensated for we check whether the resulting vector is part of a different execution history. If this is the case, then we attempt to go forward by performing the execution in accordance with the allowed sequence of actions in that execution history. As shown in green box, Fig. 6, in going backwards, and while cancelling out one action (or a set of concurrent actions) at a time, we look each time whether there is an alternative path from the vector we arrived on (after applying the compensating action(s)) leading to a maximal vector.

6.6 Results and comparison

In this chapter, we have described a formal model for analysing patterns of behaviour of Digital Ecosystem long-running transactions. The tuples-based behavioural description allows to capture the sequences of forward and compensating actions of each participant and at every stage in the course of execution of a transaction. Our formal approach to transaction semantics drives the recovery mechanism necessary to deal with failures during transaction deployment in a way that asserts transaction consistency.

In one hand, in contrast with conventional pattern behaviour analysis such as BAPC or BACC (chapter 3), this formal model for analysing patterns of behaviour of Digital Ecosystem long-running transactions, shows the behaviour of distributed coordinators of transaction model, rather than a linear analysis based on 2 phase commit protocol. On the other hand, it proves the Transaction model proposed in chapter 4 and 5 models a loosely coupled interaction of participants (because there is not any scenario for relying on local states of any services of participants) and as a result does not suffer violating the local autonomy of participants.

On the other hand, unlike other approaches to transaction semantics, such as (Butler et al., 2005), (Gray, 1992), concurrent actions are handled via a notion of independence and as a result concurrent forward actions are compensated for concurrently in our approach. We have shown how transaction languages capture the history of a transaction and this not only allows for the full recovery of a transaction in case of a failure, but also to attempt forward recovery wherever possible and avoid the sometimes costly abortion of the whole transaction. We have also highlighted the implementation aspects of the theory presented

and further details can be downloaded following (Razavi & Moschoyiannis, 2008).

We have been concerned with transactions that involve the execution of our transaction model (chapter 4) and have focused on the orderings of the underlying service interactions. The specifications of this ordering, can be found in the variety of service compositions in a Digital Ecosystem (recall chapter 3 and 4). Another aspect concerns the translation of business requirements into the transaction tree and the UML model, which are currently our starting point. Preliminary analysis shows that the use of SBVR could be harnessed in expressing business logic, specifically in terms of (orderings of) the deployment of services and their resources. This is another possible direction for future work as it would make the transactional framework accessible to a wider audience and particularly smaller businesses.

7 STABLE DIGITAL ECOSYSTEM NETWORK STRUCTURE

So far, we have provided a loosely coupled interaction model in terms of business transactions; the recovery mechanism has been developed and can reduce the cost of recovering failures. For offering "*Balance*" which, like our recovery mechanism, is "*Self-organising*" (Chang & West, 2006a), we need to provide a virtualisation for customising the business network.

The aim of this chapter is to provide the stability of the environment (network) and facilitate e-business transactions between Small and Medium Enterprises (SMEs), in a way that respects their local autonomy, within a digital ecosystem. For this purpose, we distinguish transactions from services (and service providers) by considering Virtual Private Transaction Networks (VPTNs) and Virtual Service Networks (VSNs). These two virtual levels are optimised individually and in respect to each other. The effect of one on the other, can supply us with stability, failure resistance and small-world characteristics on one hand and durability, consistency and sustainability on the other hand. The proposed network design has a dynamic topology that adapts itself to changes in business models and availability of SMEs, and reflects the highly dynamic nature of a digital ecosystem.

A number of different views exist on the development of sustainable digital ecosystems, from that of a collaborative environment for business activities, to software infrastructure for open e-business transactions, to the continuous creation of new business model categories and instants. All these different facets can challenge the current infrastructure of our software world. At the same time, the telecoms industry is moving towards the Next Generation Networks (NGNs), and this comes with yet another view of services and applications; the so-called *Next Generation Services* (NGSs) (Daho & Simoni, 2006). Our approach is trying to leverage these developments in creating a business environment which supports dynamic contexts for distributed long-lived transactions proposed for digital ecosystems in chapter 4, 5 and 6. This chapter contains material that has been published in (Razavi, Moschoyiannis, & Krause, 2007e), (Razavi et al., 2008a) and (Razavi, Moschoyiannis, & Krause, 2008b).

Current models which provide self-management capabilities at the service level (Martini, Baroncelli, & Castoldi, 2005), (Kristiansen, Hansen, & Licciardi, 2008), (Sahai et al., 2002), (Li & Mohapatra, 2004) and Quality of Service (QoS) at the virtualization levels (Daho & Simoni, 2006) can be seen to satisfy the primary requirements of such a "Digital Ecosystem" (DE) environment. However, there is little evidence that they can insulate the collaborative business activities for long-lived transactions from failure – one of the most important requirements for a DE for business – in the face of the highly dynamic business models of SMEs which cannot be expected to provide the necessary permanent platforms for a connected network.

As a result, keeping the necessary information for coordinating long-lived transactions (Razavi et al., 2007c), can be rather challenging, but also the probability of fragmentation in the network cannot be averted. This can have severe consequences in a business environment, since fragmentation in the network of course directly affects the number of

failed transactions (Razavi et al., 2007a).

In this chapter we propose a model that satisfies these requirements of a business network for SMEs in the context of digital ecosystems. We outline the main characteristics of such a business network and conventional solution in the first section. The second section is focused on the primary solution. Virtual Super peers have been proposed at the end of the next section and then discussed more fully in the third and fourth sections.

7.1 VPTNs INTERCONNECTIVITY AND NETWORK CONNECTIVITY

The purpose of a business network is to enable networked organisations to engage in distributed business transactions (Singh & Huhns, 2005), (Razavi et al., 2006), (Dini et al., 2008) that realise their core business activities. As we have seen in the previous chapters in terms of business transactions this means stronger interconnectivities for VPTNs. This is achieved when a transaction's participants can avoid failure at the supporting network level and/or alternative paths are reachable whenever service unavailability is experienced or failure occurs in one of the participants in alternative scenarios. This may lead us to increasing the local connectivity in each transaction, but the effect and side effect of changes will direct us to take into account a measurement for stability and apply any increases in terms of this measurement. This provides a dynamic and extensible method for creating a stable Digital Ecosystem that emerges through the long-running transactions that correspond to business activities between participating organisations.

Meanwhile providing a level of virtualization for applying DE conceptual foundations is necessary (recall 3.2 for primary discussions). One of the important characteristics of Digital Ecosystems is dynamicity. By their very nature, SMEs are versatile and their business model needs to be refined or adapted on a regular basis. This can transform the business domain or the nature of a business over time. Furthermore, the direct regional effect on business activity timetables (and availability of SMEs) is another property that needs to be taken into account in a digital ecosystem. These are some of the factors that make it necessary for the business network topology of such an environment to be able to adapt itself dynamically (dynamic topology). It should also be noted that loose coupling is another important characteristic of DE, as SMEs need to preserve their local autonomy, which poses further challenges in covering the above requirements.

For this reason the virtualization in our model is slightly different from that found in the general proposed models (such as (Daho & Simoni, 2006), (Martini et al., 2005), (Kristiansen et al., 2008), (Sahai et al., 2002), (Li & Mohapatra, 2004)). The business activities are at the top level of the model; business activities in terms of transactions which should be distributed (if we are to preserve the local autonomy of SMEs) and recoverable (self-healing). In previous chapters, we have introduced a transaction model with such characteristics whose design makes use of the inherent diversity of a digital ecosystem. The result of the interaction between participants of a transaction, which comes down to the composition of the corresponding services, provides a virtual connection which is useful for

the design of the underlying network. This virtual network, shown as the first conceptual level of virtualisation in Fig 7-1, is private between transaction participants, and hence the term *Virtual Private Transaction Network* (VPTN).

The second conceptual level of virtualisation is concerned with links between SMEs (service providers and actual participants of business activities) which provides the structural materials for a business activity (transaction). These structural materials are 'services'. We call such a virtual network a *Virtual Service Network* (VSN). Fig 7-1, shows the virtualisation levels of the DE business networks.

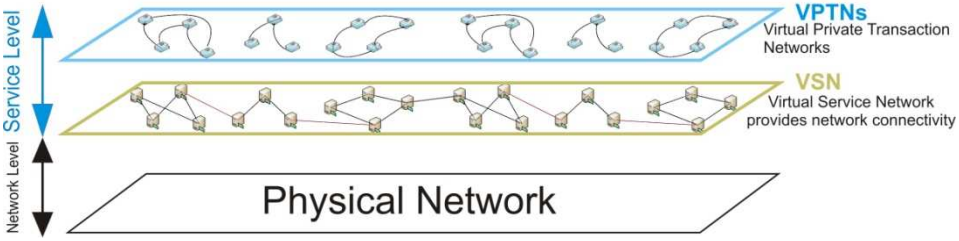

FIG. 7-1 VIRTUALIZATION IN DIGITAL BUSINESS ECOSYSTEMS

Even though SMEs may be engaging in more than one transaction at a time, a VPTN is typically a fragmented (island) network which connects the participants of a specific business activity. Therefore, nodes in VPTNs are typically in the same business domain (or strongly relevant business domains). For this reason, VPTNs potentially may improve the 'cluster coefficient' of the lower level (VSN) and in exchange VSNs can warranty the diversity for VPTNs (by providing reliable connectivity which makes alternative paths/scenarios feasible for a disconnected VPTN). This may protect business transactions against failure - using alternative paths and scenarios to avoid the costly abortions of long-running transactions is feasible as shown before and is referred to as forward recovery (more detail can be found in 5.3). It can be seen that one of the most important characteristics the VSNs should provide is 'connectivity' - ensuring that there is a network of interconnected nodes. In 3.2, we have provided more formal definition for the network of a digital ecosystem, which can now be modelled as a connected part of VSNs that are aggregations of the business activities taking place between the different partners create several virtual business networks. In the rest of this chapter we focus on the digital ecosystem network rather than using conventional VSNs metaphor.

7.1.1 LINK REPLICATION, AND CONNECTIVITY

Increasing the connectivity between participants of a transaction prevents certain types of failures in the transaction, predominantly those which are the result of the network disconnections. At the same time, alternative scenarios in terms of alternative service compositions (Chapter 3, section 1- SAt and PAt) rely on available connectivity between alternative paths (between different participants) in the network. This means increasing VPTN interconnectivity helps to provide a better chance for forward recovery (recall discussion in 5.3 and 6.4.3), and as a result avoid a full recovery even when some participants failed to provide their services. Before describing how VPTN interconnectivity is dealt with in our framework, we present the general mechanism for link replication in the

local software agent of each participant.

Normally the connections (links) to other participants of the digital ecosystem have been established by the '*global service repository*', where the address of other service providers (participants) and the description of their services have been kept. For inserting (or modifying) a new participant to the repository and its services, the 'web service information investor' component will be involved. For introducing the participant to another participant the 'web service promoter' will be used.

It can be seen that for increasing the connectivity we use three components of the component-based design of each participant. Figure 7-2 shows the relationships between these components of three participants (their software agents). Participant B ('*Agent B*'), receives all of the connections of Participant A ('*Agent A*') through its '*web service information investor*' when participant A ('*Agent A*') provides them through its '*web service promoter*'. Similarly, participant B ('*Agent B*') provides its connections (links to the other participants) to the '*web service information investor*' of participant C ('*Agent C*'). We call this procedure *Link Replication*. It is important to mention that it is possible to have *partial link replication* where there is no need to replicate or pass to other participants all of the connections of a given participant.

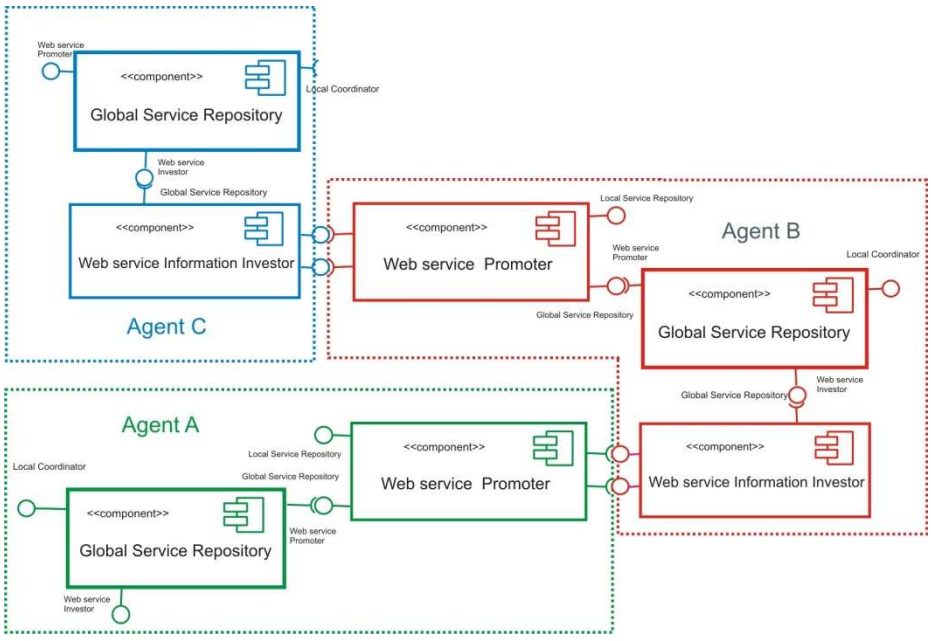

FIG. 7-2 LINK REPLICATION

7.1.2 FULLY CONNECTED VPTN AND DIGITAL ECOSYSTEM

As we mentioned in chapter 5 (5.1 and 5.4) and the beginning of this chapter, one of the significant risks for the VPTNs of a digital ecosystem (transactions of the network) is

disconnection between participants of a transaction which amounts to low connectivity inside the VPTNs. It seems the primary solution for the problem is to use the link replication procedure between the participants. By repeating link replication in each participant, within a limited time, all participants in a VPTN will be connected together. As a result we will have a fully connected VPTN, in which, if built based on transaction t and participant P_t each node will have $|P_t| - 1$ links.

Where this seems like an ideal solution for each VPTN the result can be devastating for the digital ecosystem and consequently the majority of transactions could be failed. As a digital ecosystem is a connected network through its transactions (chapter 3 section 2), the VPTNs have overlaps on some of the their participants (there are some intersections between different VPTNs' participants) and as mentioned in Chapter 4 section 1, studies show most business networks follow the power-low distribution degree (Barabási et al., 2000) which means:

The digital ecosystem relies on very few participants (nodes) to stay connected

And these small numbers of participants are involved in the majority of the transactions, i.e. these few participants will be in most VPTNs.

Now if we apply *link replication* for each VPTN ($VPTN_i(t_i, P_{t_i})$ where $VPTN_i$ is VPTN of transaction t_i and its participants are P_{t_i}), each participant in the VPTN will have

$$|P_{t_i}| - 1$$

Therefore a participant *R* which is involved in transactions

$$\{t_m, \cdots, t_n\}$$

Will have up to

$$\sum_{i=m}^{n} |P_{t_i}| - (m + n)$$

links.

Based on the second point above (point '**b**'), very few participants are involved in the majority of transactions. Therefore by applying *link replication* in this way this small number of participants will have a very large increase of links. This increases their traffic dramatically and it is highly probable they collapse as a result, which means a potentially large number of transactions will be failed. More importantly, based on the first point above (point '**a**'), as the digital ecosystem relies on them to stay connected, the whole digital ecosystem will be fragmented. Figure 18 shows this situation which his generally rather difficult for a network to recover from.

VPTNs interconnectivity and network connectivity

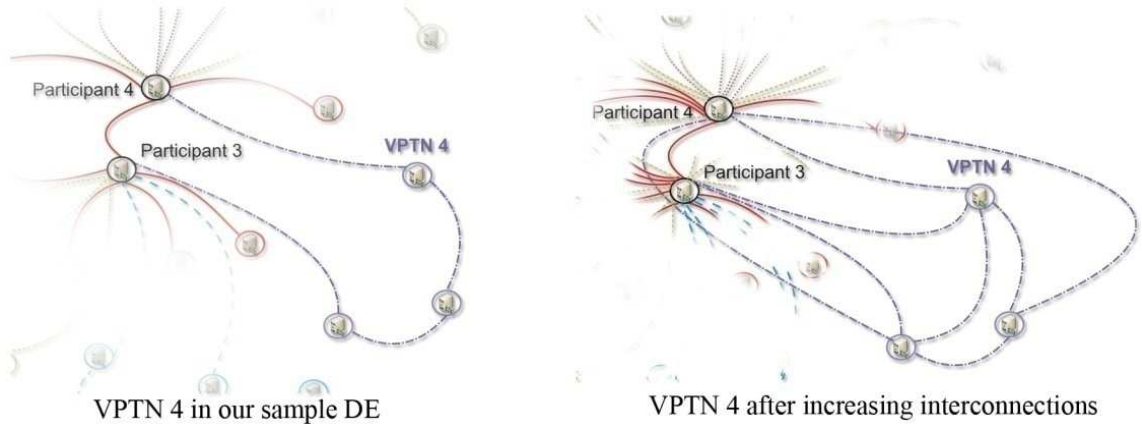

FIG. 7-3 FULLY DISCONNECTED VPTNS

On the left side of Fig 7-3 VPTN4 from the sample digital ecosystem presented in Figure 3-2, is shown, and on the right-side we can see the result of *link replication* on all participants of the VPTN4. As *link replication* in a similar way has been applied for all other VPTNs, participant 3 and participant 4 which are involved in several transactions face the large increases of links which can bring traffic complexity. While the *link replication* itself seems quite useful for increasing connectivity, the way and on which participant it is applied can be crucial for the general performance of the digital ecosystem (and even each VPTN).

Before investigating some measurements for a more careful application of *link replication* we try to review the other conventional approaches in network interactions.

7.1.3 CONVENTIONAL PEER- TO-PEER SOLUTIONS

The oldest solution to the problem of connectivity is to supply a powerful central coordinator that manages the whole network and keeps all information about all participants. However, this solution has the classic problem of *a single point of failure* as well as high cost for providing and maintaining the centralised unit. Note the cost increases as the number of nodes and associated network traffic increases.

A popular solution to the connectivity problem which is used in several P2P networks (e.g. (Yang & Garcia-Molina, 2002),(Yang & Garcia-Molina, 2003)) consists of introducing an extra layer to the network, the so-called *super peers*. These are essentially decentralised servers, each managing a significant part of the network (number of participants), and have strong links to each other. The primary necessity for having super peers is providing stable nodes which are online all of the time. This means super peers are expensive nodes with costly maintenance requirements. Furthermore, during peak time the pressure of high traffic can result in a bottleneck on super peer nodes. At off-peak times the powerful super peers will still need to be online and monitor the whole network, thus processing redundant data and producing overheads waste.

It should also be noted that the resources are used for facilitating network operation management tasks. When considering such a solution for a digital ecosystem environment involving SMEs, the question arises as to who is going to provide such nodes? Even if it were possible to find suitable SMEs willing to provide permanent nodes as super peers, these may change their business model and after some time may not find it useful to provide a permanent (and expensive) node anymore.

Perhaps even more importantly, the super peers solution results in a static topology for the network as these nodes are pre-selected and their role is pre-determined in the network. This is by no means satisfactory in a highly dynamic environment of a digital ecosystem where the idea is that the network topology changes continuously to adapt to its very usage and demands of the participating entities. The evolving nature of the DE is intended to reflect the congestion of network packages and nodes that change from time to time.

On the other hand, models which provide self-management capabilities at the service level (Martini et al., 2005), (Kristiansen et al., 2008), (Sahai et al., 2002), (Li & Mohapatra, 2004) and Quality of Service (QoS) at the virtualisation levels (Daho & Simoni, 2006) can be seen to be another extreme solution for Digital Ecosystem environments. But the network resistance against failure on the collaborative business activities for long-lived transactions from failures, (in the face of the highly dynamic business models of SMEs) has not been solved, which cannot be expected to provide the necessary permanent platforms for a connected network.

7.2 Towards a connected Digital Ecosystem

In this part, we try to provide a constructive solution for using the *link replication* mechanism to improve connectivity inside VPTNs and present the first step towards a stable digital ecosystem. In general, we can say the best candidates for *link replication* inside each VPTN and connecting VPTNs together are the most stable participants (nodes) in each VPTN. Connecting these participants and the *link replication* can be done by using the '*Global Service Repository*' of each candidate-participant from each VPTN (it has been shown in Fig. 7-3, Section 5.1).

However, we cannot warranty full stability of the network and still cannot avoid the occasional fragmentations. Even in the best case, this is still is dependent on each candidate-participant's availability and if the total online time of all stable nodes cannot cover the full '*active-time*' of the network, the network will collapse for some period of time, precisely that in which all candidates are not available. Therefore, first we try to introduce a measurement for node stability, and then use this measurement in finding the stable participants in each VPTN to cover the network's '*active-time*'. Active time we refer to the time period when any none-zero number of participants are active and working - if the digital ecosystem is large, practically this has been considered 24 hours (Razavi et al., 2007a), (Razavi et al., 2008a).

7.2.1 STABILITY MEASUREMENT FOR NODES

Since we are dealing with connectivity as a means of avoiding disconnections and fragmentation of the network, we need a measurement for node stability. It would be unreasonable (and not feasible) to expect participants (nodes) to be online all the time and thus stability is determined on the basis of declared availability.

For finding a more precise and computable measurement for node stability, we introduce the so-called *Expected Availability Time* (EAT). This is the time the node is expected to be available and online in the network. Figure 7-4 shows an example of EAT for a node in the network.

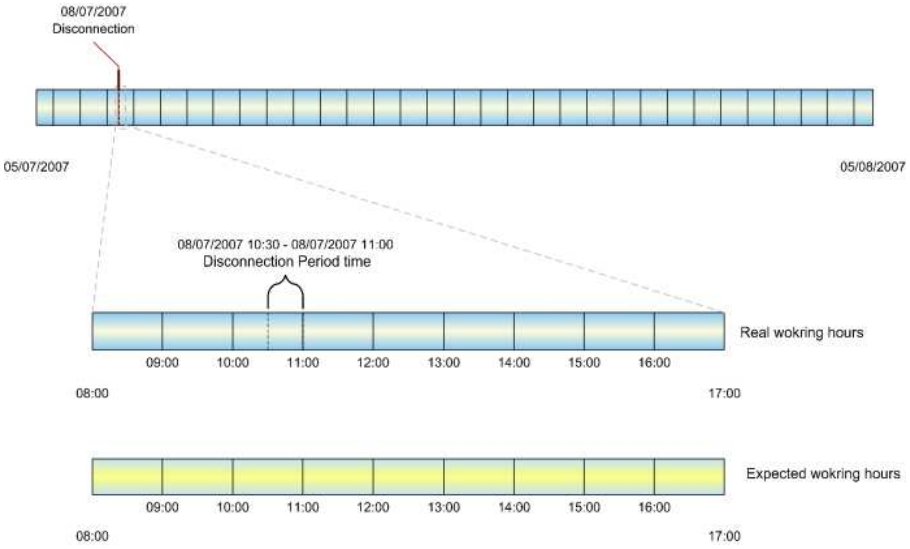

FIG. 7-4 EXPECTED AVAILABILITY TIME

The node stability is then calculated as the actual availability of the node against this expected time. These are typically different, since during its EAT the node may experience disconnections.

This will reduce stability (reliability) of the corresponding node in the final selection. This notion of stability can be simply calculated as below:

$$NodeStability = \frac{EAT - DisconnectionPeriods}{EAT}$$

It can be seen that $NodeStability \leq 1$ and the closer NodeStability gets to 1 for a node, the

more stable the node is (which can be understood as more reliable or predictable). As an additional parameter we have also considered a traffic limitation on the participants. If the infrastructural traffic reaches a specific percentage (*'K'*) of a platform bandwidth, and this can be varied depending on the ecosystem's business cluster, the participant is regarded as disconnected and this will increase its '*Disconnection Periods*'. This can be justified since the priority for each participant is its transactions and we do not want to overload the participants by infrastructural traffic and cause potential collapse and transaction failure.

In any case when a participant is involved in some transactions it is still indirectly helping the infrastructure by its VPTN links. This is one aspect that distinguishes our model from other approaches that may use a mechanism similar to *link replication* but without considering the participant's situation, which has been discussed at subsection 7.1.1, of this chapter.

In our latest simulation which is described in greater detail in chapter 8, the participant's bandwidth has been treated as a random number between 500kb to 4mb and the maximum percentage of the infrastructural traffic is %30 of the platform bandwidth but this can vary depending on the environment and average transactional traffic of VPTNs.

For calculating the stability function of a participant in the first instance we use a VPTN neighbourhood voting algorithm (where each participants in a transaction calculates its other nodes of the VPTN during each transaction life-time) to finalise this. The average availability in overlapping transactions can determine the actual availability.

We consider alternative methods for one of our implementation plans in Chapter 8, which relies on automatic calculation of availability inside of the coordinator of a transaction and this will be calculated based on overloading traffic more than 'K' percentage of the participant bandwidth or disconnections from Internet.

At the moment we have considered EAT as a part of SMEs business model which is given by each SME on joining the network. Hence, this is fixed or can only change on the account of the SME providing it. It should be noted that other approaches can be considered for calculating the EAT - for example, it is possible to use an algorithm based on their VPTNs' actual life time (their transaction life-cycle) or network neighbourhoods' estimations for calculating EAT which would allow it to vary over time.

7.2.2 Permanent Clusters and Virtual Super Peers

As mentioned before, in contrast with conventional super peers, we try in our network design to move towards a more dynamic architecture which does not rely on just a few permanent nodes. Central to our approach is finding permanent clusters on the network. More specifically, we are identifying aggregations of stable nodes, where node stability is determined as in the previous section. For doing so, the most stable nodes from different time zones must be chosen, in a way that they cover the digital ecosystem's '*active-time*' (for reasonably large ecosystem 24 hours). More specifically, we are trying to find permanent clusters through the most stable nodes.

The important part in determining permanent clusters is discovering different aggregations of these time zones which can cover 24 hour availability. Any union of the stable nodes in the aggregations (which provides the full '*active-time*' - 24 hour availability coverage) are actual permanent clusters. Fig 7-5 shows the simple situation in which the most stable nodes have been selected from two sets of time zones which can cover 24 hour service availability to form permanent clusters.

7.3 VIRTUAL SUPER PEERS

By using stable nodes from permanent clusters, as shown in Fig 7-5, we can create *Virtual Super Peers* (VSPs) which are effectively permanent clusters of nodes in the network. These can provide the desired stability for the digital ecosystem. The strong connection between the virtual super peers themselves on one hand and the connection between them and their nodes decrease the probability for fragmentation. Depending on the level of reliability required for the network, it is possible to include further redundant stable platforms from each available time zone. For example, in Fig 7-5 we have included two stable nodes from one time-zone and three stable nodes from the other one (the green and creamy coloured signs show different time zones).

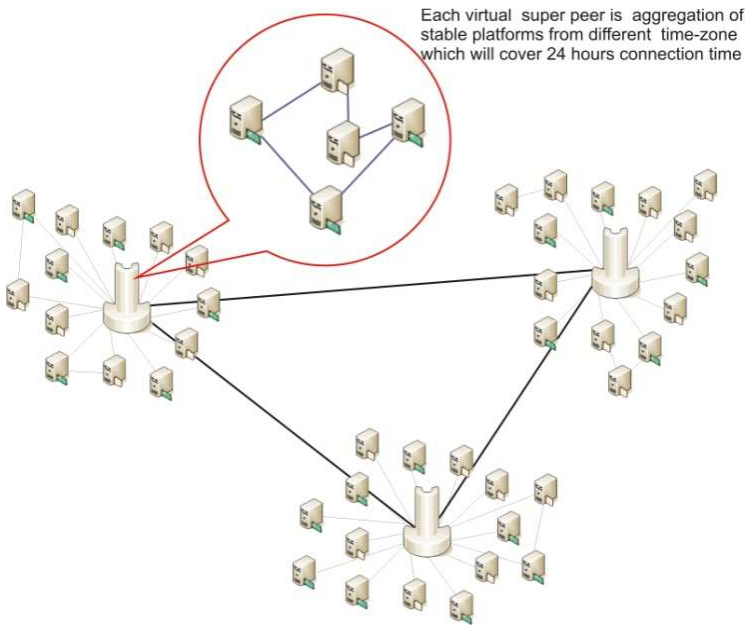

FIG. 7-5 PERMANENT CLUSTERS AND VIRTUAL SUPER PEERS

In this manner, the good connectivity can cause more reliable transactions at the VPTNs

level. Meanwhile the traffic is spread over the virtual super peers and there is less risk of bottleneck at peak time. Participants (nodes) within a virtual super peer need to keep information only about nodes in their cluster and about neighbouring VSPs so at off-peak time the amount of redundant information processing is reduced dramatically as compared to the classical Super Peers solution.

Since choosing stable participants (nodes) is done based on the stability measurement which is given by a function of EAT and the Disconnection Period of a node during EAT, whose value varies over time, is a *dynamic* process and hence it the virtual super peers are also formed dynamically. This means the topology can change from time to time and new nodes can be added to the permanent clusters as the structure of the virtual super peers changes. A node can become part of a virtual super peer, when its node stability increases and overcomes some threshold, and nodes that are super peers may not be able to cope with the increased number of connections they get, and possibly increased number of transactions they perform and lose their virtual peer status. Within a digital ecosystem for business, SMEs would be expected to invest at that time (in hardware, processing power, bandwidth etc.) and become again part of a virtual super peer in future. It is in this sense that the topology evolves to reflect the usage and demands of the participants who benefit from and contribute to the 'sustainability' of the network.

