City Research Online

City St George's, University of London

Citation: Jansen, S., Brinkkemper, S. & Finkelstein, A. (2008). Component Assembly Mechanisms and Relationship Intimacy in a Software Supply Network. Paper presented at the EurOMA 2008, tradition and innovation in operations management: connecting past and future, 15-18 Jun 2008, Groningen, Netherlands.

This is the accepted version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/26441/

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to City Research Online policy.

City Research Online: http://openaccess.city.ac.uk/ publications@citystgeorges.ac.uk

COMPONENT ASSEMBLY MECHANISMS AND RELATIONSHIP INTIMACY IN A SOFTWARE SUPPLY NETWORK

Slinger Jansen and Sjaak Brinkkemper
Department of Information and Computing Sciences
Utrecht University
Padualaan 14, De Uithof
3584CH Utrecht
{s.jansen, s.brinkkemper}@cs.uu.nl

Anthony Finkelstein
University College London
Department of Computer Science
Gower Street
London WC1E 6BT, United Kingdom
a.finkelstein@cs.ucl.ac.uk

ABSTRACT
Vendors of product software include software components, products, and services of others. These participants establish a range of different business relationships, from intimate relationships to practically disconnected relationships to arms-length purchasing. These product software vendors have also made architectural decisions about the integration of their software products. In this paper we explore the relationship between architectural integration methods and the different types of relationships between participants in a SSN. These relationships are uncovered by inventorying the relationships and architectural decisions for two specific integrated software products. Knowledge about these relationships and reuse methods enables development managers and software architects to make informed decisions on both the managerial and the architectural level, narrowing the gap between business requirements and design.

Keywords: software supply networks, software architecture, design rationale, product software

REUSE IN SOFTWARE SUPPLY NETWORKS
Increasingly, product software firms are integrating both software components and enterprise software services to achieve shorter time-to-market and higher quality for their software products. Whereas in the past product software firms were monoliths developing software systems from scratch, currently software firms are enthusiastic about component reuse, application service reuse, and purchasing software development services from others. Economic necessity has lead to reuse, to the extent that 99% of all computer instructions come from COTS products (Basili & Boehm, 2001). As new business models are being developed and firms are further specializing, relationships with participants in Software Supply Networks (SSNs) require more attention from scholars.

Interrelationships among participants in SSNs influence the daily activities of development managers, software architects, and business managers. There is a lack of awareness of the effects of decisions made by these three groups with regards to software architecture and the SSNs. This is the root cause

of three recurring problems of software vendors: the software architecture is not flexible enough to replace quickly one commercial component with another, decisions are made to cooperate at a business level without having a clear view of the implications for software development effort, and new features of subcomponents of a system cannot be made available quick enough, due to compatibility problems. New modelling methods are required to provide development managers, software architects, and business developers with clear descriptions of a product's software architecture and its SSN.

A software supply network is a series of linked software, hardware, and service organizations cooperating to satisfy market demands (Jansen, Finkelstein, & Brinkkemper, 2007). The software ecosystem (Messerschmitt & Szyperski, 2003) of a software organization are all the software supply networks in which the organization actively cooperates. We define product software as software that is built for a market in repeated releases, as opposed to software that is built for one system (Xu & Brinkkemper, 2005).

A recent survey shows that 69% of software vendors ship their products with COTS (Components-Off-The-Shelf) from third parties (Jansen, Brinkkemper, & Helms, 2008). The benchmark survey also revealed that product software vendors are still relatively immature, when it comes to management of COTS. Only 25% of those respondents that do use COTS version track those COTS included in their product. To do so these software vendors store COTS in the source tree or a versioned release repository. Similarly, in eight case studies of product software vendors and their CCU processes, conducted between 2003 and 2006, results were found that only rarely software vendors store version information and compatibilities of COTS explicitly (Jansen, 2007). When it is done, all too often they can be found using a spreadsheet, which is not readable by deployment scripts, inhibiting automated deployment problem resolution.

This paper has four contributions. In the next section a list of SSN roles is provided that can assist in defining the business value of a participant in a SSN. In the following section different methods for functionality re-use are defined, an exercise that has not been done since (Krueger, 1992). Thirdly, third-party functionality re-use is described in two software products, together with the relationships with the suppliers of the functionality. Finally, we hypothesize about the relationships between SSN relations and tight or loose coupling of third-party functionality.

