Digital Business Ecosystem

Contract n° 507953

WP 13: Business networks

D13.1: Economic models of software ecosystems

Project funded by the European Community under the “Information Society Technology” Programme

Contract Number: 507953

Project Acronym: DBE

Title: Digital Business Ecosystem

Deliverable N°: D13.1

Due date: 31/03/2005

Delivery Date: 23/04/2005

Short Description:

For the creation of a functioning software ecosystem, not only infrastructural, but also business challenges must be solved. An economic theory of how effective certain measures are in overcoming conflicts of interest, coordination and trust challenges must start from an understanding of individual actors' decision problems. Software is an economic good with peculiar characteristics, such as zero marginal costs of production, which has consequences for vendors' strategies. The report presents a hybrid game-theoretic model for strategic interaction between software vendors in their relatively long-term decisions on product portfolio and licensing.

Partners owning: FZI

Partners contributed: FZI, UniS, LSE, STU

Made available to: Project members and Commission

VERSIONING

VERSION	DATE	AUTHOR, ORGANISATION
1.0	**23/04/2005**	**TIM ROMBERG, FZI**
1.1	**26/04/2005**	**TR: Added standard header and footer. Key to Figure 1 provided. Hyperlinks activated.**
1.2	**09/05/2005**	**TR: Missing coefficient in equation 18**

Quality check
1st Internal Reviewer: Konstantinos Giannoutakis (University of Surrey)
2nd Internal Reviewer: Gordon Gow (London School of Economics)
Additional comments by Thomas Kurz (Salzburg Technical University)

Contents

1 Introduction

Digital Business Ecosystems require participating users with potential business relationships to converge on a common set of technologies to use in their business transactions. They require software developers to adopt a component-oriented mindset, both developing components with reuse by others in mind and integrating components developed by others. Much progress has been made towards solving the *infrastructural* challenges of component software and cross-enterprise service architectures, and remaining ones are addressed by other parts of the DBE project. But some *business* challenges remain:

- the *conflict of interest* challenge — business interests of important players can run counter to the establishment of a lively ecosystem. In business terms, we usually associate with a digital business ecosystem more variety, more flexibility (e.g. lower switching costs), and lower barriers to entry. While desirable from a global point of view, these can be a threat to some established players;
- the *coordination* challenge — a lack of coordination and communication in early engineering stages can lead to unintended interoperability problems, i.e. not caused by a business intention to create high switching costs and lock-in;
- the *trust* challenge — software developers generally avoid the risk of being dependent on a single third party's components. While the price of these components can be quite attractive, with small vendors there is the risk of bankruptcy and discontinued support, of bugs which the developer cannot fix him/herself. Components or frameworks from big vendors (Microsoft, Oracle, IBM, SAP) are more accepted, but do not cover specific business domains, and some developers are also afraid to become too dependent on a big vendor.

These challenges will likely have to be addressed by

- passive coordination facilities (e.g. means of communication),
- rules of *governance* which define the participants' rights and obligations with respect to the other participants and with respect to the project as a whole, and,
- at least during the start-up phase, active coordination by the project.

The intended contribution of work package 13 *business networks* is to provide, based on economic models and past experience, a theory of the effectiveness of these measures. The larger scientific question addressed is: *How are technology ecosystems based on a platform successfully created and sustained?*

1.1 Components, Platforms and Ecosystems

Software Component A well-known definition of a software component by [Szy99], which balances economic and technical aspects, is

> "Components are binary units of independent production, acquisition, and deployment that interact to form a functioning system."

Since then, the common usage has evolved a little bit. In Software Engineering, today, quite small elements of a program are often called a component, which never appear independently on the marketplace, or even exist as separate binaries as suggested by the definition above. A definition reflecting this is "A component is an object written to a specification."[1] On the other hand, the units of software which are actually traded in the marketplace, and which have component character because they need to be integrated with other components in order to produce something meaningful, like for instance relational databases, report generators, GUI element sets, are usually quite big compared to the original vision of a component 10 or more years ago. In the context of this work package, components are components in this latter economic sense; it is assumed that they expose an entire set of well-defined functionalities, which can in principle be replicated by other vendors and through other technical interfaces.

Platform Some authors define platforms as basic components which one company reuses throughout an entire product line, thereby achieving economies of scale and other benefits (one also speaks of modular design, cf. [SC01], [Fun04]). Others regard platforms as independently sold technologies, such as hardware architectures and operating systems. This latter definition is more appropriate when examining the economies achieved by specialization among vendors. In the context of this work package, more specifically:

1. a platform consists at least of a specification of a software service interface (or a set thereof), and, in many cases, its unique implementation;

2. any implementation of this service interface needs to be combined with other software components in order to provide meaningful value to the typical users. When a graph with the complementarity relationships between technology is drawn, platforms should appear as central technologies with many relationships to other technologies which have comparatively less relationships;

3. the vendor or organization offering the platform follows a platform business model, in that more than 50% of the revenues from systems built on the platform are made by vendors who do not offer an implementation of the platform themselves, but rather the components using it.

[1] Source: Wikipedia "Component Software"

Ecosystem The word "ecosystem" has appeared in several contexts recently in the business literature (cf. [Gat02], [Moo97], [Rot95]), specifically regarding technology ([MS03]). As with platforms, there is no universal definition. In the context of this work, an ecosystem shall be regarded as the field of technologies and their markets related to a given platform technology by complementarity/sustainability and supply-chain relationships.

1.2 Business challenges of interoperability in the DBE project

Interoperability problems have plagued the computer industry since its very beginning. Probably the oldest examples are differences in the representation of large integers (endianness) and text. (When representing 16-bit numbers in memory, some CPUs put the lower 8 bit before the higher 8-bit, other CPUs the other way round. This creates problems when transferring data between systems or when designing peripherals for several systems. Problems in the representation of text are visible to the end-user even today: Accented letters, special punctuation marks and line feeds are commonly affected when exchanged between systems.) These examples are probably due to unintentional, historic developments — a coordination problem. In other cases, established vendors have deliberately created obstacles for interoperability to lock their customers into their technology. (Some examples will be given in section 2.3.)

Technology platforms are often expected to solve interoperability problems. The business model of platform vendors like Microsoft or Oracle is based on the expectation, by users and complementers, that their platform provides some added value to clients who run many of their applications on it; i.e. the value of the whole system increases more than linearly with the number of applications in it. Time and again, technologies like Unix, Java, or XML have been touted to integrate the hitherto fragmented IT world by their respective vendors or consortia, and aggressively marketed with high upfront investments made by the vendors to build a complementer and user base. (These investments are an indication of positive network externalities, of the expectation that the value and revenue potential of the platform will increase considerably with the number of complementers and users.)

The principle by which platforms can solve interoperability problems and thereby connect previously disconnected networks is depicted in figure 1. A well-known example is the interoperability between applications and printers introduced in the early 1990s by Adobe's Postscript and Microsoft and Apple's similar technologies in their graphical operating systems. Although physical printer connections were quite standardized even under MS-DOS or CP/M, and the ASCII character set was universally used, each printer manufacturer used different control sequences for special characters and graphics. As a result, every application under these early operating systems had to deliver its own array of printer drivers (left hand side situation in figure 1). With the introduction of, for example, Postscript, applications could talk, using only one set of instruc-

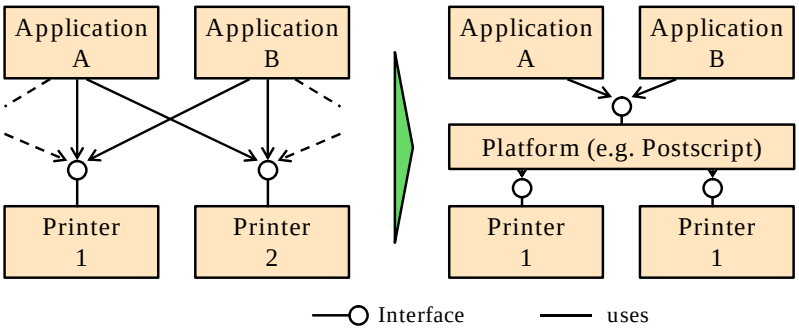

Figure 1: Utility of a printer graphics platform

tions, to all printers for which an application-independent Postscript driver was available. Interestingly, both printer manufacturers and application developers (beside the platform vendors) benefited from this economy: Lower costs for developing graphics-intensive applications meant more of them were developed, and customers' willingness to pay for printers with high-quality graphics features increased. The same story could be told with the interaction between applications and database management systems, which was facilitated by SQL [2] and, later, application interfaces such as ODBC[3].

But in component software and B2B software, the situation is different, and the track record of the technologies mentioned above is more mixed. As shown in figure 2, when trying to integrate application components into a larger application, a platform can define common interfaces for the low-level services all application components need, but it usually does not define the interfaces between the application components. As platform vendors need to achieve economies of scale, their platforms are mostly free of business domain specificity, but interfaces between application components have to contain domain semantics. This situation does not change fundamentally if application components use a common underlying communication protocol (such as SOAP, CORBA) or syntactic format (ASCII, XML) provided by the platform.

A similar observation can be made when it comes to B2B integration instead of application integration. The integration which a sector-independent B2B platform such as Oasis can provide is limited to sector-independent issues, such as billing and payment. In addition, they can provide formal description languages for sector-dependent semantics, but this does not guarantee interoperability between any two given business partners using the same language.

Therefore, a platform works best when it is enhanced by coordination between its complementers. But once again, this may be difficult due to conflicts of

[2]Structured Query Language, an ISO standard

[3]Open Database Connectivity, developed by Microsoft on the basis of the SQL call-level interface proposed by the SQL access group, and implemented on many operating systems

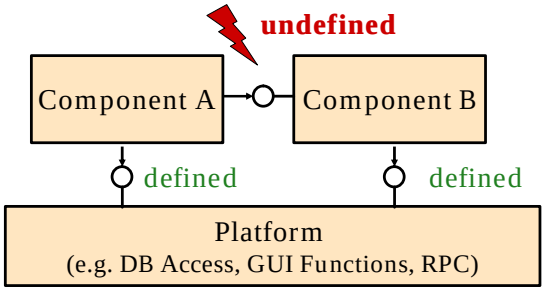

Figure 2: Platforms provide only limited application integration

interest. Vendors of application components see the world from quite a different angle than the platform vendor. What the platform vendor sells to the users as the Gospel of harmony in the software world, the component vendors may see as a thinly disguised attempt to put them at the mercy of one supplier. Also, some of the larger application vendors gain considerable competitive advantage by being able to propose a suite of well-integrated functionalities; they have therefore no interest in seeing the playing field against smaller competitors levelled by a new technology.

1.3 The role of active coordination and rules of governance

Since markets are not always able to solve interoperability problems, such as the ones just described, alone, the active engagement of the government, or publicly funded institutions, like standardization bodies, or even projects such as DBE, seems justified. The GSM mobile phone system is a positive example for such engagement[4]; it was initiated inside CEPT, the European Conference of Postal and Telecommunications Administrations, in 1982, and service startet in 1991. Today it is the dominant mobile phone system worldwide, and Europe had a lead of several years over the U.S. for a long time concerning coverage and market volume with the popular, united GSM system, while the U.S. market was fragmented between at least three different standards. So quite clearly in this case, the involvement of a government institution (in this case a supranational one) was very beneficial to both producers and consumers.

But in many other cases, centralized mediation has been less successful. In the very same year, 1982, another initiative was started inside the International Standardization Organization and International Telecommunications Union, which would subsequently be driven very much by governments (especially European ones) to integrate and supersede the proprietary computer networking technologies of that time — it was called Open Systems Interconnect, or OSI. But, while the OSI approach of dividing the protocol stack into 7 layers seemed very clean

[4]information on GSM (Global System for Mobile Communications) and OSI obtained from Wikipedia, background from [Car89]

and efficient at the time, and is still referenced today, the ambition of providing interconnectivity on all these layers was not pragmatic enough; the group's progress was slow due to the complexity of the approach, and many customers decided they could not wait. While many European organizations and corporations were encouraged to invest in the future standard, a community grown out of a military research project and start-up companies like 3Com and Cisco through a very informal, fast, almost chaotic standardization process created a well-integrated, more pragmatic solution which became the Internet, and gave the U.S. economy a big lead in networking intensity over Europe for many years. Putting all your eggs in one basket, as so often, can be a risky strategy.