Additionally, network congestion can change the maximum level of node stability (recall the discussion in sub-section 7.1 and 7.2.1) which in turn affects the selection of the most stable nodes in forming the permanent clusters. High congestion of packages can increase or decrease network reliability (higher traffic on few virtual super peers can potentially create a bottleneck and even cause fragmentation). In a digital business ecosystem, the best part of the traffic is the result of business activities which are effectively long-lived transactions. These have been virtualised in VPTNs and therefore, using the effect of VPTNs for making VSPs and their client nodes, can increase stability of each virtual super peer.

Furthermore, we expect a reasonable cluster coefficient on the account of having VPTN as the main building block which we have seen is formed from a transaction. This means its participants are in relevant domains – by connecting them to several VSPs we actually increase the probability for that. We also expect a fair distribution degree on the account of propagating links to VSPs. This means that instead of being concerned with individual links for each node, aggregate links of VSPs come into play.

Finally, reusing business activity results (or service-on-fly as result of composite services (Papazoglou, 2003) and (J. Yang et al., 2002)) and explorative service composition (Papazoglou et al., 2006) are other factors which can be considered for higher performance within a digital business ecosystem and can provide potential for creating so-called *virtual vendors* (Razavi et al., 2006).

7.4 The dynamic mechanism for choosing VSPs

In the first step, the most stable participant in each VPTN (participants of a transaction)

should be selected for keeping vital information about the transaction and its participants. In this sense, the network provide a level of durability with minimum cost from participants and it provides a greater chance for forward recovery even in terms of failure in one of participants of a transaction. Effectively, this makes our mechanism described in Section 5.3, fully effective with regard to what is referred to as (in purely transactional literature, the solved problem is called) *omitted results*,which is a problem relating to preserving as much progress-to-date as possible in the event of aborting a transaction (the details about complexity of the problem can be found in chapter 2 and our previous work (Razavi et al., 2007d)).

In the next step, by connecting the most stable nodes of each VPTN together, the first level of strong connectivity and suitable nodes for VSPs are created. Fig 7-6 shows the internal structure of each VPTN and the connection between VPTNs. The internal structure of a VPTN contains a lot of information from the transaction level such as log structures, lock schemes for ensuring consistency in recovery mentioned above, local coordinator design, formal analysis of the required interactions and recoverability (Moschoyiannis et al., 2008), along with alternative scenarios for forward recovery (chapter 6 shows this behaviour analysis). Now by using the most stable node, we allow the optimisation in transactions to be performed and any waste of resources as the result of weak connectivity will be avoided.

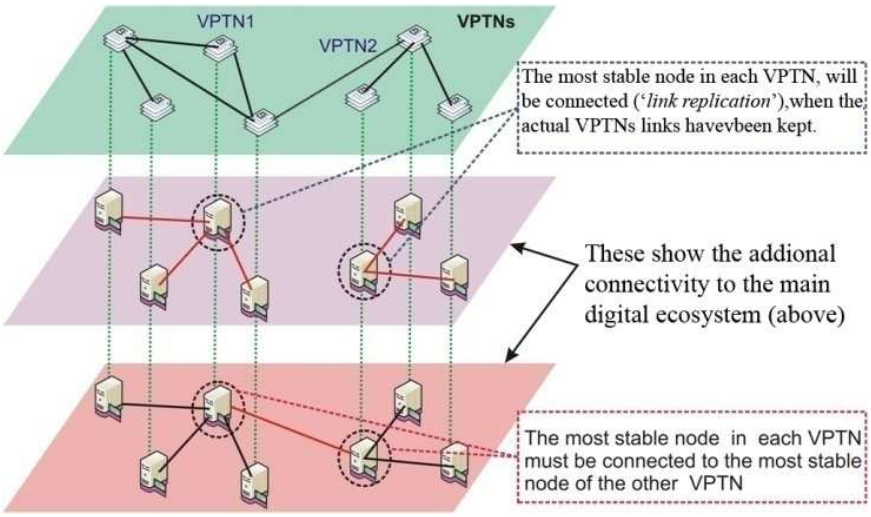

FIG. 7-6 DYNAMIC MECHANISM FOR INCREASING STABILITY

The direct effect of connecting VPTNs together is raising the cluster coefficient of the network. Conversely, connecting the most stable nodes of VPTNs together provides the opportunity of choosing the best candidate locally between these stable nodes for the permanent cluster. Choosing nodes of the permanent cluster in this way results in a virtual super peer that provides fair traffic distribution at the VSN level (each virtual super peer will take care of its local VPTNs). The main concept behind forming permanent clusters stays the

same, i.e., selecting the most stable nodes from different time zones which can cover 24 hours online time.

7.5 THE MODEL IN PRACTICE

As the most stable node in each VPTN is the best candidate for keeping the transaction information, the corresponding business activities will have increased levels of reliability. The fact that VPTNs are used initially in the design of the business network, and are connected through their most stable nodes which are determined dynamically, allows in most cases the candidate platform to avoid the full rollback or compensation of the transaction when some participants of the long-lived transaction get disconnected mid-way through its execution. This has been considered in the design of the recovery mechanism for the digital ecosystem (chapter 5).

Another expectation of the proposed network infrastructure is reducing the possibility of fragmentation. We have seen that we are dealing with a highly dynamic transactional environment where there is no central point of control. The current model can fulfil the requirement at the theoretical level. It would however be desirable to be able to somehow guide the way this topology evolves. In the next section, we try to show a realistic simulation which can compare the theoretical behaviour and practical status of the network in different situations. Furthermore the current roadmaps and prototype implementations are described.

8 IMPLEMENTATION EXPERIENCES AND ROADMAPS

The first implementation of a Digital Business Ecosystem was relying on FADA nodes (Razavi et al., 2006), (TechIDEAS, 2007) as the core infrastructure for the network. Items (service proxies) are registered in any node of the FADA cloud, and they can be searched for starting from any node in the FADA cloud. The FADA nodes create a free graph, i.e., the topology is not enforced and not even known. As a result FADA nodes are relying on node-interactions (transactions) for creating links. In this sense, the network design we propose builds on the principles of FADA, but is extended with the design concept of dynamically formed Virtual Super Peers.

As the first step, the class diagram of transaction context and consistency logs (EDG and IDG class diagrams) are presented and then the sample scenario and the roadmap of implementation by collaborating with IPTI (IPTI, 2009) have been presented. The RESTful frameworks as the next step and XMPP implementation by TeachIdeas are the other sections of this chapter. Furthermore Appendix II and III include more details about implementation and collaboration with other researchers.

8.1 SCHEMAS AND THE MODEL INFRASTRUCTURE

The primary expectation of the implementation is to clarify the consistency model of the transaction. In chapter 4 we have shown the xml schema of transaction context and a sample xml scenario (section 4.2), meanwhile the xml schemas for IDG and EDG is provided at the end of same chapter (section 4.7).

Here we provide a set of Java classes which represent foundation of parser the transaction context, IDG and EDG schemas. These classes can be imported in any java project to adding the ability for creating the transaction tree, traverse it, add elements and specify their attributes. Meanwhile by using IDG and EDG for data dependencies between different participants, it is possible to use the proposed consistency model in this report.

One may marshal or unmarshal the transaction context. Marshalling creates an XML document from a content tree, when unmarshalling an XML document means creating a tree of content objects that represents the content and organization of the document (The content tree is not a DOM-based tree. In fact, content trees produced through JAXB can be more efficient in terms of memory use than DOM-based trees).

In the implementation, easily, you can call the marshal method. This method does the actual marshalling of the content tree. When you call the method, you specify an object that contains the root of the content tree, and the output target. For example, the following statement marshals the content tree whose root is in the collection object and writes it as an output stream to the xml file. The transaction context is '*TransactionSample*'.

```
        File f = new java.io.File(this.fName);
        try {
            javax.xml.bind.JAXBContext jaxbCtx = javax.xml.bind.JAXBContext.newInstance(
                        TransactionSample.getClass().getPackage().getName());
            javax.xml.bind.Marshaller marshaller = jaxbCtx.createMarshaller();
            marshaller.setProperty(javax.xml.bind.Marshaller.JAXB_ENCODING, "UTF-8"); //NOI18N
            marshaller.setProperty(javax.xml.bind.Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
            marshaller.marshal(TransactionSample, f);
            marshaller.marshal(TransactionSample, System.out);
            this.fLenght=f.length();
        } catch (javax.xml.bind.JAXBException ex) {
            // XXXTODO Handle exception
            java.util.logging.Logger.getLogger("global").log(java.util.logging.Level.SEVERE, null, ex); //NOI18N
        }
    }
```

Now, we can explore the actual class diagram of transaction context and the consistency model (Internal Dependency Graph and External Dependency Graph).

8.1.1 TRANSACTION CONTEXT CLASS DIAGRAM

As it has shown in Fig 4-18 the root of a transaction context schema is a tree (*TransactionTree*), the root of which contains the main sub-transaction. A sub-transaction can be a simple web-service, delegation type, composition or data-composition. For presenting the nested structure of sub-transactions, we use a recursive structure which relies on a complex type that is called '*SubTransactionType*'. Fig 8-1 shows the actual class diagram of Transaction Context.

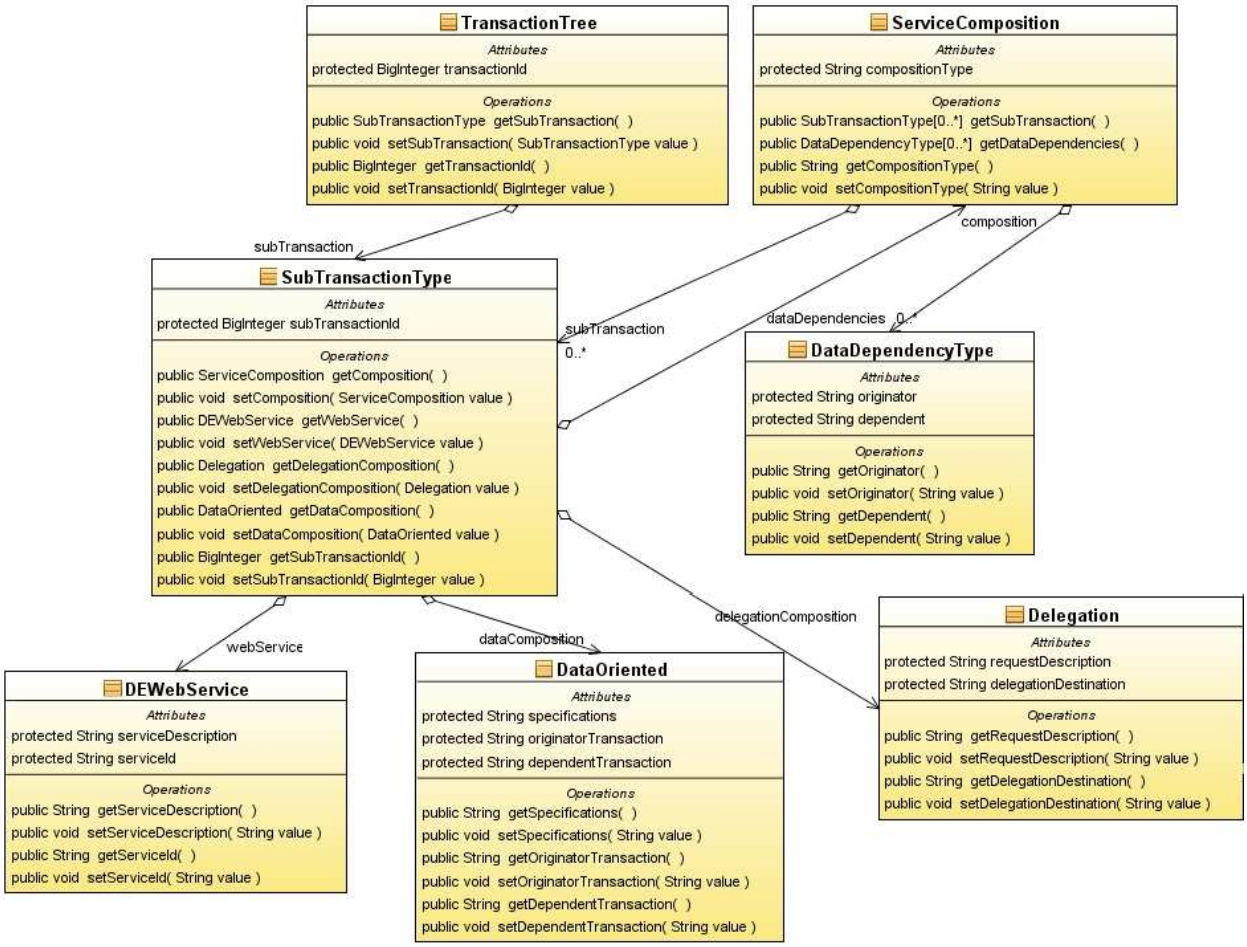

FIG. 8-1 TRANSACTION CONTEXT CLASS DIAGRAM[10]

For applying the transaction concept, we first need to create the transaction. As is shown in Fig 8-1, the root of transaction context, is *TransactionTree*. Therefore we can define an object from the Transaction tree (*TransactionSample*). Two important elements of a transaction tree are its ID and Sub-transactions:

private TransactionTree TransactionSample = new TransactionTree();

TransactionSample.setTransactionId(value);

TransactionSample.setSubTransaction(value);

The Sub-transaction type must have an ID which is unique in the transaction context and between other sub-transactions of a transaction. Meanwhile the sub-transaction type must

[10] The Class diagram has been made by Netbeans 6.1 based on its UML standard reengineering tools

reflect the recursive structure of the transaction tree. This nested structure should be reflected in the composition of other sub-transactions. This will be done by setting the composition element, where the class for creating sub-transactions is '*SubTransactionType*':

```
private   SubTransactionType    SubTransactionSample = new SubTransactionType();
SubTransactionSample.setSubTransactionId(value);
SubTransactionSample.setComposition(value);
```

For defining service composition objects, the '*ServiceComposition*' class has been defined (Fig 8-1). This represents the recursive structure of the transaction tree; when Sub-transaction can have a service composition element, a service composition will compose several sub-transactions (it has sub-transaction element). A simple example of defining an object of *ServiceComposition*' can be found as below:

```
private   ServiceComposition ServiceCompositionSample = new ServiceComposition();
ServiceCompositionSample.subTransaction.set(index, SubTransactionSample);
```

8.1.2 Internal Dependency Graph class diagram

In chapter 4, we have shown, based on the theorem, for avoiding wormholes the first necessity for internal dependency logs is the possibility for tracing dependencies. The root of such a schema is the '*internal dependency graph*', when the first element identifies the data-item which is to be shared and the second element shows the chains of dependencies. Fig 8-2 shows the class diagram of this consistency log.

Schemas and the model infrastructure

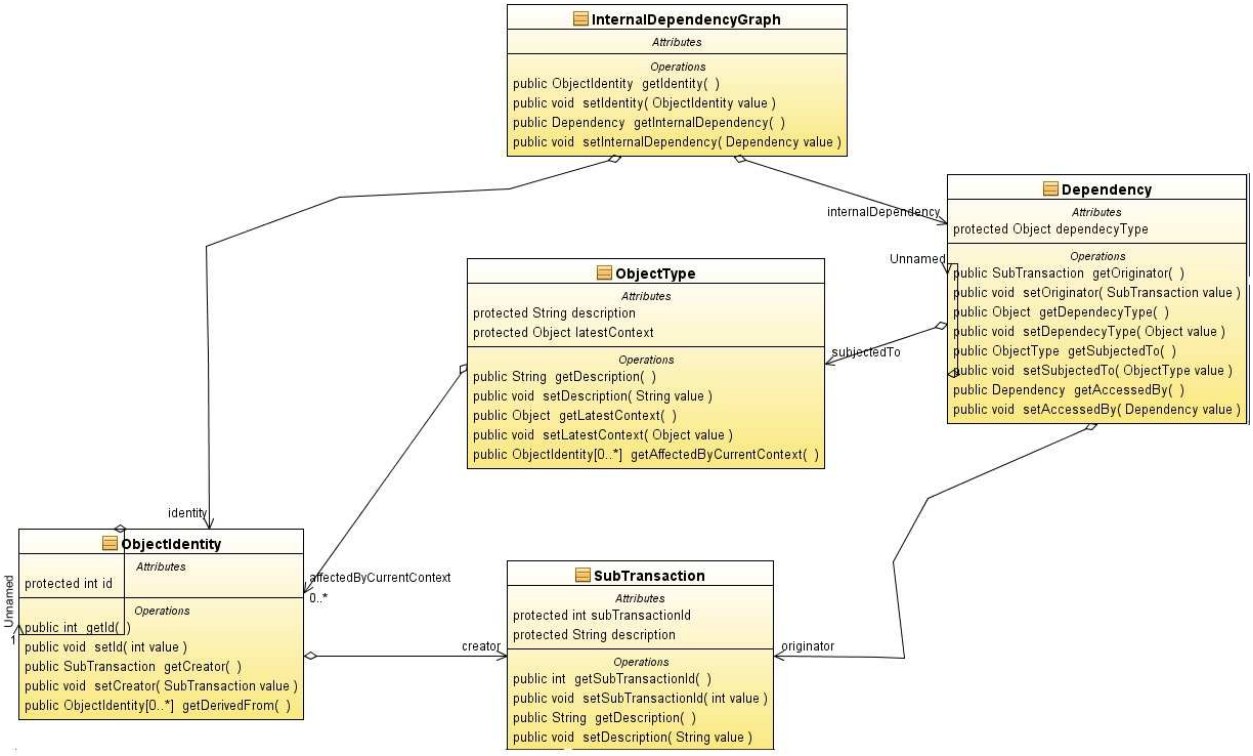

FIG. 8-2 INTERNAL DEPENDENCY GRAPH CLASSES[11]

The root of the IDG class diagram (similar to the xml schema) has been known as the *InternalDependencyGraph*, includes an identity and the internal dependency for keeping the dependencies to a particular data item:

private InternalDependencyGraph IDG_Sample = new InternalDependencyGraph();

IDG_Sample.setIdentity(value);

IDG_Sample.setInternalDependency(value);

The internal dependency in this graph is a dependency type. The structure of '*Dependency*', by using a recursive structure, creates the graph and traces the chains of dependencies for a data-item. In this way we can avoid any cycle in the graph. The dependency type has three elements; '*Originator*' (sub-transaction that share the data item), '*SubjectedTo*' (the modifications on the data-item) and '*AccessedBy*' (shows the next level of dependency on the object).

[11] The Class diagram has been made by Netbeans 6.1 based on its UML standard reengineering tools

```
private    Dependency DependencySample = new Dependency();

DependencySample.setAccessedBy(DependencySample);

DependencySample.setOriginator(SubTransaction value);

DependencySample.setSubjectedTo(ObjectType value);
```

8.1.3 External Dependency Graph class diagram

As we have explained in chapter 4 (4.6 and 4.7), partial results may be released before the actual commit of a long-running transaction. We have introduced a mechanism for releasing them; '*Conditional Commit*'. The role of the '*External Dependency Graph*' is crucial in this mechanism (recall section 4-6), and we have introduced an xml schema which provides traceability for these dependencies. Fig 8-3 shows the class diagram of this consistency log.

Schemas and the model infrastructure

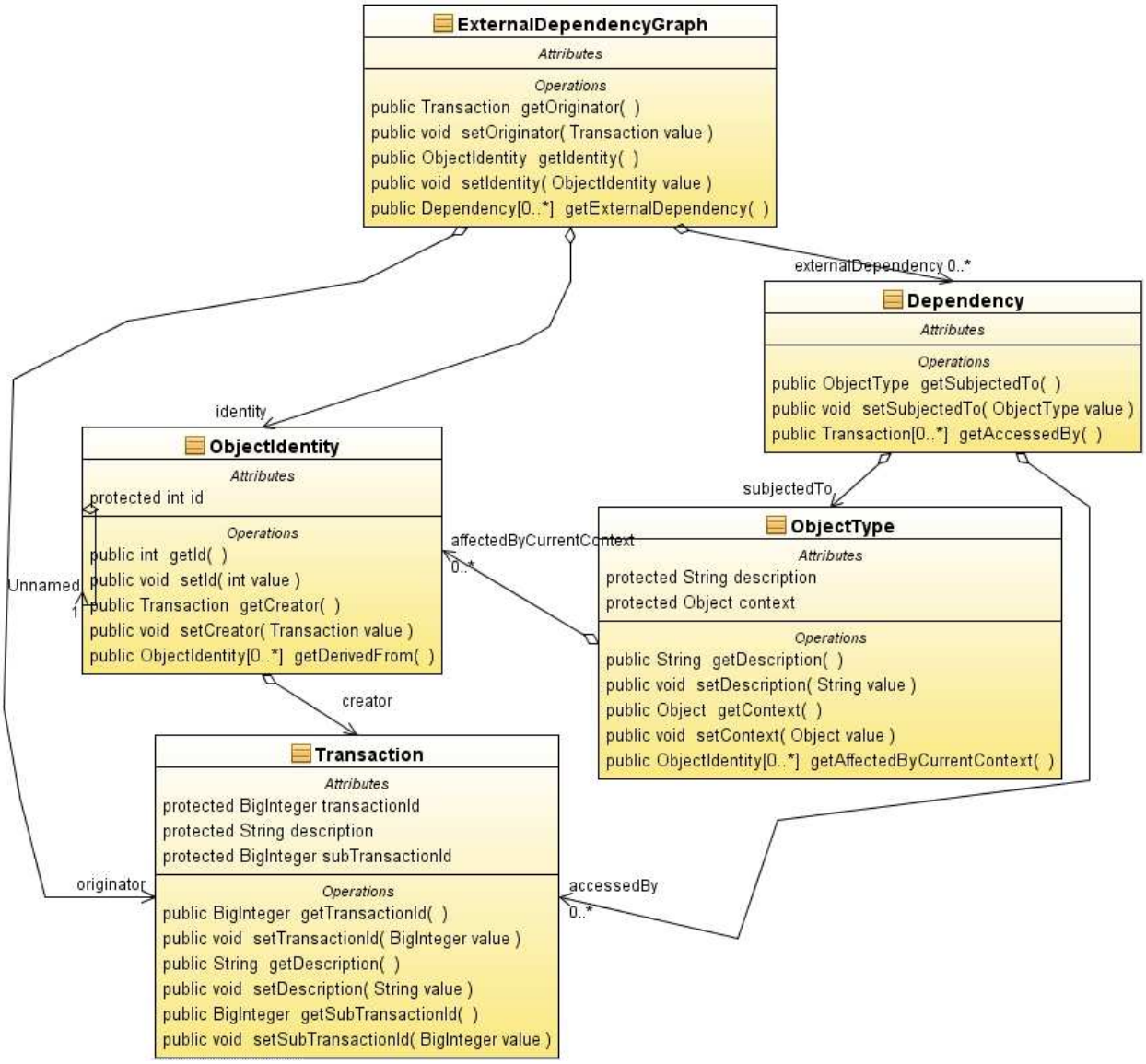

FIG. 8-3 EXTERNAL DEPENDENCY GRAPH CLASSES[12]

The root of the external dependency graph is '*ExternalDependencyGraph*', where the first element identifies the data-item which is to be shared, the second element clarifies the owner of data-item (the identity of the transaction and its particular sub-transaction, which share the data-item) and the third element shows the chains of dependencies.

private ExternalDependencyGraph EDG_Sample = new ExternalDependencyGraph();

EDG_Sample.setIdentity(value);

[12] The Class diagram has been made by Netbeans 6.1 based on its UML standard reengineering tools

EDG_Sample.setOriginator(Transaction value);

EDG_Sample.externalDependency.set(index, DependencySample);

The external dependency in this graph is a dependency type. The structure of '*Dependency*', is slightly different to the internal dependency graph, as always the dependencies are from a generator towards other transactions (recall 4-6). The current class diagram automatically avoids any recursion or loop, in this way we can avoid any cycle in the graph. Two important elements of dependency type are; '*SubjectedTo*' (information and context of the shared data-item and any probable modifications) and '*AccessedBy*' (or *Transaction* shows transaction which accessed the object/data-item).

private Dependency eDependencySample = new Dependency();

eDependencySample.setSubjectedTo(ObjectType value);

eDependencySample.Transaction.set(index, Transaction value);

8.2 JXTA AND DE MODEL

We have started to work with IPTI in Brazil ('Instituto de Pesquisas em Tecnologia e Inovação'(IPTI, 2009)) for providing an implementation of the model. The aim here is to exploit the characteristics of the transaction model, mostly in terms of interaction-based service composition and the fine-grained lock scheme, in supporting complex interactions within the collaborative platform *guigoh*("guigoh - Social Network," 2008) and improve its social network aspects. In the first instance we have been looking at reducing the traffic complexity of the interactions and adding provision for business services.

The first implementation uses JXTA protocols, which are defined as a set of XML messages which allow any device connected to a network to exchange messages and collaborate independently of the underlying network topology ("jxta: JXTA™ Community Projects," 2007). The first prototype of this work, together with preliminary documentation, can be found in the opensource project '*flypeer*' and can be downloaded from ("Flypeer - Dynamic P2P Infrastructure — Project Kenai," 2008). Based on this model, by growth of participants, the dynamicity of VSPs comes into play and the consistency model can be fully distributed. At the moment, in collaboration with IPTI we have examined the transaction context, through sample service-oriented scenarios, where the main services are optimised for creating parallel, sequential and alternative compositions of virtual online conferences. This has included the fully distributed transactional communication (exchange of messages, initiating a transaction, and terminating the transaction). The P2P relationship between participants and their services has been supported in a purely loosely coupled manner.

In the next steps, we plan to extend the implementation prototype by considering more complex scenarios and introducing additional traffic complexity through incorporating more

heavy services, such as voice, video streams. In addition, we are looking at introducing a larger number of transaction participants and work is in progress in integrating the user interface for monitoring the model.

8.2.1 INTERNAL MODEL

The core of the Peer-to-Peer (P2P) layer is based on the described model in this report (transactional model in chapter 4 and 5, the networked infrastructure in chapter 7 and distributed behaviour analysis in chapter 6), and the related papers can be found at (Razavi et al., 2009), (Razavi et al., 2007c), (Razavi et al., 2008a).

The following is a short description of our work so far and, more importantly, the work we still need to implement. At the very core of the infrastructure there is a transaction module. This is where we are most focused now. Among others, we are considering three types of transactions:

- Sequential: when actions occur one after the other;
- Alternative: when one action occur and, if it fails, another one takes place;
- Parallel: when two actions occur at the same time.

With transactions working, we define the so called Virtual Private Networks (VPTNs), based on the nodes interacting through transactions. And these VPTNs will provide the base for building Dynamic Virtual Super Peers (DVSPs), which are the final goal of the infrastructure core.

8.2.2 EXAMPLE SCENARIOS

In parallel to the implementation, we are working on two small demonstrators, to help us keep the code working while doing several big changes:

- Travel agency: a simulation of a travel agency booking hotels and flights;
- Chat: a small text chat client.

Besides that, we have EvEsim (“Evesim - Digital Ecosystems Research,” 2007) working on top of our infrastructure for simulation and evaluation purposes.

8.2.3 ADDITIONAL INTERNAL STRUCTURES

We have worked on a few internal structures that are quite important, yet they are not

finalised and may change according to further research. So here they are:

- Peer state handling: defining how each peer will keep its internal state, such as current transactions, transaction's contexts, among other things;
- Local and Remote Service Repositories: repository of the services provided by the local peer, as well as provided by remote peers; this is mapped to local and global service repository which has been introduced in chapter 3 and explored in the rest of this paper.
- Business services interfaces: we are trying to define a user-friendly interface which shall provide an easy way for users to build services on top of the infrastructure. But this work is still in its primary steps.

Meanwhile, some additional components are under customisation The important ones are listed below:

- Service Deployment: In this step, we are just embedding it in our test environment, but we have considered an evolutionary plan which will move towards a RESTful framework (8.3);
- Applet / JS communication: We currently test the infrastructure running inside browsers, but not an eventual communication model;
- Error handling: At the moment the classic failures are considered but during more extended tests, we will improve the error handling routines;

8.3 RESTful framework

We have considered the evolution of digital ecosystem towards a RESTful model as the next step of the current implementation (Razavi, Marinos, Moschoyiannis, & Krause, 2009). As this framework is in the early days, we have provided a brief explanation about the current work (more details can be found in Appendix II).

8.3.1 REST and Transactions

Representational State Transfer (REST) is a distributed computing architectural style that was defined in 1999 by Roy Fielding (Fielding, 2000) based on the architecture of the World Wide Web. REST emphasises resources identified by names, a fixed number of methods with known semantics to manipulate those resources, hypermedia as a means of traversing

the resources and statelessness in the interactions between client and server. REST has gained traction in addressing many common use cases for distributed systems. As is common with disruptive technologies, the use of REST over HTTP is evolving to compete with WS-* in increasingly advanced usage scenarios (Richardson & Ruby, 2007). Our work in this arena aims to be part of the next wave of REST evolution by defining a RESTful transaction model that is designed to operate over HTTP.