DEFINING ROLES IN THE SOFTWARE SUPPLY NETWORK

Many terms are used in the software industry to describe roles in software supply networks. Microsoft is described as an ecosystem leader, SAP is a first tier supplier, Amazon.com is listed as a value added reseller (Sturgeon, 2000), and COTS suppliers are known as lower tier suppliers or sub-contractors. Simultaneously, each of these terms has at least half a dozen synonyms. Different approaches can be taken when trying to define roles in a SSN, by looking at end-product delivery, activity scope (Sturgeon, 2000) or contract type.

When looking at end-product delivery, we require the concept of the product software production pipeline (see *Figure 1*). Seven major decoupling points (Jansen, Finkelstein, & Brinkkemper, 2007) are identified, where new customers can arise to purchase a semi-finished product. First, a development organization can outsource the requirements engineering process and/or design process (*a, b*). The developer can choose to release their source code (*c*), binaries (*d*), or assemblies of components (*e*) to another developing organization who uses these artifacts as a component to their product, or to a publisher who releases the product (common for games, where the vendor is rarely the developer). A

software vendor can also choose to release the product itself, either as a package, or as a deployed system (*f*). Finally, a vendor can decide to offer their product to its customer in an application service provider model, where the vendor sells usage of its product instead of the product itself (*g*). It is not uncommon for software products going through iterations of the decoupling points before the product is delivered to a customer. It is easy to envisage that a system designer creates a design, sells the design, and the software developer starts at the requirements phase again to see what can be added to the design.

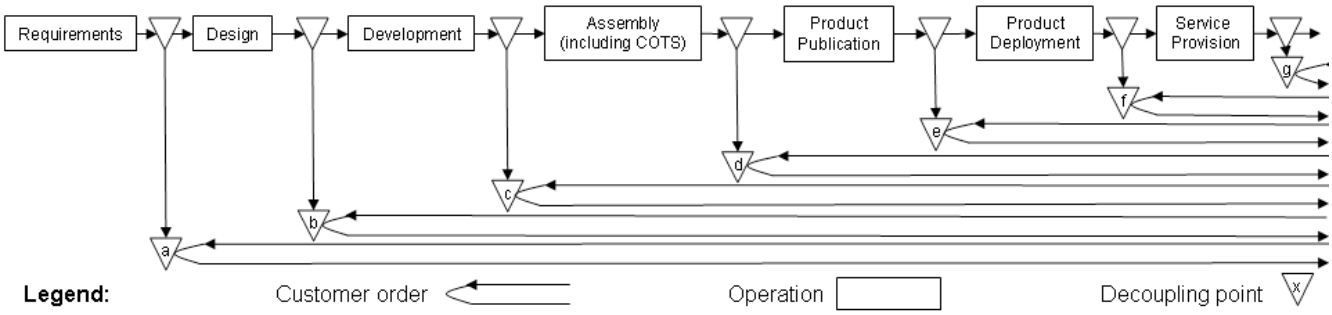

Figure 1: Product Software Decoupling Points in Product Software Production Pipeline

In Table 1 a partial list of roles in a SSN is provided. The roles are presented as if the participant is active in a SSN. They are grouped by software resellers, software service organizations, application service providers, and software vendors. Software resellers are those participants in the SSN that contract products and components and resell them to others. Service organizations are those organizations that provide services to software developers, ranging from requirements engineering to implementation services. Application service providers are those organizations that have deployed an application on their own servers, and provide the service to customers and other participants in the SSN. Finally, software vendors are those that develop software that can be sold as a product. Please note that these also include open source software developers because these also potentially add value to SSNs.

Table 1 shows that relationship intensity differs for each SSN role. A Value Added Reseller (VAR), for instance, can be an organization that buys a product, installs it on a pc, and resells the bundle, as a practically anonymous participant in the SSN. A VAR can also build extensions to a product (such as Microsoft CRM) and therefore requires a much more intimate relationship with the software supplier. Frequently, only the product makes the difference between two different terms. A COTS vendor, for instance, differs from an ISV (if at all) in that it delivers COTS instead of a final product.

