INVITED PAPER

Rethinking Components: From Hardware and Software to Systems

This paper describes a new vision of componentization that poses organizational challenges to industry practices, engineering approaches and government leadership. It reviews the history and component design methodologies and points the way forward as a basis for further research and discussion.

By David G. Messerschmitt, *Fellow IEEE*

ABSTRACT | The germ of the component idea arose in mass production as the interchangeable part, but in today's information and communications technology (ICT) industries the component can connote considerably more, such as multiple uses, opportunistic combinations with other components, design by assembly, and incremental evolution through field replacement with upgraded components. In spite of its many advantages, the component has failed to keep up with increasing scales of integration, increasing use of software, and the resulting complexity and application diversity. A rethinking of the component and associated industry practices is needed in light of modern technology and applications. Componentization has many payoffs, including as a process for industry coordination, most notably in large multivendor complex systems with fragmented administrative and ownership domains. Invigorating componentization requires abandoning antiquated concepts such as components are exclusively hardware or software or even exclusively technological, are units of manufacture and packaging, or that each component is the responsibility of an individual firm. The *system* component, which incorporates hardware, software, and oftentimes even human process or organizational elements, whatever is necessary to achieve a coherent body of functionality, is the appropriate perspective today.

KEYWORDS | Complex systems; components; design; industrial organization; modularity

I. INTRODUCTION

Components in their simplest form are elements of functionality sold as a unit and incorporated into multiple uses. In the purest form of componentization, systems are constructed entirely from acquired (not locally implemented) components, which are assembled together without modification. A component creates value when assembled with other components (possibly combined with locally developed technology) to form a system, which in turn provides a turnkey solution satisfying some need, like a network or a personal computer. Component-based design is the antithesis of handcrafting, where a new system is designed entirely from scratch for a single purpose.

Components can be traced back several centuries to the interchangeable part that arose in the munitions industry and reached its pinnacle in the automobile industry. While the component played an important role in the earlier history of information and communications technologies (ICT), the component methodology has failed to keep pace with changing technology. Older readers can remember when systems of scale and complexity at the limits of consumer affordability could be assembled entirely from off-the-shelf components. Witness the Heathkit Company, which had an extensive catalog of compelling consumer electronics products (like a color television and high fidelity sound systems) offered to the consumer as a box of components with assembly

Manuscript received December 11, 2006; revised April 5, 2007.
The author is with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA USA and visiting the Department of Computer Science and Engineering, Software Business Laboratory, Helsinki University of Technology, 94720-1770 Helsinki, Finland (e-mail: messer@eecs.berkeley.edu).

Digital Object Identifier: 10.1109/JPROC.2007.898862

instructions. Heathkit abandoned this business in 1992 after a long decline [1]. Today components are popular in hardware, although not as ubiquitous as before, and componentization is an active area of research and development in software engineering [2].

There are much bigger opportunities than electronic kits. Stepping back and examining the ICT industries critically, large and complex systems seem to hit an architectural "wall" where innovation and progress is slowed. Both operating systems [3] and cellular wireless networks [4] have evolved a series of "generations," each a major redesign (although often sharing technology with the older generations and seeking to offer backward compatibility). This carries the heavy burden of offering sufficient added value with each generation to justify the heavy development and provisioning costs, a challenge that becomes more difficult with maturity. The increasing costs and time associated with achieving backward compatibility further undermine this approach. For example, the Windows Vista operating system has been delayed, attributed in part to the difficulty in maintaining backward compatibility with numerous complementary products (such as peripherals and applications) [5]. But this same voluminous complementarity is largely responsible for Windows' large market share. Apple Computer updates its MacOS operating system more often, in part because the Macintosh and its peripherals are more proprietary, but this characteristic contributes to its relatively small market share. The Internet has had impressive longevity, with an expanding suite of applications. However, the latest upgrade to version six of its core protocol (IPv6) is being deployed slowly [6]. It has no built-in security architecture and other shortcomings, stimulating efforts to redefine its architecture [7] following a "clean slate" approach that is unlikely to gain direct market acceptance.

Clearly, our methodologies for architecting large complex ICT systems have room for improvement. A major opportunity is to better align these methodologies with market forces in a multivendor environment with multiple owners and operators. This issue is even more critical in distributed business and social applications like business-to-business e-commerce and social networks, where change is rapid. This paper asserts that componentization offers a promising framework for addressing these issues and offers a number of other benefits. The component as it was once defined and practiced is much less relevant to today's ICT, but with a rethinking in light of today's realities it can once again play a significant role.