It would seem counterintuitive and disturbing if centralized planning in general lead to superior results especially in the new markets of technology, as among the advantages typically associated with free market capitalism is the distribution of risk, i.e. the risk of misjudging the future needs of customers and the risk of bad planning, among many actors. The ideal pursued by the DBE project seems to be to identify some background conditions which can be actively set by the project to ensure that the decentralized market dynamics can come into full play and lead to efficient results. This is in line with the popular trend of "active deregulation" in economic policy today — a concrete example in a different area is the creation of a market for emission certificates in order to limit global warming, an intervention which uses the market to efficiently allocate available resources for emission reduction.

The DBE vision, as represented in [Nac02] and the project proposal, does not draw the line very clearly yet between areas where to adopt a "hands off", or let-it-self-optimize attitude, and where to adopt an "active optimization" attitude. On the one hand, the proverbial dynamism, swarm intelligence[5] and growth potential of SMEs is to be leveraged, on the other hand, there is the idea that SMEs need a little push and a little help to succeed. (By somebody even smarter than they?) On the one hand Open Source software, and software made of substitutable components are promoted, on the other hand there should also be some profit potential for small software companies.

But one of the innovative characteristics of the DBE project is that it does contend itself with engineering a technical solution to a business problem which is assumed to be well represented by the explicit requirements of future users, but instead tries to understand and transform the dynamics of entire networks of companies and markets. Even if one rejects the idea of a "omniscient optimizer", a digital business ecosystem will likely need some new universally offered services and/or a code of conduct which all participants live by in order to provide added value over what is available today (e.g. "the Internet"). The aspects just mentioned are part of the question of governance of digital business ecosystems, and these rules of governance should be designed consciously based on a de-

[5] swarm intelligence refers to achieving high performance on a global scale through locally optimizing agents with local, limited information

tailed understanding of their effects.

1.4 Objectives and contribution of work package 13

The *project-related objective* of work package 13 is to provide a theory of how effective the measures mentioned (active and passive coordination, rules of governance) are in fostering the growth and vitality of the ecosystem. What is the best reference discipline to build this theory upon? The ecosystem in question, one could argue, is similar to an Open Source community. These communities appear to be best understood from a social sciences point of view, starting from an understanding of the personality structure of hackers and the mutual rewards of community involvement ([Ray99a]). On the other hand, one could liken the ecosystem to a biological ecosystem; so it would be understood in terms of, for example, the exchange of certain quantities of chemicals by the metabolism of each agent, by the laws of heredity or population dynamics. In this work package, economics is considered the most adequate reference discipline. Arguments will therefore be based on the assumption that participants or agents in the ecosystem are rational and utility/profit-maximizing, and able to reason about and learn from their decision situations. The discussion concentrates on software and software-based services, taking into account the specific characteristics of software as an economic good.

The *scientific objective* is answering the research question: *How are software ecosystems successfully created and sustained?* This includes the questions:

1. *What is a platform ecosystem?* What is the anatomy of a platform ecosystem? What are typical lifecycles of platform ecosystems?
2. *What is the specific utility of platforms?* If the overall question is to be interesting, platforms must have a specific utility to at least some stakeholders.
3. *How can ecosystems be created and sustained?* What should interested market participants, including public authorities, do?

The *research strategy* consists three conceptual steps:

1. Understand the decision problems of individual actors
2. Understand the general dynamics of their strategic interaction
3. Understand the influence of background variables on target variables through the market dynamics

The research question can be asked both from the perspective of the public (how to maximize welfare) and from the perspective of the platform vendor (how to maximize profit). The DBE project plays an ambivalent role, as it has been set up with a welfare maximization objective, but is, at the same time, proponent of one particular solution which competes against other, existing offerings.

[MF00], among others, informally present the idea that vendors' success is determined by understanding and proactively influencing the network of complementarity relationships between technologies, and position oneself at the "critical" nodes (thereby becoming a platform vendor). For example, it is said that Intel and Microsoft concentrated on the "right" components within the Personal Computer architecture created by IBM, and that IBM was wrong to outsource these components. This is rather easy to see after the fact, but what are the criteria to identify the "critical" technologies in tomorrow's technology networks? How does a platform vendor identify the technologies he needs to keep proprietary and closed, the ones he can outsource to several partners, where to seek longterm partnerships, what to license for free? There is, as far as we know, no accepted framework or theory today that could systematically answer those questions.

1.5 Overview of this report

The next section presents an overview of the issues which participants in the software market face according to existing literature. The main section presents the progress made so far in modelling strategic interaction between vendors' product portfolio and licensing decisions in software component markets. Using this model as well as historical cases of platform ecosystems (such as Lotus Notes, Oracle Forms), this work package's later reports will provide answers to the research question mentioned above.

2 Decision problems in the software market

2.1 The nature of software

Why is it justified to write about decision problems in the software market? Aren't many of these general problems that occur in almost any market? This is certainly so, but on the other hand, there are many phenomena and patterns of competitive behavior which were unknown prior to the birth of the software industry — history does not tell us, for instance, of Open Source restaurants or car makers.

First of all, software falls into two larger categories of goods known to the economic literature, and represents their major intersection: Information goods and technology. Topics in information goods include rights management, and specific pricing, versioning and distribution strategies (see [SV99] for a very good overview); topics in technology include core competencies, R&D management, and technology lifecycles. Important characteristics of software as an economic good are:

- *Zero marginal costs* — making the first copy of a piece of software can cost millions, each further copy comes at practically zero cost

- *Long lifecycle and no wearing out* — unlike other technology goods, software does not wear out and can supply a given functionality for decades. However, in only a few areas software is actually used that long; in many other areas, for example in personal computing, the advances in hardware capability, and the desire for extended functionality have triggered new generations of software in the past. However, the absence of wearing out has often created a challenge for companies to generate revenue from their existing customer base.

- *Experience good* — customers have difficulty assessing the value of a piece of software before actually using it, and the utility of a piece of software is usually highly specific to an individual user. Hence the importance of brands and the practice of free trials;

- *Switching costs* — customers take time to become productive with a given piece of software - just one of many forms of costs customers incur when switching to a different product;

- *Network externalities* — software usually becomes more valuable when there are many others using it; there is better support, more complements are available, fellow users may be able to share files or do better business together; these *positive* network externalities are typical across all types of software; for specific types of software, there may be many positive *and* negative effects (for example due to viruses or piracy), which are described more accurately by differentiating between different types of users and products. See, for some examples [Shy01, chapter 3].

- *Functional complexity* — there are not many physical restrictions to what can be achieved with software, as is typical for other engineering disciplines, and the few existing physical boundaries keep being pushed back by better hardware. Together with the longevity mentioned above, this allows the construction of ever more complex "cathedrals". Economically, this complexity can be a cause of uncertainty, or high cost of finding out about the software's characteristics.

Software as Product or Service and instantiation levels A notorious question is: Is Software a product or a service? Until the early 1970s, the most common answer would have been "a service" (delivered with the hardware product). Then came the era of software in shrink-wrapped boxes; software as a product. Recently, organizations find it less and less necessary to run software in their own data centers; software is a "managed service" delivered "on demand". We have difficulty finding a final answer to this question because of the strange immaterial nature of software. One can differentiate between at least three levels of instantiation:

1. the unique master copy of the source code, with the artefacts that engineers work on;
2. a copy of the compiled executable, e.g. on a CD or on a hard drive;
3. the running instance on a certain computer.

There are one-to-many relationships between these levels: Many copies of the executable exist for each master copy of the source code; many running instances can be started from the same executable file; and even when there is only one running instance of the entire executable, on the inside, there are usually many run-time objects instantiated on top of one set of binary instructions in the executable (class methods in object-oriented programming).

For ordinary industry products, one could say there are similar levels: Blue prints for the product and the manufacturing process developed by the engineers; the instantiated manufacturing process / assembly line (there can be several around the globe); and the individual product coming out of this process. For services, for example an airport or a hospital, there is the specification of how to perform security checks or, respectively, childbirths with a given technical equipment; the airport / hospital building and the staff trained according to these specifications; and the individual passenger journey or patient case being handled. The big difference is the cost structure: For software, almost the entire cost is development cost. And in industry, we always buy and sell the products or pre-products — nobody would confuse the mobile phone with the assembly line for the mobile phone. In the software industry, we typically buy and sell compiled executables (similar to the assembly lines or airports), but sometimes there are several choices. It is very common to see these levels mixed up and metaphors from industry being used in strange ways (e.g. speaking of "software factories" as if software development were a largely repetitive process).

Technical relationships between units of software The relevant technical relationships between pieces of software depend on the instantiation level. On the source code level, most programming languages have something like modules, or packages. A package comprises definitions and implementations. Between packages, the standard relationship is "uses" (this is also reflected in the Unified Modelling Language used extensively in the DBE project). Package A uses Package B if the symbols defined by B are visible to A. One can refine this relationship by differentiating between the definitions inside the package and the implementation. If B is only referenced inside the implementation of A, then other packages can access A without knowing about B (but the environment has to ensure that whenever a package wants to access A, B's implementation is available). Otherwise any package C wanting to use A also needs to import B. A could also reference only B's definitions (e.g. by defining a class that implements an interface defined by B), in which case A will not need B on the executable level.

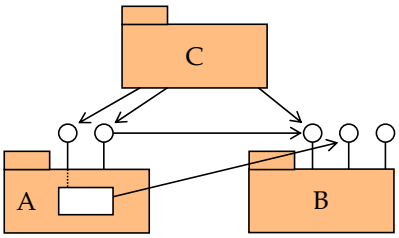

Figure 3: Example relationships on the source code (static) level

On the executable level, several packages usually get compiled together into main executables ("EXE") or dynamic libraries ("DLL"), and there is also a "uses" relationship here. The common case is that, when EXE1 uses DLL1, then the packages contained in EXE1, on a source code level, use the packages contained in DLL1. But this is not necessarily the case. EXE1 might only share a common interface definition with DLL1. EXE1 might have discovered DLL1 through a service discovery mechanism. Finally, on the level of running instances, an instance can communicate in a symmetric or asymmetric fashion with other instances. A common asymmetric way of communication is clientship (procedure call). When instance a makes a procedure call to instance b, then a is called the client and b the server (the same terms are used for the computers they are running on if they run on different computers). This normally means that a's implementation is described by some package A which has visibility (uses) a package B describing a part of b's implementation. But again, there are other cases, when, via a common interface, the instance of a package B used by A calls an instance of A (callback). Instances can also communicate in a symmetric way with one another, where each party can initiate communication.

2.2 Characterisitics of software markets

Critical mass For goods which derive a major part of their utility from the network of users, like telephones, faxes, or B2B software, it is quite difficult to find early users. It is even more difficult for software with its zero marginal cost, which means serving these few early users already costs a good part of what it costs to serve the whole market. But once a critical mass of early users is reached, demand often explodes.

Tippiness When there are several competing incompatible technologies with comparable functionality, once one technology leads the others in market share, it becomes very hard for them to catch up, and the market tends to tip towards one winner. The effect is enforced (not created!) by the fact that users tend to not only evaluate a technology based on the current number of users, but also based on the future number of users, and so if they expect the market to tip, it will all

the more probably. One consequence of this is that vendors engage in fierce *price wars* in early phases to capture as much market share as possible. Tippiness also enforces the critical mass effect, because potential early users are very wary of getting stuck with an unpopular technology.

Effects of standardization Users thus tend to welcome standardization which reduces this risk and ensures that the network will comprise all users in the market. For vendors, this can also be beneficial since the total market potential can be multiplied by standards in many cases. One example for this is the market for mobile phone software and services, which has, in Europe, been plagued by fragmentation and overprotection by the operators (in stark contrast to the *voice* service on GSM mentioned earlier), while in Japan, the fact that there is one application interface all phone manufactures and operators adhere to – i-mode – this industry has flourished and operators have not been poorer for it. The general detrimental effects of standardization according to [SV99] are, for users, a reduction of variety and an end of price wars; for vendors, they level the playing field, which can be a much less comfortable situation, most of all for big vendors. For software, again because of zero marginal cost and because of its immaterial nature, standardization can destroy a market completely. For physical goods, there are capacity restraints, restraints in geographic distribution etc. which allow vendors in an oligopoly to make some profit. But for a software product category which is completely defined in all its aspects, even in a market with only two vendors the competition is disastrous. Competition can only work if there are some other aspects in which vendors can differentiate themselves. The functionality and access method for Relational Database Management Systems (RDMS), for example, has been standardized to a large extent by SQL, and many of the former mid-range RDMS are given away for free today[6]. There are commercial vendors left thanks to the physical limitations of size and performance which RDMS all deal with in a different way, and which are important characteristics in the view of professional users.