Others (Messerschmitt & Szyperski, 2003) (Sturgeon, 2000) have defined different groups and classifications for organizations in supply networks. Software vendors (ISVs) are often seen as first-tier suppliers or Original Equipment Manufacturers (OEMs). COTS vendors are seen as second- or lower-tier suppliers and service organizations are seen as turn-key solution providers. Beside these synonyms, (Sturgeon, 2000) defines companies as being 'lead firms' and 'integration firms', e.g., IBM, Microsoft, etc. Note that these terms can coexist and are in fact fundamental to this work. Most software vendors agree that they are 'integration firms', integrating some or performing all product software development activities (from Figure 1) up to the final provision of the service itself (example: Salesforce.com). We have left out terms that define the size or scale at which a participant operates, such as 'eco-system leader' (Microsoft) and leave this to future work.

When firms decide to cooperate in a SSN they can do so at different levels of interaction intimacy. These relationships can range from completely cold, for example when a software component is reused from an open source community that is unaware of that reuse, to very warm, for instance where two software vendors lay out their release schedules and plan their releases cooperatively. Many software firms themselves have developed loyalty programs with resellers and co-developers and are consciously encouraging software vendors to become (more active parts) of their eco-systems.

Table 1: A Partial List of SSN Roles

Group	Term	Relationship intensity	Activity	Contractual Relationship				Deliverable
				Reseller	Vendor	Enterprise Service Supplier	Service Supplier	
Resellers	*(i) Value added reseller*	Any	Add functionality and resell	X				(Rebranded) Product or service
	(ii) Reseller	Any	Buy and resell	X				Original Product
	(iii) Software assembler	Intense	Assemble and resell	X	X			COTS or product assembly
	(iv) Software publisher	Intense	Rebrand and resell	X	X			(Rebranded) Product or service
Service organizations	*(v) Software Designer*	Intense	Supply service			X		Software design
	(vi) Requirements engineer	Intense	Supply service			X		Requirements documents
	(vii) Software developer, Outsourcing partner	Intense	Develop and supply service		X	X		Source code
	(viii) Product deployer	Intense	Resell, deploy, implement		X	X		Deployed product
Service providers	*(ix) Application Service Provider (ACADVendor)*	Intense	Provide computer service				X	Service
Software vendor	*(x) Independent Software Vendor (ISV)*	Any	Build and Sell		X			Product
	(xi) COTS vendor	Any	Build and sell		X			COTS
	(xii) Original Design Manufacturer (ODM)	Intense	Design, develop, and sell		X	X		Rebranded Product

SOFTWARE FUNCTIONALITY EXTENSION MECHANISMS

Several mechanisms exist that extend software functionality with third party functionality: component calls, service calls, source code inclusion, and shared data objects. Component calls can be direct to a component, or indirect through a component bus or another (glue) component. Components are typical

in that they are dormant until invoked, compared to services that need to be live to be (re)used. Components can run independently (think of a shell script) or can require a component framework or virtual machine to be instantiated. Data object sharing, though popular, has the drawback that the database is required to be 'smart' (locking and transactions are at least required) and that software objects cannot independently evolve the data model.

Services tend to be live when called upon. This requires that they are running locally (think of MySql running on the same server as a PHP program) or remotely (think of a currency conversion service). The same holds for services as for components, that they can be called upon directly, for instance through a SOAP call or indirectly through a service bus. We cannot avoid the difference in trend here between services and components, in that services are said to be less tied to a location than components are. Furthermore, it is much easier to facilitate dynamic composition and updating of service configurations than it is for component configurations (Ajmani, Liskov, & Shrira, 2006).

Contrary to others (Tomer, Goldin, Kuflik, Kimchi, & Schach, 2004), we explicitly exclude source code inclusion because it has serious drawbacks. One can simply copy a method, class, or full package into a component configuration. By doing this, tight coupling is established, with all its downsides, such as the fact that an update of the third-party source requires another copy-paste action. We have developed a classification showing the different reuse patterns used for software products (see Table 2 and Figure 2). The presented mechanisms have been taken from other comprehensive overviews of software functionality extension such as Krueger's work on software reuse (Krueger, 1992), Szyperski's work on components (Szyperski, 1997), and the work of Shaw et al. on architectural connections (Shaw, DeLine, & Zelesnik, 1996).