While transactions are concerned with the constraints of maintaining the ACID properties, REST adheres to its own set of constraints. To create a truly RESTful transaction model, it is necessary to satisfy both the constraints of REST and the constraints relevant to the ACID properties of transactions. This is the primary contribution of this section.

When two (or more) transactions access the same resource, they may produce two (or more) different versions of that resource (*lost update*), or simply they may work with the out-of-date version of the resource (*dirty* GET and *unrepeatable* GET). Fig 8-4 shows these three faulty scenarios, which are typically called *wormholes* as they lead the system to an inconsistent state.

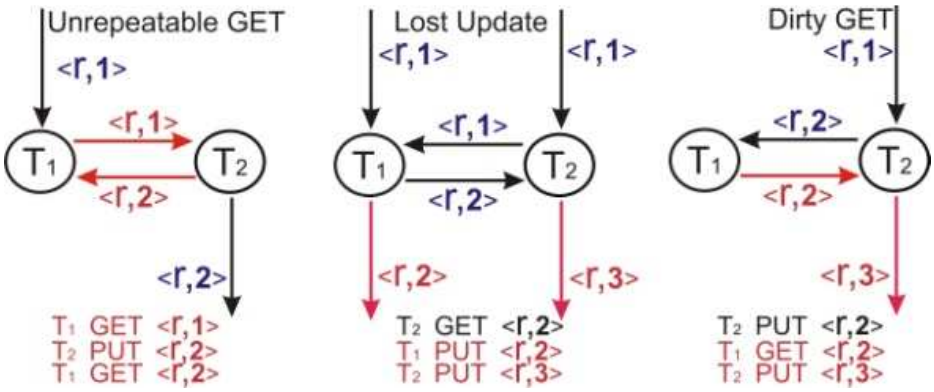

FIG. 8-4 INCONSISTENT SCENARIOS

8.3.2 APPLYING THE ISOLATION THEOREMS TO REST

One transaction instance *T* is said to depend on another transaction *T'* in a history *H* if *T* GET (reads) or PUT (writes) data-resources previously PUT (written) by *T'* in the history *H*, or if *T* PUT (writes) a resource previously GET (read) by *T'*.

We can formalise such dependencies using a directed graph where nodes are 'transactions', arcs indicate 'transaction dependencies', and labels on arcs denote 'resource versions'.

The *version* of a resource *r* at step *k* of a history *H* is an integer and is denoted *V(r,k)*. In the beginning each resource has version zero (*V(r,0)*=0). At step *k* of *H*, resource *r* has a version equal to the number of writes of that resource before step *k*. Formally:

$$V(r,k) = \left|\left\{\langle t_j, a_j, r_j\rangle \in H \middle| j < k and a_j = \text{PUT} and r_j = r\right\}\right|$$

(The outer vertical bars represent the set cardinality function.)

Each history, *H*, for a set of transactions $\{T_i\}$ defines a ternary *dependency relation*, denoted by DEP(*H*), as follows.

Let *T*1 and *T*2 be any two distinct transactions, let *r* be any resource, and let *i*, *j* be any two steps of *H* with $i < j$. Suppose step $H[i]$ involves action *a*1 of *T*1 on resource *r*, step $H[j]$ involves *a*2 of *T*2 on *r*, and suppose there is no PUT on *r* by any transaction between these steps (there is no $\langle T', PUT, r\rangle$ in $H[i+1], \ldots, H[j-1]$). Then DEP(*H*) is defined as:

$$\langle T, \langle r, V(r,j)\rangle, T'\rangle \in \text{DEP}(H)$$

if *a*1 is a PUT and *a*2 is a PUT

*a*1 is a PUT and *a*2 is a GET

*a*1 is a GET and *a*2 is a PUT.

PUT→PUT, PUT→GET and GET→PUT dependencies.

The dependency relation for a history defines a directed *dependency graph*, where transactions are the nodes of the graph, and resource versions are label on the edges. If $\langle T, \langle r, j\rangle, T'\rangle \in \text{DEP}(H)$, then the graph has an edge from node *T* to node *T'* labelled by $\langle r, j\rangle$. Two histories are equivalent, if they have the same dependency relation.

The dependency relation of a history defines a time order of the transactions. Conventionally this ordering is signified by <<< and it is the *transitive closure* of $<<<_{\text{H}}$. It is the smallest relation satisfying the equation $\text{T} <<<_{\text{H}} \text{T}'$ if $\langle \text{T}, \text{r}, \text{T}'\rangle \in \text{DEP}(H)$ for some resource version r, or $\text{T} <<<_{\text{H}} \text{T}''$ and $\langle \text{T}'', \text{r}, \text{T}'\rangle \in \text{DEP}(H)$ for some transaction T'' and some resource r. Whenever $T <<< T'$ there is a path in the corresponding dependency graph from transaction *T* to transaction T'. The <<< ordering defines the set of all transactions that run before or after *T* as follows.

$$\text{BEFORE}(T) = \{T' | T' <<< T\}$$

$$\text{AFTER}(T) = \{T' | T <<< T'\}$$

If *T* runs fully isolated (ex: it is the only transaction, or it GET and PUT resources not accessed by any other transactions), then its BEFORE and AFTER sets are empty (it can be scheduled in any way). When a transaction is both after and before the other distinct transaction, it is called *wormhole transaction* (T' here):

$$T' \in \text{BEFORE}(T) \cap \text{AFTER}(T)$$

Therefore, a cycle in a dependency graph is a wormhole. Using a well-formed and two phase locking mechanism is a conventional method for avoiding wormholes. In the next section we introduce a locking mechanism which produces a history that is equivalent to a serial history (Gray & Reuter, 1993). Serial histories do not present wormholes.

8.3.3 A MODEL FOR RESTFUL HTTP TRANSACTIONS

For an API to be characterized as RESTful, according to the hypermedia constraint, it must allow a client to interact with the service solely by being given a single entry URI and understanding of the relevant media types. This results in extremely loosely coupled systems with a minimum of assumptions. Additionally, it is desirable that the always-available and backwards compatible nature of the web is preserved and not be made unavailable by the introduction of locks. To accommodate these requirements, a number of resources and resource collections have been defined in addition to the resources that the transactions interact with. These resources are necessary for the transactions to take place in a RESTful environment.

Lockable Resource (R)	A resource that locks can be applied to Operations: GET, [By XLOCK owner: PUT]
Resource Lock Collection (R-Lc)	The collection of locks that apply to a particular resource. Operations: GET, POST
Lock Resource (R-L)	The representation of a specific lock Operations: GET
Conditional Resource Representation (R-C)	The potential representation of a locked resource, once its lock is committed. Operations: GET, [By XLOCK owner: PUT, DELETE]
Transaction Collection (Tc)	The collection of transactions on the server. Operations: POST
Transaction Resource (T)	The representation of a specific transaction. Operations: GET
Transaction Lock Collection (T-Lc)	The collection of locks connected to a specific transaction. Operations: GET, [By transaction owner: DELETE]
Owner Collection (Oc)	The collection of owners of a specific transaction Operations: GET, [By transaction owner: POST]

TABLE 2 – RESOURCE TYPES AND ALLOWED OPERATIONS

The resources required can be seen in the left column of Table 2, while the right shows the relevant HTTP operations available for each resource.

RESTful framework

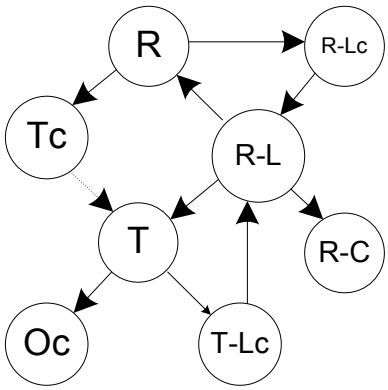

FIG. 8-5 RESOURCE RELATIONS

Having defined all the resources, it is easy to see that a network arises. Fig 8-5 displays the interconnections of the resource graph (more extended version can be found in the Appendix II). It can be observed that having a URI for R is enough to locate all other resources in the network. The connection from Tc to T is different from the other connections as there is no GET ability for the Tc resource. The URI of T is returned as a response to a POST operation on Tc performed by the transaction's owner.

Ideally any resource that can be served by an HTTP server should be potentially lockable. This however would require the HTTP protocol to carry the metadata for the locking mechanism. Since we wish to preserve the HTTP protocol, we limit ourselves to a snippet of XML that is to be included in an XML representation of a resource which could look something like the following:

```
<lockable>
<link rel="lock_collection" href="http://example.org/resource/locks/">
<link rel="transaction_collection" href="http://example.org/transactions/">
</lockable>
```

Namespaces could also be utilized to avoid namespace collision but this would make the approach difficult to apply to other (e.g. JSON) representations which do not have support for namespaces.

As it is expected, a history should not complete a lock action on a resource when that resource is locked by another transaction. But if two or more transactions want to just read (GET) the content of a resource, they do not change the resource version (state). This will not cause any conflict or access to dirty data (data/resource which has been PUT by another transaction) but the transaction has not committed and may change the version of the resource again (recall Fig. 8-5). Table 3 shows the lock compatibility. The inferred rules constrain the set of allowed histories. Histories that satisfy the locking constraints are called *legal histories*.

The model on an XMPP implementation

		Mode of Preceding Lock	
		Share	*Exclusive*
Mode of New Lock	*Share*	*Yes*	*No*
	Exclusive	*No*	*No*

Table 3. Lock Compatibility

8.3.4 Future Work

We have provided a RESTful framework for business transactions by adapting the conventional locking mechanism to work within the architectural style of REST. We have shown that this locking mechanism is well formed and two-phase through the application of the Isolation theorems. Further details have been provided in Appendix II and the next step of this framework is applying our extended long-running lock mechanism to this model for covering more complicated transactions.

8.4 The model on an XMPP implementation

Our work on distributed coordination of long-running transactions involving the deployment of services started in the Digital Business Ecosystem (DBE) project. The support for a distributed transaction model initially targeted the DBE Studio[13] which can be understood as a service container for search and deployment of services from various services providers, and specifically SMEs. The DBE Studio implemented by TechIdeas[14] uses the so-called FADA network. Experience with FADA has shown the network to become unstable in certain respects. Subsequent analysis and further experimentation under the real DBE studio implementation revealed certain problems relating to connectivity and fragmentation. Our own simulations have also highlighted such aspects.

Currently, the work led by TechIdeas is steered towards using XMPP[15] protocols and a first implementation of this is a new platform called *Sironta*[16]. XMPP at its core is a technology

[13] The DBE Studio is an Integrated Development Environment (IDE) for the Digital Business Ecosystem (DBE). It includes eclipse plugins that allow business services to be analysed, and corresponding software services to be defined, developed and deployed; http://dbestudio.sourceforge.net/

[14] http://techideas.es/

[15] Extensible Messaging and Presence Protocol (XMPP) is an open, XML-based protocol originally aimed at near-real-time, extensible instant messaging (IM) and presence information (e.g., buddy lists), but now expanded into the broader realm of message oriented middleware; http://xmpp.org/

[16] http://www.sironta.com/

for streaming XML over a network. Our transaction framework is concerned with optimising transactions in terms of the XML context, the consistency model for the interactions. In the OPAALS project [17] there is a roadmap for integrating this work with Techideas' implementation in order to provide a customised infrastructure for the service-oriented platform Sironta.

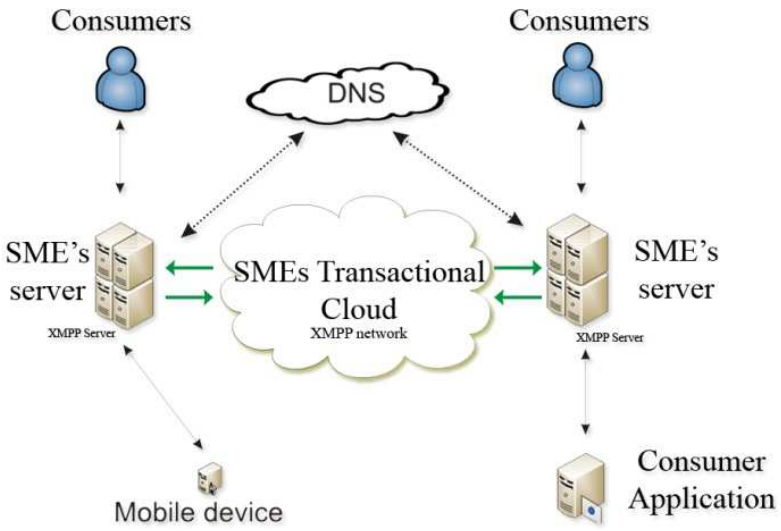

FIG. 8-6 XMPP IMPLEMENTATION FOR DIGITAL ECOSYSTEM

Fig 8-6 outlines the general idea behind the integration of the two approaches which involves modelling XMPP servers onto customised SME's servers. In this case, the digital ecosystem infrastructure described in this report can act as an SME transactional cloud, which dynamically optimises itself to respond to the usage that is being made of it based on the transactions taking place between participating organisations.

[17] OPAALS is a global Network of Excellence formed around multi-disciplinary research into Digital Ecosystems. OPAALS research covers social science, linguistics, computer science, software engineering, and biology. Part-financed for four years by the European Union's 6th Framework Programme, OPAALS launched on 1st June 2006. The current partners in the network are listed below; http://www.opaals.org/

9 CONCLUSIONS AND FUTURE WORK

In this report, a loosely coupled solution has been described in the context of digital ecosystems, and in particular we have been concerned with services, transactions, and network support within the digital ecosystem initiative. The structure of the interaction network within the architecture we have proposed emerges through the local interactions that take place in the context of long-running business transactions.

Particular care has been taken to ensure that transaction and network (environment) support in our approach satisfies certain requirements that are pertinent to the adoption of digital ecosystems by SMEs. The absence of a central point of command and control (and by virtue of that, also governance), and consequently the absence of a single point of failure, the distributed coordination, the usage of loosely-coupled services, the resilience to fragmentation and smart attacks, and allowing for a dynamic topology that continuously adapts to reflect the actual usage of the network in terms of business transactions are the main features that figure prominently in the proposed digital ecosystems' design.

In part, this has been achieved by considering a distributed model for the coordination of long-running transactions, and the provision for fine-grained lock schemes and recovery procedures. The transaction model feeds into the corresponding Virtual Private Transaction Networks (VPTNs) which are the main building block for the underlying network that supports these complex interactions between participating entities.

The basic design feature of the digital ecosystem network has to do with the Virtual Super Peer (VSP) construct. These clusters of stable nodes are used instead of the conventional super peer solution and allow for creating a connected network without generating dependency on a single (or a few) network infrastructure provider(s). They are formed dynamically and this means the VSP solution also allows the network to reconfigure itself and withstand certain types of failure that typical scale-free networks find it difficult to recover from such as fragmentation or smart attacks.

Some of the basic features of the VPTNs go beyond the domain of business transactions and are relevant in more general complex interactions for example involving knowledge services. For this reason we have focused on correctness of the transaction model.

This is important not only because the VPTNs are the main building block for the underlying network but for other applications over digital ecosystem networks. For instance, a prerequisite for online collaborative editing of documents (or content more generally) is that changes made by one participant are visible to the rest and not overwritten by their concurrent edits. The proposed lock scheme, here considered within the transaction model, allows for such simultaneous editing on different parts of the document, and this is something that is being incorporated in the guigoh[18] or OKS[19], e-

[18] http://www.guigoh.com/ has been designed by IPTI

[19] http://oks.opaals.org/ officially part of OPAALS project

learning collaborative platform in collaboration with IPTI as outlined in chapter 8 (8.2).

As mentioned before, a digital ecosystem is highly dynamic environment for a variety of reasons and failures of various types are to be expected. Therefore, our efforts so far have been targeted at providing a stable network that exhibits increased connectivity and resilience to fragmentation. For instance, time-zones and a pool of candidate nodes factor in the formation of VSPs and hence the proposed digital ecosystems' network design works best as the number of nodes increases. Further experimentation is under way with regard to a number of parameters in the proposed framework such as the minimum number of nodes that afford desired stability levels, the period of time needed for forming layers of VSPs and this is in addition to other considerations such as adapting efficient search algorithms into our framework. In terms of transactions, we are looking at other variable parameters such as the time-out in T-Lock and the overall execution period of a given transaction. Although these tend to be domain-specific, we are keen to exploit the interrelationship with the underlying network topology in providing a more stable environment for business activities and open collaboration.

Another factor that adds to the dynamicity of a digital ecosystem for business is that nodes (especially when considering SMEs) may be joining or leaving the network continuously, and in some cases abruptly. The proposed model is essentially an unstructured network, which only inherits from the VPTN network structures. In this report we have only touched upon the issues of birth and growth.

In addition, the network topology itself evolves continuously based on the dynamically formed VSPs. In order to get a handle on how the network topology evolves under the events of nodes joining and leaving the network, we have been looking at biological models of growth in living organisms. Of particular interest seems to be the study of molecular networks of lipids and proteins in (Rzhetsky & Gomez, 2001), (Gomez, Lo, & Rzhetsky, 2001) which exhibit scale-free characteristics and has interesting properties with respect to connectivity.

Preliminary investigations show that these aspects are driven by the major evolutionary events in growth in molecular networks; namely domain duplication and innovation. We are currently examining ways to inform the reaction of the network, possibly in terms of the neighbouring nodes, to the event of a node joining the network or leaving. This can be coupled with the component-based design of the local software agent on each participating node, and this is certainly an aspect of the work that we are keen to investigate further.

Conceptually, the proposed model satisfies the main properties of Digital Ecosystems. The Transaction model provides a practical framework for *interaction and engagement* of businesses. Despite their size they can use our coordination model as a *loosely couple* business interaction model which respect their local autonomy. This is the main reason for this model to be used for coordination model in OPAALS project (Razavi et al., 2007b). The *self-organising* procedure for creating Virtual Super Peers provides the sustainability of the environment through the stability function and satisfies *Balance* requirement (Chang & West, 2006a) of Digital Ecosystems. Further more for tolerating the dynamicity of environment, especially when the participants are Small and Medium Enterprises, the

optimised recovery mechanism have been provided which offers a self recovery method that is able to avoid full recovery when there is an alternative path for the interaction. As ongoing activities, new criteria such as RESTful frameworks and detailed simulations have been considered for exploring the model in different environment (Appendix II and III).

9.1 KEY CONTRIBUTION AND FUTURE PLANS

The main contribution of this report is to provide a loosely coupled solution, based on the context of digital (business) ecosystems. Specifically, it has been concerned with services, transactions, and network support within the digital ecosystem initiative. The proposed coordination model enables businesses to trigger their business transactions, when the consistency and recoverability of transactions are sustained. In term of the infrastructural support, the proposed dynamic topology of the network provided a fully distributed solution for stable environment which is resistant against failure.

The foremost features that figure prominently in this proposed digital ecosystems' design are:

- **The absence of a central point of command and control:** in term of network we have provided virtual super peers (chapter 7) and propose a distributed concurrency model (chapter 6 shows their pattern behaviours).
- **The absence of a single point of failure:** as the business activities are using a distributed framework for consistency (chapter 4) and recoverability (chapter 5) on one hand and our dynamic topology for infrastructure one the other hand avoid any single point of failure in theoretical level.
- **The distributed coordination model with full consistency and recoverability properties:** the behaviour of our coordination model (chapter 6) shows how the model can work in a fully distributed manner, when it is consistent (4.4) and recoverable (5.2).
- **The usage of loosely-coupled services:** Our model avoids the conventional pattern behaviours (3.3) and provides a loosely coupled solution which respect SMEs local autonomy.
- **The resilience to fragmentation:** The system has considerable resilience against fragmentation, this has been done by a dynamic algorithm for choosing VSPs (7.4) and the current implementation and simulations provide more evidences in that respect.
- **Allowing for a dynamic topology that continuously adapts to reflect the actual usage of the network in terms of business transactions:** The primary proposal of such a system can be found in introducing the digital ecosystem in chapter 3 but the full model has been finalised in chapter 7, where

choosing the VSPs' nodes are based on their stability in each VPTN.

As future plans, we can mention the continuous implementation paths of current research and multidisciplinary approach for infrastructure. Meanwhile other relevant approaches in term of deadlocks and potential overheads are another dimension for future work:

- **Extending the network topology by looking at biological models of growth in living organisms (Rzhetsky & Gomez, 2001), (Gomez, Lo, & Rzhetsky, 2001)**
- **Contributing in three different implementation paths (JXTA, RESTful & XMPP)**
- **Exploring the formal modelling for analysing new criteria such as algorithm complexity, deadlock issue, etc.**

References

Applegate, L., Austin, R., & McFarlan, F. W. (2002). *Corporate Information Strategy and Management: The Challenges of Managing in a Network Economy* (6th ed., p. 320). McGraw-Hill, Inc. New York, NY, USA.

Attiya, H., & Welch, J. (2004). *Distributed Computing: Fundamentals, Simulations and Advanced Topics* (2nd ed.). WileyBlackwell, New Jersey, USA

Barabási, A. L., & Albert, R. (1999). Emergence of Scaling in Random Networks. *Science*, *286*(5439), 509.

Barabási, A. L., Albert, R., & Jeong, H. (2000). Scale-free characteristics of random networks: the topology of the world-wide web. *Physica A: Statistical Mechanics and its Applications*, *281*(1-4), 69-77.

Bernstein, P. A., Hadzilacos, V., & Goodman, N. (1987). *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Bernstein, P. A., & Newcomer, E. (1997). *Principles of transaction processing*. Morgan Kaufmann. USA

Boley, H., & Chang, E. (2007). Digital Ecosystems: Principles and Semantics, *IEEE International Conference on Digital Ecosystems and Technologies.* Cairns, Australia. Inaugural IEEE-IES (pp. 398-403).

Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C.,Orchard, D. (2004). Web Services Architecture, W3C Working Group Note 11 February 2004. *World Wide Web Consortium, article available from: http://www. w3. org/TR/ws-arch*.

Bruni, R., Melgratti, H., & Montanari, U. (2005). Theoretical foundations for compensations in flow composition languages. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (pp. 209-220). ACM New York, NY, USA.

Butler, M., Hoare, T., & Ferreira, C. (2005). A trace semantics for long-running transactions. In *Proceedings ofCommunicating sequential processes: the first 25 years: Symposium on the Occasion of 25 Years of CSP*, London, UK, July 7-8, 2004 (Vol. 25, pp. 133–150). Springer -Verlag New York Inc.

Cabrera, L., Copeland, G., Feingold, M., Freund, R., Freund, T., Johnson, J., Joyce, S., Kaler, C., Klein, J., & Langworthy, D. (2005a). *Web Services Atomic Transaction (WS-AtomicTransaction)*. IBM, US: IBM . Retrieved from http://www-128. ibm. com/developerworks/library/specification/ws-tx/# atom.

Cabrera, L., Copeland, G., Feingold, M., Freund, R., Freund, T., Johnson, J., Joyce, S., Kaler, C., Klein, J., Langworthy, D., Little, M., et al. (2005b). Web Services Coordination (WS-Coordination). IBM DeveloperWorks. Retrieved from http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-tx/WS-Coordination.pdf.

Cabrera, L., Copeland, G., Feingold, M., Freund, R., Freund, T., Joyce, S., et al. (2005c). *Web Services Business Activity Framework (WS-BusinessActivity). 2005*. IBM DeveloperWorks.

Cabrera, L., Copeland, G., Johnson, J., & Langworthy, D. (2004, January 28). Coordinating Web Services Activities with WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity. Microsoft Corporation. Retrieved from http://msdn.microsoft.com/en-us/library/ms996526.aspx.

Ceponkus, A., Dalal, S., Fletcher, T., Furniss, P., Green, A., Pope, B., Inferior, A. (2002). Business transaction protocol V1.0. *OASIS Committee Specification, 3 June 2002*.

Chandler, A. D. (1990). *Scale and Scope: The Dynamics of Industrial Capitalism*. Harvard University Press. USA

Chang, E., Quaddus, M., & Ramaseshan, R. (2006). *The vision of DEBI Institute: digital ecosystems and business intelligence*. DEBII publication.

Chang, E., & West, M. (2006a). Digital Ecosystem - A next generation of the collaborative environment. In *The Eight International Conference on Information Integration and Web-Based Applications & Services, books@ ocg. at* (Vol. 214, pp. 3-23).

Chang, E., & West, M. (2006b). Digital Ecosystems and comparison to existing collaboration environment. *WSEAS Transactions on Environment and development*, *2*(11), pp. 1396-1404.

Chang, E., West, M., & Hadzic, M. (2006). A Digital Ecosystem for Extended Logistics Enterprises. In *Proceedings of the 11th International Workshop on Telework*. pp. 28-31 August 2006

Cournot, A. A. (1960). *Researches into the mathematical principles of the theory of wealth, 1838* (p. 213). A.M. Kelley.

Daho, Z. B., & Simoni, N. (2006). Towards Dynamic Virtual Private Service Networks: Design and Self-Management. In *10th IEEE/IFIP Network Operations and Management Symposium, 2006. NOMS 2006* (pp. 1-4).

Dalal, S., Temel, S., Little, M., Potts, M., Webber, J., & Technologies, A. (2003). Coordinating business transactions on the web. *IEEE Internet Computing*, *7*(1), 30-39.

Date, C. (2003). *An Introduction to Data Base Systems* (8th ed., p. 1024). Pearson Education.

Davey, B. A., & Priestley, H. A. (1990). *Introduction to Lattices and Order*. Cambridge University Press. UK

Deacon, A., Schek, H. J., & Weikum, G. (1994). Semantics-based multilevel transaction management in federatedsystems. In *Data Engineering, 1994. Proceedings. 10th International Conference* (pp. 452-461).

DEBII. (2009). DEBI Institute - Digital Ecosystems and Business Intelligence Institute. Retrieved May 8, 2009, from http://www.debii.curtin.edu.au/.

Dillard, D. D. (1967). *Economic Development of the North Atlantic Community: Historical Introduction to Modern Economics/Dudley Dillard*. Prentice-Hall. USA

Dillon, T.S., Wu, C. and Chang, E. (2007a). GRIDSpace: semantic grid services on the web - evolution towards a softgrid. *Third international conference on semantics, knowledge and grid*, 7-13, Xi'an, China, October 29-31, 2007.

Dillon, T.S., Wu, C. and Chang, E. (2007b). Reference architectural styles for service oriented computing. *Network and parallel computing: IFIP international conference*, 543-555, Dalian, China, September 18-21, 2007.

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

Eder, J., & Liebhart, W. (1994). A transaction-oriented workflow activity model. In *Proc. of the Ninth Int. Symposium on Computer and Information Sciences, Antalya, Turkey*.

Elmagarmid, A. K. (1992). *Database transaction models for advanced applications*. Morgan Kaufmann. USA

European Commission. (2005). *Recommendation 2003/361/EC: SME Definition*. ENTERPRISE AND INDUSTRY PUBLICATIONS (European Commission). Retrieved May 20, 2009, from http://ec.europa.eu/enterprise/enterprise_policy/sme_definition/index_en.htm.

European Commission. (2008). Technologies for Digital Ecosystems - Innovation Ecosystems Initiative - Specific Aims of the Digital Business Ecosystem. Retrieved May 8, 2009, from http://www.digital-ecosystems.org/.

Evesim - Digital Ecosystems Research. (2007). . Retrieved April 4, 2009, from http://www.evesim.org/.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures, 2000*. Thesis, University of California - Irvine.

Flypeer - Dynamic P2P Infrastructure — Project Kenai. (2008). . Retrieved April 4, 2009, from http://kenai.com/projects/flypeer.

Furnis, P., Dalal, S., Fletcher, T., Green, A., Ceponkus, A., & Pope, B. (2004). Business Transaction Protocol, version 1.1. 0 (November 2004). *URL http://docs. oasis-open. org/business-transaction/business_transaction-btp-1.1-spec-cd-01. pdf.–(20.07. 2006)*.

Furnis, P., & Green, A. (2005). Choreology Ltd. Contribution to the OASIS WS-Tx Technical Committee relating to WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity (November 2005), Ihttp. *www. oasis-open. org/committees/download. php/15808*.

Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., & Salem, K. (1991). Coordinating activities through extended sagas: a summary. In *Compcon Spring '91. Digest of Papers* (pp. 568-573). doi: 10.1109/CMPCON.1991.128867.

Gomez, S. M., Lo, S. H., & Rzhetsky, A. (2001). Probabilistic prediction of unknown metabolic and signal-transduction networks. *Genetics*, *159*(3), pp. 1291-1298.

Gray, J. (1992). *Benchmark handbook: for database and transaction processing systems*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA.

Gray, J., & Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann. USA

guigoh - Social Network. (2008). *Guigoh Social Network*. Retrieved April 4, 2009, from http://www.guigoh.com/Home.do.

Haghjoo, M. S. (1996). Scheduling and Scripting Mega Transaction. In *Proceedings of the 2nd International Computer Conference, Iran Computer Society, Amir Kabir University, Iran*.

Haghjoo, M. S., & Papazoglou, M. P. (1992). TrActorS: a transactional actor system for distributed queryprocessing. In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on* (pp. 682-689).