[MP88] have developed a model of the effect of a systems market, where users can mix & match components from different vendors instead of buying entire systems. (This mix & match also necessitates an interoperability standard). The interesting result is that under certain conditions, the users whose needs correspond to components of the same vendor are worse off because the individual components are sold more expensively than entire systems would be sold.

2.3 Decision problems of software vendors

Pricing Network effects and its resulting tippiness lead to penetration pricing. Vendors forego current profits for future profits. This is quite a clear case if there is only one company with a new product which needs to establish itself vice versa

[6]For ex. the former Adabas D which is now freely distributed as Max DB

an old product with a large installed base. If there are two or more competitors with new products, a price war can be expected. Price wars are hard to get out of, even if a vendor knows it cannot win because the other vendors have more capital resources. In real cases, there are usually other factors beside price which influence buyers' decisions.

Price Differentiation, Customer Segmentation & Versioning Software being an experience good leads to the practice of free trials. Free trials can also be seen as a form of price differentiation. Price differentiation means that the same, or slightly differing versions of the same software product are sold for widely differing prices. The reason for this is, of course, that different customer groups generate very different amounts of value from the same package, and thus have widely differing price elasticities. The practice of price differentiation is known from other sectors, but the effect in the software market is stronger due to the zero marginal costs of production, which means that making additional sales even for a very low price still increase profit, as long as one can effectively shield against cannibalization.

A related practice is *bundling*. The best-known example is the introduction of the Microsoft Office bundle, including Word, Excel and PowerPoint, programs that had previously been sold separately. The underlying assumption is that buyers' valuations for the bundle vary less (relatively) than their valuations for individual products in the bundle. This assumption does not include a complementarity between the products in the bundle. Although in the Office case, an obvious selling argument was that products in the Office suite work seamlessly with each other, while it would be more difficult to exchange data with non-Microsoft Office programs. (This point could be more deeply examined.)

These practices also effectively close possible niches for competitors. For example, in the 80ies, when the industry had already standardized on MS-DOS for PC operating systems, Apple (and later: NeXT) provided very cheap licenses for Universities for their products, expecting that 1. Universities would be more responsive to low prices, 2. they would be more willing to deviate from the industry standard, 3. they, being a relatively closed network, it would be easier to reach critical mass within it, 4. this established, university students would be more willing to invest in Apple technology later, when working for the industry after their graduation. In the 1990s, Microsoft effectively closed this niche by providing their software for a fraction of the price to both academic institutions and students.

Openness of Intellectual Property Traditionally, high technology vendors have adopted a "protect whatever you can" strategy with respect to intellectual property. But in the software industry, there are famous examples of "open" technologies which have been very profitable for their inventors, due to their attractiveness for users and complementers. Software vendors therefore, like in

other areas, need to trade off between "enlarging the pie" and "ensuring one's share of the pie". Software companies have been very innovative in recent years in finding new ways to divide their intellectual property into open and closed parts. See for example the Open Source business models presented in [Ray99b]. Assets which may be desirable to protect include:

- Design ideas / Engineering innovations (by patents)
- Source code (by copyright & by keeping it secret)
- Look & Feel, menu structures etc.
- Interoperability specifications
- Support information

Conformance to Standards or Third-Party APIs, Dependence on Third-Party components On the other hand, one has to decide to what degree intellectual property of other vendors are integrated into one's product. The advantage of this is usually access to that vendor's user base. Building one's product on components of another vendor will require a reselling license and may put new features of one's product at the mercy of the other vendor's development roadmap. Adopting a standard usually increases competition on price and non-standardized performance features. As mentioned above, if a standard more or less completely defines a product's functionality, it can destroy the market for that product. Sometimes it might therefore even be necessary to invest into making the product *less* interoperable: Manufacturers of ink jet printers have practiced this for quite a while, making it as difficult as possible to use ink cartridges from other vendors. For instant messaging software and online music shops, there have been cases of active investments into interface obfuscation[7], albeit not to defend a margin on the software itself.

Vertical and horizontal scope of activity; product & component portfolio Before all the questions above become relevant, a vendor has to decide what markets to engage in, what belongs to his[8] business and what does not. Regarding the vertical focus, a basic result from industrial organization is that, in a vertical value chain, it does not pay a producer of a non-commodity good to integrate the upstream or downstream production of a commodity good. Rather, if he is at present producing such a good, he should divest from this production, and

[7] Apple's iTunes music store changed the (secret) client-server interface after it had been reverse engineered by the Open Source project PyMusique in order to offer an alternative iTunes client for Linux. Multi-protocol instant messaging clients like Trillian are regularly locked out by networks like Yahoo by slight changes in the protocol.

[8] "He" usually refers to abstract actors, players etc. in this report — usually they are corporations.

source it from the free market. On the other hand, if the upstream or downstream product (or service) is not a commodity, then both producers and consumers benefit from integration. This effect is limited by some restrictions. The original argument assumes that the two vendors are exclusively dealing with each other along the supply chain, which is hardly realistic. In Business Administration, it is generally not recommended to merge two companies with completely different characteristics, as this increases management complexity without the benefits of economies of scale which are attained when merging two similar companies. In practice, one can observe that powerful buyers (e.g. automobile manufacturers) who have to deal with powerful suppliers can try to build up new suppliers, so that the supplied product becomes more commodity-like, instead of integrating with these suppliers.

Both arguments can be applied to complementary products. I.e. when buyers (e.g. consumers) buy a vendors non-commodity product together with other vendors commodity products, the non-commodity vendor already makes all the profit he could make even if he integrated with the other vendors.

2.4 Decision problems of users

Estimating use value Assessing the economic benefits of a piece of software can be challenging for users because of its *experience good* nature and the network externalities.

Timing of sourcing decisions Being among the early adopters of a software product with network effects is riskier than waiting for widespread adoption. However, one also foregoes the benefits of the software during the waiting period, and one's sourcing decision will influence other potential users, especially one's suppliers and customers. A powerful user with high visibility with the other potential users (for example a big automotive manufacturer with respect to the automotive industry as a whole), by adopting early, can hope to establish his preferred technology as an industry standard, and can usually obtain large discounts.

Avoiding lock-in Most smaller user companies benefit from using only one technology for a given purpose throughout the company, in order to save costs for integration and training. But very large corporations can be observed to foster some degree of variety in order to minimize the risk of being left out of some vendor's innovation, and to improve their position in negotiating with vendors.

User-driven standardization activities As users generally benefit more from standardization than technology vendors, it would seem natural for them to engage in it themselves. But on the other hand, one's partners in a standardization

attempt are one's competitors in normal life; and standardization efforts often demand bringing in talent and expertise which one is eager to protect.

3 A method for the analysis of software component markets

3.1 Definition of the problem

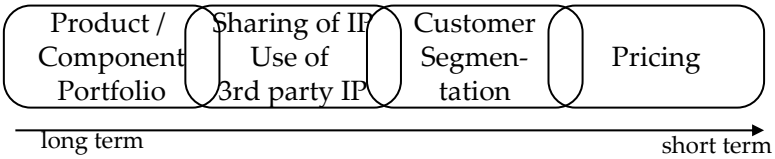

Figure 4: Rough chronological order of vendors' decision problems

Figure 4 shows the vendors' decision problems identified in the last section in rough chronological order, from long-term to short-term decisions. This should not be understood as "a vendor first chooses his product portfolio, then worries about intellectual property", but rather, a product portfolio is a long term engagement, hard to change once it has been chosen; in choosing it, a vendor certainly already thinks about the prices he can later obtain for his products etc. This section examines the interference between the first two stages — the development portfolio decisions and the licensing decisions — and the strategic interaction between a limited number of vendors in a component industry. The defining feature of a component industry is that each vendor does not design/produce the entire product by itself, but combines third-party components with its own, specific components into a finished product. (As mentioned in the introduction, we are not merely concerned with a *modular* approach, also known as product-line approach.)

In reality, most development decisions are not taken at any single point in time; rather, development roadmaps are sketched and then refined according to reactions by customers/licensees and competitive offerings. Rather than tackling a formal treatment of this dynamic process directly, we start by formalizing the problem in a static way. As a second simplification, we consider only licenses for a fixed price, not per unit resold by the licensee, for example. This form of licensing is common practice in markets with a limited number of vendors, where per-unit-licensing easily creates a double marginalization problem [Spe50].

Definition 1 *A component based software market is defined by*

- *a set of vendors* $N = \{1, 2, ..., n\}$,
- *a set of viable product categories* $\mathcal{P} = \{a, b, ...\}$

- *a set of viable sub-product-level component functionalities* $\mathcal{C} = \{\mathrm{v}, \mathrm{w}, ...\}$
- *a development technology, in input requirements set notation*

$$T : N \times 2^{\mathcal{P} \cup \mathcal{C}} \rightarrow 2^{2^{\mathcal{C}} \times \mathbb{R}_0^+}, \textit{ where}$$

$$T_1(\{\mathrm{a}, \mathrm{v}\}) = \{(\{\mathrm{x}\}, 50), (\{\mathrm{x}, \mathrm{y}\}, 20)\}$$

denotes that in order to be able to offer a product of category a *and a component of functionality* v*, vendor 1 has to either have access to an implementation of* x *and invest 50 (of whatever monetary unit) into development, or has to have access to* x *and* y *and invest 20.*

- *a revenue function* $r : N^{\mathcal{P}} \rightarrow (\mathbb{R}_0^+)^{\mathcal{P}}$*, where, in*

$$r(n_1, n_2, \ldots) = (r_1, r_2, \ldots),$$

r_i *denotes the revenue each vendor of product category i can make, given that there are* n_1 *vendors for product category 1,* n_2 *vendors for product category 2 etc.*

Note that T above is only defined in terms of functionalities accessible and provided; it does not specifically refer to developed components as exposing and requiring interfaces, and requiring resources for their implementation.

Even though T formally resembles a production technology from classical economic theory, there are important differences: There is no notion of production volume — rather, one either has access to a functionality or has not. Inputs are not consumed by the development process — a component can be reused again and again in any product without additional cost. Some common-sense properties of T are:

1. $(\emptyset, 0) \notin T(S), S \supset \emptyset$: No free lunch
2. $T(\emptyset) = \{(\emptyset, 0)\}$: Possibility of inaction
3. $(R_1, m_1) \in T(S_1) \wedge (R_2, m_2) \in T(S_2) \Rightarrow (R_1 \cup R_2, m) \in T(S_1 \cup S_2)$ where $m \leq m_1 + m_2$: Additivity of component requirements, subadditivity of development costs

The revenue function defined as above abstracts from sales volumes, network effects and details of the pricing mechanism. All these factors can be accounted for in constructing the revenue function, but the further discussion does not depend on the way it was created. Note that the revenue from any product category i can depend on the number of vendors for *all* other product categories — allowing for complementarity and substitutivity between product categories.

Definition 2 *A* license deal *is a tuple* (i, j, k)*, with* $i \in N$ *the licensor (seller),* $j \in N$ *the licensee (buyer) and* $k \in \mathcal{C}$ *the component of category k produced by i. The licensor sells his license for a fixed fee.*

For a market as defined above, we look for plausible development and licensing strategies for each player, and plausible resulting outcomes of the game. The payoff of each vendor is his product revenues plus licenses fees received minus license fees paid.

3.2 Classification of game types

In order to turn the above problem definition into a formal game, several additional assumptions have to be made which divide the game space into several classes:

Relative timing of development and licensing decisions. Development and licensing decisions can be made together or in sequence. There are good arguments both for the "develop, then license" and for the "license, then develop" assumption. On the one hand, before engaging in development of a product, a vendor will likely try to secure licenses for required components. On the other hand, it is unlikely that a vendor will license a component which has not been developed yet. The question is not so much the physical transaction, but the time at which the payoff effect of a decision becomes irreversible compared to the time when the decisions of other vendors are known.

Enforceability of contracts. Mainstream, non-cooperative game theory assumes absence of any authority (state, court) which can help enforce agreements. The frequency of litigations between clients and providers of made-to-order software, and between licensors and licensees of software technologies suggests that contracts and authorities enforcing them do have an important role to play in the software market. On the other hand, there may be important limits to this:

Extent of contracts. In general, not all agreements can be enforced; some agreements constitute anti-competitive behavior and are therefore illegal. For example, bilateral agreements between vendor 1 and vendor 2 may not be allowed to contain clauses restricting their interaction with vendor 3. Where agreements are illegal, the parties can sometimes be observed to engage in *tacit collusion*, where a similar anti-competitive behavior arises as the stable result of a non-cooperative game.