Table 2 - Software Functionality Extension Mechanisms

Unit of Inclusion	Mechanism	Interaction Method	Example
Component	(a) Direct comp. call	Pipe and filter	Call runnable component and feed it data
	(b) Direct comp. call	Shared data object(s)	Call runnable component on data source, report when finished
	(c) Direct comp. call	Plug-in architecture	Java Plug-in architecture
	(d) Direct comp. call	Component frameworks	CORBA call
	(d) Direct comp. call	Component library reuse (method calls & class use)	DLL, libraries, jars, etc
	(e) Indirect comp. call	Component Bus	Component invocation bus, that is active like a service
	(f) Indirect comp. call	Glue code	Glue code is written between the application and the extending component, aka adaptor
Service	(g) Direct serv. call	SOAP call, specific integration, forward form submits, web page inclusion, etc	MySql database service, online currency converter
	(h) Indirect serv. call	Enterprise service bus	SOAP request that is evaluated and directed to three different services, of which one is found most suitable to handle the request

There are three reasons for classifying the different mechanisms for third-party functionality inclusion. First, the classification can be used to see whether organizations drive business differently when they use different types of inclusion. Secondly, when studying opportunistic COTS reuse, we can establish

whether the organizations trying to assemble these COTS require a specific type of software architecture. Thirdly, the reuse mechanisms enable us to define coupling tightness. The more different mechanisms that are used and the more code that is required in the original product, the tighter coupling becomes (a crude adaptation from (Pressman, 1982)).

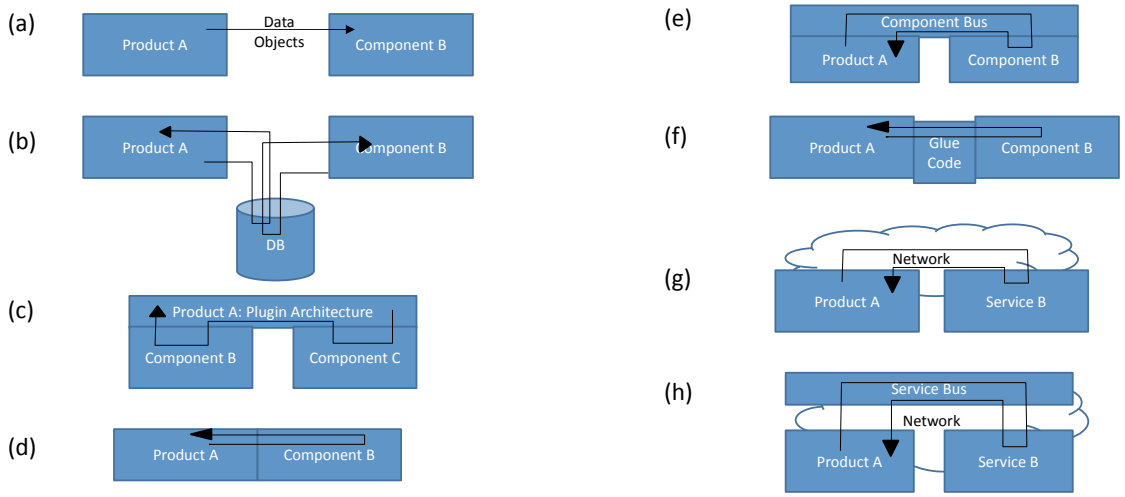

Figure 2: Graphical Representation of Software Extension Mechanisms

THE CASE STUDIES: AN ERP VENDOR AND A 3-D CAD VENDOR

To produce the results in this paper two descriptive case studies (Yin, 2003) were performed at software vendors in the Netherlands. These case studies resulted into two case study reports. Facts have been collected from interviews, document study, software study, and direct observations at the vendor. The validity threats to our case studies are construct, internal, external, and reliability threats. With respect to construct validity, the same protocol was applied to each case study, which was guarded by closely peer reviewing the case study process and database. With regard to internal validity we have defined the terms in this research in earlier work (Jansen, Finkelstein, & Brinkkemper, 2007) and provided these definitions to the case study participants. With regards to external validity we took two average sized product software vendors from the Netherlands that build extensions to larger products, each with a small number of partners. Finally, in regards to reliability, we would gather similar results if we redid the research, because we use a case study protocol, a structured interview outline, and a case study database. Both the companies in the case studies were anonymised.