Haghjoo, M. S., Papazoglou, M. P., & Schmidt, H. W. (1993). A semantic-based nested transaction model for intelligent andcooperative information systems. In *Intelligent and Cooperative Information Systems, 1993., Proceedings of International Conference on* (pp. 321-331).

Hannon, B. (1997). The use of analogy in biology and economics: From biology to economics, and back. *Structural Change and Economic Dynamics*, *8*(4), pp. 471-488.

Hoare, C. A. R. (1985). *Communicating Sequential Processes.* Prentice Hall.

Hoffman, P., & Bray, T. (2006). Atom Publishing Format and Protocol (atompub). *IETF - Retrieved from< http://www. ietf. org/html. charters/atompub-charter. html*.

Iansiti, M., & Levien, R. (2004). *The Keystone Advantage: What the New Dynamics of Business Ecosystems Mean for Strategy, Innovation, and Sustainability*. Harvard Business School Press. USA

Iansiti, M., & Levien, R. (2004). *The Keystone Advantage: What the New Dynamics of Business Ecosystems Mean for Strategy, Innovation, and Sustainability* (p. 304). Harvard Business School Press. USA

IPTI. (2009). IPTI - Instituto de Pesquisas em Tecnologia e Inovação. Retrieved April 4, 2009, from http://ipti.org.br/site/.

Josephson, M. (1962). *The Robber Barons: the Great American Capitalists: 1861-1901*. A Harvest Book, Reprint, New York: Harvest Book, 1962

jxta: JXTA™ Community Projects. (2007). . Retrieved April 4, 2009, from https://jxta.dev.java.net/.

Kakeshita, T., & Xu, H. (1992). Transaction sequencing problems for maximal parallelism. In *Research Issues on Data Engineering, 1992: Second International Workshop onTransaction and Query Processing* (pp. 215-216).

Khare, R., & Taylor, R. N. (2004). Extending the representational state transfer (rest) architectural style for decentralized systems. In *Proceedings of the 26th International Conference on Software Engineering* (pp. 428-437). IEEE Computer Society

Washington, DC, USA.

Krause, P., Marinos, A., Moschoyiannis, S., Razavi, A., Zheng, Y., Kurtz, T., et al. (2008). Full Architecture Definition for Autopoietic P2P Network Version 1. Project Acronym: OPAALS, European Community, Framework 6, Contract No: 034824. Retrieved from http://files.opaals.org/OPAALS/Year_2_Deliverables/WP03/D3.3.pdf.

Kristiansen, L., Hansen, P. F., & Licciardi, C. (2008). TINA 1.0 Specifications. *TINA Service Architecture and Specifications*. TINA 1.0 DELIVERABLES AND SPECIFICATIONS, . Retrieved March 25, 2009, from http://www.tinac.com/specifications/specifications.htm.

Kshemkalyani, A. D., & Singhal, M. (2008). *Distributed Computing: Principles, Algorithms, and Systems* (illustrated edition.). Cambridge University Press, UK.

Kuhn, O., Elmagarmid, A. K., & Kuhn, E. (1993). Implementation of the Flex Transaction Model. *Bulletzn of the IEEE Technical Committee on Data Engsneering*, *16*(2), pp. 28-32.

Lewin, R. (1999). *Complexity: Life at the Edge of Chaos*. University of Chicago Press. USA

Li, Xu, H., & Liu, T. L. (2002). Coordinating transaction model in CDBMS. In *The 7th International Conference on Computer Supported Cooperative Work in Design, 2002.* (pp. 421-425).

Li, Z., & Mohapatra, P. (2004). QRON: QoS-aware routing in overlay networks. *IEEE Journal on Selected Areas in Communications*, *22*(1), pp. 29-40.

Limthanmaphon, B., & Zhang, Y. (2004). Web service composition transaction management. In *Proceedings of the 15th Australasian database conference - Volume 27* (pp. 171-179). Australian Computer Society, Inc. Darlinghurst, Australia, Australia.

Martini, B., Baroncelli, F., & Castoldi, P. (2005). A novel service oriented framework for automatically switched transport network. In *9th IFIP/IEEE International Symposium on Integrated Network Management, 2005. IM 2005.* (pp. 295-308).

Mazurkiewicz, A. (1977). Concurrent Program Schemes and their Interpretation. Technical report PB-78, DAIMI, Arhus University. *Computer Science Department,Denmark (July 1977)*.

Mazurkiewicz, A. (1988). Basic notions of trace theory, in de Bakker, de Roever and Rozenberg (eds.), Linear Time, Branching Time and Partial Orders in Logics and Models for Concurrency. *Springer Lecture Notes in Computer Science*, *354*, pp. 285-363.

Milner, R. (1980). *A calculus of communicating systems, volume 92 of Lecture Notes in Computer Science*. Springer-Verlag, New York, Inc. Secaucus, NJ, USA.

Moore, J. F. (1993). Predators and Prey: A New Ecology of Competition. *Harvard Business Review*, *71*, pp. 75-83.

Moschoyiannis, S. (2004). A set-theoretic framework for component composition. *Fundamenta Informaticae*, *59*(4), pp. 373-396.

Moschoyiannis, S. (2005). *Specification and Analysis of Component-Based Software in a True-Concurrent Setting*. PhD thesis, University of Surrey, UK

Moschoyiannis, S., & Darking, M. L. (2008). Consensus detailed architecture of the (OPAALS) Digital Ecosystems . Project Acronym: OPAALS project, European Community, Framework sixth, Contract Number: 034824. Retrieved from http://files.opaals.org/OPAALS/Year_2_Deliverables/WP03/D3.6.pdf.

Moschoyiannis, S., Razavi, A., & Krause, P. (2007). Transaction Vectors; A Non-Interleaving Semantics for Long-Running Transactions. OPAALS Conference, Rome, Italy.

Moschoyiannis, S., Razavi, A., & Krause, P. (2008). Transaction Scripts: Making Implicit Scenarios Explicit. *In ETAPS 2008 - FESCA'08, ENTCS. Elsevier, 2008. In Press*.

Moschoyiannis, S., Razavi, A., Zheng, Y., & Krause, P. (2008). Long-running Transactions: Semantics, schemas, implementation. In *Digital Ecosystems and Technologies, 2008. DEST 2008. 2nd IEEE International Conference on* (pp. 20-27). doi: 10.1109/DEST.2008.4635168.

Moschoyiannis, S., Shields, M. W., & Krause, P. J. (2005). Modelling Component Behaviour with Concurrent Automata. *Electronic Notes in Theoretical Computer Science*, *141*(3), pp. 199-220.

Moss, J. E. B. (1985). *Nested transactions*. MIT Press Cambridge, Mass.

Moss, J. E. B. (1987). Log-based recovery for nested transactions. In *Proc. 13th Int'l Conf. on Very Large Data Bases* (pp. 427-432).

Mundlapudi, B. (2006). Implementing High Performance Web Services Using JAX-WS 2.0. Sun Microsystems, developer & technical articles for web services, Retrieved May 29, 2009, from http://java.sun.com/developer/technicalArticles/WebServices/high_performance/.

Nachira, F. (2002a). Towards a Network Of Digital Business Ecosystems Fostering the Local Development. *European Commission Discussion Paper. Bruxelles*. Retrieved from http://hdl.handle.net/2038/529.

Nachira, F. (2002b). *Discussion paper, e-Business Unit, IST Thematic Priority, European Commission*. Brussels, Belgium.

Nachira, F., Dini, P., & Nicolai, A. (2007). A Network of Digital Business Ecosystems for Europe: Roots, Processes and Perspectives. *Digital Business Ecosystems. European Commission, Bruxelles. www. digital-ecosystems. org/book/de-book2007. html*.

Nachira, F., Nicolai, A., Dini, P., Le Louarn, M., & Leon, L. R. (2007). *Digital Business Ecosystems* (Vol. 9, p. 2008).

Nielsen, M., Plotkin, G. D., & Winskel, G. (1981). Petri nets, event structures and domains. *Theoretical Computer Science*, *13*(1), pp. 85–108.

Nodine, M. H. (1993). Supporting long-running tasks on an evolving multidatabase usinginteractions and events. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems, 1993.,* (pp. 125-132).

Nodine, M. H., & Zdonik, S. B. (1994). Automating compensation in a multidatabase. In *Proceedings of the Twenty-Seventh Hawaii International Conference onSystem Sciences, 1994. Vol. II: Software Technology,* (Vol. 2).

Papazoglou, M., Dells, A., Bouguettaya, A., & Haghjoo, M. (1997). Class library support for workflow environments and applications. *IEEE Transactions on Computers*, *46*(6), pp. 673-686.

Papazoglou, M. P. (2003). Service-Oriented Computing: Concepts, Characteristics and Directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering* (Vol. 3). Washington: IEEE Computer Society Press, December.

Papazoglou, M. P. (2005). Extending the Service-Oriented Architecture. *Business Integration Journal*, *7*(1), pp. 18-21.

Papazoglou, M. P., Delis, A., Haghjoo, M., & Bouguettaya, A. (1996). Language support for long-lived concurrent activities. In *Proceedings of the 16th International Conference onDistributed Computing Systems, 1996.,* (pp. 698-705).

Papazoglou, M. P., & Georgakopoulos, D. (2003). Service-Oriented Computing. *Communications of the ACM*, *46*(10), pp. 25-28.

Papazoglou, M. P., Traverso, P., Dustdar, S., Leymann, F., & Kramer, B. J. (2006). Service-oriented computing: A research roadmap. *Service Oriented Computing (SOC)*, Dagstuhl Seminar Proceedings (05462) pp. 1-29

Payton, I. J., Fenner, M., & Lee, W. G. (2002). *Keystone Species: The Concept and Its Relevance for Conservation Management in New Zealand*. Dept. of Conservation.

Power, M. E., Tilman, D., Estes, J. A., Menge, B. A., Bond, W. J., Mills, L. S., et al. (1996). Challenges in the quest for keystones. *Bioscience*, *46*(8), pp. 609-620.

Prochazka, M. (1999). Advanced transactions in enterprise javabeans. *Lecture Notes in Computer Science, Volume 1999/2001, Springer Berlin / Heidelberg,* pp. 215-230.

Prochazka, M. (2001). Advanced Transactions in Enterprise JavaBeans. In *Engineering Distributed Objects* (pp. 215-230). Retrieved April 23, 2009, from http://dx.doi.org/10.1007/3-540-45254-0_19.

Ramakrishnan, R., & Gehrke, J. (2003). *Database management systems*. McGraw-Hill, USA.

Ramampiaro, H., & Nygård, M. (1999). Cooperative database system: A constructive review of cooperative transaction models. In *Proceedings of the 1999 International Symposium on Database Application in Non-Traditional Environment (DANTE 99)* (pp. 315–324).

Ravasz, E., & Barabási, A. L. (2003). Hierarchical organization in complex networks. *Physical Review E*, *67*(2), pp. 26112.

Razavi, A. R. (1999). *Design and Implementation of Recovery Management for Mega Transaction and Model implementation*. MSc Thesis, Iran University of Science and Technology, Tehran, Iran.

Razavi, A. R. (2009). PPNA - An Open dgital environment to support business ecosystems (Source code for this Paper in Peer-to-Peer Networking and Applications Springer Journal) - http://personal.cs.surrey.ac.uk/personal/pg/A.Razavi/ppna/. Retrieved March 17, 2009, from http://personal.cs.surrey.ac.uk/personal/pg/A.Razavi/ppna/.

Razavi, A. R., Krause, P., & Moschoyiannis, S. (2006). Deliverable D24.5: DBE Distributed Transaction Model. Project Acronym: DBE, European Community, Framework, Contract No: 507953.

Razavi, A. R., Malone, P. J., Moschoyiannis, S., Jennings, B., & Krause, P. (2007). A Distributed Transaction and Accounting Model for Digital Ecosystem Composed Services. In *Digital EcoSystems and Technologies Conference, 2007. DEST'07. Inaugural IEEE-IES* (pp. 63-66).

Razavi, A. R., Marinos, A., Moschoyiannis, S., & Krause, P. (2009). RESTful Transactions supported by the Isolation Theorems, *The Ninth International Conference on Web Engineering (ICWE 2009)*, San Sebastian, Spain (in press).

Razavi, A. R., & Moschoyiannis, S. (2008). FESCA WORKSHOP 2009 xml Sources and relevant Java source codes for presented paper in the workshop, Retrieved December 22, 2008, from http://www.computing.surrey.ac.uk/personal/st/S.Moschoyiannis/trnscripts/.

Razavi, A. R., Moschoyiannis, S., & Krause, P. (2007a). Preliminary Architecture for Autopoietic P2P Network focusing on Hierarchical Super-Peers, Birth and Growth Models. OPAALS Project (OPAALS project Deliverable D3.1). Retrieved from http://files.opaals.org/OPAALS/Year_1_Deliverables/WP03/OPAALS_D3.1-final_submitted.pdf.

Razavi, A. R., Moschoyiannis, S., & Krause, P. (2007b). Report on formal analysis of autopoietic P2P network, together with predictions of performance. OPAALS project (Deliverable D3.2). European Community, Framework sixth, Contract No: 034824. Retrieved from http://files.opaals.org/OPAALS/Year_1_Deliverables/WP03/OPAALS_D3.2_final.pdf.

Razavi, A. R., Moschoyiannis, S., & Krause, P. (2007c). A Coordination Model for Distributed Transactions in Digital Business EcoSystems. *Digital Ecosystems and Technologies (DEST 2007), IEEE Computer Society Press, Los Alamitos*.

Razavi, A. R., Moschoyiannis, S., & Krause, P. (2007d). Concurrency Control and Recovery Management for Open e-Business Transactions. In *Communicating Process Architectures 2007: WoTUG-30: Proceedings of the 30th WoTUG Technical Meeting, 8-11 July 2007, University of Surrey, Guildford, United Kingdom* (p. 267).

Razavi, A. R., Moschoyiannis, S., & Krause, P. (2007e). The Agent-based Digital Business Ecosystem: towards the Semantic Web. OPAALS Conference, ROME.

Razavi, A. R., Moschoyiannis, S., & Krause, P. (2008a). A scale-free business network for digital ecosystems. In *Digital Ecosystems and Technologies, 2008. DEST 2008. 2nd IEEE International Conference on* (pp. 241-246).

Razavi, A. R., Moschoyiannis, S., & Krause, P. (2008b). A Self-Organising Environment for

Evolving Business Activities. In *Computing in the Global Information Technology, 2008. ICCGI '08. The Third International Multi-Conference on* (pp. 277-283). doi: 10.1109/ICCGI.2008.37.

Razavi, A. R., Moschoyiannis, S., & Krause, P. (2009). An open digital environment to support business ecosystems. *Peer-to-Peer Networking and Applications Springer Journal*.

Richardson, L., & Ruby, S. (2007). *Restful web services* (p. 446). O'Reilly. Retrieved April 1, 2009, from http://portal.acm.org/citation.cfm?id=1406352.

Rothschild, M. (1995). *Bionomics: Economy As Ecosystem* Henry Holt & Company; Reissue edition (April 1995), New York, USA.

Rzhetsky, A., & Gomez, S. M. (2001). *Birth of scale-free molecular networks and the number of distinct DNA and protein domains per genome* (Vol. 17, pp. 988-996). Oxford University Press, UK.

Sahai, A., Machiraju, V., Sayal, M., Van Moorsel, A., Casati, F., & Jin, L. J. (2002). Automated SLA monitoring for web services. *Lecture notes in computer science*, 28-41.

Service-oriented architecture - Wikipedia, the free encyclopedia. (n.d.). . Retrieved December 21, 2008, from http://en.wikipedia.org/wiki/Service-oriented_architecture.

Shields, M. W. (1985). Concurrent machines. *The Computer Journal*, *28*(5), pp. 449-465.

Shields, M. W. (1997). *Semantics of parallelism*. Springer, New York, USA.

Shields, M. W. (1979). *Adequate path expressions*. *Lecture Notes in Computer Science, Volume 70/1979,* Springer Berlin / Heidelberg, pp. 249-265.

Singh, M. P. (1998). A customizable coordination service for autonomous agents. *Lecture Notes in Computer Science*, *1365*, pp. 93-106.

Singh, M. P., & Huhns, M. N. (2005). *Service-Oriented Computing: Semantics, Processes, Agents*. WileyBlackwell, USA.

Strømmen-Bakhtiar, A., & Razavi, A. R. (2008). Emerging Problems in the Digital Business Ecosystem. *OPAALS conference 2009*, Tampere, Finland.

Tanenbaum, A. S., & Steen, M. V. (2008). *Distributed Systems: Principles and Paradigms* (2nd ed.). Pearson Education, USA.

TechIDEAS. (2007). fada - Federated Advanced Directory Architecture. Retrieved March 29, 2009, from http://fada.sourceforge.net/.

Toynbee, A., & Jowett, B. (1887). *Lectures on the Industrial Revolution of the 18th Century in England* Longmans, Green and Co.; fifth edition (January 1, 1896), UK. p. 263.

Verharen, E. M., & Papazoglou, M. P. (1998). Introducing contracting in distributed transactional workflows. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences, Volume 7, 6-9 Jan. 1998 pp. 324 - 333*.

Vogt, F. H., Zambrovski, S., Gruschko, B., Furniss, P., & Green, A. (2005). Implementing Web Service Protocols in SOA: WS-Coordination and WS-BusinessActivity. *CECW*, *5*, 21–

28.

Xiao W., Lu Z., Li B., & Sarem, M. (2001). Transaction management in mobile multidatabase systems. In *Proceedings. 2001 International Conference onComputer Networks and Mobile Computing, 2001.* pp. 513-518.

Yang, B., & Garcia-Molina, H. (2002). Improving search in peer-to-peer networks. In *INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS* (Vol. 22, pp. 5-14). IEEE Computer Society; 1999.

Yang, B., & Garcia-Molina, H. (2003). Designing a super-peer network. In *Proceedings of the International Conference on Data Engineering* (pp. 49-62). IEEE Computer Society Press; 1998.

Yang, J., Papazoglou, M. P., & van den Heuvel, W. J. (2002). Tackling the Challenges of Service Composition in e-Marketplaces. In *Proceedings of the 12th International Workshop on Research Issues on Data Engineering: Engineering E-Commerce/E-Business Systems (RIDE-2EC 2002), San Jose, CA, USA*.

APPENDICES

10 Appendix I: Formal modelling for Pattern behaviour of Long-running Transactions

In chapter 6, the analysing pattern behaviour has been explored. For extending the formal model to investigate transaction pattern behaviour the vector language by Shields (Shields, 1997) and Moschoyiannis (Moschoyiannis, 2005) is adapted. The proposed model in this appendix, is Moschoyiannis' motivation, which during the collaboration with Razavi in OPAALS has been introduced:

- A. Razavi, S. Moschoyiannis and P. J. Krause. Report on formal analysis of autopoietic P2P network, together with predictions of performance. OPAALS project Deliverable D3.2, 2007.
- S. Moschoyiannis, A. Razavi, Y. Zheng and P. Krause (2008) Long-Running Transactions: semantics, schemas, implementation, In Proc. of IEEE Int'l Conf. on Digital Ecosystems and Technologies (IEEE-DEST 2008), IEEE Computer Society.
- S. Moschoyiannis, A. Razavi, and P. Krause (2008) Transaction Scripts: making implicit scenarios explicit. In Proc. of ETAPS 2008 - Formal Foundations of Emebedded Software and Component-Based Software Architectures (FESCA'08), ENTCS, Elsevier

The appendix is a brief version of full formal analysis in OPAALS project Deliverable D3.2.

10.1 Transaction vectors: a language-based representation

We may now start to describe the pattern actions of long-running transactions more formally. As mentioned before, our objective is to get a thorough understanding of the behaviour the underlying service composition need to exhibit for successful commit or compensation of the transaction as a whole.

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 construction, and there is no need for additional notation. A transaction T, then, is associated with a set of leaves L which consists of a set of basic services S, a set of data-oriented coordinators D and a set of delegation coordinators Dlg. Hence, $L = S \cup D \cup Dlg$.

In this appendix we introduce a formal language for describing long-running transactions. 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. Note that services are offered in a digital ecosystem for business from different service providers and it is important that we defer from interfering with the local state of the service execution. The service-oriented architecture for distributed transactions reinforces our interest in all environmentally observable actions inside and outside a transaction. That 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.

A transaction may thus be associated with a finite set of events or *significant events* (Singh, 1998) or *actions* that may occur (on its subtransactions) upon activation, e.g. service invocation, initialisation, commitment, service return, release result (return), termination, abort, etc. We denote this set of actions of a transaction by M.

These actions take place on the leaves and therefore it seems appropriate to say that each leaf is in turn associated with a set of actions 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 actions formed over the corresponding set $\mu(l)$, $l \in$ L. We may thus describe the behaviour of the transaction by assigning such sequences to each of its leaves.

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

By $\mu(l)^*$ we denote 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 actions that have occurred on that leaf.

When an action occurs on a leaf of the transaction tree, that is to say when an action associated with some subtransaction takes place, it appears on a new transaction vector at the appropriate coordinate. For example, the vector

$$(s_1, \Lambda, \Lambda)$$

describes that portion of behaviour of the transaction in which an action s_1 (e.g. service invocation) has taken place on the corresponding service allocated to the first coordinate. The vector

$$(s_1, s_2, \Lambda)$$

describes that portion of behaviour in which both s_1 and s_2 have happened on the corresponding services while the vector

$$(s_1 s_3, s_2, \Lambda)$$

describes an occurrence of s_1 and an occurrence of s_3. on the service corresponding to the first coordinate, and an occurrence of s_2 on that of the third coordinate. Nothing has happened on the service corresponding to the third coordinate.

In this sense, each transaction vector provides a *snapshot* of behaviour in which the transaction has executed the actions appearing on the vector's coordinates – the vector tells us what actions have already occurred and on which part of the transaction tree.

This vector-based description of behaviour allows recording the actions of a transaction as these occur on the multiple services involved in the execution of the transaction. Readers familiar with process algebras like CSP or CCS can understand each particular coordinate of the vector description as a sequential CSP process. In this sense, the transaction vectors can be understood as the Cartesian product of sequential processes describing each leaf in a transaction tree.

It can be seen from the examples given above that there is already an ordering among actions on a particular subtransaction (e.g. s_1 followed by another s_1). This vector-based behavioural description of transactions can also capture the orderings between different subtransactions, which amounts to actions appearing on different vector coordinates. This requires however a more careful consideration of the mathematical properties of such vectors which we briefly describe in the following section.

Before examining the mathematical properties of our construction so far, we introduce a specific kind of transaction vectors, which is used in our model to describe actions (events or activations) within a transaction.

Definition 2. (Column vectors.) Let T be a transaction and V_T its set of transaction vectors. We define

$$A_T = \{\underline{a} \in V_T \setminus \{\underline{\Lambda}_T\} : l \in L \Rightarrow |\, \underline{a}(l) \,| \leq 1\}$$

where $|\, x \,|$ denotes the length of sequence *x*. We refer to elements of A_T as *column vectors*.

Thus, the vectors of Definition 2 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 (s_1, Λ, Λ) represents the occurrence of an action s_1 on the

sub-transaction associated with the first coordinate.

We will use the term *transaction language* to refer to a subset V of all possible vectors V_T formed over a given transaction T. Hence, a transaction T comes with a language V, where $V \subseteq V_T$. The idea is that the particular set of transaction vectors for a specific transaction expresses the ordering constraints necessary in the corresponding service orchestration.

10.2 ORDER-THEORETIC PROPERTIES OF TRANSACTION VECTORS

In what follows we describe the basic order-theoretic properties of transaction vectors since this is what allows us to define operations on vectors. These are important in determining the coordination of the transaction in terms of its underlying service invocations. We will see how the order structure of sets of such vectors expresses ordering constraints on actions inferred by the execution of the various subtransactions inside a long-running transaction.

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.

First, let us establish our notation. If x and z are sequences, we write x.z for the concatenation of x and z. As is well known this operation on sequences is associative with identity Λ, where Λ denotes the empty sequence. We also have a partial order on sequences given by $x \leq z$ if and only if there exists a sequence y such that x.y = z, and this partial order has a bottom element Λ. It is also well-known that the operation '.' is cancellative, which means that if $x \leq z$, then the sequence y such that x.y = z is unique. We shall denote this sequence by z / x. Finally, recall that if x, y, z are sequences such that $x, y \leq z$, then either $x \leq y$ or $y \leq x$.

We may now lift these well-known operations on sequences onto transaction vectors. This is done formally in the following definition.

Definition 3. (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 $\underline{w}(l) = \underline{u}(l).\underline{v}(l)$, for each $l \in L$ (*concatenation*)
- $\underline{u} \leq \underline{v}$ iff $\underline{u}(l) \leq \underline{v}(l)$, for each $l \in L$ (*prefix ordering*)
- *glb*($\underline{u},\underline{v}$) to be the vector $\underline{w}$ such that $\underline{w}(l)=\min(\underline{u}(l),\underline{v}(l))$, for each $l \in L$
- *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 $l \in L$
- if $\underline{u} \leq \underline{v}$, then we define $\underline{v}$ / $\underline{u}$ to be the unique $\underline{z} \in V_T$ 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,

$$(s_1 s_3, s_2, \Lambda).(\Lambda, s_4, \Lambda) = (s_1 s_3, s_2 s_4, \Lambda)$$

Transaction vectors can be seen to be built up from the empty vector $\underline{\Lambda}_T$ by a series of concatenations with column vectors (Definition 2) that represent actions. In fact, in describing the behaviour of a transaction we are interested only in those vectors describing (orderings of) actions that we expect the transaction to engage in during the course of its execution. This is the subset of all possible transaction vectors, over a given T, we referred to as *transaction language*.

For example, consider a transaction with three leaves (basic services) in which a service s1 is intended to execute first, then a service s2 (which uses results of s1, and is thus dependent on s1) and after that execution continues with another service s3. This kind of (sequential) behaviour can be modelled by a series of concatenations. We assume the actions are labelled by the service name here, so we have actions a1 = (s1, Λ, Λ) for service invocation s1, a2 = (Λ, s2, Λ) for service invocation s2, and a3 = (Λ, Λ, s3) for service invocation s3.

Initially nothing has happened. This is described by the empty vector $\underline{\Lambda}_T$ = (Λ, Λ, Λ).

Then, s1 occurs. This is described in a vector $\underline{v}$ which is obtained by concatenating $\underline{\Lambda}_T$ with the column vector representing the action s1. Hence, we have

$$\underline{\Lambda}_T.\underline{a}_1 = (\Lambda, \Lambda, \Lambda).(s1, \Lambda, \Lambda) = (s1, \Lambda, \Lambda) = \underline{v}$$

Then, s2 occurs. This is described in a vector $\underline{u}$ which is obtained by concatenating $\underline{v}$ (the latest behaviour we have) with the column vector representing the corresponding action s2, here $\underline{a}_2$. Hence, we have

$$\underline{v}.\underline{a}_2 = (s1, \Lambda, \Lambda).\ (\Lambda, s2, \Lambda) = (s1, s2, \Lambda) = \underline{u}$$

Then, s3 occurs. This is described in yet another vector $\underline{w}$ which is obtained by concatenating $\underline{u}$ (the latest behaviour we have) with the column vector representing the corresponding action s3, here $\underline{a}_3$. Hence, we have

$$\underline{u}.\underline{a}_3 = (s1, s2, \Lambda).\ (\Lambda, \Lambda, s3) = (s1, s2, s3) = \underline{w}$$

In the next Section we shall impose conditions on transaction languages that ensure they

comprise transaction vectors which are obtained in this way, and the coordination of the underlying services it determines corresponds to intended behaviour of the transaction only.

The ordering amongst vectors is defined in terms of the usual prefix ordering operation on sequences appearing on their coordinates. For example,

$$(s_1, s_2, \Lambda) \leq (s_1 s_3, s_2, \Lambda) \text{ since } s_1 \leq s_1 s_3 \text{ and } s_2 \leq s_2 \text{ and } \Lambda \leq \Lambda$$

In other words, the vector $\underline{v}$ '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. For example,

$$(s_1 s_3, s_2, \Lambda) \text{ and } (s_1 s_5, s_2, \Lambda)$$
or
$$(s_1, \Lambda, \Lambda) \text{ and } (\Lambda, s_2, \Lambda)$$

It turns out that such vectors describe either parallel or alternative behaviours of the transaction in question, and this will be further discussed in Section 10.4.

It is important to note that these two fundamental operations, *concatenation and prefix-ordering*, on transaction vectors are performed *coordinate-wise* in our model and this simplifies the mathematics of it and allows for relatively straightforward proofs.

The operations *glb*() and *lub*() of Definition 3 give the greatest lower bound and the least upper bound, respectively of $\underline{u}$, $\underline{v} \in V_T$, in the usual sense of lattices and domain theory (Davey & Priestley, 1990). For example, for vectors $\underline{v}$ = (s1, s2, Λ) and $\underline{u}$ = (s1, Λ, s3) the *glb*($\underline{u}$, $\underline{v}$) is computed as follows,

$$glb(\underline{u}, \underline{v}) = glb(\text{ (s1, s2, Λ), (s1, Λ, s3) }) = (s1, \Lambda, \Lambda)$$

since

$$\min(\underline{u}(1), \underline{v}(1)) = s1$$

$$\min(\underline{u}(2), \underline{v}(2)) = \Lambda$$

$$\min(\underline{u}(3), \underline{v}(3)) = \Lambda$$

Similarly, their least upper bound *lub*($\underline{u}$, $\underline{v}$) is computed as follows,

$$lub(\underline{u}, \underline{v}) = lub(\text{ (s1, s2, Λ), (s1, Λ, s3) }) = (s1, s2, s3)$$

since

$$\max(\underline{u}(1), \underline{v}(1)) = s1$$

$$\max(\underline{u}(2), \underline{v}(2)) = s2$$

$$\max(\underline{u}(3), \underline{v}(3)) = s3$$

These operations are central to the treatment of concurrency in our approach and also have an important role to play in defining the properties that ensure the well-formedness of the behavioural description, as will be discussed in the next Section.