Multilaterality and timing of contracts. A related issue is the question of contracts (provided they exist) being bilateral or multilateral. Especially in the case of bilateral contracts, there is the question of timing between the agreements of one vendor, since it is unrealistic to assume that a vendor negotiates two contracts simultaneously and independently, i.e. the definitive outcome of neither contract is known when making one's move on the other.

Symmetry. Vendors can be assumed to be identical in their technological capabilities and initial set of accessible components, leading to a symmetric game. Symmetric games allow for some optimizations in the solution algorithm [CRVW04]. Although real-world cases seldom involve vendors with equal characteristics, the symmetric case can be an interesting reference case to gauge the effect of the differences between them.

Completeness of information. In the case of asymmetric games, a vendor may or may not have complete information about the technologies and initial components available to the other vendors.

Interoperability costs Even if we assume that there is a fixed set of functionalities, and that the products and components offered by different vendors are indistinguishable with respect to customers' preferences, their APIs may be different — they *will* be different if they are not explicitly coordinated due to the complexity of the design process. Likewise, a vendor makes assumptions about the third-party APIs he integrates with, or sometimes uses an integration framework which allows linking his software to a variety of third-party APIs. Obviously, there is a cost associated with this integration, and with changing these decisions late in the development process.

Redistribution of licensed components In the typical case examined here, component vendors charge an overall license fee and then allow free redistribution of their components with any number of products. But several assumptions are possible regarding whether the redistributed component is still "visible" and can be used without the product in came with.

3.3 Modelling contracts

In summary, the question how contracts are modelled adds a lot of complexity to the game space. Game theory gives us several alternative tools to deal with contracts:

In non-cooperative game theory contracts are not part of the "language" itself and have to be modelled using other elements. One possibility is to micromodel the *bargaining process* as a dynamic game with consecutive steps of offering and, eventually accepting offers. The "enforcement" happens through the design of the game tree, where depending on the moves in the bargaining steps, different options are left for action and thus different sets of payoffs. Or, the enforcing agency is modelled as an additional player, as in some games of *mechanism design*[9]. Both techniques in general multiply the complexity of any given game.

[9] see [FT91, chapter 7] for an introduction

Also, it can be quite difficult to deal with continuous action spaces, as in setting prices.

There is also the weaker notion of *correlated equilibrium*[10], which can be used to consider games with prior communication between players and common observation of a random event, but without enforceable contracts. This is interesting because in their production decisions, vendors often face a "battle of the sexes"-type situation: Vendors gain from choosing different sets of components to develop — no matter which — and lose if they choose the same set of components -.. no matter which; but the symmetry of their situation may lead them to the latter. They can avoid this situation either using "communication & correlation" or using enforceable contracts.

In cooperative game theory, payoffs depend on combinations of players forming coalitions, not on individual moves.[11] The game is defined by the set of players N and the *characteristic function* $v(S)$ with $S \subseteq N$ which gives the utility a coalition S can jointly achieve. The solution one looks for is not a plausible set of strategies for each player, but a plausible or fair distribution between players of the coalitional gain. (More precisely, this applies to games with transferable utility, such as the given problem, where vendors can negotiate payments for licenses.) Several concepts exist for what constitutes a "fair" distribution, but generally two conditions must be satisfied: It must be efficient (no payoff is lost) and it must be stable in that neither individual or coalition is able to easily reach better distributions by deviating. These criteria most often assume that contracts are multilateral, although one can sometimes reach the same result by considering bilateral negotiations [Ste69]. The criteria differ notably in what constitutes an effective deviating coalition. For example, the *core* of a coalitional game is defined to be robust to any deviating coalition, regardless of whether that coalition is robust itself.

This identification of relevant alternatives is also a key question for the given problem. The range of a license fee which is both acceptable to the licensor and the licensee clearly depends on the difference between what payoffs they can achieve with the license compared to the payoffs they can achieve without the license. For example, if the licensee can make some additional product k once he acquires the license, and the licensor also makes this product, then the licensor will lose some revenue; so the minimum price the licensor will charge is greater than 0. (Remember that, in the software market, there are virtually no production, but only development costs.) But this is seldom a *ceteris paribus* comparison, since the licensee usually wants the license because of an investment in a component or product which uses the licensed component. Therefore, if turned down by one licensor, he will either try to license the component from a different vendor (who may be more than ready to provide it), or he will try to revise his development

[10] cf. [Mye91, chapter 6]

[11] See [Mye91, chapter 9] for an introduction.

portfolio — in case this is still possible. If it is not possible to revise (this depends on the timing assumption), he will usually at least regret it, which means we are not dealing with an equilibrium alternative either. But in order to evaluate these more complicated, "real" alternatives, one has to again know about the likely license fee ranges under the alternative conditions, so the alternatives of the alternatives have to be considered, and so on.

3.4 A set of consistent assumptions

We shall consider the following set of consistent assumptions, which will allow relating the problem to existing game theoretic solution concepts, and extend them:

Assumption 1 *First, each vendor decides (non-cooperatively) on his production portfolio. Then, vendors negotiate (cooperatively) about licensing conditions.*

The association of the production decision with a non-cooperative game, and of negotiation with a cooperative game is quite natural. A vendor does not strictly "need" other vendors in order to make his production decision, and vendors cannot directly observe what goes on in another vendor's lab. The production decision is a long-term decision, a license is a deal which can be made quickly. The only other real alternative within the framework of a static game would be a complete coordination between vendors both on production and on licensing, which would mean they act as one vendor. (The only remaining question would be how assymmetry in technologies and initial assets affect final payoffs.) If vendors negotiate licensing *first*, then this automatically also involves commitments on their production plan, so it is only slightly different from the simultaneous, complete coordination case.

The resulting game is, in principle, what [BS04] call a *biform game*. Biform games are defined as two-stage games with a first, non-cooperative stage and a second, cooperative stage. They can often be applied when actors try to make long-run strategic choices to shape the short-run competitive environment, an idea often informally called "changing the game" in the business strategy literature. The game has to be solved backwards, i.e. one first solves, for each possible pure-strategy profile of the first stage, the corresponding cooperative second stage subgame, then uses the expected payoffs to solve the first stage. One could say that, in fact, the license negotiations are the first step after all — namely the ones taking place virtually, in the minds of the product marketing managers. These "daydream" negotiations are all the more realistic and informative because of the following assumption:

Assumption 2 *All vendors have complete information, i.e. each vendor knows all vendors' available technologies, costs, etc. They also know that the other vendors know all of this, and that all vendors know that all vendors know all of this, and so on. Of course, nobody directly knows the other vendors' strategies beforehand.*

But, the cooperative licensing game does not present itself directly in a solvable form. In cooperative game theory with transferable utility, payoffs are formalized by the characteristic function $v(S)$ with $S \subseteq N$; i.e. the total payoff of a coalition is only determined by the set of players in the coalition. But in the licensing game, with production decisions given, payoffs are determined by the set of bilateral licensing agreements. A given set of players, e.g. 1, 2 and 3, can become linked together by licenses in different configurations. For example, player 1 and 3 can have a direct cooperation link or only be both linked to player 2. And the payoffs are not only determined by the configuration of linked players, but by the individual licenses; player 1 and player 2 in general have several deals they can make, and payoffs will be different depending or which are. Likewise, all payoff transfers are originally bilateral (which in itself does not exclude any particular payoff distributions however).

The characteristic function does not capture the effect that a license deal between player 1 and player 2 has on the payoff of player 3 who does not have a link with either player 1 or player 2. [Mye91, section 9.2] describes several techniques (representations) by which a general n-player static game (which *can* represent this effect) can be converted into the characteristic function representation. But all these techniques assume, in order to calculate $v(S)$, that there is a complementary coalition $N \setminus S$ in place, acting together in various degrees of offensiveness towards S, rather than individual players or several smaller coalitions. This seems to be quite a strong assumption, and it is not needed as shall be shown. Perhaps it can be made in games where the space of available actions that payoffs depend on is independent of the coalition structure. In our case, components cannot be licensed without negotiations (i.e. coalitional links). However, the techniques, such as Neumann & Morgenstern's minimax rule ([vNM44]) *are* needed later in order to calculate payoffs in the case where several competing coalitions are given. This leads us to

Definition 3 *The licensing subgame pre-transfer payoff function is a function*

$$u^q : N \times \mathcal{L}^q \rightarrow \mathbb{R}_0^+$$

where $u_i^q(L)$ denotes the pre-transfer payoff for player i where q is the production profile*, i.e. the profile of production plans (strategies) for each player which were determined during the first stage of the game, and L is the set of licenses dealt (see Definition 2 above), with $\mathcal{L}^q$ the set of possible license deals under the given production profile.*

The above reasoning means that we do not assume that u_i^q only depends on the licenses $\{(j_1, j_2, k) | (j_1 = i) \vee (j_2 = i)\}$, i.e. involving player i. The production profile will be formalized below.

What we would like to get to is a modified characteristic function

$$v_S^q(C), \text{with } C \subseteq L_2(N) = \left\{\{i, j\} | i \in N, j \in N, i \neq j\right\}$$

which gives us the joint payoff a coalition $S \subseteq N$ can expect to achieve in the coalition structure C (denoted by the set of bilateral *cooperations*) under the production profile q. This function contains all the necessary information about the bargaining power of each vendor without all the details about individual licenses. The only necessary assumption is

Assumption 3 *In bilateral negotiations between any two vendors, they discuss all their possible license deals together (at zero negotiation cost), and their final agreement covers their entire cooperation.*

To see this, and the way v can be constructed from u, consider a game with only two vendors first. The two vendors, under a given production profile q, have two possible license deals $\mathcal{L} = \{l_1, l_2\}$, with $l_1 = (1, 2, \mathrm{v}), l_2 = (2, 1, \mathrm{w})$. Consider first case a, with vendors being collectively best off when making both deals, and worst off when making no deal at all ($u_N^q(\{l_1, l_2\}) > u_N^q(\{l_1\}) \geq u_N^q(\{l_2\}) > u_N^q(\emptyset)$). Which deal set will the vendors agree on, and what is the possible range of total license payments agreed to?

[bhtp]

(u_1, u_2)	not l_1	l_1
not l_2	$(0, 0)$	$(-1, 4)$
l_2	$(2, 0)$	$(2, 2)$

Table 1: two-vendor case a

[bhtp]

(u_1, u_2)	not l_1	l_1
not l_2	$(0, 0)$	$(-1, 4)$
l_2	$(2, 0)$	$(1, 1)$

Table 2: two-vendor case b

Imagine that one of the vendors proposes a non-optimal agreement; for example, vendor 2 proposes $\{l_1\}$ with a payment of 1.5 to vendor 1. This would lead to an effective payoff of 0.5 for vendor 1 and 2.5 for vendor 2. Will vendor 1 accept this offer? After all, it is more attractive than no agreement at all. But in fact, vendor 1 could make a counter-offer of $\{l_1, l_2\}$ with a payment of 1 to vendor 2 which would be better for both, leading to a payoff of 3 for vendor 1 and 1 for vendor 2. It is easy to see that this is true for any non-optimal agreement which gives both vendors a payoff above the no-agreement case; the vendor answering the proposal can always propose the jointly optimal agreement and divide the joint gain between the two. So in case b, $\{l_1\}$ is certain to be the result of negotiations. And since both vendors are rational and know this, the pre-transfer payoffs of all other license deal sets should not matter when determining the range of possible license payments. All that matters are the payoffs in the no-agreement case. Therefore, in case a, the range of feasible payments from vendor 2 to vendor 1 is $(-2, 2)$ (where a -1 denotes a payment of 1 from vendor 1 to vendor 2). In case b, it is (1,4).

We lose no strategic content, then, in representing case a as

$$v_S^q(C) = \begin{cases} 0 & \text{if } C = \emptyset \\ 4 & \text{if } C = \{\{1,2\}\} \end{cases}$$

and case b as

$$v_S^q(C) = \begin{cases} 0 & \text{if } C = \emptyset \\ 3 & \text{if } C = \{\{1,2\}\}. \end{cases}$$

The ranges of possible effective (post-transfer) payoffs to both vendors are completely defined by the modified characteristic function. We only need the individual pre-transfer payoffs in u if we want to calculate the actual transfer payment. If there are several optimal license deal sets, then they will all result in the same ranges of effective post-transfer payoffs, so from a strategic point of view they are all equivalent. However, the transfer payment may be different for each deal set.