The reused functionality under study was selected carefully. In one of the case studies an on-line market is used for semi-finished components from which source code is copied vis-à-vis. These components and suppliers, though interesting, were not studied because of their merging with the main software product itself. Ten reused components and services were distinguished in the case studies.

Case Study 1: ERP Product ERPVendor

The company ERPVendor is an ERP vendor that sells a large product that extends the popular Microsoft CRM product. ERPVendor currently has around 600 customers worldwide and 38 employees. The product is sold through resellers and through an internal sales department. ERPVendor's product includes external functionality from an online timesheet application (OTS), an office integration application, and a PDF creation application. OTS, MS CRM and ERPVendor's product all depend on Microsoft SQL Server (MS SQL).

Case Study 2: Technical CAD Application Plug-in CADVendor

CADVendor develops and distributes CAD software products for the building services industry. CADVendor's product is now used by more than 7000 end-users daily in the Netherlands and Belgium. CADVendor employs approximately 100 employees. The software development activities are performed in the Netherlands and Romania.

Hypotheses and Results

A subset of the results of the two case studies is listed in Tables 3 and 4. Table 3 lists the methods used by ERPVendor and CADVendor to reuse functionality from others into their products. Table 4 displays some of the characteristics of relationship intimacy by looking at typical methods of contacting third-party software providers. The two cases are used to confirm or discredit a number of hypotheses we have about software supply networks. Table 3 shows some unexpected results. We were under the impression that functionality is generally reused with one reuse method, instead of many. In the most extreme example (ERPVendor's product reusing MS CRM functionality) four methods of inclusion are used. MS CRM is a well known CRM product. It is installed as a web server and requires MS SQL Server. ERPVendor's product works beside MS CRM, which means that the products can function independently from each other, even though customers will generally not be aware of that. Interaction with MS CRM is initiated in different ways. Regularly, data objects that are shared are synchronized, such that parts of the data in the MS CRM database are the same as the data in ERPVendor's product's database. Furthermore, ERPVendor's product includes pages from MS CRM. Also, MS CRM enables plug-ins, which are used when MS CRM requires data from ERPVendor's product. Finally, both products make available a number of services from the MS CRM service bus. Both products call upon each other's services. The two products are tightly coupled.

Table 3: Architectural Relationships Components and Coupling

Case	ERPVendor				CADVendor					
Component Name	MS CRM	OTS	WC	PDFC	CR	SS	XML	IC	AC	DWG
Feature Criticality										
Alternative solutions	Some	Many	Many	Many	Some	Many	Many	None	None	Some
Criticality	High	Med.	Med.	Low	Med.	High	Low	High	High	Med.
Coupling	Very Tight	Tight	Loose	Med.	Med.	Loose	Loose	Tight	Tight	Tight
Estimated Development Switching Effort (hrs)	5000	500	100	100	480	100	100	5000+	5000+	100
Component Reuse Method										
Pipe and filter										
Shared data object(s)	Y	Y			Y					
Component frameworks								Y	Y	
Component library reuse				Y	Y	Y	Y			Y
Plug-in architecture	Y							Y	Y	
Component Bus										
Glue code					Y	Y				Y
Service Reuse Method										
SOAP call						Y				
Web page inclusion	Y	Y								
Service bus	Y		Y	Y						

H1: Relationship intimacy between a component supplier and a product software vendor is directly related to the tightness with which components are coupled. In both tables can be seen that tight coupling is at least related to the contact intimacy. The relationship does not seem to work inversely, in that a warmer relationship between the two software organizations implies that coupling between their components is tight. When looking at the effort that is required to change from one component to an alternative, this relationship does not change.