In the following, componentization is approached from a number of perspectives, including history, value, and design process. In addition, some industry cooperation, industry coordination, and business and economics issues are considered because componentization has particular relevance in these dimensions.

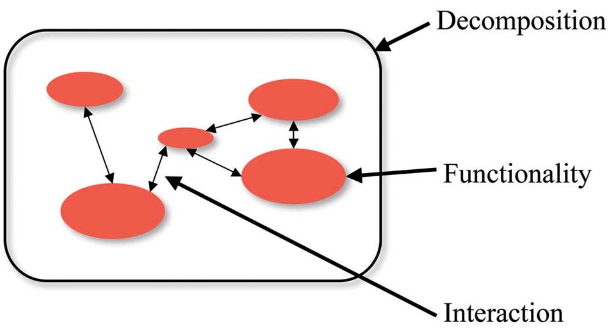


Fig. 1. ***Three aspects of decomposition into modules.***

II. BACKGROUND

Before discussing opportunities and challenges for componentization, it is helpful to define some terminology and give a general definition of the component.

A. Modularity

The component is a more ambitious form of modularity. As shown in Fig. 1, modularity [8] as a design construct (our perspective here) seeks to "divide and conquer" the design through decomposition of the overall system design into smaller modules. There are two primary objectives in this decomposition: weak coupling and strong cohesion.

Weak Coupling: Modules should have as little dependency on one another as possible. One design goal is a *division of labor*, or the ability to assign the design of separate modules to different individuals or groups, with a minimum of interaction among those groups. The weaker the coupling, the more each design can focus on local properties and ignore nonlocal or global properties.

Strong Cohesion: A module should group functionality that has strong dependencies. This supports another aspect of division of labor: *specialization*. The group designing each module can employ or develop special expertise and accumulate experience and thus deal with the design more efficiently and effectively.

Any decomposition has three facets shown in Fig. 1: identifying the participating modules, the functionality of each module, and the interaction among modules.

Each module's interface defines the capability of the module exposed to the outside world, within the context of a specific modularity. An interface has both an external and internal purpose. Internally, it defines what functionality has been promised to other modules and thus guides the module implementation. Externally, the interface defines what functionality is available to be invoked by other modules and thus guides the interactive portion of their design. A general goal of interface design is abstraction, or the hiding of unnecessary or irrelevant internal details of a module, including its implementation. Abstraction further reduces the dependency among module designs, especially as to their implementation choices. In ICT there are much richer possibilities (Appendix A).

The management and economics literature defines modularity in other ways, including the degree to which a system's elements can be separated and recombined [9] or the degree to which a system internally conforms to open standards [10]. For example, applying the first definition, an automobile tire is not modular because its elements (rubber and steel bands) cannot readily be recombined back into a tire after separation. An automobile wheel (tire and rim) is modular because the tire can be dismounted from the rim and later remounted. Applying the second definition, the wheel is modular because there exist industry standards for rim dimensions and other attributes that allow multiple manufacturers to manufacture tires for multiple rims. Although these definitions are not entirely suitable for modern ICT, they offer useful insights.

B. Composition

Modules are designed such that they can be composed with other modules in a system integration step. There are two distinct aspects to this problem [11]: interoperability and complementarity. Interoperability is the ability of modules to successfully invoke actions and exchange information using mutually agreed protocols. Complementarity implies module behaviors and functionality that is mutually reinforcing and works to common purposes. For example, consider a memory and processor. Their interoperability is manifested when the processor reads and writes data in memory at a specified address. Their complementarity is manifested by the ability of the memory to store program instructions, which are retrieved and executed by the processor.

C. Hierarchy

Choosing the modularity of a system involves a tradeoff between two granularities: fine grain (large number of small and relatively simple modules) and coarse grain (small number of large and relatively complex modules). Finer grain modules are simpler to implement and are thus preferred by module implementers. However, decomposition into a large number of fine-grain modules makes the interaction among modules voluminous and more difficult to understand. Thus, both system designers (those who establish the decomposition in the first place) and system integrators (those who make the modules work together after their implementation) prefer coarse-grain modularity because the interactions among modules are fewer and devoted to purposes directly related to system capability.