This situation changes a little bit when we look at 3-player games. Consider for example the situation in Table 3. Vendor 2 has two components to license to vendor 1, and one component to license to vendor 3: $\mathcal{L}^q = \{(2,1,\mathrm{v}),(2,1,\mathrm{w}),(2,3,\mathrm{x})\} = \{l_{11}, l_{12}, l_3\}$. We will denote

$$\mathcal{L}_S^q = \{(i,j,k) | (i,j,k) \in \mathcal{L}^q \wedge i \in S \wedge j \in S\}$$

the license deal set available to a set of players S (so $\mathcal{L}^q$ is in fact shorthand for $\mathcal{L}_N^q$).

		$L \cap \mathcal{L}^q_{\{1,2\}}$			
	(u_1,u_2,u_3)	$\emptyset$	$\{l_{11}\}$	$\{l_{12}\}$	$\{l_{11},l_{12}\}$
$L \cap \mathcal{L}^q_{\{2,3\}}$	$\emptyset$	$(0,0,0)$	$(1,0,3)$	$(2,0,0)$	$(2,0,-1)$
	$\{l_3\}$	$(-1,0,1)$	$(-1,0,4)$	$(0,0,1)$	$(0,0,0)$

Table 3: three-vendor case

Note that in this example, vendor 2 always has a pre-transfer payoff of 0, which makes the discussion a little bit easier. First, consider the situation where vendor 2 only cooperates with vendor 1. There are two optimal deal sets for them: $\{l_{12}\}$ and $\{l_{11}, l_{12}\}$. But, unlike in the two-vendor case, these two are not strategically equivalent! Vendor 3 gets 0 (pre- and post-transfer) payoff in case of one, and -1 in case of the other. Which one is the "valid" one, i.e. what value should we assign to $v^q_{\{3\}}(\{\{1,2\}\})$? The "traditional" solution, and the one that makes the most sense for the way we shall use v, is to assume that the coalition $\{1,2\}$ will aggressively minimize 3's payoff (especially because it does not cost them anything). This will make sense because $v^q_{\{3\}}(\{\{1,2\}\})$ will represent the baseline payoff to vendor 3 that vendor 2 will use as an argument in negotiations with vendor 3. The lower it is, the more advantageous the result of these negotiations for vendor 2. Some of the approaches presented in [Mye91, section 9.2],

for example the minimax rule, go even further to assume that a vendor cannot even rely on the other vendors only considering the moves that maximize their payoff, but must expect them to use even more aggressive moves towards him (in the example above, there are none).

This leads us to the case where vendor 2 cooperates with both vendor 1 and vendor 3. If coalitions $\{1,2\}$ and $\{2,3\}$ were distinct players, there would be a prisoner's dilemma situation between them: $L = \{l_{11}, l_3\}$ would be the jointly optimal solution with a total payoff of 4, but the moves necessary to produce it are strictly dominated, so that the likely outcome would be $L = \{l_{11}, l_{12}, l_3\}$ with zero total payoff. But of course, vendor 2 will not be aggressive against himself. By maximizing the total payoff, he can hope to maximize his own, since if ever one of his partners is worse off in the maximizing outcome than what he can realistically hope to achieve without vendor 2's "coordination work", the gain in payoff of the other two will exceed this loss, so vendor 2 can "bribe" him while not losing his other partner, and still be better off. A dilemma can only occur in the case of a fixed negotiation order: Let's say vendor 2 successfully negotiated licensing l_{11}, not l_{12} to vendor 1, hoping to gain from maximizing the pie, and foregoing a payment of up to 1 from vendor 1 for licensing l_{12} as well. Once this agreement made, all that is left to determine is the joint payoff of 2,3, and 3 can bribe 2 using any payment between 0 and 1 to license l_3, so vendor 1 will lose 2. This result is dependent on the order chosen. It is realistic to assume that under these conditions, vendor 1 will talk to vendor 2 about l_3 as well and make vendor 2's behavior a condition of their agreement. But obviously vendor 2 cannot conclude an agreement dictating his behavior towards vendor 3 before negotiating with him — this would undermine his negotiating position. A more complicated mechanism is needed. But the defining characteristic of cooperative game theory is not to care about the exact mechanism, but rather reason about the feasible payoff distributions based on different coalition structures. All we need to know for the moment is that there is some "efficient" mechanism which will ensure that the maximum total payoff will be reached. In more general terms:

Definition 4 *Let S/C denote a partition of the set of vendors S into the sets of vendors connected directly or indirectly by the cooperation relationships in C.*[12] *The elements of $S/C = \{T_1, T_2, \dots\}$ are called coalitions resulting from C.*

Assumption 4 *For any given cooperation structure C, every coalition in N/C will effectively maximize its total payoff within the limits of that structure.*

One result of this assumption is that the modified characteristic function v is superadditive, i.e. by establishing a cooperation link, the participating coalition(s) are never collectively worse off:

$$v_S^q(C) \geq v_S^q(C \setminus \{i,j\}), \text{for } S \in N/C, i \in S, j \in S \tag{1}$$

[12] This notation is also introduced in [Mye91, section 9.2, p.446]

Before moving on to the specific issues of even larger games, let's see how we can solve the coalitional game defined by v for likely payoff distributions between the vendors. Solution concepts for cooperative game include:

1. the core
2. the stable set (introduced in [vNM44])
3. the Shapley value ([Sha53]
4. the bargaining set ([AM64])
5. the kernel ([DM65])
6. the nucleolus ([Sch69]).

(All of these are described in [Mye91].) [BS04] suggests the use of the core for biform games, but mentions that other solution concepts can be used and have been used in specific cases — without providing a general treatment of biform games. For example [HM90] uses the Shapley value.

The core of a coalitional game defined by the characteristic function $v(S)$ is defined as the set of payoff distributions x which satisfy

$$\sum_{i \in N} x_i = v(N) \qquad \textit{efficiency}, \text{ and} \tag{2}$$

$$\sum_{i \in S} x_i \geq v(S), \forall S \subseteq N \qquad \textit{robustness.} \tag{3}$$

For many games, the core does not give a unique solution, but either no solution at all or a whole range of payoff distributions. This is quite difficult to treat in biform games. [BS04] introduce so-called confidence indices which indicate, in the case of a range of payoffs in the core, whether a player tends to be rather optimistic or rather pessimistic about the payoff he will be able to capture within that range. But the question of whether these confidence indices are independent of the first stage outcome, and if they have to be consistent, creates whole new dimensions of complexity. By consistency we mean the question whether we can assume that, for example, all players are optimistic. According to [BS04], this is okay, because the confidence indices represent subjective views. But it seems to us that this question is closely linked with the solution concept used in the first stage game and its interpretation. For example, the Nash equilibrium is often interpreted as the result of a learning process, or a process of evolution (cf. [FT91, section 1.2.5]), so it seems that in this case this learning or evolution should also apply to the confidence indices, and that in equilibrium, players should not be optimistic or pessimistic, but realistic regarding their negotiation skills. (It is of course not a necessity to use Nash equilibria as the first stage solution concept, or to interpret Nash equilibria in this way.) An additional difficulty is the possibility of empty cores, for which [BS04] provide no solution. Also, we have experienced

difficulty extending the core concept to games with a cooperation structure, as given by the modified characteristic function.

Luckily, a modified version of the Shapley value has been proposed which has almost all the properties we need. The Shapley value can be interpreted in two ways: 1. As a weighted average of the payoffs resulting from all possible sequences of coalition forming, starting from isolated players; 2. as resulting from a set of axioms of desirable properties of a payoff allocation (for example, that the numbering of players should not matter for the result). The Shapley value always delivers exactly one payoff distribution, which can be regarded as the "fair" distribution. The drawback, of course, is that it does not give us any idea of the real risk that players face in the negotiations, as expressed by the size of the feasible payoff interval for the other solution concepts.

The modified version is presented in [Mye91, section 9.5] and based on [Mye77]. The coalitional value (extended Shapley value) $\phi_i(v, S)$ is defined to be the unique solution to the following conditions:

$$\sum_{i \in S} \phi_i(v, S) = v(S), \forall S \subseteq N \tag{4}$$

$$\begin{aligned} \phi_i(v, S) - \phi_i(v, S \setminus \{j\}) = \phi_j(v, S) - \phi_j(v, S \setminus \{i\}), \\ \forall S \subseteq N, i \in S, j \in S \end{aligned} \tag{5}$$

where the first could be called an *efficiency*, the second a *balanced contributions* condition. The Shapley value, or "fair" payoff for player i is defined as $\phi_i(v, N)$.

Note that the balanced contributions condition is stated in terms of ϕ only, not directly in terms of the worth v. We look for a global rule determining payoffs to be expected by each player in *any* coalition situation, and we expect players who find themselves in a given coalition situation to share the worth by thinking of what they would get by this rule in the coalition situations "nearby". ϕ can actually be calculated starting with the expected payoffs when there are no coalitions — $\phi_i(v, \{i\})$ — and then iterating "upwards" through all coalitions of 2, all coalitions of 3, and so on.

The extended Shapley value can be further extended to describe fair payoffs in a graphical cooperation structure, just like defined by our problem, where it matters if players 1 and 3 cooperate directly or are merely part of the same coalition because they both cooperate with player 2. The fair allocation rule introduced by [Mye77] (now called the *Myerson value*) is defined by the following conditions:

$$\sum_{i \in S} \psi_i(v, C) = v(S), \forall C \subseteq L_2(N), S \in N/C; \tag{6}$$

$$\begin{aligned} \psi_i(v, C) - \psi_i\left(v, C \setminus \{\{i, j\}\}\right) = \psi_j(v, C) - \psi_j\left(v, C \setminus \{\{i, j\}\}\right) \\ \forall C \subseteq L_2(N), \{i, j\} \in C. \end{aligned} \tag{7}$$

where N/C is the partition operator introduced earlier.

But this definition still assumes that the worth of a coalition S only depends on its members (only the *fair payoff* also depends on its internal cooperation structure), and does not on its internal cooperation structure, nor on the cooperation structure of the other players. This is then the only missing piece we have to add to reach a fair allocation rule for our problem:

Definition 5 *The* fair allocation rule *for the cooperative component licensing game defined by the modified characteristic function $v_S(C)$ is a function ψ satisfying the following conditions:*

$$\sum_{i \in S} \psi_i(v, C) = v_S(C), \forall C \subseteq L_2(N), S \in N/C; \tag{8}$$

$$\begin{aligned} \psi_i(v, C) - \psi_i\left(v, C \setminus \{\{i, j\}\}\right) = \psi_j(v, C) - \psi_j\left(v, C \setminus \{\{i, j\}\}\right), \\ \forall C \subseteq L_2(N), \{i, j\} \in C. \end{aligned} \tag{9}$$

Provided v is superadditive, $\psi_i(v, L_2(N))$ is the fair payoff vendor i can expect.

Note that the balanced contributions condition has remained exactly the same as above.

Applying this to the two-vendor case in table 1:

$$\begin{aligned} v_{\{1\}}(\emptyset) = 0 = \psi_1(v, \emptyset) \\ v_{\{2\}}(\emptyset) = 0 = \psi_2(v, \emptyset) \\ v_{\{1,2\}}\left(\{\{1,2\}\}\right) = 4 = \psi_1\left(v, \{\{1,2\}\}\right) + \psi_2\left(v, \{\{1,2\}\}\right) \\ \Rightarrow \psi_1\left(v, \{\{1,2\}\}\right) = \psi_2\left(v, \{\{1,2\}\}\right) = 2 \end{aligned}$$

3.5 Application of the fair allocation rule to the three-vendor case

Since the three-vendor case is as order of magnitude more complex than the two-vendor case, we introduce some special notation which will make things much easier to read. The vendors are depicted in a little triangle. Cooperations are shown as lines. The vendors which we calculate v or ψ for are shown as black disks, the others as empty circles. The production profile q is not written explicitly, although one must always keep in mind that v depends on it. So, for example $v_{\{1,3\}}(\{\{1,3\}\})$ is written as $v(\circ\!\!-\!\!\bullet)$ and $\psi_1(v, \{\{1,2\}, \{2,3\}\})$ as $\psi(\bullet\!\!-\!\!\circ\!\!-\!\!\bullet)$. Since the $v\{i\}(\emptyset)$ can vary with q, we do not assume them to be 0, which would be customary for pure cooperative games.