H2: More mature organizations will reuse functionality from more loosely coupled components than younger organizations. ERPVendor is a younger company than CADVendor and has only had one release of their main product. CADVendor has had five major releases of the product under study and the product is mature. Many efforts have been made to make coupling between external products looser. This cannot be found in the results, however. Another aspect seems to have greater influence: ERPVendor builds its product on top of a more service oriented web-based product, whereas CADVendor builds its product on top of a stand-alone CAD product. We speculate that less effort is required from ERPVendor to achieve loose coupling between their own product and functionality of others than in the case of CADVendor.

Table 4: Organizational Relationship and Relationship Intimacy

Case	ERPVendor				CADVendor					
Component Provider	MS	OTS	WC	PDFC	CR	SS	MS	IC	AC	DWG
Access to source code		Y	Y					Y		
Access to early releases	Y	Y		Y				Y	Y	
Access to release planning	Y	Y		Y		Y		Y	Y	
Access to online dev portal	Y	Y				Y		Y	Y	Y
Manual feedback forwarding	Y	Y	Y	Y		Y		Y	Y	Y
Cooperative Development		Y						Y		
Contact with sales department	Y	Y		Y		Y	Y		Y	
Contact with developers		Y				Y		Y	Y	Y
CEO level contact		Y						Y		Y
Contact with Helpdesk	Y	Y	Y	Y	Y	Y		Y	Y	Y
Company Size	79,000	20	20	10[1]	6,200	1,100	79,000	40[1]	5,169	40[1]
Relationship Type	(*x*)	(*xi*)	(*xi*)	(*xi*)	(*xi*)	(*xi*)	(*xi*)	(*x*)	(*x*)	(*xi*)
Sees its customers as	(*i*)	(*iii*)	(*iii*)	(*iii*)	(*iii*)	(*iii*)	(*iii*)	(*i*)	(*i*)	(*iii*)

H3: Software vendors only reuse application services when the service does not contain any critical functionality for the software product. The reuse of application services implies that a vendor loses control over some of the functionality (and data) the vendor is trying to reuse. This makes software vendors reluctant to reuse services from others. For both cases the only application services that are being reused are application services over which the customer or the vendor has full control, i.e., the application service is installed on the customer (MS CRM, WC, PDFC) or vendor site (SS).

H4: The larger the component supplier, the colder the relationship. Larger software vendors appear never to do cooperative development with ERPVendor and CADVendor. Also, they are less likely to provide direct access to their CEOs. Table 4 shows that size is not related to whether a product software vendor will get into contact directly with a software vendor's development department or not. Whereas ERPVendor has no contact with Microsoft's developers, CADVendor does

[1] Estimates

have contact with developers from AC. Another fact that tends to distort views on this relationship is that larger software firms tend to have many different products (acquired through the years), run by relatively independent product teams. In Table 4 this holds true, since both Microsoft and CR are large organizations that sell a large range of products. The product development groups of MS CRM and CR's product are small organizations within larger ones.

DISCUSSION & CONCLUSIONS

A relationship arises from Table 4, in that a software vendor's and a component supplier's perception of each other is related. When a software vendor is defined by ERPVendor or CADVendor as an ISV, the ISV defines the ERPVendor or CADVendor as VARs. Simultaneously, COTS vendors see ERPVendor or CADVendor as component assemblers. When looking at the fact that ERPVendor's and CADVendor's products plug into the products from ISVs this is not surprising. These relationships are related to the product context (Brinkkemper, Soest, & Jansen, 2007) of a software product. The roles presented in this paper can be modeled as participants with *requires* and *provides* interfaces (for example *requires* source code, *provides* COTS assemblies). The activities a participant fulfills can be modeled using value net modeling techniques (Vilminko & Kinnula, 2005) to complete the SSN model. We plan to improve business modeling using value net modeling techniques, to enable product software vendors and other participants in the SSN to further develop their business models and discover novel opportunities.

Basili and Boehm (Basili & Boehm, 2001) show that although glue-code development usually accounts for less than half the total CBS software development effort, the effort per line of glue code averages about three times the effort per line of developed applications code. This was confirmed by both the software vendors, who in some cases hired specialized consultants to integrate the COTS. Furthermore, this supports H1 because once components become more tightly coupled, there is an increase in demand for knowledge from the component supplier. Secondly, Basili and Boehm confirm that non-development costs, such as licensing fees, are significant and projects must plan for and optimize them. Both vendors experienced this, in that some functionality comes at the price of a relatively large sum per developer seat, whereas others charge a relatively small fee per end-user. This further confirms that more research is needed in this area, to create financial models that can show whether the re-use model is truly healthy (Brinkkemper, Soest, & Jansen, 2007). Basili and Boehm's final statement that CBS is currently a high-risk but profitable activity because COTS integration projects generally have larger effort and schedule overruns than conventional development projects, only further confirms the contribution of this paper.