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. It is also particularly useful, together with the ordering '$\lhd$' (cf Definition 6), in deriving a transition relation that allows to associate the vector-based description of behaviour with automata and asynchronous transition systems (Shields, 1985), in giving a state-based description of the interactions involved (Moschoyiannis et al., 2005).

It can be shown (by an adaptation of the proof found in (Moschoyiannis et al., 2005), which is in turn based on that originally perceived in (Shields, 1997)) that a set of transaction vectors equipped with the operations of concatenation and prefix ordering of Definition 3 forms a monoid[20] and a partial order. $\underline{\Lambda}_T$ 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

1. a monoid under '.' and identity $\underline{\Lambda}_T$
2. a partial order under $\leq$ and bottom element $\underline{\Lambda}_T$

Proof.

For (1), it suffices to show that V_T is closed under '.' and that '.' is associative. We argue coordinate-wise. Let $\underline{u}, \underline{v} \in V_T$ and $l \in L_T$. Since $\underline{u}(l), \underline{v}(l) \in \mu(l)^*$ we have that $(\underline{u}.\underline{v})(l) \in \mu(l)^*$. Hence, $\underline{u}, \underline{v} \in V_T$, proving that V_T is closed under '.'. Now, for associativity, if $\underline{u}, \underline{v}, \underline{w} \in V_T$, then for each $l \in L_T$ we have

$$(\underline{u}.(\underline{v}.\underline{w}))(l) = \underline{u}(l).(\underline{v}.\underline{w})(l) = (\underline{u}(l).\underline{v}(l)).\underline{w}(l) = (\underline{u}.\underline{v})(l).\underline{w}(l) = ((\underline{u}.\underline{v}).\underline{w})(l)$$

Since $(\underline{u}.(\underline{v}.\underline{w}))(l) = ((\underline{u}.\underline{v}).\underline{w})(l)$, for all $l \in L_T$, we have that $\underline{u}.(\underline{v}.\underline{w}) = (\underline{u}.\underline{v}).\underline{w}$. so '.' is associative.

For (2), we need to show that '$\leq$' is reflexive, antisymmetric and transitive. Again, we

[20] Recall that a monoid is a semi-group with identity.

argue coordinate-wise. Let $\underline{u}$, $\underline{v}$, $\underline{w} \in V_T$. Since $\underline{u}(l) \leq \underline{u}(l)$, for all $l \in L_T$, we have that $\underline{u} \leq \underline{u}$, giving reflexivity. If $\underline{u} \leq \underline{v}$ and also $\underline{v} \leq \underline{u}$, then $\underline{u}(l) \leq \underline{v}(l)$, for all $l \in L_T$, and $\underline{v}(l) \leq \underline{u}(l)$, for all $l \in L_T$, so we deduce that $\underline{u}(l) = \underline{v}(l)$, for all $l \in L_T$, which implies that $\underline{u} = \underline{v}$, proving antisymmetry. Finally, if $\underline{u} \leq \underline{v}$ and $\underline{v} \leq \underline{w}$, then $\underline{u}(l) \leq \underline{v}(l)$, for all $l \in L_T$, and $\underline{v}(l) \leq \underline{w}(l)$, for all $l \in L_T$, so $\underline{u}(l) \leq \underline{w}(l)$, for all $l \in L_T$, which in turn implies that $\underline{u} \leq \underline{v}$, proving transitivity. □

We note that a transaction language $V \subseteq V_T$ is not a monoid in general as it is not closed under '.' unless it contains the empty vector $\underline{\Lambda}_T$. We will see in Section 10.3 that this is the case in *discrete* (cf Definition 5) transaction languages.

The incomparable vectors in the partial order $(V_T, \leq)$ allow to introduce a notion of independence between transaction vectors, which is central to expressing true-concurrency within our model. This builds on earlier work on describing parallel behaviour in Shield's *behaviour vectors*(Shields, 1997) where the notion of independence found in *Mazurkiewicz traces*(Mazurkiewicz, 1988) is lifted onto vectors. This development is the topic of Section 10.4 where we are concerned with modelling concurrent actions of a long-running transaction.

10.3 WELL-FORMEDNESS OF THE BEHAVIOURAL DESCRIPTION OF A TRANSACTION

In describing the behaviour of transaction we are interested in the actions (activations) on its sub-transactions. These are captured in our model using column vectors (Definition 2). 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 us the behaviour of the transaction in terms of activations or actions of its sub-transactions and can be used to enforce the coordination of the underlying services.

We have seen that transaction vectors are obtained by coordinate-wise concatenation (Definition 3), for example

$$(x_1, x_2, x_3).(y_1, y_2, y_3) = (x_1 y_1, x_2 y_2, x_3 y_3)$$

In such a behavioural description of a transaction, transaction vectors can be seen to be built up from the empty vector by a series of concatenations with the column vectors (Moschoyiannis, 2005), each of whose coordinates is either empty or contains a single event/action.

For example, the column vector $\underline{a} = (s_1, \Lambda, \Lambda)$ represents the activation of the leaf corresponding to the first coordinate. If s1 is intended to occur only after both s3 and s4 have, then this is described in the transaction vector $\underline{v} = (s_1, s_3, s_4)$ which is obtained as

$$\underline{u}.\underline{a} = (\Lambda, s_3, s_4).(s_1, \Lambda, \Lambda) = (s_1, s_3, s_4)$$

In order to ensure that vectors associated with a transaction are the result of concatenations with column vectors only, the set of transaction vectors must satisfy certain properties, namely *discreteness* and *local left-closure*. We introduce these properties next.

In describing the behaviour of a transaction in terms of the coordination of its sub-transactions, we want to capture the fact that a system's computations always have a starting point, and ensure that only a finite number of events 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 on the set, then their least upper and greatest lower bounds are also in the set. This is formally put in the following definition.

Definition 4. (**Discreteness.**) Let $V \subseteq V_T$, then V is *discrete* if and only if, $\underline{\Lambda}_T \in V$ and whenever $\underline{u}, \underline{v}, \underline{w} \in V$ such that $\underline{u}, \underline{v} \leq \underline{w}$ then

(i) $lub(\underline{u}, \underline{v}) \in V$

(ii) $glb(\underline{u}, \underline{v}) \in V$

Note that $lub(\underline{u}, \underline{v}) \in V$ is understood as asserting that $lub(\underline{u}, \underline{v})$ is defined, i.e. the least upper bound of $\underline{u}, \underline{v}$ exists. This property builds on the notion of consistently complete subsets, as discussed in (Shields, 1997), and further requires that the least upper and greatest lower bounds belong to the set.

It can be seen that discreteness imposes a finiteness constraint in the sense that it excludes infinite ascending or descending chains of actions with respect to time ordering. It ensures that situations like those resulting in Zeno-type paradoxes will never arise. The famous Zeno paradoxes, in which the philosopher seeks to demonstrate the impossibility of motion, are examples of a non-discrete representation of system behaviour[21].

Consider a transaction T which involves the execution of two basic services, and let

$$V_T = \{(\Lambda, \Lambda), (s_1 s_1, \Lambda), (\Lambda, s_2 s_2), (s_1 s_1, s_2 s_2)\}$$

We observe that V_T is discrete (by checking against Definition 4). Indeed, V_T is a lattice in which greatest lower bounds are computed coordinate-wise. However, the corresponding transaction language has the counter-intuitive property that although four actions have occurred, there are only two elements, namely ($s_1 s_1$, Λ) and (Λ, $s_2 s_2$) to represent that portion of behaviour. The two vectors represent the second of the two actions only, at each service. We would like to eliminate such situations.

[21] Zeno's paradox with arrow and that involving Achilles and the tortoise are discussed in view of computer science in [Shi97]. The conclusion drawn from Zeno's arguments in this case is not that motion is impossible, but that the behavioural description used is not *discrete*.

In order to obtain a precise description of discrete behaviour, we further require that every occurrence of an action (e.g. service invocation, partial result, commitment) 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.

Definition 5. (**Local left-closure.**) Let $V \subseteq V_T$, $l \in L$ and $x \in \beta(l)^*$. Then, V is locally left-closed if and only if, whenever $\underline{v} \in V$ and $\Lambda < x \leq \underline{v}(l)$, then there exists $\underline{u} \in V$ such that $\underline{u} \leq \underline{v}$ and $\underline{u}(l) = x$.

The above definition says that whenever there is a sequence of actions on some sub-transaction (or local coordinator) which is less or equal than some other sequence appearing in some transaction vector in *V*, then there is some other vector in *V* which describes an earlier part of behaviour and has that sequence on the corresponding coordinate. In fact, 'local' comes from the fact the property is considered at the vector coordinate level and thus applies to individual sub-transactions or local coordinators and 'left-closure' reflects the fact that earlier parts of a given behaviour are themselves behaviours.

Effectively, 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, as demonstrated in the example given prior to Definition 5. 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, and on each appropriate leaf of the transaction tree.

Primes play a central role in the more general theory of parallelism (Shields, 1997) and in particular with respect to associating vector languages with behavioural presentations. For the purposes of the present report, 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 among vectors in a transaction language V is based on the relation '$\lhd$' which we define next.

Definition 6 (**Cover.**) Suppose that $\underline{u}, \underline{v} \in V \subseteq V_T$. We shall say that $\underline{v}$ covers $\underline{u}$ in V, and we shall write $\underline{u} \lhd_V \underline{v}$, if

1. $\underline{u} \leq \underline{v}$ and $\underline{u} \neq \underline{v}$
2. If $\underline{z} \in$ V such that $\underline{u} \leq \underline{z} \leq \underline{v}$, then $\underline{z} = \underline{u} \vee \underline{z} = \underline{v}$

We will omit subscript V's when the language is clear from context.

Intuitively, the covers relation '$\lhd$' provides an ordering among elements of V, in which one is 'immediately beneath' the other, allowing no other vector in V to exist in between them.

The set $\mathrm{Pre}_V(\underline{v})$, defined for $\underline{v} \in$ V, is used to denote all vectors that are related to $\underline{v}$ by '$\lhd$' in V, or more simply the set of predecessors of the vector $\underline{v}$. Hence,

$$\mathrm{Pre}_V(\underline{v}) = \{\underline{u} \in V \mid \underline{u} \lhd_V \underline{v}\}$$

A technical lemma shows that if $\underline{u}$ is an earlier part of $\underline{v}$, but not a predecessor of $\underline{v}$, then there is some predecessor of $\underline{v}$ that is larger than $\underline{u}$.

Lemma 1. Suppose that $\underline{u}, \underline{v} \in V \subseteq V_T$ such that $\underline{u} \leq \underline{v}$ and $\underline{u} \lhd \underline{v}$, then there exists $\underline{w} \in \mathrm{Pre}_V(\underline{v})$ such that

$\underline{u} \leq \underline{w}$.

Proof.

Since the number of leaves in a transaction tree is finite, the set $\{\underline{z} \in V \mid \underline{u} \leq \underline{z} \leq \underline{v}\}$ is finite. It is also non-empty since it contains $\underline{u}$. Thus, it has a maximal element. Let $\underline{w}$ be the maximal element of this set, then $\underline{w} \in \mathrm{Pre}_V(\underline{v})$. □

The following result relates the '$\lhd$' relation with the right-cancellation operator '/' of Definition 3.

Proposition 2. Suppose that $\underline{u}, \underline{v} \in V \subseteq V_T$. If $\underline{u} \lhd \underline{v}$, then $\underline{v} / \underline{u} \in A_T$.

Proof.

Since $\underline{u} \lhd \underline{v}$, we have that $\underline{v} / \underline{u} \neq \Lambda_T$. we want to show that $\underline{v} / \underline{u} \in A_T$. If $\underline{v} / \underline{u} \notin A_T$, then for some $l \in$ L, $\underline{v}(l) = \underline{u}(l).w_1.w_2$, where $w_1, w_2 \in > \Lambda$. By local left-closure, (and take $\underline{u}(l).w_1.w_2$ as x) there exists $\underline{w} \in$ V such that $\underline{w} \leq \underline{v}$ and $\underline{w}(l) = \underline{u}(l).w_1$.

Let $\underline{z} = lub(\underline{u}, \underline{w})$. Since V is discrete we have $\underline{z} \in$ V, and also $\underline{u} \leq \underline{z}$. But now $\underline{u}(l) < \underline{u}(l).w_1 = \underline{w}(l) = \underline{z}(l)$ and hence, $\underline{u} \leq \underline{z}$. Also, $\underline{z}(l) = \underline{w}(l) = \underline{u}(l).w_1$ and hence, $\underline{z} \leq \underline{v}$. This implies that $\underline{u} < \underline{z} < \underline{v}$, which is a contradiction to $\underline{u} \lhd \underline{v}$. □

This important result establishes that what takes a transaction vector and extends it to its successor is a column vector, representing an action of the transaction. This means that vectors in a normal transaction language are built by a series of concatenations with column vectors – in other words, the behavioural description is constructed by considering slices of behaviours that arise through the occurrence of actions as determined by the subtransactions of a transaction.

To anticipate further, these results will be useful in the development concerning the *t*-decompositions in our mathematical framework, which are central to the handling of compensations in our approach. This is the topic of Section 2.3.7 and 2.3.8.

Finally, 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* if and only if 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 scenarios of execution are still possible. This draws upon previous work on vector languages in (Moschoyiannis, 2005).

10.4 SEQUENTIAL ACTIONS

The prefix ordering (Definition 3) among 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 appearing on its coordinates.

This can be expressed more succinctly by saying that $\underline{u} \leq \underline{v}$ in a transaction language means that $\underline{u}$ is an earlier part of behaviour leading to $\underline{v}$.

If in addition the transaction language is normal, i.e. discrete and locally left-closed, then we can say more than that. In particular, we have seen (Proposition 2) that whenever $\underline{v}$ covers $\underline{u}$ in a normal transaction language, then what takes $\underline{u}$ and 'stretches it up' to $\underline{v}$ is a column vector representing the occurrence of an action (in fact, the model can express the occurrence of a simultaneity class of actions, based on [Shi97], but we have abstained from going this far for the moment). This allows us to model dependent actions. That is, occurrence of one action depends on the previous occurrence of the other. Recall the example in Section 2.3.1 where service s3 feeds service s4. It is in this sense that we talk about actions occurring in sequence (one after the other).

Suppose that during a long-running transaction a series of actions have already been executed and the resulting behaviour (up to that point) is described by a transaction vector $\underline{u}$ = (a1, Λ, c1, Λ). Then, occurrence of $\underline{a}_1$ = (Λ, b1, Λ, Λ) followed by occurrence of $\underline{a}_2$ = (a2, Λ, Λ, d1) can be modelled by

- first, concatenating vector $\underline{u}$ with $\underline{a}_1$ and
- then concatenating the resulting vector $\underline{v}$ with $\underline{a}_2$

In terms of our mathematical framework this amounts to operations $\underline{u}.\ \underline{a}_1 = \underline{v}$ and then $\underline{v}.\ \underline{a}_2 = \underline{w}$.

Considering the Hasse diagram for the order structure of the corresponding transaction language V, where lines between vectors denote an ordering relation in which the topmost vector is greater than the lower one, this would result in the portion of the diagram shown

in Fig 10-1.

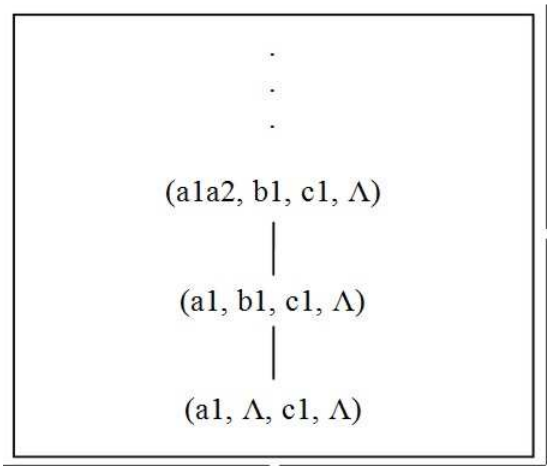

FIG. 10-1 AN ACTION B1 FOLLOWED BY A2

It is important to make the observation that the actual ordering between actions appearing in different coordinates of a transaction vector is determined by context - by what other vectors are included in the language. In other words, the relationship between transaction vectors and associated order theoretic structures is very much dependent on what other vectors are in the set V (unlike the behaviour vectors in (Shields & Laboratory, 1979), (Shields, 1997)where this relationship is independent of context).

For instance, in transaction vector $\underline{v}$ = (a1, b1, c1, Λ) we may immediately derive that the action a1 has happened on the leaf corresponding to the first coordinate, action b1 has happened on the leaf corresponding to the second coordinate and action c1 has happened on the leaf corresponding to the third. To determine the relationship between these actions however, we need the rest of the language.

The following discussion illustrates this by means a small example.

Assume that V is given by the set

$$V = \{(\Lambda, \Lambda, \Lambda, \Lambda), (a1, \Lambda, c1, \Lambda), (a1, b1, c1, \Lambda), (a1a2, b1, c1, d1)\}$$

Notice that adding in (Λ, Λ, Λ, Λ) is essential, and in this case is also sufficient for making V discrete and locally left-closed. Now the presence of $\underline{u}$ = (a1, Λ, c1, Λ) for which we have $\underline{u} \triangleleft \underline{v}$, tells us that action b1 on the subtransaction corresponding to the second coordinate occurs only after both actions a1 and c1 have taken place.

Now suppose that the intended behaviour of the transaction prescribed that a1 must occur before c1. This is captured in the corresponding transaction language by adding in the

vector (a1, Λ, Λ, Λ). In the resulting language

$$V = \{(\Lambda, \Lambda, \Lambda, \Lambda), (a1, \Lambda, \Lambda, \Lambda), (a1, \Lambda, c1, \Lambda), (a1, b1, c1, \Lambda), (a1a2, b1, c1, d1)\}$$

which continues to be normal. The presence of $\underline{w}$ = (a1, Λ, Λ, Λ) for which $\underline{w} \triangleleft \underline{u}$ dictates that a1 on the first coordinate occurs strictly before c1 does on the service corresponding to the third coordinate.

10.5 Concurrent actions

Our approach towards modelling concurrent actions, actions that can happen in parallel, draws upon the concepts in *Mazurkiewicz trace* languages (Mazurkiewicz, 1977),(Mazurkiewicz, 1988) where the ordering of concurrent events is considered subjective and thus is not distinguished, in contrast to CSP trace theory where it is assumed that observations are sequential in nature leading to the interpretation that concurrent events occur in either order.

For systems that exhibit concurrency, different external observers may disagree on the ordering of concurrent events. This may be seen more clearly in Einstein's famous thought-experiment[22] involving two trains travelling at constant speed in opposite directions along a pair of parallel tracks. Observers O1 and O2 are sitting in the middle of each train. A third observer O3 is sitting on the embankment. At a given moment, the two observers on the trains are on a line at right angles to the third observer. At that moment, two bolts of lightning strike on either end of the first train in such a way that O3 sees them strike at exactly the same time. Observer O1 travelling towards the light coming from the strike on the front end of the train he is on, sees that light before he sees the light of the strike on the rear end of the train. Observer O2 travelling towards the light coming from the strike on the rear end, sees that light before she sees the light coming from the strike on the front end.

Now from the point of view of observer O1 there are three distinct behaviours of the "system". One is when nothing has happened yet, another when he has seen the lightning bolt from the front end of the train, and another when he has seen both lightning bolts. Likewise, observer O2 has seen a behaviour in which nothing has happened yet, a behaviour where she has seen the lightning strike on the rear end and a behaviour where she has seen both lightning bolts. From the point of view of observer O3 there are only two distinct behaviours. One is when nothing has happened yet and the other is when both have. Thus, all four distinct behaviours can be observed for the same system; nothing has happened, one event has happened, the other event has happened, and both events have happened.

[22] This thought-experiment was given by A. Einstein to demonstrate the non-objectivity of contemporaneity in relativistic mechanics. It has been considered in view of concurrency in [Shi97] and our description of the experiment here is based on that.

The point to be made here is that observations on systems exhibiting concurrency largely depend on the relative position of the observer or the actual timing of execution. Such differences are non-objective and do not allow to infer the actual ordering between the events. On this basis, any particular ordering between concurrent events is irrelevant. On the contrary, the ordering between causally related events is objective (independent of the observer) and should be distinguished.

Returning to the treatment of concurrency within our component model, this takes up on Mazurkiewicz traces (Mazurkiewicz, 1977),(Mazurkiewicz, 1988), 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 (understood as events here), which describes potential concurrency.

Definition 7. (**Concurrent alphabet**) Let A denote a (finite) set. A *concurrent alhapbet* is an ordered pair (A, ι) where the binary relation $\iota \subseteq A \times A$ satisfies

- $a \iota b \Rightarrow b \iota a$ (symmetry)
- $a \iota b \Rightarrow a \neq b$ (irreflexivity)

This definition gives an independence relation on action symbols from a set (alphabet) A. Symmetry requires that concurrency is always mutual while irreflexivity prohibits considering an action being concurrent with itself.

Transaction vectors are essentially tuples of sequences, as discussed before. Thus, we find it useful to consider the extension of the relation ι to sequences, based on (Mazurkiewicz, 1977).

Given a concurrent alphabet (A, ι), a relation $\equiv_\iota^{(1)}$ can be defined on the set of all sequences over A, denoted by A*, by

$$x \equiv_\iota^{(1)} y \Rightarrow \exists\ u, v \in A^*, \exists\ a, b \in A \text{ such that } a \iota b \ \wedge\ x = uabv \ \wedge\ y = ubav$$

Let $\equiv_\iota$ be the reflexive, transitive closure of $\equiv_\iota^{(1)}$. By definition, $\equiv_\iota$ is an equivalence relation[23] on A*. The set of all sequences in A* that are related by $\equiv_\iota$ to a sequence $x \in A^*$ is called the equivalence class of x. We denote the equivalence class of a sequence $x \in A^*$ by $\langle x \rangle_\iota$. The set of equivalence classes of A with independence relation ι is denoted by $A_\iota^* = \{\langle x \rangle_\iota \mid x \in A^*\}$. Any subset L of A_ι^* is called a *Mazurkiewicz tracelanguage*.

Therefore, the independence relation ι on A gives rise to an equivalence relation $\equiv_\iota$ on sequences formed over A. We make use of this construction in terms of sequences formed

[23] Recall that an equivalence relation on a set is a binary relation that is symmetric, reflexive and transitive.

over the sets of actions $\mu(l)$, for each $l \in L$, associated with a transaction. It might be instructive at this point to revisit the definition of an independence relation given in compensating CSP (Butler et al., 2005).

Intuitively, the equivalence relation on sequences of actions says that any two consecutive actions are allowed to permute, providing they are independent. Note that when the independence relation is empty in the sets $\mu(l)$, for each $l \in L$, no two actions can be permuted in the corresponding sequences $\mu(l)^*$, which amounts to our understanding of sequential systems (e.g. as described by processes in CSP (C. A. R. Hoare, 1985) and its extension with compensations in (Butler et al., 2005)).

The equivalence relation on the set of actions A of a transaction equates all, and only those, sequences from $\mu(l)$, for each $l \in L$, which differ in the order of concurrent events. Drawing upon the extension of the independence relation ι to behaviour vectors in (Shields, 1997), the notion of independence between events in *Mazurkiewicz traces* can be readily interpreted into transaction vectors in our approach.

Definition 8. (**Independence**) For $\underline{u}, \underline{v} \in V \subseteq V_T$, we define

$$\underline{u} \text{ ind } \underline{v} \Leftrightarrow \forall\, l \in L :\ \underline{u}(l) > \Lambda \Rightarrow \underline{v}(l) = \Lambda$$

The intuition is that the behaviours described by $\underline{u}$ and $\underline{v}$ may occur independently. In the case of column vectors (recall Definition 2), independence captures the fact that actions appearing in one vector may occur independently of those appearing in the other. If in addition they occur consecutively, then they are concurrent. Thus, whenever two consecutive actions permute, their corresponding column vectors commute, i.e. $\underline{a}_1.\underline{a}_2 = \underline{a}_2.\underline{a}_1$, and the resulting behaviours are concurrent. In fact, (A, ind) is a concurrent alphabet, in the sense of Definition 7.

For example, suppose that a transaction with 3 leaves has experienced a fragment of behaviour described by $\underline{u}$ = (s1, Λ, Λ) and after that may engage in $\underline{a}_1$ and $\underline{a}_2$ concurrently, where $\underline{a}_1$ = (Λ, s2, Λ) and $\underline{a}_2$ = (Λ, Λ, s3).

We make the observation that $\underline{a}_1$ ind $\underline{a}_2$ (recall Definition 8) and consequently,

$$\underline{a}_1.\,\underline{a}_2 = (\Lambda, s2, \Lambda).(\Lambda, \Lambda, s3) = (\Lambda, s2, s3) = (\Lambda, \Lambda, s3).(\Lambda, s2, \Lambda) = \underline{a}_2.\,\underline{a}_1$$

Thus, we have $\underline{u}.\,\underline{a}_1.\,\underline{a}_2 = \underline{w} = \underline{u}.\,\underline{a}_2.\,\underline{a}_1$.

Indeed,

$$\underline{u}.\,\underline{a}_1 = (s1, \Lambda, \Lambda).(\Lambda, s2, \Lambda) = (s1, s2, \Lambda) = \underline{v}_1$$

and

$$\underline{v}.\,\underline{a}_2 = (s1, s2, \Lambda).(\Lambda, \Lambda, s3) = (s1, s2, s3) = \underline{w}$$

We also have that

$$\underline{u}.\underline{a}_2 = (s1, \Lambda, \Lambda).(\Lambda, \Lambda, s3) = (s1, \Lambda, s3) = \underline{v}_2$$

and

$$\underline{v}_2.\underline{a}_1 = (s1, \Lambda, s3).(\Lambda, s2, \Lambda) = (s1, s2, s3) = \underline{w}$$

In the resulting behaviour $\underline{w}$ the actions s1 and s3 are concurrent. The situation is depicted in Fig 10-2.

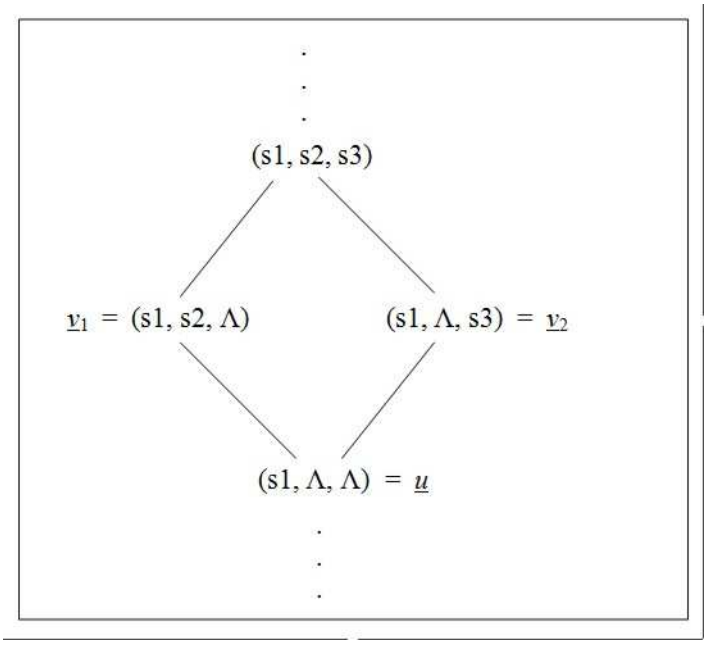

FIG 10-2 CONCURRENT ACTIONS IN A TRANSACTION

Note that if the actions were not concurrent, then we would have the lower part of the diamond shape in the diagram but not the upper half. The upper half is obtained only when the column vectors corresponding to the actions in question commute (that is to say, equivalently, that they are independent) *and* represent actions that occur consecutively. Both these requirements have to be met for the actions to be concurrent. This is then reflected in the order structure of the corresponding transaction language by the presence of the vector forming the upper half of the diamond. This vector is the resulting common behaviour, after the concurrent actions have taken place. If either of these two requirements is violated, then the transaction would never exhibit the common behaviour described by $\underline{w}$. The point to be made here is that independence alone does not guarantee concurrency. (The case of non-independence is more obvious.)

As depicted in Fig 10-2, the transaction as a whole experiences both actions (on each appropriate leaf) and the ordering is irrelevant. The corresponding concatenations result in a unique transaction vector (sitting on the top of the diamond) in which both actions have

occurred in *no* particular order. The incomparable transaction vectors in the middle of the diamond (i.e. $\underline{v}_1$, $\underline{v}_2$) represent behaviour arising during concurrent execution. These two vectors are bounded above by the vector in which both concurrent actions appear (i.e. $\underline{w}$).