We calculate bottom up, i.e. starting from no cooperation to one cooperation, then two, then all three.

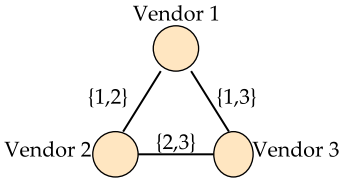

Figure 5: Notation for three vendor cooperation structures

No cooperation By equation (8),

$$\psi_i(v, \emptyset) = v_{\{i\}}(\emptyset), \text{or, equivalently}$$

$$\psi(\bullet\circ\circ) = v(\bullet\circ\circ); \psi(\circ\bullet\circ) = v(\circ\bullet\circ); \psi(\circ\circ\bullet) = v(\circ\circ\bullet).$$

One cooperation Every vendor gets to keep what he had under no cooperation and gets half of the single cooperation's gain, provided he is part of it:

$$\psi(\bullet\bullet\circ) = v(\bullet\circ\circ) + \tfrac{1}{2}[v(\bullet\bullet\circ) - v(\bullet\circ\circ)] \quad (10)$$

$$= \tfrac{1}{2}[v(\bullet\bullet\circ) + v(\bullet\circ\circ) - v(\circ\bullet\circ)] \quad (11)$$

$$\psi(\bullet\circ\bullet) = v(\bullet\circ\bullet) \quad (12)$$

(We assume the reader will be able to permutate the vendors to see that, for example, $\psi(\circ\bullet\bullet) = \frac{1}{2}[v(\circ\bullet\bullet) + v(\circ\circ\bullet) - v(\circ\bullet\circ)]$.)

Two cooperations For obtaining $\psi(\bullet\bullet\bullet)$, the following system of equations must be solved:

$$\psi(\bullet\bullet\bullet) + \psi(\bullet\bullet\bullet) + \psi(\bullet\bullet\bullet) = v(\bullet\bullet\bullet) \quad (13)$$

$$\psi(\bullet\bullet\bullet) - \psi(\bullet\bullet\circ) = \psi(\bullet\bullet\bullet) - \psi(\bullet\bullet\bullet) \quad (14)$$

$$\psi(\bullet\bullet\bullet) - \psi(\bullet\circ\bullet) = \psi(\bullet\bullet\bullet) - \psi(\bullet\bullet\bullet) \quad (15)$$

Stepwise replacement of the other unknowns in equation (13) yields:

$$v(\bullet\bullet\bullet) = 3\psi(\bullet\bullet\bullet) - \psi(\bullet\bullet\circ) - \psi(\bullet\circ\bullet) + \psi(\bullet\bullet\bullet) + \psi(\bullet\bullet\bullet) \quad (16)$$

$$\psi(\bullet\bullet\bullet) = \tfrac{1}{3}[v(\bullet\bullet\bullet) + \psi(\bullet\bullet\circ) + \psi(\bullet\circ\bullet) - v(\bullet\bullet\bullet) - v(\bullet\bullet\bullet)] \quad (17)$$

$$= \tfrac{1}{6}[2v(\bullet\bullet\bullet) + 2v(\bullet\circ\circ) + v(\bullet\bullet\circ) - 2v(\bullet\bullet\bullet) - v(\circ\circ\bullet) + v(\circ\bullet\bullet) - 2v(\bullet\bullet\bullet) - v(\bullet\circ\circ)] \quad (18)$$

For $\psi(\bullet\bullet\circ)$, the equations are:

$$\psi(\bullet\bullet\circ) + \psi(\bullet\bullet\circ) + \psi(\bullet\bullet\bullet) = v(\bullet\bullet\bullet) \quad (19)$$

$$\psi(\{1,2\}) - \psi(\{3\}) = \psi(\{1,2,3\}) - \psi(\{1,2,3\}) \quad (20)$$
$$\psi(\{1,2\}) - \psi(\{1\}) = \psi(\{1,2,3\}) - \psi(\{2,3\}) \quad (21)$$

And the solution:

$$v(\{1,2\}) = 3\psi(\{1,2\}) + 2\psi(\{3\}) - 2\psi(\{3\}) + \psi(\{2,3\}) - \psi(\{1\}) \quad (22)$$
$$\psi(\{1,2\}) = \tfrac{1}{3}[v(\{1,2\}) - 2\psi(\{3\}) + \psi(\{1\}) + 2v(\{3\}) - v(\{2,3\})] \quad (23)$$
$$= \tfrac{1}{6}[2v(\{1,2\}) + 4v(\{3\}) - 2v(\{2,3\}) - 2v(\{3\}) + v(\{1\}) - v(\{3\}) + 2v(\{3\}) - v(\{3\})] \quad (24)$$

Full cooperation Finally, for $\psi(\{1,2,3\})$:

$$\psi(\{1,2,3\}) + \psi(\{1,2,3\}) + \psi(\{1,2,3\}) = v(\{1,2,3\}) \quad (25)$$

$$\psi(\{1,2,3\}) - \psi(\{3\}) = \psi(\{1,2,3\}) - \psi(\{3\}) \quad (26)$$
$$\psi(\{1,2,3\}) - \psi(\{1,2\}) = \psi(\{1,2,3\}) - \psi(\{1,2\}) \quad (27)$$
$$\psi(\{1,2,3\}) - \psi(\{1,2\}) = \psi(\{1,2,3\}) - \psi(\{1,2\}) \quad (28)$$

leading to

$$v(\{1,2,3\}) = 3\psi(\{1,2,3\}) - \psi(\{1,2\}) + \psi(\{2,3\}) - \psi(\{3\}) + \psi(\{3\}) \quad (29)$$
$$\psi(\{1,2,3\}) = \tfrac{1}{3}[v(\{1,2,3\}) + \psi(\{1,2\}) - \psi(\{2,3\}) + \psi(\{3\}) - \psi(\{3\}) \quad (30)$$
$$= \tfrac{1}{6}[2v(\{1,2,3\}) - 2v(\{3\}) + 4v(\{3\}) - 2v(\{3\}) + v(\{1\}) - 2v(\{2,3\}) + v(\{3\}) + v(\{3\}) - 2v(\{3\}) + v(\{3\})] \quad (31)$$

The term

$$v(\{1,2\}) - 2v(\{2,3\}) + v(\{3\}) \quad (32)$$
$$= [v(\{1,2\}) - v(\{2,3\})] + [v(\{3\}) - v(\{2,3\})] \quad (33)$$

can be interpreted as the sum of the absolute advantage of coalition $\{1,2\}$ over its competitor 3 and the relative competitor damage of their cooperation from a state of no cooperation. The payoff a player can expect according to equation (31) can therefore be said to be $1/3$ of the overall worth, adjusted positively by one half, for each of his coalitions, of the sum of this coalition's absolute advantage and its relative competitor damage, and adjusted negatively by the full sum for the coalition excluding him. We will denote the term in equation (32) as $\theta_{\{1,2\}} = \theta(\{1,2\})$.

To see that, indeed, the cooperation structure matters, consider

$$\psi(\{1,2,3\}) - \psi(\{1,2\}) = \tfrac{1}{6}[2v(\{1,2,3\}) - 2v(\{1,2\}) + v(\{3\}) - v(\{3\}) - v(\{3\}) - 2v(\{3\}) + 2v(\{3\})] \quad (34)$$

In words, player 1's gain of cooperating with player 3 is one third of:

- the gain in overall worth of this cooperation
- plus one half of the private gain of this cooperation from a state of no cooperation
- plus the full damage this cooperation does to its competitor from a state of no cooperation

In order to apply this to the example in table 3, we have to calculate v first.

- If nobody cooperates, all payoffs are 0.
- If only 1 and 2 cooperate, the outcome is $(2, 0, -1)$, so $v(\{1,2\}) = 2, v(\{2\}) = -1$.
- If only 2 and 3 cooperate, the outcome is $(-1, 0, 1)$, so $v(\{2,3\}) = 1, v(\{3\}) = -1$.
- If only 1 and 3 cooperate, payoffs are 0, as they cannot license anything.
- If 1 cooperates with 2, and 2 with 3, the outcome is $(1, 0, 3)$, so $v(\{1,2,3\}) = 4$.
- An interesting case is the cooperation structure $\{\{1,2\},\{1,3\}\}$. Although vendor 1 and 3 have no licenses to deal in, vendor 3 can "bribe" vendor 1 to only offer license l_{11} to vendor 2 (a clear indication of the importance of the *extent of contracts* assumption), thus reaching outcome $(1, 0, 3)$, $v(\{1,2,3\}) = 4$.
- For $C = \{\{1,3\},\{2,3\}\}$, the two possible outcomes are equivalent with respect to v, $v(\{1,3\}) = 0$. The cooperation between vendors 1 and 3 does not add any worth.
- In the case of full cooperation, the outcome is (1,0,3), so $v(\{1,2,3\}) = 4$. Cooperation $\{1,3\}$ does not add any worth.

In order to calculate the values, we calculate the θs first:

$$\theta(\{2\}) = v(\{1,2\}) - 2v(\{2\}) + v(\{3\}) = 2 - 2 \cdot (-1) + 0 = 4$$
$$\theta(\{3\}) = v(\{2,3\}) - 2v(\{3\}) + v(\{3\}) = 1 - 2 \cdot (-1) + 0 = 3$$
$$\theta(\{1,3\}) = v(\{1,3\}) - 2v(\{1,3\}) + v(\{1,3\}) = 0 - 0 + 0 = 0$$

The payoffs, therefore, are:

$$\begin{aligned}
\psi(\text{3-clique}) &= \tfrac{1}{6}[2v(\text{3-clique}) - 2\theta(\text{2-path}) + \theta(\text{edge}) + \theta(\text{edge})] \\
&= \tfrac{1}{6}[2 \cdot 4 - 2 \cdot 3 + 4 + 0] \\
&= 1 \\
\psi(\text{3-clique}) &= \tfrac{1}{6}[2v(\text{3-clique}) - 2\theta(\text{edge}) + \theta(\text{edge}) + \theta(\text{2-path})] \\
&= \tfrac{1}{6}[2 \cdot 4 - 2 \cdot 0 + 4 + 3] \\
&= \tfrac{15}{6} = 2,5 \\
\psi(\text{3-clique}) &= \tfrac{1}{6}[2v(\text{3-clique}) - 2\theta(\text{edge}) + \theta(\text{2-path}) + \theta(\text{edge})] \\
&= \tfrac{1}{6}[2 \cdot 4 - 2 \cdot 4 + 3 + 0] \\
&= \tfrac{3}{6} = 0,5
\end{aligned}$$

It fits with intuition that vendor 2, even though his pre-transfer payoff is always 0, is able to gain a majority of the overall worth, being the only one who has something to offer to both other vendors.

3.6 Solving the first stage production game

In principle, we can now solve the two stage software component production and licensing game as suggested by [BS04]: Calculate $\boldsymbol{\psi}^q = (\psi_1^q(L_2(N)), \ldots, \psi_n^q(L_2(N))$ as the outcome of the pure production profile q, using the modified Myerson value from the previous section, then solve the resulting static non-cooperative game by finding the Nash equilibria.

The (pure) production strategy q_i of each vendor is a tuple $(q_i^{\text{out}}, q_i^{\text{in}})$, with $q_i^{\text{out}} \in \mathcal{P} \cup \mathcal{C} \ \wedge \ q_i^0, q_i^{\text{in}} \in \mathcal{C} \ \wedge \ \exists m | (q_i^0 \cup q_i^{\text{in}}, m) \in T(q^{\text{out}})$. So the vendor commits himself to a set of product and component functionalities q^{out} he would like to provide, and the set of component functionalities q^{in} from other vendors that he will rely upon in addition to the functionalities q_i^0 he already has.

If he can successfully obtain all the necessary licenses, his pre-transfer payoff will be

$$u_i^q = -m + \sum_{j \in q_i^{\text{out}} \cap \mathcal{P}} r_j(n_1, n_2, \ldots) \; ; \tag{35}$$

revenues minus costs, in other words, where the revenues are given by the revenue function, and depend on the other vendor's offerings, and the costs are chosen with the production strategy. Obtaining *more* licenses than necessary will not change anything in the pre-transfer payoff.