Three conclusions can be drawn from this research. We show that re-use in practice is generally not done through clean APIs alone for product software. Furthermore, we show that software vendors are reluctant to use application services from others as long as this implies giving up full control over its functionality. Finally, we show that if functionality is tightly coupled with a software product, the software vendor will have a warm relationship with the component supplier.

ACKNOWLEDGEMENTS

We would like to thank both the companies for their participation in the case studies. Furthermore, we would like to thank Wilco van Duinkerken and Henk van der Schuur for their inspiring ideas that contributed to this paper.

REFERENCES

Ajmani, S., Liskov, B., & Shrira, L. (2006). Modular Software Upgrades for Distributed Systems. *European Conference on Object-Oriented Programming (ECOOP).*

Basili, V. R., & Boehm, B. (2001). COTS-Based Systems Top 10 List. *IEEE Computer* , 2-4.

Brinkkemper, S., Soest, I. v., & Jansen, S. (2007). Modeling of product software businesses: Investigation into industry product and channel typologies. *In The Inter-Networked World: ISD Theory, Practice, and Education, proceedings of the Sixteenth InteInternational Conference on Information Systems Development (ISD 2007).* Springer-verlag.

Jansen, S. (2007). *Customer Configuration Updating in a Software Supply Network.* PhD Thesis, Utrecht University, ISBN: 978-90-393-4666-2.

Jansen, S., Brinkkemper, S., & Helms, R. (2008). A Benchmark Survey into the Customer Configuration Updating Processes and Practices of Product Software Vendors in the Netherlands. *7th International Conference on Composition-Based Software Systems (ICCBSS).* Madrid, Spain: IEEE.

Jansen, S., Finkelstein, A., & Brinkkemper, S. (2007). Providing transparency in the business of software: A modelling technique for software supply networks. *Proceedings of the 8th IFIP Working Conference on Virtual Enterprises.* Gumares, Portugal: IFIP.

Krueger, C. W. (1992). Software Reuse. *ACM Computing Surveys (CSUR), Volume 24, Issue 2* , 131-181.
Messerschmitt, D., & Szyperski, C. (2003). *Software Ecosystem.* Cambridge, Massachusetts, 424 pages: MIT Press.

Pressman, R. S. (1982). *Software Engineering - A Practitioner's Approach.* ISBN 0-07-052182-4.

Shaw, M., DeLine, R., & Zelesnik, G. (1996). Abstractions and Implementations for Architectural Connections. *Third International Conference on Configurable Distributed Systems*, (pp. 2-10). Annapolis, Maryland.

Sturgeon, T. J. (2000). How Do We Define Value Chains and Production Networks? *Proceedings of the Bellagio Value Chains Workshop* (pp. 1-22). MIT IPC Globalization Working Paper 00-010.

Szyperski, C. (1997). *Component Software: Beyond Object-Oriented Programming.* 411 pages: Addison-Wesley Professional .

Tomer, A., Goldin, L., Kuflik, T., Kimchi, E., & Schach, S. R. (2004). Evaluation Software Reuse Alternatives: A Model and its Application to an Insdustrial Case Study. *IEEE Transactions on Software Engineering* , 601-612.

Vilminko, S., & Kinnula, M. (2005). Managing the Software Market Evolution - a Network Approach to Value Creation in Software Business. *Proceedings of the Frontiers of E-Business Research Conference* (pp. 825-839). Tampere: Cityoffset.

Xu, L., & Brinkkemper, S. (2005). Concepts of Product Software: Paving the Road for Urgently Needed Research.

Proceedings of the The first International Workshop on Philosophical Foundations of Information Systems Engineering (pp. 523-528). LNCS.

Yin, R. K. (2003). *Case Study Research - Design and Methods.* SAGE Publications, 3rd ed.