In terms of the order theoretic properties of transaction vectors discussed in previous sections, this vector is their least upper bound. In the example of Fig 10-2, we have

$$lub(\underline{v}_1, \underline{v}_2) = lub(\ (s1, s2, \Lambda), (s1, \Lambda, s3)\ ) = (s1, s2, s3) = \underline{w}$$

Their greatest lower bound (sitting at the bottom of the diamond) is the vector in which none of the concurrent events have occurred but are both available. In our example, we have

$$glb(\underline{v}_1, \underline{v}_2) = glb(\ (s1, s2, \Lambda), (s1, \Lambda, s3)\ ) = (s1, \Lambda, \Lambda) = \underline{u}$$

We note that this non-interleaving representation of concurrent behaviour manifests itself in the structure of the automata associated with this kind of vector languages, as described in (Moschoyiannis et al., 2005).

The fundamental difference in expressing concurrency should now be apparent. By departing from classic CSP concurrency, we are able to consider concurrency within a long-running transaction, and without the need to consider sequences of actions within a transaction as in compensating CSP (Butler et al., 2005). In CSP, and related process algebras, concurrency arises through composition. Here we have not yet been concerned with composing sequences from subtransactions of different transactions, though this may also produce concurrency. We are simply describing the case that subtransactions of the same transaction engage in concurrent actions, a phenomenon common in most B2B scenarios for example. The notion of composition within this vector language – based behavioural description has been described in (Moschoyiannis, 2004).

In what follows, we again discuss concurrent actions in connection to the context of the corresponding ctransaction language. Consider the transaction language

$$V = \{\ (\Lambda, \Lambda, \Lambda), (s1, \Lambda, \Lambda), (s1, s2, \Lambda), (s1, \Lambda, s3), (s1, s2, s3)\ \}$$

It can be easily checked that V is discrete and locally left-closed. Its order structure is (in part) depicted in Fig 10-2. We have that $\underline{u} \lhd \underline{v}_1$ and $\underline{u} \lhd \underline{v}_1$. Also, we have $\underline{v}_1 \lhd \underline{w}$ and $\underline{v}_2 \lhd \underline{w}$. We have seen that the actions s2 on the second leaf and s3 on the third are concurrent.

Now consider the transaction language,

$$V = \{\ (\Lambda, \Lambda, \Lambda), (s1, \Lambda, \Lambda), (s1, s2, s3)\ \}$$

In this language, which is also discrete and locally left-closed, the actions s2 on the second leaf and s3 on the third are simultaneous rather than concurrent. This is because the transaction vector $\underline{w}$ = (s1, s2, s3) in which both actions have taken place is obtained directly from $\underline{u}$ = (s1, Λ, Λ) in which neither of the actions have occurred yet. Hence, what takes $\underline{u}$ and stretches it up to $\underline{w}$ is the column vector $\underline{a}$ = (Λ,, s2, s3) in which s2 and s3 are simultaneous actions. This case can be understood as cutting through the diamond of Fig 10-2.

Next, consider the transaction language

$$V = \{ (\Lambda, \Lambda, \Lambda), (s1, \Lambda, \Lambda), (s1, s2, \Lambda), (s1, \Lambda, s3) \}$$

In this language, which is also discrete and locally left-closed, the actions s2 and s3 are neither concurrent nor simultaneous. The transaction in this case, after doing s1 on the leaf corresponding to the first coordinate, has a choice between doing s2 on the second coordinate or s3 on the third. This case can be understood as having only the lower half of the diamond in Fig 10-2, and brings about the issue of alternative actions and mutual exclusion in a long-running transaction. This is discussed in the following section.

10.6 Alternative actions

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 in the middle of the diamond represent concurrent behaviour. The fact that the two incomparable vectors are in the middle of the diamond implies that they are bounded above in the set (by the transaction vector sitting on top of the diamond).

Whenever this latter requirement does not hold we may talk about events in conflict. In terms of pictures and associated Hasse diagrams, we are essentially getting rid of the upper part of the diamond and keeping the lower part, the branches of which represent a choice between doing one or the other action. In effect, this amounts to ensuring that the behaviours they represent is not (an early) part of the same behaviour. Therefore, in what follows we examine when two transaction vectors are not bounded above in a component language.

Let us first consider the case where the column vectors in question are not independent. Then, they do not agree on the non-empty coordinates corresponding to the same leaves of the transaction tree. This entails that there is no causality between the two and at the same time it is not possible for both of them to occur (since they engage the same leaf of the tree).

For example, the actions represented by $\underline{a}_1$ = (s1, Λ, Λ) and $\underline{a}_2$ = (s4, Λ, Λ) could both be available (or possible to occur) when the transaction has already exhibited the behaviour described by a transaction vector, say, $\underline{u}$. In other words, after $\underline{u}$ the transaction may engage in either $\underline{a}_1$ or $\underline{a}_2$. Note that it cannot do both since s1 and s4 (s1 ≠ s4) are actions associated with the same service (the one corresponding to the first coordinate). Considering the Hasse diagram for the order structure of the corresponding transaction language, this situation would result in the fragment of the diagram shown in Figure 2.18.

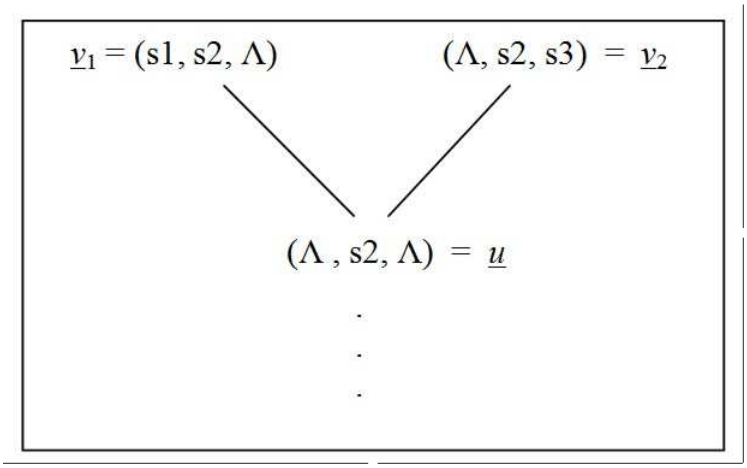

FIG. 10-3 ALTERNATIVE ACTIONS IN A TRANSACTION

In further explanation, v1 = (s1, s2, Λ) is the behaviour resulting from occurrence of a1 while the vector v1 = (s4, s2, Λ) is the behaviour resulting from occurrence of a2, after u. In terms of our mathematical framework, we have u. a1 = v1 and u. a1 = v1 but only one of these behaviours may take place during an execution of the long-running transaction in question.

We now turn our attention to actions whose corresponding column vectors are independent (recall Definition 8. This case is a bit more subtle. In principle, independent column vectors represent actions which are in no way related to each other. For example, consider the actions given by column vectors $\underline{a}_1$ = (s1, Λ, Λ) and $\underline{a}_2$ = (Λ, Λ, s3). If they are both offered after the transaction has engaged in behaviour described by vector $\underline{u}$, then they represent a choice between doing s1 on the leaf corresponding to the first coordinate and action s3 on the leaf corresponding to the third coordinate. Unless they are bounded above!

To ensure that the two independent events are not bounded above, effectively, that they are not part of a subsequent common behaviour, they must not occur consecutively. In other words, the actions succeeding $\underline{a}_1$ must not be $\underline{a}_2$ and, dually, the action succeeding $\underline{a}_2$ (on the other branch) must not be $\underline{a}_1$. Otherwise, they lead to a common behaviour $\underline{w}$ which inadvertently bounds $\underline{v}_1$ and $\underline{v}_2$ (forcing them to be concurrent as discussed before).

The situation is depicted in Fig 10-4(i) where the actions s1 and s3 are alternative. Compare with Fig 10-4(ii) where the actions s1 and s3 are concurrent.

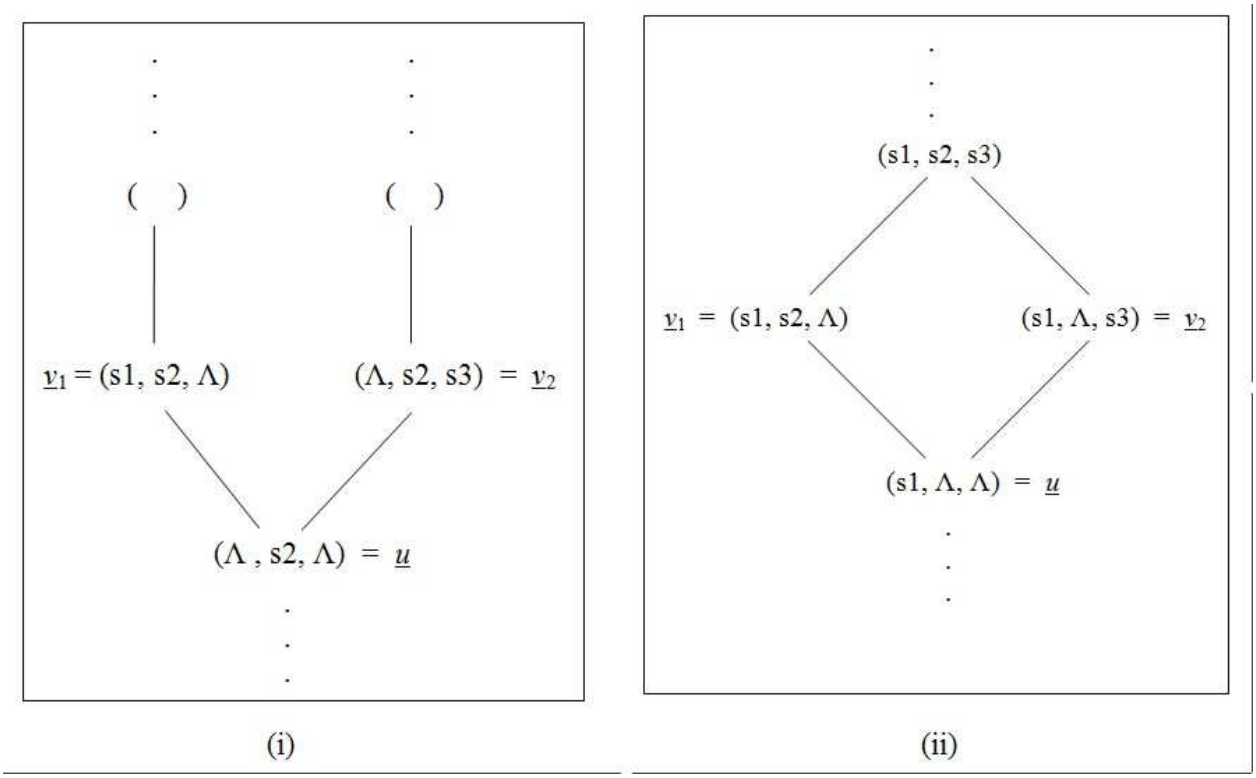

FIG. 10-4 ACTIONS S1 AND S3 ARE CONCURRENT IN (II) BUT NOT IN (I)

10.7 COMPENSATION IN TRANSACTION VECTORS

In this section we describe work in progress on the handling of compensations in our formal model for long-running transactions. We have seen that the occurrence of an action is recorded in a the vector description by (coordinate-wise) concatenation of the existing vector (or vectors), describing the behaviour of the transaction before the action occurred, with the column vector representing the action in question.

We have also seen (Definition 3) that the right-cancellation operator '/' on vectors can be used to isolate the behaviour that arises in between vectors, so 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 $\underline{v}$. Further, in a normal (discrete and locally left-closed) transaction language we have seen (recall Proposition 2)) that if $\underline{u}$ is an immediate predecessor of $\underline{v}$, i.e. $\underline{u} \lhd \underline{v}$ in the sense of Definition 6, then the application of the right-cancellation operator on $\underline{v}$ produces a vector which is a column vector, i.e. it isolates the last action(s) that went into obtaining $\underline{v}$ from $\underline{u}$.

The following result shows that the column vector $\underline{v} / \underline{u}$ is unique.

Lemma 2. Suppose that $\underline{v}, \underline{w} \in V \subseteq V_T$ such that $\underline{v} \leq \underline{w}$, then there exists unique vector $\underline{u} \in$ V such that $\underline{v}.\underline{u} = \underline{w}$. We denote this element by $\underline{w} / \underline{v}$.

Proof.

For each $l \in$ L, define $\underline{u}(l) = \underline{w}(l).\underline{v}(l)$. We have, for all $l \in$ L,

$(\underline{v}.\underline{u})(l) = \underline{v}(l).(\underline{w}(l) / \underline{v}(l)) = \underline{w}(l)$

so $\underline{v}.\underline{u} = \underline{w}$. Furthermore, if $\underline{u}' \in$ V such that $\underline{v}.\underline{u}' = \underline{w}$, then for each $l \in$ L, $\underline{v}(l).\underline{u}(l) = \underline{w}(l)$, so that

$\underline{u}'(l) = \underline{w}(l) / \underline{v}(l) = \underline{u}(l)$, and $\underline{u}' = \underline{u}$,

establishing uniqueness.

This result together with the main result of Proposition 2 in previous Section allows us to define compensations by using the right-cancellation operator on vectors that differ only in an action (or, more generally, in a simultaneity class of actions). This can be done since the application of '/' on a vector undoes the last action that took place in obtaining the behaviour described by that vector.

In this way our approach uses (coordinate-wise) concatenation to model the occurrence of an action (the activation of a service invocation point in the transaction), by

$$\underline{u} \xrightarrow{\underline{a}} \underline{v}$$

Instead of introducing separate notation and associated semantics for cancelling actions that have already taken place, the idea is to use we use right-cancellation on vectors to perform the compensation action, hence,

$$\underline{v} \xleftarrow{\underline{v} / \underline{u}} \underline{u}$$

which will be invoked if a failure later in the long-running transaction makes it necessary. In other words, the application of the right-cancellation operation on a vector $\underline{v}$ obtained by

$\underline{u}$, produces $\underline{u}$, since

$$\underline{v} / (\underline{v} / \underline{u}) = \underline{u}$$

In what follows we show that compensations performed in this way will cancel all forward actions (modelled using concatenation on vectors), leaving only vectors where actions for which their compensations have also been executed are no longer visible in the end result.

Intrinsic to the development of the theory for compensations in our vector model is the fact that vectors describing the behaviour of a long-running transaction can be seen to be built up from the empty vector by a series of concatenations with column vectors representing appropriate actions. Therefore, vectors are essentially formed by a series of concatenations with column vectors. We find it useful to describe this more formally now, since the subsequent application of the right-cancellation would remove (or undo) each action in turn.

We have seen in Proposition 2, which is concerned with building up the vectors, the interaction between '$\lhd$' and '/' – that is, if $\underline{u} \lhd \underline{v}$ in a normal transaction language, then $\underline{v} / \underline{u}$ is an action, represented by a column vector (Definition 2). The following proposition states a similar result for the interaction between '$\lhd$' and '.'.

Proposition 3. Suppose that $V \subseteq V_T$ is a normal transaction language, then

1. $\underline{\Lambda}_T \in V$
2. If $\underline{u}, \underline{v} \in V$ such that $\underline{u} \lhd \underline{v}$, then there exists $\underline{a} \in A_T$ such that $\underline{v} = \underline{u}.\underline{a}$.

Proof.

For (1). $V \neq \emptyset$, so there exists $\underline{v} \in V$. By local left-closure (Definition 6) of V, for each leaf $l \in L$, there exists $\underline{u}_l \in V$ such that $\underline{u}_l \leq \underline{v}$ and $\underline{u}_l(l) = \Lambda$. By discreteness (Definition 5) of V, the greatest lower bound of the $\underline{u}_l$, which must be $\underline{\Lambda}_T$, belongs to V.

For (2). Let $\underline{a} = \underline{v} / \underline{u}$, so that $\underline{v} = \underline{u}.\underline{a}$. This means that $\underline{a} \neq \underline{\Lambda}_T$. If $| \underline{a}(l) | > 1$, then we can find $x, y \in \mu(l)^*$ such that $x, y \neq \Lambda$ and $\underline{a} = x.y$. By local left-closure, there exists $\underline{x} \in V$ such that $\underline{x} \leq \underline{v}$ and $\underline{x}(l) = \underline{u}(l).x$. Let $\underline{w} = glb(\underline{u},\underline{x})$. We have that $\underline{w}$ exists, $\underline{w} \leq \underline{v}$ and $\underline{w} \in V$ by consistent completeness (of the definition of discreteness), since $\underline{u}, \underline{x} \leq \underline{v}$.

Now, $\underline{u} \leq glb(\underline{u},\underline{x}) \leq \underline{v}$, which can be written as $\underline{u} \leq \underline{w} \leq \underline{v}$, and $\underline{w}(l) = \max(\underline{u}(l),\underline{x}(l)) = \underline{u}(l).x$ so that $\underline{u}(l) < \underline{w}(l) < \underline{v}(l)$. This implies that $\underline{u} < \underline{w} < \underline{v}$, and we have a contradiction since $\underline{u} \lhd \underline{v}$.

A corollary of Proposition 3 can give a formal description of the way transaction vectors

are actually obtained.

Corollary 1. Suppose that $V \subseteq V_T$ is a normal transaction language and $\underline{u}$, $\underline{v} \in$ V such that $\underline{u} \leq \underline{v}$ (and also $\underline{u} \neq \underline{v}$), then there exists $\underline{a}_1, \ldots, \underline{a}_n \in A_T$ such that

1. $\underline{u}.\underline{a}_1 \ldots \underline{a}_n = \underline{v}$
2. $\underline{u}.\underline{a}_1 \ldots \underline{a}_i \in$ V, i = 1..n
3. $\underline{u} \lhd \underline{u}.\underline{a}_1$ and $\underline{u}.\underline{a}_1 \ldots \underline{a}_{i-1} \lhd \underline{u}.\underline{a}_1 \ldots \underline{a}_i$, i = 2..n.

Proof.

If $\underline{u} \lhd \underline{v}$, then the corollary holds with n = 1 and $\underline{a}_1 = \underline{v} / \underline{u}$, by Proposition 3.

Otherwise, there exists $\underline{w} \in$ V such that $\underline{u} \leq \underline{w} \lhd \underline{v}$, by Lemma 1 (in Section 2.3.3). By induction, there exists $\underline{a}_1, \ldots, \underline{a}_{n-1} \in A_T$ such that

$$\underline{u}.\underline{a}_1, \ldots \underline{a}_{n-1} = \underline{w}.\underline{a}_1, \ldots \underline{a}_i \in \text{V}, \text{ i} = 1, .. \text{ n-1},$$

and

$$\underline{u} \lhd \underline{u}.\underline{a}_1, \text{ and } \underline{u}.\underline{a}_1, \ldots \underline{a}_{i-1} \lhd \underline{u}.\underline{u}.\underline{a}_1, \ldots \underline{a}_i \text{ , i} = 2, \ldots, \text{n-1}$$

If we now let $\underline{a}_n = \underline{v} / \underline{w}$, then by Proposition 3, $\underline{a}_n \in A_T$, $\underline{u}.\underline{a}_1 \ldots \underline{a}_n \in$ V and $\underline{u}.\underline{a}_1 \ldots \underline{a}_{n-1} \lhd \underline{u}.\underline{a}_1 \ldots \underline{a}_n$. This means that $\underline{a}_1, \ldots, \underline{a}_n$ have the desired properties.

We may now formally define transaction vectors as series of concatenations with column vectors.

Definition 9. (***t*-decompositions**) Suppose that $\underline{u}$, $\underline{v} \in V \subseteq V_T$ such that $\underline{u} \leq \underline{v}$. We shall define sequences such as $\underline{a}_1 \ldots \underline{a}_n$ in Corollary 1 as *t-sequences* from $\underline{u}$ to $\underline{v}$. In the case that $\underline{u} = \underline{\Lambda}_T$, we describe such sequences as *t-decompositions* of $\underline{v}$.

Since the transaction vectors used in describing the behaviour of a long-running transaction are built up from the empty vector, they are *t*-decompositions. This means that the recursive application of the right-cancellation, on each action that has gone into obtaining the vector in question, cancels out the initial actions leaving only vectors containing no observable actions as a result, i.e. leaves the corresponding transaction language only with the empty vector $\underline{\Lambda}_T$, effectively returning the system to (an approximation of) the state it was before the long-running transaction started.

It should be noted that in this report and Deliverable D3.1 (Razavi et al., 2007a) we have described an extended lock mechanism for recovery management and concurrency control. Thus, in our overall approach to long-running transactions in digital ecosystems we go beyond the assumption that a compensating action undoes the effects of its associated action. The corresponding locks, namely the internal lock (I_Lock), conditional-commit lock (C_Lock) and recovery lock (R_Lock), together with the corresponding IDG and EDG graphs, show how the dependencies between subtransactions due to data sharing are handled. In addition, the time-out lock (T_Lock) covers the cases of sequential alternative service composition and allows for forward recovery. Further details can be found in chapter 3, 4 and 5 and (Razavi et al., 2007a), (Razavi et al., 2009). Such aspects have not been discussed in the corresponding formal setting, but work is in progress on their formal treatment.

10.8 MODELLING FORWARD AND COMPENSATING BEHAVIOUR OF A TRANSACTION

In the previous section we described how the ordering relation between different vectors of a transaction reflects the orderings between activations of its sub-transactions. The vector-based description of behaviour in our formal model for long-running transactions makes it possible to express sequential, parallel and alternative behaviour of a transaction.

We have seen that the ordering relation between transaction vectors is given in terms of the coordinate-wise prefix ordering relation '$\leq$' of Definition 3. This turns the set of vectors associated with a transaction to a partially ordered set $(V, \leq)$ (recall Proposition 1). Since each vector describes the part of behaviour of the component in which the actions appearing in it have taken place, it is appropriate to say that whenever $\underline{u} \leq \underline{v}$, then $\underline{u}$ describes an earlier part of the behaviour described by $\underline{v}$. Further, in a normal transaction language, if $\underline{u} \lhd \underline{v}$then the vector $\underline{v}$ describes the behaviour of the transaction in which a single action has (or concurrent actions have) occurred since $\underline{u}$.

Since $(V, \leq)$ is a partially ordered set some vectors may be incomparable. For example, consider the vectors $\underline{u} = (s_1, \Lambda, \Lambda)$ and $\underline{v} = (\Lambda, s_2, \Lambda)$ for which neither $\underline{u} \leq \underline{v}$ or $v \leq \underline{u}$. Such vectors describe either alternative behaviour (there is a choice between the last actions that went into forming each) or concurrent behaviour (the last actions that went into forming each are concurrent). Any pair of incomparable vectors stands in one relation or the other, and this is determined by what other vectors are in the set of vectors associated with a given transaction.

If the incomparable vectors are bounded above – in other words, if they describe earlier parts of some common later behaviour – then they describe concurrent behaviours. If they are not bounded above, then they describe alternative behaviours. It is important to stress that this is determined by context, by what other vectors are included in the set for a transaction.

This is illustrated in Fig. 10-5 which uses Hasse diagrams to depict the order structure of different sets of transaction vectors for a transaction with 3 leaves. It can be seen that s1

and s2 are sequential (s2 can only be activated after s1) in Fig. 10-5(i) while they are mutually exclusive (alternative) in Fig. 10-5(ii) and they are concurrent in Fig. 10-5(iii).

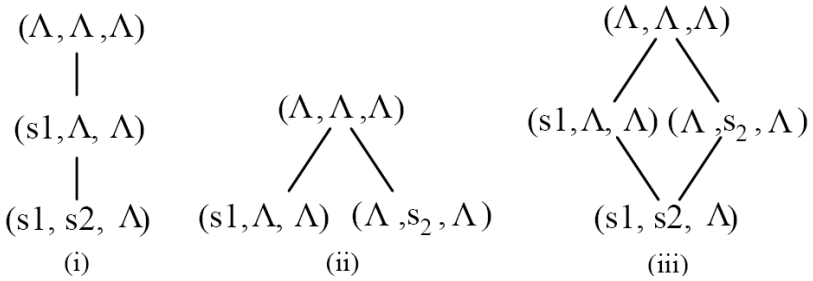

FIG. 10-5 ORDER STRUCTURE OF TRANSACTION VECTORS

Notice that the set of vectors in case (i) does not include the vector (Λ, s_2, Λ). This, in addition with the fact that (s_1, s_2, Λ) is included, implies that s2 can only happen after s1 has (sequential dependency).

The set of vectors in case (ii) does not include (s_1, s_2, Λ). This has as a consequence that the vectors $\underline{u} = (s_1, \Lambda, \Lambda)$ and $\underline{v} = (\Lambda, s_2, \Lambda)$ are not bounded above in this case. Hence, the actions s1 and s2 are independent but do not take place consecutively in this case (one immediately after the other). This implies that there is a choice between doing s_1 and doing s_2 on the respective coordinates (alternative execution).

In case (iii) where the vector (s_1, s_2, Λ) is included, the vectors are bounded above and this implies that they describe the concurrent execution of actions s_1 and s_2 leading to the behaviour described by the vector (s_1, s_2, Λ). This is indicated by the familiar lozenge shape (or diamond) which marks the characteristic structure of a finite lattice (Davey & Priestley, 1990). The incomparable vectors sitting at the middle of the lozenge are both available after the same behaviour (that is $(\Lambda, \Lambda, \Lambda)$ in this case) and occur consecutively leading to the behaviour described by the vector sitting at the bottom of the lozenge shape, i.e. (s_1, s_2, Λ).

Fig. 10-5 might be instructive with regard to the subtle distinction between independence and concurrency. Independent events are concurrent only if they are both offered after the same behaviour (both enabled at the same time) and do occur one after the other (consecutively). Otherwise, they may be mutually exclusive or even sequential.

It might also be worth pointing out that the lozenge shape in Fig. 10-5(iii) exhibits the characteristic structure of a finite lattice, which is a requirement of the discreteness property (Definition 4) in the case that the vectors $\underline{u}, \underline{v}$ are independent. The vector at the bottom of the lozenge is the least upper bound of the vectors in the middle, while the vector at the top is their greatest lower bound. This shows that discreteness – in the case of independent vectors bounded above in the set – is a property inherently related to

concurrency.

For representing the pattern behaviour of the presented transaction model, we use a relavant example which uses partial results by applying EDG and IDG (see chapter 4 and 5). The left transaction tree shown in Fig. 10-6 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. 10-6) does not meet the desired outcome, then s_3 and s_4 are executed in succession (sequentially) followed by d_2.

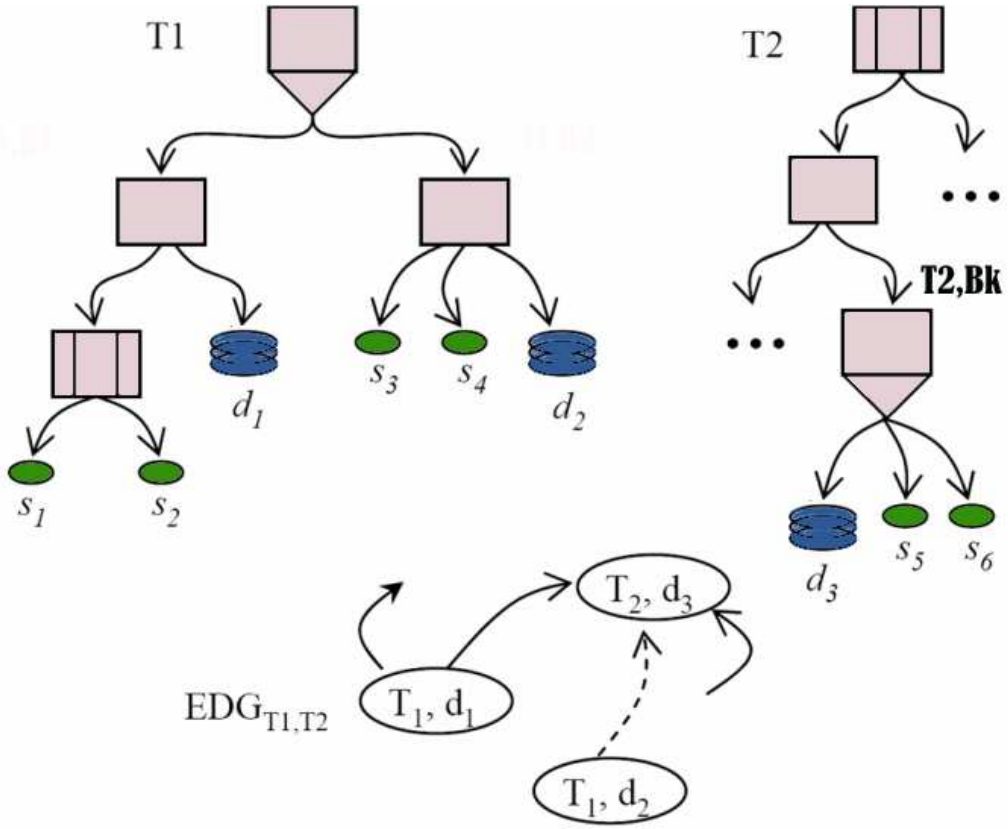

FIG. 10-6 DATA ORIENTED AND PARTIAL RESULT BEHAVIOUR

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. 10-7, which describe all possible series of subtransaction activations in performing the transaction T1 (given in Fig. 10-6).