Due to the special nature of the biform game, we need additional assumptions in order to calculate the pre-transfer payoff for the case when the licenses obtained fall short of the required functionalities q_i^{in} of the production strategy. This is because T does not tell us about its long-term vs. its short-term flexibility, and we also have not said "how much later" the licensing negotiations will take place after the production strategy has been chosen. If the vendor had very bad

software engineers, every software asset they build would crucially depend on every required functionality, so that, if any outside piece is missing, no functionality can be offered. At the other extreme, a large part of m might be integration costs which only occur when a third party component is actually integrated, so that the production strategy can be reduced "on the fly" to another one with the same costs as if it had been chosen in the first place.

To make things both realistic and intuitive, we choose an explicit technology model which mimics the dependency structures actually declared in component-oriented software artefacts (cf. section 2.1). In the production plan, the resources are in fact committed to a set of individual components. Each component implements exactly one component or product functionality (we call the latter components *product-level components*), consumes a defined amount of development cost, and requires a fixed set of other components. There are no economies of scale beyond the possibility to use a self-produced or licensed component in several other components or products, i.e. the development costs simply add up. The general solution method presented here does not depend on this particular model; any other model will do, as long as it gives us explicit payoffs for cases in which licenses obtained fall short of the production strategy.

n_a	r_a
1	100
2	40
3	10

		n_b			
r_b, r_c		0	1	2	3
n_c	0	0,0	80,0	30,0	0,0
	1	0,60	50,40	10,10	0,0
	2	0,20	20,10	0,0	0,0
	3	0,0	0,0	0,0	0,0

Table 4: Example revenue function

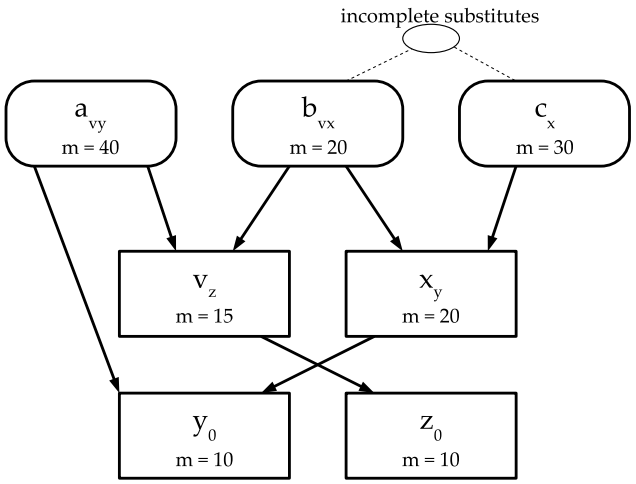

Figure 6: Example technology structure

An example is shown in figure 6. There are 3 product categories; each can be produced by one product-level component, which are labelled according to their required component functionalities: a_{vy}, b_{vx}, and c_x. The cost of development for each component is written in their respective box. The revenue function for product category *a* is independent of the other product categories. Product categories *b* and *c* are partial substitutes, so the revenue for *b* decreases with an increasing number of vendors for *c* and vice versa. The space of product-level components shall be denoted $\tilde{\mathcal{P}}$, the space of sub-product-level components $\tilde{\mathcal{C}}$.

The production strategy of a vendor is then more explicitly represented by the set of components he chooses to produce — the $(q^{\text{out}}, q^{\text{in}})$ representation follows from that as the set of product and component functionalities provided by the chosen components, and the set of component functionalities required by, but not provided by either one of the chosen components. Once a production strategy $\tilde{q} \subseteq \tilde{\mathcal{P}} \cup \tilde{\mathcal{C}}$ chosen, a vendor can offer all products categories for which he produces a corresponding product-level component, and for that component, all the required component functionalities are *available* in his production or licensing portfolio. A component is available if:

- either it is licensed from another vendor (the other vendor is then responsible to deliver all the functionality required by the component, the licensee should not have to worry),
- or he produces a sub-product-level component himself which provides the functionality, and for which all the required component functionalities are available.

For example, if a vendor's production strategy is $\{a_{vy}, b_{vx}, v_z\}$, then $q^{\text{in}} = \{x, y, z\}$. In case he manages to license only *x*, *a* cannot be offered because *y* is missing — even though the vendor providing *x* may be doing so using the same technology, so that a y_0 is hidden somewhere within his component; and *b* cannot be provided because, with *z* missing, the produced v_z is not available. If he is able to license *x* and *z*, then he can offer *b* (because the *x*'s vendor will be responsible for all the necessary lower-level components), but not *a*. Nevertheless, in both cases, we will assume that he will incur the full development cost for his production strategy.

3.7 A test run for the three vendor case

The described method has been implemented and run for some test cases. The Haskell program computing the expected post-transfer payoffs $\boldsymbol{\psi}$ (Myerson values) is presented in appendix A. The other parts of the program (most of them written in C#) are not particularly interesting. The game size is a formidable challenge: Games with less than the 7 total components shown in the above example do not seem to be very interesting. But with 7 components, each vendor has 128 possible pure strategies, so there are $2^{21} \approx 2,1$ million outcomes (slightly more

than 1/6 of that for symmetric games) which need to be calculated and stored. We could not find any existing software which can calculate all or an interesting subset of mixed Nash equilibria for games this big. The leading software package, Gambit ([MMT04]), for 3-player games, can manage up to about 30 strategies per player. [CRVW04] propose quite efficient algorithms for finding all the symmetric equilibria for symmetric games. However, in the case of a software component market, symmetric behavior is highly unlikely. We expect vendors to divide the market up, not to all settle for the same components to produce. Until a different computational breakthrough is found, we therefore only look for all pure Nash equilibria, which can be found quite easily. Pure equilibria are interesting because they are more intuitively understood, and, contrary to theory, one cannot observe people randomizing their strategies very often in real life. Also, one has to remember that we do not account for the risk yet, that payoffs can come out differently than predicted by the Myerson value, which seems a more pressing issue to address than mixed equilibria, which would themselves add a risk aspect.

Assuming that all three vendors have the technology described in the example, and no initial components, the resulting symmetric game has 92 undominated strategies and the following two pure Nash equilibria (except for permutations):

- One vendor produces everything except c_x, the others nothing. He receives 65, the others nothing.
- Vendor 1 produces a_{vy} and all the necessary components v_z, y_0, and z_0, and receives 25. Vendor 2 produces b_{vx} and all the necessary 4 components, and receives 5. Vendor 3 produces and receives nothing.

The tendency to cooperate and cross-license seems not to be very strong in this technology network, perhaps because it is too dense.

3.8 Games with more than three players

When there are more than four or more players, non-cooperative games reappear in the step from u (pre-transfer payoffs for each set of license deals) to v (pre-transfer payoffs for each cooperation structure). For example, consider the case in table 5 where we seek $v_S(\{\{1,2\},\{3,4\}\})$. $\mathcal{L}^q_{\{1,2\}} = \{l_1\}, \mathcal{L}^q_{\{3,4\}} = \{l_2\}$. Which is the value to assign to v in this case, since there are several pure Nash Equilibria? We need additional assumptions to treat cases like this. The techniques presented in [Mye91, section 9.2], as Neumann & Morgenstern's minimax rule may also be helpful here.

$(u_1 + u_2, u_3 + u_4)$	not l_1	l_1
not l_2	$(3,1)$	$(0,0)$
l_2	$(0,0)$	$(1,2)$

Table 5: two vs. two cooperations *battle of the sexes* situation

4 Conclusion and further research plans

For the DBE project to be successful in creating a software ecosystem, not only technological, but also business challenges must be solved. These challenges are typical for the introduction of new platforms. An economic theory of how effective certain measures are in solving these challenges needs to start from an understanding of the decision problems of the individual actors. The software vendors' decisions can be roughly ordered by their time horizon. The main section of this report has shown the technical feasibility as well as some of the difficulties, of applying the hybrid game theoretic approach of biform games to strategic interaction between software vendors, and the interference of product portfolio and licensing decisions. The approach treats contracts and license fees much more elegantly than it would be possible with pure non-cooperative game theory. Although there are many parameters which need to be set before applying this method, this is still less problematic than with agent simulations, which replace rational by mechanistic agents (rational in the sense of being able to reflect on the game situation as a whole). However, the presented approach is limited to games with a handful of actors and components due to its computational complexity.

There are many obvious ways of extending the presented approach to include more of the market complexity described in the first two sections: A crucial one will be the consideration of integration costs, which are completely ignored at the moment, i.e. the production strategy is only chosen based on third-party *functionalities*, not based on third-party *APIs*. But the more pressing issue is to link the results to existing empirical evidence and to seek specific evidence from historical platform/ecosystem cases.

Acknowledgements The author thanks Konstantinos Giannoutakis (University of Surrey), Gordon Gow (London School of Economics) and Thomas Kurz (Salzburg Technical University) for their valuable feedback, and Ben Romberg for technical assistance.

A A Haskell program for solving the second stage coalitional software component game

Haskell[13] is a functional programming language, hence quite compact and similar to the mathematical formulation in the text. The program consists of four parts: Declarations of some useful generic functions, definition of the game, the solution algorithm and the loops iterating through the production strategies and writing the payoffs for each combination to the output file (optimized for symmetry).

```
module CompGame where

import List
import IO

--
-- Generics
--

accum :: Num b => (a -> b) -> [a] -> b
accum f = foldl (\x y -> x + f y) 0  -- = sum(map(f x))

maxFunc :: (a -> Int) -> [a] -> (a,Int)
maxFunc f = foldl (\(ma,mb) xa
 -> if (f xa) > mb then (xa, f xa) else (ma, mb)) (error "maxFunc",-999999)

maxIndex :: (a -> Int) -> [a] -> a
maxIndex f = (\(x,y) -> x) . maxFunc f

combineAdd :: [[a]] -> a -> [[a]]
combineAdd xss x = xss ++ [x:ys | ys <- xss]

combine :: [a] -> [[a]]
combine [] = [[]]
combine (x:xs) = combineAdd (combine xs) x

mcombineAdd :: [[a]] -> [a] -> [[a]]
mcombineAdd xss x = xss ++ [x ++ ys | ys <- xss]

mcombine :: [[a]] -> [[a]]
mcombine [] = [[]]
mcombine (x:xs) = mcombineAdd (mcombine xs) x

-- Basic project parameters
data Player = P1 | P2 | P3 deriving (Enum,Eq,Show)

data Component = V | X | Y | Z deriving (Enum,Eq,Show)

data Product = A | B | C deriving (Enum,Eq,Show)

allPlayers = [P1, P2, P3]
```