Hard choices can be avoided through hierarchical design. A hierarchical modularity is illustrated in Fig. 2, in which modules are themselves decomposed into smaller modules. A similar modular hierarchy is characteristic of many types of complex systems and indeed seems to be central to managing complexity in natural and social systems as well as in ICT [12]. A system with hierarchical modularity can be viewed at different granularity's, from courser grain at the top of the hierarchy to a finer grain at the bottom. The goal is to constrain the interacting modules at each level to an understandable number while avoiding constraints on the total number of modules. System design is top-down: first the coarse-grain modularity is established, and at each successive phase the next level of hierarchy is established by decomposition of the modules at the next higher level. System implementation, on the other hand, is bottom-up: Only the modules at the leaves of a hierarchy are actually implemented, while each module above is integrated from existing modules below, starting at the bottom.

Hierarchy offers a compromise between generality and specificity. The top level of hierarchy has functional requirements emanating from the system context, like a network or an enterprise accounting application. As we move down the hierarchy, the modules become less system specific, and more technology specific (for example, software, semiconductor packages, or communication links). While the system itself may not have either a single ownership or single supplier (think of a network), at lower levels of hierarchy both ownership and supply become singular. Thus, hierarchy offers architectural support for the realities of the market, where multiple suppliers contribute to a large complex system and contribute a variety of specialized knowledge. It also supports a supply chain in which suppliers themselves utilize suppliers (for example, semiconductor manufacturers who supply chips to equipment manufacturers, who supply equipment to systems integrators). The articulation of industry organization with hierarchical modularity is a critical issue discussed further in Section III-B.

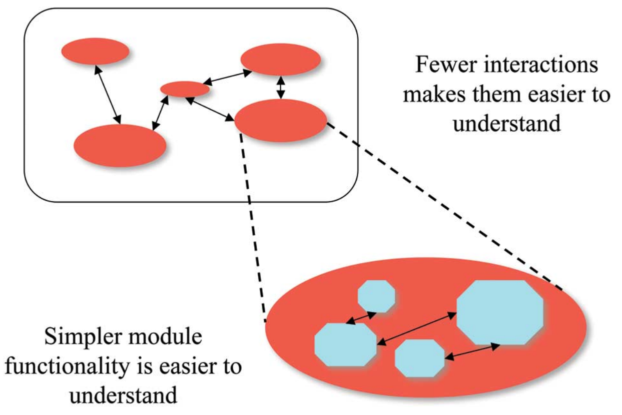

Fewer interactions makes them easier to understand
Simpler module functionality is easier to understand

Fig. 2. ***Hierarchical decomposition decomposes each module into submodules.***

D. Components

The degree to which a system's modules can be called components is measured in part by the ability to combine those components in different ways, mixing and matching components from other systems to realize entirely new systems. For example, an automobile wheel's tire and rim can be called components to the extent that rims and tires from different sources can be combined, and the argument becomes stronger as tires with different characteristics and purposes successfully mount on the same rim. This multisource property is enabled by industry standards for rim attributes (like dimensions and strength) that serve to coordinate the multiple rim and tire manufacturers.

From a design perspective, modules seek to admit implementation changes and functional improvements without affecting other modules. A component is a module, but its design seeks the more ambitious goal of opportunistic composition with other components, even where the components were not designed with specific knowledge of one another. Modules are designed for a specific use in the context of a set of other modules visible to the designer, and components are designed instead for multiple uses, including unanticipated ones. These and other design objectives are discussed in Section VI. There is not a hard and fast distinction between the component and modularity; it is a matter of degree, because there are inevitable tradeoffs.

Defining modules by decomposition results in modules suited specifically to the system's objectives and requirements. In contrast, a component taken from inventory or purchased or licensed from an outside firm and assembled with other components was not tailored to specific system requirements. The value of componentization increases when the many available versions of a component in the market are not perfect substitutes (that is, competing only on price and quality measures), but rather are differentiated functionally so that choosing among the components offers substantive differentiation in system purpose or functionality. The system design stage focuses on the question "what system functionality can be achieved by a composition of the available components." Hopefully, there is a wide variety of components available with varying functionality and characteristics, conveying a rich set of possibilities in system design. Of course, it is possible (and common) to mix components with handcrafted modules. This allows degrees of customization or ability to meet special system requirements not possible by component assembly alone.

In summary, components are designed explicitly for composition with other components, and this implies that the design of each component can be accomplished in a different time and place (and often business unit or firm) from other components. Composition of components is opportunistic, so the components can be designed without specific knowledge of one another. In organizational terms, components are designed without collaboration with the designers and implementers of other components, as is usually the case in modularity.