In Fig. 10-7 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 T1. This choice is deterministic and will be resolved on the basis of whether d_1 satisfies the desired outcome.

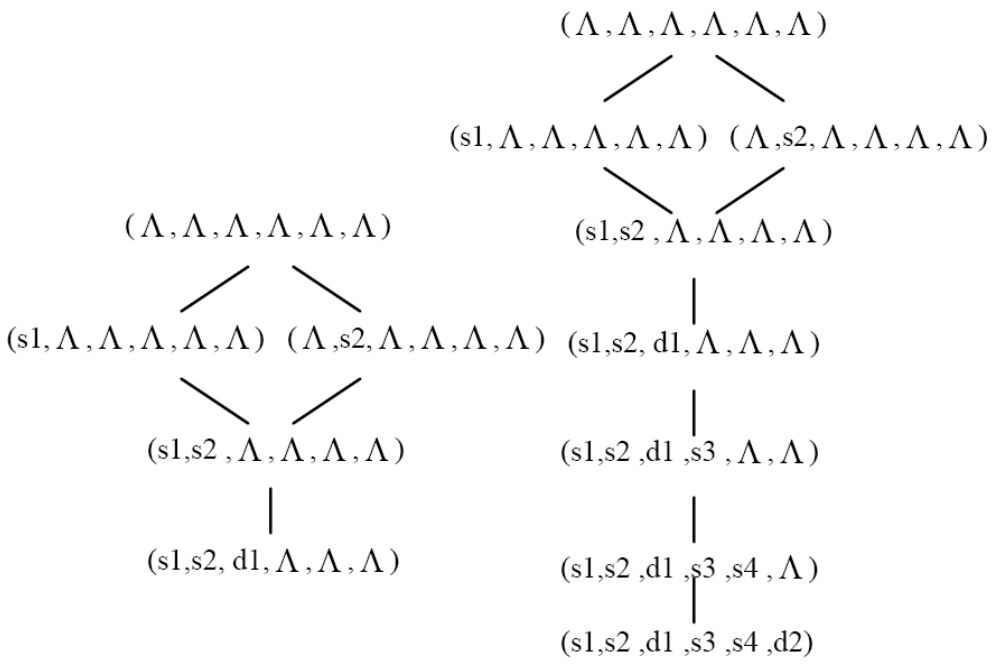

FIG. 10-7 TRANSACTION VECTORS FOR T1

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, Λ, Λ) $\leq$ (s_1, s_2, d_1, s_3, s_4, Λ) which means that s_4 can only happen after s_3 has (sequentially).

The Hasse diagram depicting the order structure of the transaction vectors for T1 can be readily used for checking against discreteness and local left-closure. For discreteness (recall Definition 4), we concentrate on vectors which have more than one vector immediately underneath. Then, that vector together with its immediate predecessors (the vectors immediately below it) must constitute a finite lattice. This will be the case when the immediate predecessors are bounded above (least upper bound) and below (greatest lower bound) by some vector in the set. In our example, such a vector is (s_1, s_2, Λ, Λ, Λ, Λ) which has two distinct immediate predecessors, namely (s_1, Λ, Λ, Λ, Λ, Λ) and (Λ, s_2, Λ, Λ, Λ, Λ). These vectors are bounded above by (s_1, s_2, Λ, Λ, Λ, Λ) and are bounded below by (s_1, s_2, Λ, Λ, Λ, Λ). Hence, the set V_{T1} is discrete.

For local left-closure (recall Definition 5), we look at each coordinate of the vectors. We concentrate on those which have a sequence of length greater than one. In such case, there must be some other vector in the set which, at the specific coordinate, has the same sequence but reduced by one. It can be readily checked diagrammatically that this is the case for the order structure depicted in Fig 10-7.

Furthermore, in case some subtransaction fails, the vector-based description is used in

providing compensations for long-running transactions, taking up on the "do-compensate" and "validate-do" behaviour patterns. We have seen that compensations are performed by applying the right-cancellation operator '/' of Definition 3, which produces a unique vector, by Lemma 2. This unique vector is the immediate predecessor of the vector whose last action is compensated for. With successive applications of the right-cancellation operator we effectively move backwards along the Hasse diagram of Fig 10-7, removing the last action each time from each vector.

For example, consider the vector $\underline{v}$ = (s1, s2, d1, Λ, Λ, Λ). By applying Definition 9, with $\underline{u} = \underline{\Lambda}_T$ we have that the *t*-decomposition of $\underline{v}$ is $\underline{a}_1.\ \underline{a}_2.\ \underline{a}_3$, where $\underline{a}_1$ = (s1, Λ, Λ, Λ, Λ, Λ), $\underline{a}_2$ = (Λ, s2, Λ, Λ, Λ, Λ) and $\underline{a}_3$ = (Λ, Λ, d1, Λ, Λ, Λ). If a failure occurs after the action associated with doing d1, the actions s2 and s1 also have to be compensated since they are dependent on d1. This is done by applying the right-cancellation operator as follows.

First, we identify the last action that went into forming $\underline{v}$. This is done by looking for vectors in V which are immediate predecessors of $\underline{v}$. In this case it is vector $\underline{u}$ = (s1, s2, Λ, Λ, Λ, Λ) for which $\underline{u} \lhd \underline{v}$. By Proposition 2, we have that $\underline{v}$ / $\underline{u}$ is a column vector – in this case it is $\underline{a}_3$ = (Λ, Λ, d1, Λ, Λ, Λ). Hence,

$$\underline{v} / (\underline{v} / \underline{u}) = \underline{v} / \underline{a}_3 = (s1, s2, d1, \Lambda, \Lambda, \Lambda) / (\Lambda, \Lambda, d1, \Lambda, \Lambda, \Lambda) = (s1, s2, \Lambda, \Lambda, \Lambda, \Lambda) = \underline{u}$$

Similarly, by application of '/' on vector $\underline{u}$ we have,

Now, vector $\underline{u}$ = (s1, s2, Λ, Λ, Λ, Λ) is interesting in that it has two immediate predecessors, namely $\underline{x}$ = (s1, Λ, Λ, Λ, Λ, Λ) and $\underline{y}$ = (Λ, s2, Λ, Λ, Λ, Λ). Despite it being involved in a diamond the same process applies. In this case,

$$\underline{u} / \underline{x} = \underline{a}_2 = (\Lambda, s2, \Lambda, \Lambda, \Lambda, \Lambda) \text{ and } \underline{u} / \underline{y} = \underline{a}_1 = (s1, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda)$$

Hence, we have

$$\underline{u} / (\underline{u} / \underline{x}) = \underline{u} / \underline{a}_2 = (s1, s2, \Lambda, \Lambda, \Lambda, \Lambda) / (\Lambda, s2, \Lambda, \Lambda, \Lambda, \Lambda) = \underline{y}$$

$$\underline{u} / (\underline{u} / \underline{y}) = \underline{u} / \underline{a}_1 = (s1, s2, \Lambda, \Lambda, \Lambda, \Lambda) / (s1, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda) = \underline{x}$$

Note we are now in the middle of the diamond appearing in the Hasse diagram of Figure 2.21. We apply similar development to both $\underline{x}$ and $\underline{y}$. The vector $\underline{x}$ has $\underline{\Lambda}_T$ as its immediate predecessor, and $\underline{x} / \underline{\Lambda}_T = \underline{a}_1$. So does $\underline{y}$, for which $\underline{y} / \underline{\Lambda}_T = \underline{a}_2$.

Hence, we have

$$\underline{x} / (\underline{x} / \underline{\Lambda}_T) = \underline{x} / \underline{a}_1 = (s1, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda) / (s1, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda) = (\Lambda, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda) = \underline{\Lambda}_T$$

and

$$\underline{y} / (\underline{y} / \underline{\Lambda}_T) = \underline{y} / \underline{a}_2 = (\Lambda, s2, \Lambda, \Lambda, \Lambda, \Lambda) / (\Lambda, s2, \Lambda, \Lambda, \Lambda, \Lambda) = (\Lambda, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda) = \underline{\Lambda}_T$$

In this way all actions that had occurred before failure, as described in vector $\underline{v}$ = (s1, s2, d1, Λ, Λ, Λ), have been compensated for. It can be seen that there are no longer any visible observable actions, as illustrated by $\underline{\Lambda}_T$.

It might be instructive to compare with the way compensations are handled in the approaches discussed in chapter 3. In our approach there is no need for enforcing the sequence of actions to be performed in the reverse order. This is inherent to the way operations are defined on transaction vectors (right-cancellation, concatenation, ordering relations). Further, and perhaps even more importantly, no additional notation or formal construction is required in handling compensations for concurrent actions. In both compensating CSP (Butler et al., 2005) and the approach taken by (Bruni, Melgratti, & Montanari, 2005), require additional syntax and a separate semantics in order to perform compensations for sequential actions that are composed in parallel. Finally, note that there is no need to consider different sequences of actions within a transaction and compose them in order to model concurrency in our approach. Concurrency is handled in terms of actions themselves, and there is no need for all actions within a transaction to be independent.

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 subtransactions in terms of actions on the leaves (e.g. service invocations) and the coordination between them. The resulting behavioural patterns (see Fig. 10-5) 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 (Moschoyiannis, 2005), (Moschoyiannis et al., 2007). We have also addressed compensations in an intuitive and relative straightforward manner which does not require further formal constructions – it is again based on (coordinate-wise) operations on transaction vectors.

It might be worth pointing out that our formal description of the distributed transaction model here we have been concerned with modelling individual transactions, albeit in a way that allows to capture the release of partial results to other transactions. In other words, we have been mostly concerned with the dependencies within a transaction rather than between transactions. For the latter, it would appear that we need to consider the vectors from each and compose, in a principled way, in order to get the resulting inter-transaction behaviour. Previous work on composition within the vector-based representation of behaviour (Moschoyiannis, 2004) could be exploited in this respect. Further, we note that the properties discussed in this appendix, discreteness and local left-closure, are shown to be preserved under composition of vectors in Moschoyiannis, 2004), (Moschoyiannis, 2005), (Moschoyiannis et al., 2007) .

11 APPENDIX II: RESTFUL TRANSACTION FRAMEWORK FOR DIGITAL ECOSYSTEM

As RESTful interaction model has been considered as the evolution of the next step of the current implementation of digital ecosystem. Razavi and Marinos has started to provide a primary framework for the RESTful transaction model:

- Razavi, A. Marinos, S. Moschoyiannis and P. Krause (2009) RESTful Transactions supported by the Isolation Theorems, The Ninth International Conference on Web Engineering (ICWE 2009), San Sebastian, Spain (in the processed to be published).
- Marinos, A. Razavi, S. Moschoyiannis and P. Krause (2009) RETRO: A Consistent and Recoverable RESTful Transaction Model, IEEE 7th International Conference on Web Services (ICWS 2009), Los Angeles, CA, USA (in the processed to be published).
- A. Razavi, A. Marinos, S. Moschoyiannis and P. Krause (2009), Recovery management in RESTful Interactions, 3rd IEEE International Conference on Digital Ecosystems and Technologies, IEEE Computer Society, Istanbul, Turkey (in the processed to be published).

This appendix provides a brief presentation of the framework (customised version of the above-third paper).

11.1 Introduction

Preservation of loose coupling and local autonomy within digital ecosystem can cause considerable algorithm complexity (Razavi et al., 2007a). Hence, there is a need for alternative frameworks to increase the feasibility of such requirements. *REpresentational State Transfer* (REST) was introduced by Roy Fielding (Fielding, 2000), to provide a formal description of the architectural style that had emerged in the World Wide Web. Simplicity and reliance on stateless resources, shifts the complexity from tightly coupled algorithms to loosely coupled resource interactions. However, the modelling of resource interactions to support recoverable transactions, in a way that offers consistency and isolation of concurrent transactions, is a crucial necessity for using REST in digital ecosystems. In this paper, we have provided a RESTful framework for recovery management which includes a locking concurrency control. This appendix is structured as follows. Section 2 provides the necessary literature for transactional challenges in related areas. Section 3 introduces the recovery model for REST. Section 4 applies the locking mechanism in a RESTful framework. In section 5 the proof of correctness of constraints is applied. Finally Section 6 illustrates our approach with an example scenario.

11.2 Transactional approaches and Digital Ecosystems

The general term ‘Transaction’ is usually defined by the four properties contained in the ACID

acronym (Gray & Reuter, 1993). A transaction that is started when a system is in a consistent state may make the state temporarily inconsistent, but it must terminate by producing a new consistent state. Consistency is the C in ACID. Temporary inconsistency must not affect other concurrent transactions. This maintains the illusion that each transaction runs in isolation; the I in ACID.This means that the inputs and consequent behaviour of some may be inconsistent, even though each transaction executed in isolation would be correct. It follows that concurrent execution should not cause application programs to malfunction, which is the first law of concurrency control. Equally, the transaction should either run to completion, or if some operation within a transaction should fail it should automatically undo all previous actions and return to the original consistent state. This property is Atomicity, the A in ACID. Also, none of the updates or messages of committed transactions should be lost. Durability is the D in ACID.

Despite concurrency in a transactional environment, the consistency should be preserved, and the aborted transactions can be rerun. The complication of digital ecosystem as a service-oriented architecture is a loosely coupled environment, that have been discussed in different approaches (recall chapter 3), (Razavi et al., 2007b), (Chang et al., 2006), (Singh & Huhns, 2005). Especially in term of WS standards, practical implementations have been purposed (Cabrera, Copeland, Feingold, Freund, Freund, Johnson, Joyce, Kaler, Klein, & Langworthy, 2005), (Furnis et al., 2004), (Cabrera, Copeland, Feingold, R. Freund, Freund, Johnson, Joyce, Kaler, Klein, Langworthy, Little, et al., 2005). The application of the transactional concept in WS-* adds considerable complexity to the required coordination framework (Furnis & Green, 2005), (Razavi et al., 2007c). This can be seen more clearly when analysing the pattern behaviour for the recovery model (*compensation*) (Razavi et al., 2007d), (Vogt et al., 2005).

In our previous works (presented in chapter 4, 5 and 6), (Razavi et al., 2007d), (Razavi et al., 2007c), (Razavi et al., 2009), we have provided general platform design, pattern behaviour, concurrency control and recovery management for a service-oriented environment, by focusing on specific requirements of Digital Ecosystems (recall chapter 1), (Nachira, 2002a), (Chang et al., 2006). Introducing Representational State Transfer (REST) as a distributed computing architectural style (Fielding, 2000), has created a new opportunity for achieving Digital Ecosystem requirements. In contrast with WS protocols, REST works directly with resources. This is in line with the semantics of the basic theorems in conventional transaction processing (Gray & Reuter, 1993). Transactions rely on read/write operations on objects, and RESTful HTTP, likewise, provides GET (equivalent to 'read') and DELETE, POST, PUT (equivalents of 'write') methods. Various approaches have been proposed for handling RESTful transactions. The traditional approach is to simply design a new resource that can be used to trigger the desired transaction on the server side. For example, when trying to transfer funds from one bank account to another, there could be a 'transfer request' resource to which new 'transfer requests' can be posted. While it can be very simple to implement at design time, it constrains users to the predictive ability of the developers at design time. Furthermore, in scenarios where a large or unpredictable variation of transactions may take place, it cannot be expected that all the necessary resources have been designed beforehand. This situation is similar to the static versus dynamic allocation debate found in the database and transaction literature (Bernstein & Newcomer, 1997),(Gray, 1992). The approach completely breaks down however, when a transaction exceeds the scope of a single provider, the case of distributed transactions. Other approaches such as (Khare & Taylor, 2004) suggest extending REST to include mutex locks, but this would necessitate extending HTTP as well. The alternative to these approaches is to introduce locks on resources by modelling

them as resources themselves (Richardson & Ruby, 2007). While this approach looks much more capable, the details of its implementation and its extension into transactions have neither been fleshed out nor proven. In this paper we describe how this approach can be extended to produce a fully specified and theoretically robust RESTful transaction model. In (Razavi et al., 2009) we have provided a primary framework for consistent RESTful transactions and in this paper we focus on Recovery issue.

11.3 From General Recovery model to RESTful

Conventionally in transaction management, Recovery procedure needs sophisticated API, with high algorithm complexity. Figure 1 shows the conventional paradigm of Recovery Management in the transactional environment (Gray & Reuter, 1993).

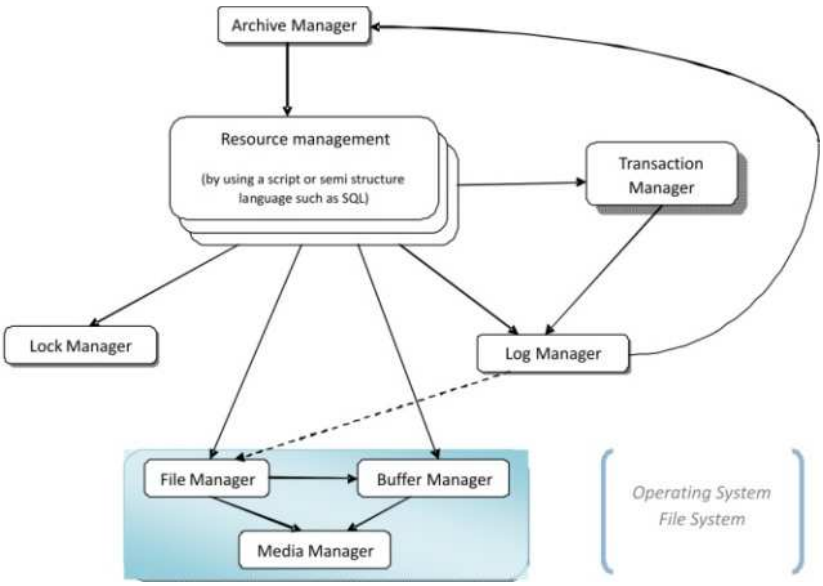

FIG. 11-1 CONVENTIONAL RECOVERY MANAGEMENT [(GRAY & REUTER, 1993)-PP495]

In Fig 11-1, '*Log Manager*' keeps step-by-step logs of operation of run-time transactions, when '*Lock Manager*' supports the consistency of concurrent transactions. The entry to '*Log Manager*' is created by '*Transaction Manager*', when a transaction has been created and logs will be archived ('*Archive Manager*') when the transaction commits. The Transaction will be built by using a script language or any semi structure language (such as SQL). The actual data-items and meta-data will be managed by operating system and distributed or stand-alone file systems.

When a failure happens, '*Lock Manager*' by isolating the written data-items, avoids spreading the failures between concurrent transactions. '*Log Manager*', offers necessary logs for rolling back the failed transaction, and the transaction entry can be deleted by '*Transaction Manager*'. This

conventional framework needs complex API, where all components are tightly coupled. For providing loosely coupling (as one of the most important requirements of Digital Ecosystems), the extra complexity will be added to the usual complex recovery algorithm which may harm SMEs as the main participants of Digital Ecosystems (Razavi et al., 2007b). Fig 11-2 shows our RESTful approach, which instead of using API for each manager, introduces suitable resources to increase the feasibility of Recovery Management and Transactional implementations for SMEs.

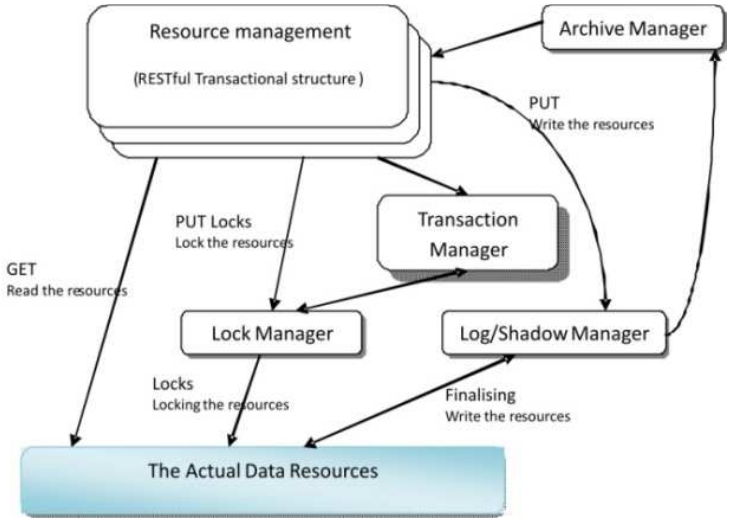

FIG. 11-2 RESTFUL RECOVERY MANAGEMENT

In the RESTful recovery management, interactions are represented by standard HTTP operations on resources. So a user applies GET to the resource to retrieve it. This also alters the nature of the managers. The Lock manager and the Transaction Manager are actually a Lock Collection per resource and a separate Transaction Collection. New transactions are POSTed to a transaction collection and new locks on a resource are POSTed to the lock collection of the resource. Since new locks have to reference the transaction they belong to, the locks that belong to a transaction can be accessed through the transaction lock collection. Each exclusive lock is accompanied by a shadow resource that will become the new representation of the resource when the transaction is finalised. That shadow resource can be manipulated as a regular resource via PUT and DELETE. When the transaction is finalised, the shadows become the normal resource representations but also the entire resource structure relevant to a transaction, including the initial state of the resources that were modified is archived. In this way, the RESTful paradigm has implemented the functionality of the managers into lightweight resources that can potentially be subject to transactions themselves.

11.4 LOCKS IN RESTFUL HTTP

In order to handle concurrency challenges in HTTP, we introduce the concept of locks. This is done

in a way that does not affect the always available and backwards compatible nature of the web

11.4.1 LOCKING RESOURCES

For an API to be characterized as RESTful according to the hypermedia constraint, it must allow a client to interact with the service solely by being given a single entry URI and understanding of the relevant media types. This enforces loose-coupling and elimination of assumptions.

Lockable Resource (R): Ideally, any resource that can be served by an HTTP server should be lockable regardless of serialization format. This however would require the HTTP protocol to carry the metadata for the locking mechanism. Since we wish to preserve the HTTP protocol, we opt for a fragment of XML that is to be included in an XML representation of a resource. This approach could potentially be extended to other formats such as JSON [5] but not to binary files such as images or zip archives. The information that should be in the fragment is the location of the lock collection and the location of the transaction collection. The inclusion of this fragment (Fig 11-3) makes any resource lockable. Namespaces could also be utilized to avoid namespace collision but this would limit the approach to serializations that support namespaces.

```
<lockable>
<link        rel="lock_collection"          href="http://example.org/resource/locks/">
<link        rel="transaction_collection"   href="http://example.org/transactions/">
</lockable>
```

FIG. 11-3(R) XML FRAGMENT

Lock Resource (R-L): The lock resource is represented by a dedicated media type and should contain the elements in Table 1.

ResourceURI: a link back to the resource that this lock affects.
TransactionURI: a link to the transaction that controls the lock.
Type: "S" or "X" depending on the type of the lock.
PrevLockURI: a link to the previous lock in the lock sequence.
Timestamp: Server's timestamp when the lock was granted.
Duration: Indicates the interval that the lock has been granted for.
ConditionalResourceURI: A link to the representation of the resource that will come into effect once the lock is committed.

TABLE 4 - ELEMENTS OF R-L

The type element can take one of two values, X or S, corresponding to the available lock types. X stands for **XLOCK:** eXclussive Lock, and S stands for **SLOCK:** Shared Lock. To place a new lock, the

server must authenticate the user as the owner of the transaction that is referenced by the lock. The length of time of effectiveness that is granted to a lock is dependent on the maximum length of time that the server is prepared to grant a guarantee to the client. Once the duration of the lock expires, the lock is aborted. The result of the GET operation does not change until a lock of type X is committed. In this sense, the locks and transactions are transparent to the GET which on commit reacts as if a simple PUT was applied. This was a specific design objective. PUT and DELETE operations return a *'405 Method Not Allowed'* HTTP response for the duration of a lock's effect. GET requests should still return successfully. This behaviour maintains backwards compatibility, with the understanding that if a client requires further guarantees on the future state of the resource, the client should seek to place a lock. In all other cases, the semantics of GET are unaffected, as a GET on a resource does not guarantee that the state will remain unchanged for any period of time.

11.4.2 WELL-FORMED COLLECTIONS OF LOCKS

As expected, a transaction cannot lock a resource that is locked by another transaction. But if two or more transactions want to GET the content of a resource, they are not going to change the resource state. This will therefore not cause any conflict or access to data which has been PUT to a resource by another transaction, but the first transaction has not committed and may change the version of the resource again). Table 2 shows the lock compatibility. The inferred rules constrain the set of allowed histories. Histories that satisfy the locking constraints are called *legal histories*.

		Mode of Preceding Lock	
		Share	*Exclusive*
Mode Of New Lock	*Share*	*Yes*	*No*
	Exclusive	*No*	*No*

TABLE 5 – LEGAL LOCK SEQUENCES

Resource Lock Collection (R-Lc): The R-Lc contains locks in sequences that follow the compatibility rules stated in Table 2, rendering the transaction well-formed. The lock collection is represented as an Atom Feed (Hoffman & Bray, 2006). Since ATOM does not support sequencing entries, we use the 'PrevLockURI' element of the lock resource to create a linked list of locks that can be represented as an ATOM Feed. The client can retrieve the lock collection via GET to determine if the resource is locked. An empty feed indicates an unlocked resource. New locks can be submitted to the resource collection via the POST method.

GET Returns the resource's collection of locks.

POST	Adding a new lock to the related resource

TABLE 6 - AVAILABLE OPERATIONS FOR R-LC

11.5 TWO PHASE LOCKING AND RECOVERABILITY

So far our model provides a well-formed locking mechanism. However, to provide a consistent RESTful Transaction Model, we need to provide a wormhole-free locking mechanism, clarify the scope of each transaction with respect to the locking mechanism and facilitate recoverability in the model. In this section, we demonstrate why two phase locking has been chosen to offer a consistent environment, as it is wormhole-free, and then we extend the model to provide two phase locking and recoverability.

11.5.1 TWO PHASE LOCKING IS WORMHOLE FREE

As has been described in 11.4, our model provides a well-formed locking system for GET and PUT. We now show that by adding two-phase locking, the model becomes wormhole-free (Gray & Reuter, 1993). We showed in '*Lock Resource (R-L)*' (section 11.4.1), that each transaction can use two different types of Locks for its resources (SLOCK for GET and XLOCK for PUT). Therefore in $H = \langle\langle t, a, r\rangle_i | i = 1, \dots, n\rangle$ we can consider two extra actions for 'a': SLOCK; and, XLOCK. Meanwhile as these locks at some point should be released, we have UNLOCK as another action for 'a'. Based on 'Two-Phase Locking', each transaction can use locking in two phases. In the first phase (Growth), it can acquire locks for resources (SLOCK or XLOCK) and in the second phase (Shrink), it releases them. These two phases should not have any overlap. When the transaction starts to UNLOCK a resource, it cannot lock any more resources under any circumstances. So, unlocking resources means that the transaction is either successfully committing or aborting. Now, we want to show that if all transactions are well-formed and two-phase, any legal history will be isolated (wormhole-free).the proof relies on contradiction. Suppose *H* is a legal history of the execution of a set of transactions, each of which is well-formed and two-phase. For each transaction, *T*, define SHRINK(*T*) to be the index of the first unlock step of T in history *H* ; formally:

$$\text{SHIRINK}(T) = min(i | H[i] = \langle T, \text{UNLOCK}, r\rangle \text{for some resource}))$$

Since each transaction *T* is non-null and well-formed, it must contain an *UNLOCK* step. Thus SHRINK is well defined for each transaction.

Transaction (T) : The transaction resource is represented by a dedicated media type (e.g. application/vnd.retro-transaction+xml). It should contain the elements in Table 4.

TransactionCollectionURI	

OwnerURI
TransactionLockCollectionURI
TransactionStatus (Active | Committed | Aborted)

TABLE 7 - ELEMENTS OF T

These 3 elements identify the collections of information vital to the execution of a transaction. When the transaction is created its status is set to 'Active'. The owner of the transaction can GET the transaction resource as a means of locating these collections and reviewing its status. Deleting a transaction with active locks will result in the transaction being committed. Its status will change to 'Committed' when the commitment process is complete. A committed transaction and its related resources cannot be altered after the transaction is committed as the resources now serve as an archive of the transaction.If the lock collection of a transaction is deleted, the status of the transaction becomes 'Aborted'.

Transaction Collection (Tc): The transaction collection is a resource where new transactions are submitted via the POST operation which creates a new transaction and returns the URI for its representation. The resource itself cannot be accessed via GET as the clients that need to know the location of a specific resource are informed at the time of POSTing.

Transaction Lock Collection (T-Lc) : The transaction lock collection contains links to the locks that belong to a specific transaction, formatted as an Atom feed. Clients cannot abort single locks directly but must do so by Deleting the T-Lc which aborts all the locks of a transaction, leaving the transaction void and is equivalent to aborting the transaction.

GET	Returns the collection of locks relevant to a transaction
DELETE	Aborts all the locks of the relevant transaction. This can only be performed by an owner of the transaction and results in the transaction being aborted.