[13] http://www.haskell.org

```
allComponents = [V, X, Y, Z]
allProducts = [A, B, C]

type Productions = (Player -> [Product], Player -> [Component])
type License = (Component, Player, Player)
    -- first Player licenses component to second player
type Cooperation = (Player, Player)
type Moves = (Player -> [Product], Player -> [Component], [Cooperation])
type MovesLic = (Player -> [Product], Player -> [Component], [License])

--
-- Game definition
--

-- Definition of component structure
prRequires :: Product -> [Component]
prRequires A = [V,Y]
prRequires B = [V,X]
prRequires C = [X]

requires :: Component -> [Component]
requires V = [Z]
requires X = [Y,Z]
requires Y = []
requires Z = []

-- Definition of demand (revenue) function based on number of suppliers
demand :: (Product -> Int) -> Product -> Int
demand suppliers A = case suppliers A of
    0 -> 0
    1 -> 100
    2 -> 40
    3 -> 10
demand suppliers B = case (suppliers B, suppliers C) of
    (0,_) -> 0
    (3,_) -> 0
    (1,0) -> 80
    (2,0) -> 30
    (1,1) -> 50
    (2,1) -> 10
    (1,2) -> 20
    (2,2) -> 0
    (1,3) -> 0
    (2,3) -> 0
demand suppliers C = case (suppliers C, suppliers B) of
    (0,_) -> 0
    (3,_) -> 0
    (1,0) -> 60
    (2,0) -> 20
    (1,1) -> 40
    (2,1) -> 10
    (1,2) -> 10
    (2,2) -> 0
    (1,3) -> 0
```

```
    (2,3) -> 0

-- Definition of cost function
prCost :: Product -> Int
prCost A = 40
prCost B = 20
prCost C = 30

cost :: Component -> Int
cost V = 15
cost X = 20
cost Y = 10
cost Z = 10

--
-- Definition of the algorithm
--

-- Determine number of suppliers for given product
numberOfSuppliers :: MovesLic -> Product -> Int
numberOfSuppliers myMoves myProduct =
    accum (\x -> if canSellProduct myMoves x myProduct then 1 else 0) allPlayers

-- Determine if product can be sold
canSellProduct :: MovesLic -> Player -> Product -> Bool
canSellProduct (myProducesProduct, myProducesComponent, myLicenses) myPlayer myProduct =
    (elem myProduct (myProducesProduct myPlayer))
        -- we have produced the product's main "component"
    && (all (canUseComponent (myProducesComponent, myLicenses) myPlayer)
        (prRequires myProduct)) -- and we have access to all required components

canUseComponent :: (Player -> [Component], [License]) -> Player -> Component -> Bool
canUseComponent (myProducesC, myLicenses) myPlayer myComponent =
    ( -- we have access to the component itself
        (elem myComponent (myProducesC myPlayer)) -- either because we produced it
        || any (\(c,x,y) -> ((c == myComponent) && (y == myPlayer))) myLicenses
            -- or because we have an agreement with somebody who has
    )
    && (all (canUseComponent (myProducesC, myLicenses) myPlayer) (requires myComponent))
        -- and we have access to all components required by that component

possibleLicensesOne :: (Player -> [Component]) -> Cooperation -> [License]
possibleLicensesOne myProducesC (seller, buyer) = [(c,seller,buyer) |
    c <- myProducesC seller, c `notElem` (myProducesC buyer)]

possibleLicenses  :: (Player -> [Component]) -> [Cooperation] -> [License]
possibleLicenses myProducesC = foldl (\xs y -> (possibleLicensesOne myProducesC y) ++ xs) []

rawProfitLic :: MovesLic -> Player -> Int
rawProfitLic myMoves myPlayer =
    - accum prCost (myProducesProduct myPlayer)
    - accum cost (myProducesComponent myPlayer)
    + accum (demand (numberOfSuppliers myMoves))
        (filter (canSellProduct myMoves myPlayer) (myProducesProduct myPlayer))
```

```
   where (myProducesProduct, myProducesComponent, myLicenses) = myMoves

rawProfitLicM :: Productions -> [License] -> [Player] -> Int
rawProfitLicM (myProducesP, myProducesC) myLics myPlayers =
    accum (rawProfitLic (myProducesP, myProducesC, myLics)) myPlayers

coalitionFromCoops :: [Cooperation] -> [Player]
coalitionFromCoops myCoops = nub [x | (x,y) <- myCoops]
    -- This implementation only works for up to 3 players

licensesFromCoops :: Productions -> [Cooperation] -> [License]
licensesFromCoops myProductions myCoops = case (length coalition) of
    0 -> []
    2 -> maxIndex
        (\x -> 256 * (rawProfitLicM myProductions x coalition)
            - (rawProfitLicM myProductions x (allPlayers \\ coalition)))
        (combine (possibleLicenses myProducesC myCoops))
          -- "256 *" is a little trick to do minimax, relies on payoffs being in the range 0-200
    3 -> maxIndex
        (\x -> rawProfitLicM myProductions x coalition)
        (combine (possibleLicenses myProducesC myCoops))
    where   coalition = coalitionFromCoops myCoops
        (myProducesP, myProducesC) = myProductions

rawProfitCoop :: Productions -> [Cooperation] -> [Player] -> Int
rawProfitCoop myProductions myCoops myPlayers
    = rawProfitLicM myProductions (licensesFromCoops myProductions myCoops) myPlayers

utility :: Productions -> Player -> Int
utility myProds myPlayer =
      2 * (rawProfitCoop myProds [(myPlayer,p2),(p2,myPlayer),
            (myPlayer,p3),(p3,myPlayer),(p2,p3),(p3,p2)] [myPlayer,p2,p3])
    + 4 * (rawProfitCoop myProds [(p2,p3),(p3,p2)] [myPlayer])
    - 2 * (rawProfitCoop myProds [(p2,p3),(p3,p2)] [p2,p3])
    - 2 * (rawProfitCoop myProds [] [myPlayer])
    +     (rawProfitCoop myProds [(myPlayer,p2),(p2,myPlayer)] [myPlayer,p2])
    - 2 * (rawProfitCoop myProds [(myPlayer,p2),(p2,myPlayer)] [p3])
    +     (rawProfitCoop myProds [(myPlayer,p3),(p3,myPlayer)] [myPlayer,p3])
    - 2 * (rawProfitCoop myProds [(myPlayer,p3),(p3,myPlayer)] [p2])
    +     (rawProfitCoop myProds [] [p2,p3])
   where (p2:p3:rest) = (filter (/= myPlayer) allPlayers)


--
-- Iterating through production strategies and output
--

allCoalitions :: [[Cooperation]]
allCoalitions = mcombine [[(P1,P2),(P2,P1)],[(P1,P3),(P3,P1)],[(P2,P3),(P3,P2)]]

allProductions :: [([Product],[Component])]
allProductions = [(x,y) | x <- combine allProducts, y <- combine allComponents]

tupleToFunctionProduct :: (([Product],[Component]),([Product],[Component]),
```

```
    ([Product],[Component])) -> Player -> [Product]

tupleToFunctionProduct ((ps1,cs1),(ps2,cs2),(ps3,cs3)) py = case py of
    P1 -> ps1
    P2 -> ps2
    P3 -> ps3

tupleToFunctionComponent ((ps1,cs1),(ps2,cs2),(ps3,cs3)) py = case py of
    P1 -> cs1
    P2 -> cs2
    P3 -> cs3

iteratePl3 :: Handle -> ([Product],[Component]) -> ([Product],[Component])
    -> [([Product],[Component])] -> IO ()
--iteratePl3 handle pl1Prods pl2Prods [] = do {}
iteratePl3 handle pl1Prods pl2Prods (pl3Prods:pl3Rest) = do {
    let {prs = (pl1Prods, pl2Prods, pl3Prods)};
    hPutStrLn handle ((show prs) ++ ":" ++ (show (map (utility ((tupleToFunctionProduct prs),
        (tupleToFunctionComponent prs))) allPlayers)));
    if (length pl3Rest) > 0
        then iteratePl3 handle pl1Prods pl2Prods pl3Rest
        else putStr "."
}

iteratePl2 :: Handle -> ([Product],[Component]) -> [([Product],[Component])] -> IO ()
--iteratePl2 handle pl1Prods [] = do {}
iteratePl2 handle pl1Prods (pl2Prods:pl2Rest) = do
    iteratePl3 handle pl1Prods pl2Prods (pl2Prods:pl2Rest)
    if (length pl2Rest) > 0
        then iteratePl2 handle pl1Prods pl2Rest
        else putStrLn (show pl1Prods)

iteratePl1 :: Handle -> [([Product],[Component])] -> IO ()
--iteratePl1 handle [] = do {}
iteratePl1 handle (pl1Prods:pl1Rest) = do
    iteratePl2 handle pl1Prods (pl1Prods:pl1Rest)
    if (length pl1Rest) > 0
        then iteratePl1 handle pl1Rest
        else hPutStrLn handle ""

printAllUtilities :: IO ()
printAllUtilities = do
    myH <- openFile "compgameout.txt" WriteMode
    iteratePl1 myH allProductions
    hClose myH
```

The output of the program looks as follows:

```
(([],[]),([],[]),([],[])):[0,0,0]
(([],[]),([],[]),([],[Z])):[0,0,-60]
(([],[]),([],[]),([],[Y])):[0,0,-60]
(([],[]),([],[]),([],[Y,Z])):[0,0,-120]
(([],[]),([],[]),([],[X])):[0,0,-120]
```

⋮ ∼ 2 million lines

```
(([A,B,C],[V,X,Y]),([A,B,C],[V,X,Y]),([A,B,C],[V,X,Y,Z])):[-810,-810,1410]
(([A,B,C],[V,X,Y]),([A,B,C],[V,X,Y,Z]),([A,B,C],[V,X,Y,Z])):[-810,-330,-330]
(([A,B,C],[V,X,Y,Z]),([A,B,C],[V,X,Y,Z]),([A,B,C],[V,X,Y,Z])):[-630,-630,-630]
```

The payoffs in the output have to be divided by 6 to give the actual payoff. The program was run on the Glasgow Haskell Compiler and Runtime.

References

[AM64] R.J. Aumann and M. Maschler. The bargaining set for cooperative games. *Advances in Game Theory*, 1964.

[BS04] Adam Brandenburger and Harborne Stuart. Biform games. August 2004. Available from: `http://pages.stern.nyu.edu/%7Eabranden/biformgames081504.pdf`.

[Car89] Carl F. Cargill. *Information Technology Standardization: Theory, Organizations and Process*. Digital Pr, 1989.

[CRVW04] Shih-Fen Cheng, Daniel M. Reeves, Yevgeniy Vorobeychik, and Michael P. Wellman. Notes on equilibria in symmetric games. Technical report, 2004. Available from: `www.sci.brooklyn.cuny.edu/~parsons/events/gtdt/gtdt04/reeves.pdf`.

[DM65] M. Davis and M. Maschler. The kernel of a cooperative game. *Naval Research Logistics Quarterly*, 12:223–259, 1965.

[FT91] Drew Fudenberg and Jean Tirole. *Game Theory*. The MIT Press, 1991.

[Fun04] Jeffrey L. Funk. The product life cycle theory and product line management: the case of mobile phones. *IEEE Transactions on Engineering Management*, 51(2):142–152, May 2004.

[Gat02] Bill Gates. A growing technology ecosystem. Online Essay, April 2002. Available from: `http://www.microsoft.com/issues/essays/2002/04-23ecosystem.asp`.

[HM90] Oliver Hart and John Moore. Property rights and the nature of the firm. *J. of Political Economy*, 98:1119–1158, 1990.

[MF00] C.R. Morris and C.H. Ferguson. How architecture wins technology wars. *Harvard Business Review*, 71(2):86–96, 2000. Open Research Question: How can you operationally recognize architecture-defining, thus strategic components? Relation to article about video games that delivers some arguments why things can be different in different industries.

[MMT04] R. D. McKelvey, A. M. McLennan, and T.L. Turocy. Gambit: Software tools for game theory. v0.97.0.7, 2004.

[Moo97] James F. Moore. *The Death of Competition: Leadership and Strategy in the Age of Business Ecosystems*. HarperBusiness, 1997.

[MP88] Carmen Matutes and Regibeau P. Mix and match: Product compatibility without network externalities. *Rand Journal of Economics*, 1988.

[MS03] David G. Messerschmitt and Clemens Szyperski. *Software Ecosystem : Understanding an Indispensable Technology and Industry*. The MIT Press, 2003.

[Mye77] Roger B. Myerson. Graphs and cooperation in games. *Mathematics of Operations Research*, 2:225–229, 1977.

[Mye91] Roger B. Myerson. *Game Theory: Analysis of Conflict*. Harvard University Press, 1991.

[Nac02] Francesco Nachira. Towards a network of digital business ecosystems fostering local development. discussion paper, 2002. Available from: `http://www.digital-ecosystems.org/de/refs/ref_books.html`.

[Ray99a] Eric Raymond. Homesteading the noosphere. In *The Cathedral and the Bazaar*. O'Reilly, 1999. Available from: `http://www.catb.org/~esr/writings/cathedral-bazaar/`.

[Ray99b] Eric Raymond. The magic cauldron. In *The Cathedral and the Bazaar*. O'Reilly, 1999. Available from: `http://www.catb.org/~esr/writings/cathedral-bazaar/`.

[Rot95] Michael Rothschild. *Bionomics: Economy As Ecosystem*. Henry Holt & Company, 1995.

[SC01] Ron Sanchez and Robert P. Collins. Competing - and learning - in modular markets. *Long Range Planning*, 34(6):645–, Dec 2001.

[Sch69] D. Schmeidler. The nucleolus of a characteristic function game. *SIAM J. of Appl. Math.*, 17:1163–1170, 1969.

[Sha53] L.S. Shapley. A value for n-person games. In *Contributions to the theory of games II*. Princeton University Press, 1953.

[Shy01] Oz Shy. *The economics of network industries*. Cambridge University Press, 2001.

[Spe50] J. Spengler. Vertical integration and anti-trust policy. *J. of Political Economy*, 58:347–352, 1950.

[Ste69] R.E. Stearns. Convergent transfer schemes for n-person games. *Transactions of the American Mathematical Society*, 134:449–459, 1969.

[SV99] Carl Shapiro and Hal Varian. *Information Rules*. Harvard Business School Press, 1999.

[Szy99] Clemens Szyperski. *Component Software*. Addison-Wesley Professional, 1st ed. edition, 1999.

[vNM44] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.