TABLE 8 - AVAILABLE OPERATIONS FOR T-LC

11.5.2 RECOVERABILITY

Based on the Rollback Theorem, a transaction that unlocks an exclusive lock and then performs a '*Rollback*' is not well-formed and can potentially cause a wormhole unless the transaction is degenerated. As the theorem is well-known, we refer the interested reader to (Gray & Reuter, 1993) for the actual proof. The important point of the theorem is that we have to degenerate the transaction to effect rollback. For this purpose, our model does not store potential updates on the actual resources but works on the shadow of the locked data, called a *conditional resource representation*.

InitialResourceRepresentation(R-L-IR): A resource that is of identical media type as the locked resource and stores the initial state. The initial representation is archived together with the lock to

represent the change caused by the commit of the lock and enable rollback.

GET	Returns the representation that was the initial state of the resource before locking

TABLE 9 - AVAILABLE OPERATIONS FOR R-C

ConditionalResourceRepresentation(R-L-CR): A resource that is of identical media type as the locked resource. The conditional resource representation is essentially the state that will be applied to the resource once the XLOCK is committed.

GET	Returns the representation that will be committed if the relevant XLOCK is committed.
PUT	Creates a new conditional state that will replace the current state of the locked resource once the linking XLOCK is committed.
DELETE	Deletes the conditional state. If the XLOCK is committed, there will be no write action performed.

TABLE 10 - AVAILABLE OPERATIONS FOR R-C

11.5.3 Operation Coverage

In a database environment, a transactional workflow is nearly identical to its non-transactional counterpart, differing only by the fact that it is executed in a transactional context. The database platform can then infer the implications on locking that are implied. In a RESTful environment, this luxury does not exist, as statelessness is a stated goal. It therefore is not possible to simply start a transaction and perform all the operations we wish to and assume transactional guarantees. Each operation must be performed in a manner that explicitly places it in the transactional context. Due to the differing semantics of each HTTP operation, this must be applied separately for each type. GET and HEAD operations are positioned in the transactional context, as long as the requested resources are locked (S or X) by the containing transaction. In order for a PUT in its update incarnation to be within a transaction, it needs to be applied to the conditional representation of the XLOCK of the resource. The lock must belong to the containing transaction. Similarly, a DELETE can be performed by deleting the conditional representation. A PUT that aims to create a resource is similar to an updating PUT. The difference is that the lock refers to a resource that does not exist. Any updates to the conditional representation of the resource will become the initial state of the newly created resource at the time of commitment of the transaction. These patterns aim to expand the scope of the covered operations and remain intuitive while remaining within the statelessness constraint of REST. POST operations are not yet covered as they require a new top-level activity for the transaction, different from a lock. Additionally the URL that the server creates becomes a data dependency for the

transaction. POST operations are currently outside the scope of our model, as they require significant additions that will be the focus of future work.

11.5.4 Model overview

Having defined all the resource types, it is easy to see that an interconnected network arises. Fig. 11-4 displays the interconnections of the resource graph. It can be observed that having a URI for R is enough to locate all other resources in the network. The connection from Tc to T is different from the other connections as there is no GET ability for the Tc resource, as for security reasons no user can receive a list of all current transactions. The URI of a given T is only returned as a response to the initial POST operation on Tc performed by the transaction's owner.

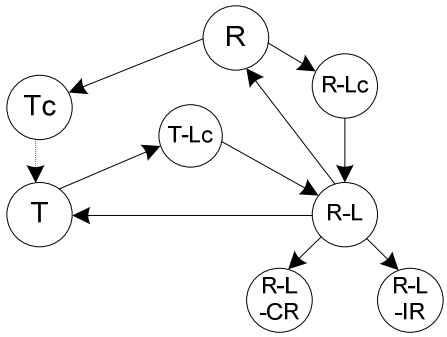

FIG. 11-4 RESOURCE HYPERMEDIA CONNECTIONS

Table 8 summarizes all the relevant resource types that comprise our model together with a short description and a list of the allowed operations.

Lockable Resource (R)	A resource that locks can be applied to Operations: GET, [By XLOCK owner: PUT]
Resource Lock Collection (R-Lc)	The collection of locks that apply to a particular resource. Operations: GET, POST
Lock Resource (R-L)	The representation of a specific lock Operations: GET
Conditional Resource Representation (R-L-CR)	The potential representation of a locked resource, once its lock is committed. Operations: GET, [By XLOCK owner: PUT, DELETE]
Initial Resource Representation (R-L-IR)	The potential representation of a locked resource, once its lock is committed. Operations: GET, [By XLOCK owner: PUT, DELETE]
Transaction Collection (Tc)	The collection of transactions on the server. Operations: POST
Transaction Resource (T)	The representation of a specific transaction. Operations: GET

Transaction Lock Collection (T-Lc)	The collection of locks connected to a specific transaction. Operations: GET, [By transaction owner: DELETE]

TABLE 11 – RESOURCES AND OPERATIONS

11.6 EXAMPLE SCENARIO

A transaction, such as in fig 11-5, can be aborted at any point by the client. Throughout the course of the lock's validity, the data of the locked resources (R1, R2) remain safe from alteration from the transaction itself or from external clients. When the transaction is aborted, all conditional representations are deleted with no effect on the actual state of the resource. In fig 11-6 an example of a transaction committing can be seen. On commit, the conditional representation of an active XLOCK becomes the new version of the locked resource. The transaction resource as well as all other relevant resources need not be deleted but can be archived for future rollback as they constitute an effective recording of all the changes made by the transaction. They also include the state previously in effect through the R-L-IC resource and therefore can be used to reverse the changes if needed.

Client	Operation	Resource	Response	Description
A	GET	R2	200 OK	GETting R2 to extract location of TC and R2-LC
A	POST <new transaction>	TC	201 CREATED {Location: T1}	Creating a new transaction
A	POST <LOCK {type:X}>	R2-LC	201 CREATED {Location: R2-L1}	POSTing an XLOCK to R2-LC
A	GET	R1	200 OK	GETting R1 to extract location of R1-LC
A	POST <LOCK {type:S}>	R1-LC	201 CREATED {Location: R1-L1}	POSTing an SLOCK to R1-LC
B	PUT <new version>	R1	405 Method Not Allowed	Another client attempting to modify R1
A	GET	R1	200 OK	GETting the locked representation of R1
A	GET	R2	200 OK	GETting the locked representation of R2
A	GET	R2-L1	200 OK	GETting R1-L1 to extract location of L1-CR
A	PUT <new version>	R2-L1-CR	201 CREATED	Creating a conditional Representation of R2
A	GET	T1	200 OK	GETting T1 to extract location of T1-LC
A	DELETE	T1-LC	200 OK	Abort: Deleting R2-L1-CR and Unlocking R1 and R2

FIG. 11-5 EXAMPLE OF A TRANSACTION ABORTING

Client	Operation	Resource	Response	Description
A	GET	R2	200 OK	GETting R2 to extract location of TC and R2-LC
A	POST <new transaction>	TC	201 CREATED {Location: T1}	Creating a new transaction
A	POST <LOCK {type:X}>	R2-LC	201 CREATED {Location: R2-L1}	POSTing an XLOCK to R2-LC
A	GET	R1	200 OK	GETting R1 to extract location of R1-LC
A	POST <LOCK {type:S}>	R1-LC	201 CREATED {Location: R1-L1}	POSTing an SLOCK to R1-LC
B	PUT <new version>	R1	405 Method Not Allowed	Another client attempting to modify R1
A	GET	R1	200 OK	GETting the locked representation of R1
A	GET	R2	200 OK	GETting the locked representation of R2
A	GET	R2-L1	200 OK	GETting R1-L1 to extract location of L1-CR
A	PUT <new version>	R2-L1-CR	201 CREATED	Creating a conditional Representation of R2
A	DELETE	T1	200 OK	Commit: R2-L1-C becomes R2 and R1 and R2 unlock

FIG. 11-6 EXAMPLE OF A TRANSACTION COMMITTING

11.7 Future Work

We have provided a RESTful framework for Recovery management which includes a locking concurrency control. Our locking mechanism protects the write-access to resources in one hand and offers the safe copy of resources for transactional access which easily is recoverable. This work can be considered as the necessary primary step for using RESTful transaction in Digital Ecosystem. Supporting Long-running transaction, purposing more user friendly interface languages (instead of pure RESTful script for *Resource Management*) and analysing complexity of traffic are expected future extensions to this work.

12 APPENDIX III: SIMULATIONS AND IMPLEMENTATION OF DIGITAL ECOSYSTEMS

This appendix includes two sections; the first section is the result of collaborative work of OPAALS' team with Zheng for implementing the first version of Digital Ecosystem Transaction model:

- S. Moschoyiannis, A. Razavi, Y. Zheng and P. Krause (2008) Long-Running Transactions: semantics, schemas, implementation, In Proc. of IEEE Int'l Conf. on Digital Ecosystems and Technologies (IEEE-DEST 2008), IEEE Computer Society.
- P. Krause, A. Marinos, S. Moschoyiannis, A. Razavi, Y. Zheng, T. Kurtz, et al. (2008). Full Architecture Definition for Autopoietic P2P Network Version 1. Project Acronym: OPAALS, European Community, Framework 6, Contract No: 034824.

The second section is a primary result of one of the early simulation of Digital Ecosystem.

12.1 A SCENARIO WITH SEQUENTIAL SERVICE DEPENDENCY

We have used JAX-WS 2.1, Java 5, and Netbeans IDE 5.5.1 in our implementation framework. The JAX-WS is a new standard for message passing, it supports both synchronous and asynchronous message passing by the polling model and the call-back model, respectively. In the polling model, the client continuously polls the service response. In the call-back model, the client creates a call-back handler. JAX-WS is shown to perform better than JAX-RPC in certain aspects (Mundlapudi, 2006). Furthermore, JAX-WS supports static and dynamic generation of web service client stubs.

In our implementation, the JAX-WS call-back message passing and dynamic client stub generation are used. The call-back message passing is suitable for asynchronous message passing, which is important for long-running transactions. The dynamic WS client creation enables web service invocation chains. Three participants are required in order to make a service X work: 1) X web service; 2) X TransactionAgent web service; 3) the client stubs for the X web service. The dependencies between web services are captured in an XML file, as discussed before.

From the transaction vector schema, we can create different transaction scenarios (as an XML file). The generated xml file shows the various states of a transaction for a specific scenario. We have designed a software agent coordinator which can perform the coordination of the service invocations as prescribed by the transaction vector language in a fully distributed manner. As a case study we analyse a simple transaction, with a sequential service composition, which involves the interactions shown in the sequence diagram of Fig. 12-1.

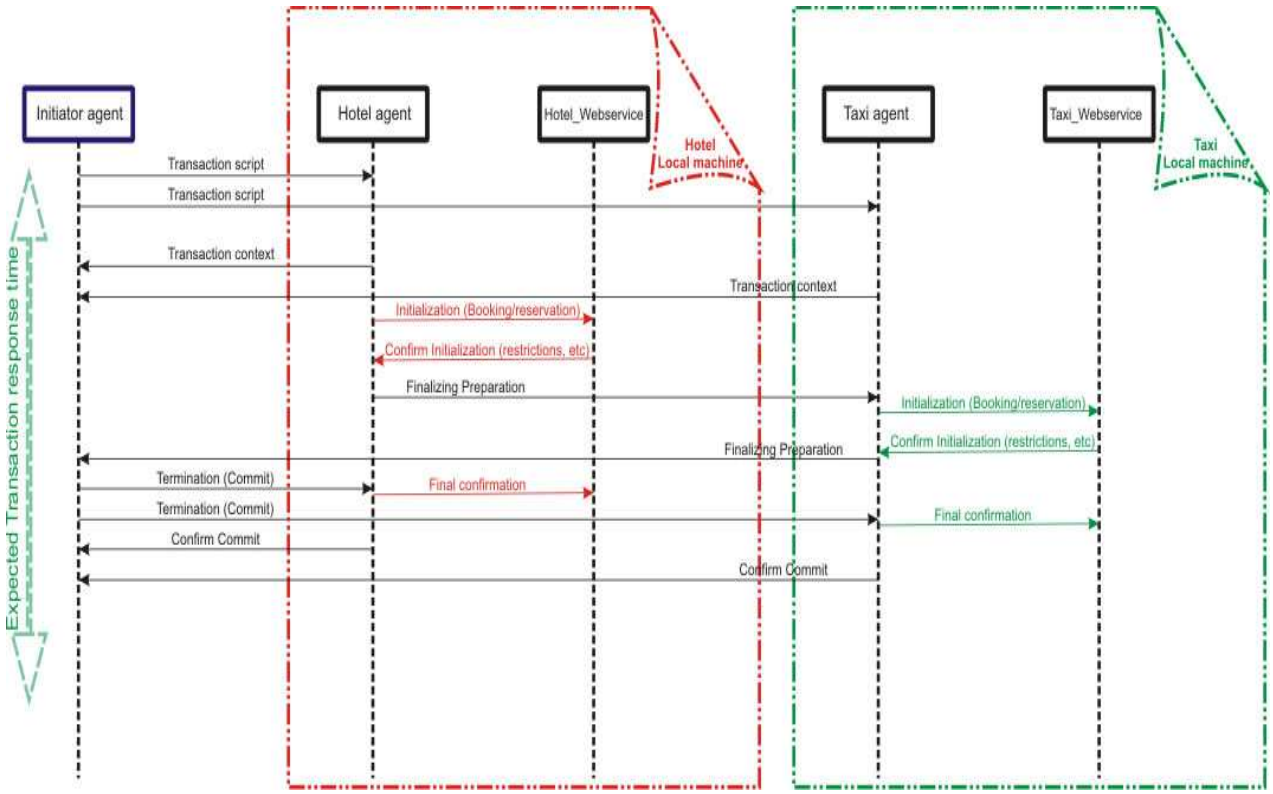

FIG. 12-1 A SIMPLE TRANSACTION WITH A SEQUENTIAL SERVICE COMPOSITION

Our simple transaction scenario involves two participants, located at two different SMEs (Hotel and Taxi services). The initiator agent has an AgentHelper that communicates with the TransactionAgents. The transaction involves four players: HotelTransactionAgent (HTA; local coordinator of the Hotel), HotelService (webservice of the hotel), TaxiTransactionAgent (TTA; local coordinator of taxi), and TaxiService (webservice of the taxi). The AgentHelper plays the Initiator role and it starrts the transaction by sending messages (setting the transaction context) to both HTA and TTA. This specifies the first web service should be deployed on the first access point (HTA) and the second will be deployed only after receiving the successful confirmation of the first (sequential service composition).

The following information about the required distributed coordination can be derived: 1) the AgentHelper will know that after it initiates the HTA and TTA, it needs to wait for a response from the TTA for the final preparation status; 2) the HTA will know that it depends on TTA so it needs to send a message to TTA asking for final preparation status; 3) the TTA will know that it gets a request message from the HTA and then sends its response message back to the AgentHelper for the final preparation status.

Fig 12-2 shows the performance analysis of the case study in terms of CPU performance, memory usage, and thread usage when the client performs a commit transaction. The Netbeans 5.5.1 source is available following (Razavi & Moschoyiannis, 2008).

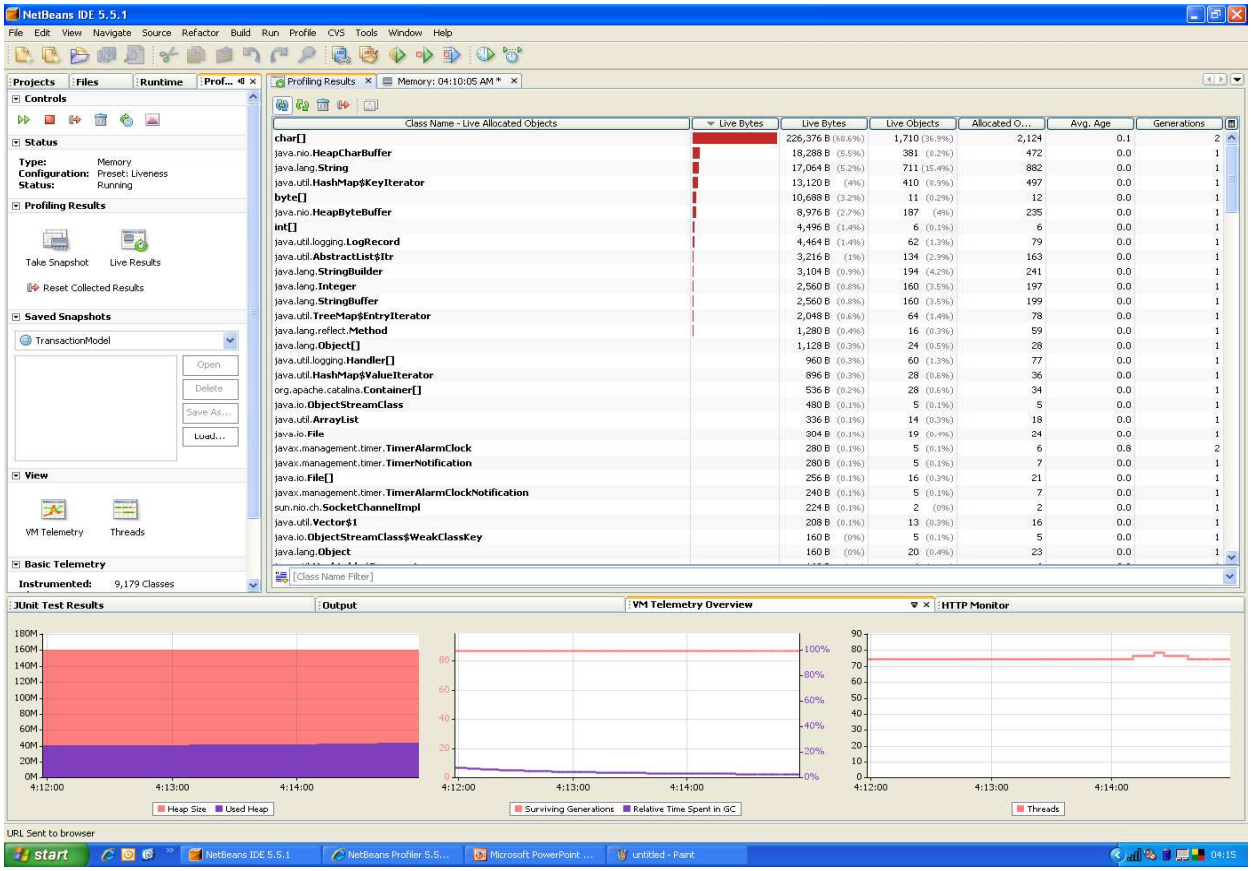

FIG. 12-2 RUN-TIME BEHAVIOUR OF THE SCENARIO

12.1.1 CONCLUDING NOTE

In this section we have been concerned with the distributed coordination of the service compositions involved in long-running transactions in DEs, and have described schemas which can be used to derive XML representations of the underlying formal model (transaction vectors) and guide the implementation of the proposed transaction model. This was illustrated with a simple example involving sequential service dependencies in a transaction between three participants.

When talking about service composition, in SOA terms but also more generally, we need to take into account dependencies that may arise due to: (i) ordering (of service invocations) and, (ii) data dependencies due to exchange or sharing of results. In this section we have dealt with dependencies that arise due to ordering. In previous work, reported in detail in Deliverable D3.2 of OPAALS (Razavi et al., 2007b), and subsequently published in (Razavi et al., 2007c), we have described an extended lock scheme that is used to handle data dependencies in a transactional

setting with the desired capabilities.

Similarly, dealing with the order of service compositions in long-running transactions comes with a need for coordinating the required service invocations in a principled manner. This is not only relevant for ensuring that the set-up of the transaction includes no more than the desired behavioural scenarios (we have more to say on this in the closing paragraph of this section) but is necessary for the complete specification of a long-running transaction, in terms of expressing both *forward* and *compensating* behaviour. So in addition to modelling the sequences of service invocations required for a successful outcome, it is also necessary to be able to model the sequences of compensating actions required when some forward action of the transaction fails (compensating behaviour). We have not dealt with compensating behaviour in this section, and therefore refer the reader to Deliverable D3.2 of OPAALS (Razavi et al., 2007b) which deals with this additional dimension of a multi-service transaction setting and gives a detailed account of how we go about handling failures in our model.

Our model for long-running transactions, to the extend it has been described in this section, provides a way of expressing the service coordination implied in a given transaction along with a schema representation for the formal modelling of such coordination, which can be used to derive XML descriptions of the required orderings on service invocations, the so-called *transaction scripts*. This aspect of the OPAALS distributed transaction model can be extended in interesting ways. In particular, we have been concerned with the refining the initial specification of a transaction, which may be given as a UML model of service interaction scenarios, to ensure that the actual order in which services are invoked respects the required orderings of service invocations in the specification. Our ideas on the gradual elaboration of behavioural scenarios in long-running transactions can be found in (Moschoyiannis et al., 2008).

12.2 Simulation and expectation

This section is a primary result of one of the early simulation of Digital Ecosystem:

- Razavi, S. Moschoyiannis, and P. Krause (2008) A Self-Organising Environment for Evolving Business Activities, The First International Workshop on Computational P2P Networks: Theory & Practice (ICWMC 2008 // ICCGI 2008 // ComP2P), 2008, July 27 - August 1, 2008 - Athens, Greece.

We focus on the final result, in comparison with a conventional implementation of Digital Ecosystem; for current comparison is with a conventional unstructured FADA network (Razavi et al., 2006), (TechIDEAS, 2007)).

12.2.1 Analysing the result

In the first step, by using the frequency distribution of links per node we try to clarify the distribution of links among nodes. This enables us to compare similar snapshots of nodes for FADA and our network in a similar situation.

For clearer comparison, we use cumulative frequency analysis to examine the network situation during a critical situation. For example, in the next section we investigate the impact in case of failure of crucial nodes on the network, and specifically its connectivity. Despite of the facility for comparison between two different types of networks and questioning the topology reaction in a decisive situation, this can steer us to formulate the confidence statements (as the indication for the reliability of network) as future work.

12.2.2 A conventional Digital Ecosystem

FADA (TechIDEAS, 2007) as a conventional implementation of a digital ecosystem is a scale-free network, which is relying on a few hubs. The effect of this for SMEs is an inevitable bottleneck at peak time. As a simple example Fig. 3-2 shows how the core infrastructure may indeed rely on a few hubs. This not only causes high traffic on peak time (and as a result instability of hubs during this time), but also the possibility of fragmentation and creating islands in the network grows. Especially when we take into account the regular unavailability of SMEs based on their business model and regional working hours. Fig. 12-3 shows the relationships between the number of nodes (on the vertical axis) and the number of links (horizontal axis). Clearly, a few number of nodes have the most number of links (high degree) while the majority of nodes have just a few links (and these few links mostly end up to a high degree node).

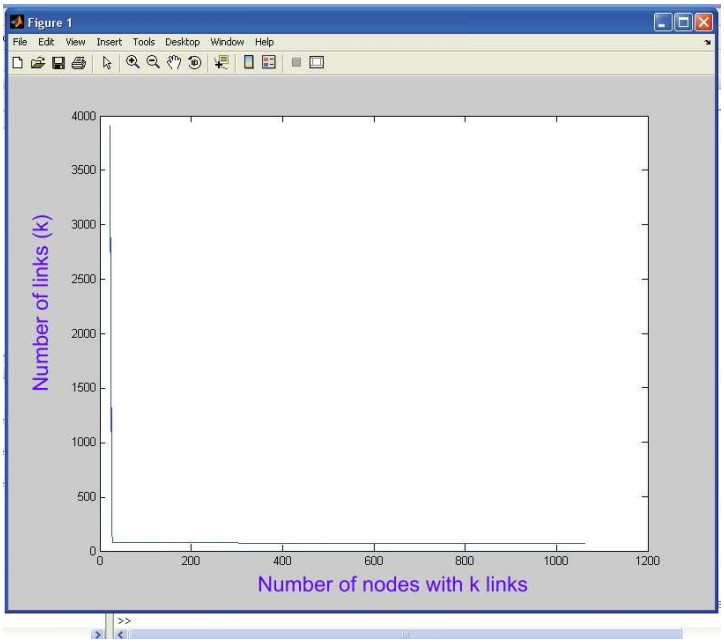

FIG. 12-3 DISTRIBUTION DEGREE OF A CONVENTIONAL DIGITAL ECOSYSTEM

Based on Fig 12-3, in a conventional Digital Ecosystem, a few number of nodes have the most links (8 nodes have more than 2500 links) and majority of nodes have a few links (more than 4000 nodes with less than 5 links). As a result the network has to rely on a few hubs which may not be the most reliable nodes. Conceptually, they have been hubs because of the high number of business transactions which they participate in – the result is close to chapter 7 estimation.

As a typical scale-free network whose distribution degree follows a power law distribution, any failure (or high traffic complexity) on hubs can cause immediate disruption at the transactional level (abortion of the majority of transactions) and fragmentation of the network. These problems are addressed in our current design and the use of virtual super peers shows significant improvement on the infrastructure of the digital ecosystem. As demonstrated in the next section, the dynamic topology of the network can react in response to failures or attacks on the virtual hubs.

12.2.3 DYNAMIC VSPS MODEL FOR A DIGITAL ECOSYSTEM

By using a dynamic measurement for choosing nodes in VSPs, the dependency on a few nodes with higher distribution degree decreases dramatically. Fig. 12-4 shows an example of a DE where links are propagated on different nodes.

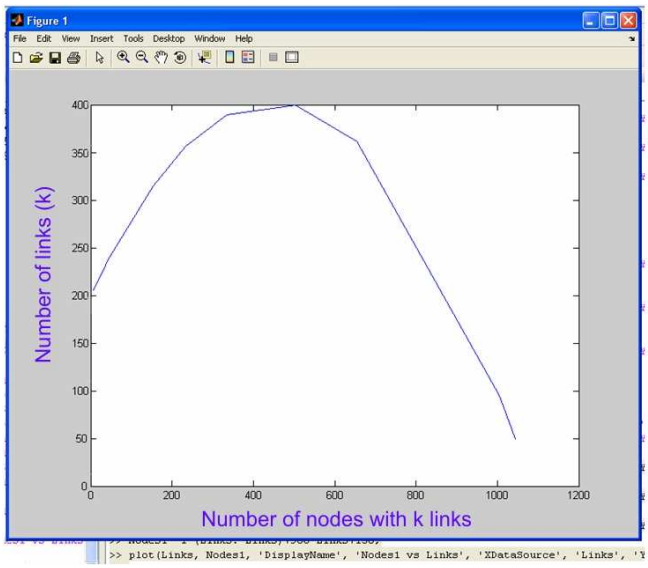

FIG. 12-4 DISTRIBUTION DEGREE OF DGITAL ECOSYSTEM WITH VSPS

Our primary results with the same number nodes (5000), shows spectacular shift in comparison with the typical FADA infrastructure (compare with Fig. 12-3) when more than 800 nodes have more than 350 links (nodes which made virtual super peers) and even their neighbouring nodes (which have node stability measurement close to 1) have large number of links (about 900 nodes with more than 300 links). These become good candidates for joining the virtual super peers by substituting existing member nodes during failures or attacks on current VSPs. As a result, using Virtual Super Peers and relying on node stability, rather than the business activity, provides a fairer distribution degree. As depicted in Fig. 12-4 Considerable numbers of nodes have higher number of links.

12.2.4 FAILURES AND REACTIONS

Fig. 12-5 shows the result of a simulation of 800 simultaneous failures to all VSPs nodes (nodes in permanent clusters forming Virtual Super Peers). These nodes lose all of their links and then only through performing a few business transactions they rejoin the network. As a result of these failures they have very weak stability (NodeStability close to 0). Therefore, despite of their transactions they will not receive link replications and still have a low number of links (less than 75 in the simulation of Fig. 12-5).

Meanwhile their neighbourhood nodes have been substituted in the VSPs structure and their links increased (Fig. 12-5 shows this effect). But the interesting point is that still the network has not suffered any fragmentation. We may experience some longer response time on business transactions execution but the Digital Ecosystem does not suffer the full failure. Typically, such a severe attack resulting in this type of simultaneous failures would cause several fragmentations.

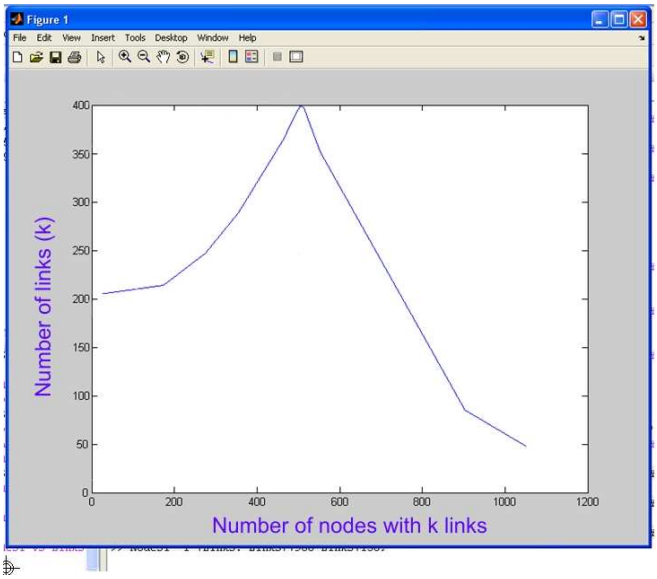

FIG. 12-5 DISTRIBUTION DEGREE AFTER FAILURE ON VSPS