E. Reference Architecture

While one goal of component design is to divorce it from a specific system context, in many cases it is necessary to position a component within a targeted system architecture, which we term *reference architecture*. A reference architecture specifies sufficient details of the general system context to enable interoperability and complementarity with other components also targeted for the same reference architecture. The reference architecture thus provides at least the minimum level of coordination among component designers necessary to achieve composibility. One goal is to specify the minimum architectural detail necessary to get by, leaving as much freedom as possible to individual component designs.

Finer grain components may need a minimal application-independent reference architecture. An example is the composition of transistor components into a circuit, where interoperability follows from simple connections and specification of voltage levels, but complementarity (such as Boolean logic and timing) is left entirely to the specifics of the composition (that is, the pattern of connections). So, reference architecture can be as simple as "transistors are connected by wires with certain constraints." Coarse-grain components, on the other hand, typically require a reference architecture that is more context or application dependent. For example, the reference architecture for a communications network would acknowledge certain generic and necessary functions within networks in general, such as transmission, routing, switching, and so forth, but would stop short of specialization to a particular set of network objectives, such as delivery of telephony or data or video services.

Reference architectures can be hierarchical—a component can be an internal composition of components. The defined modularity at one level of hierarchy specifies a system context for each of its constituent modules. The system context for a module should exclude anything above the next-higher level of hierarchy.

F. Infrastructure

Infrastructure captures operational commonalities and makes these capabilities available separately to be used by any and all. Applications are a prime candidate for capturing commonalities and are typically built "on top" of infrastructure. Examples include processing and storage (the computer) and communication (the network). Infrastructure also includes tools such as (in hardware) computer-aided design and (in software) programming languages and compilers and human resources (trained experts who can execute designs).

The sum of technological capabilities assumed to be available to support the deployment of a component is collectively called the *platform*. In the software realm there are three competing component technology environments and platforms: CORBA Component Model (Object Management Group), Enterprise JavaBeans (Sun Microsystems), and .NET (Microsoft) [2].

Infrastructure plays an important intermediation role. For example, in personal productivity applications, "cut and paste" is intermediated by the operating system and would be difficult to implement within the applications because this would require coordination between each and every pair of applications exchanging information in this manner. Similarly, software component's interactions are typically intermediated by the platform to avoid a combinatorial explosion of direct interfaces that would have to be defined and maintained [11].

G. Lifecycle

It is useful to distinguish four phases in the life of a given component. The first phase is *development*, which includes both design and implementation. Objectives are established and a set of requirements and architecture is established in the design. Those requirements are matched to a specific technology in the implementation. The outcome of development is a *realization* of the component. A single component may have two or more independently developed realizations (for example by competing firms). The second phase is *replication*, in which copies of the component realization are created. These replicas may be identical or may differ as to configuration options (built-in alternatives left to be chosen at the time of replication). For physical components replication involves fabrication or manufacture, whereas in software replication is called instantiation (the replicas are called instances). Software instantiation is inexpensive (memory and processing cycles are all that are consumed) so that replication costs are insignificant relative to design costs. The third phase is *deployment*, where a component replica is introduced into an operational context. Deployment often requires and includes assembly with other components. The fourth phase is *maintenance and upgrade*, in which a deployed component is replaced by a newer realization that repairs design defects and adds functionality.

III. SOME RATIONALE

The rationale for componentization has changed dramatically with technology advances. The sources of value have changed, the realities of ICT have changed, and the fundamental concepts of componentization need rethinking to adjust to new realities.

A. What Has Changed

Why are components much less ubiquitous?

Large-Scale Integration: In pre-integrated circuit electronics, the component was a unit of manufacture and packaging—a component was manufactured all at once at the same place and was sold in a physical package much like the interchangeable part in mass production (Section V). Today, a unit of manufacture and packaging in ICT is usually at minimum a large complex subsystem or system encapsulated in a large-scale integrated circuit chip. Retaining a historical component definition makes little sense, for a couple of reasons. The component would be coarse grain, and for the most part only coarse grain, with both granularity and levels of hierarchy constrained by a changing technology. This definition would also rule out software components, which have a logical or informational realization and no physical "packaging."

Software: In the golden age of electronic components, software was relegated to large and expensive computers in the data center. Software is now a major element of everyday products. Today more and more functionality is implemented in software, in virtually all ICT products. Software is easy and cheap to replicate so "manufacture" in the historical sense is not an issue.

Outsourcing Alternative: In early industrial history (Section V), interchangeable parts enabled a move away from the vertically integrated firm (every aspect of a product produced internally) to a supply chain (multiple firms contribute manufactured content to a single product). In today's ICT, purchasing parts has been largely supplemented by purchasing outside services. For example, design, semiconductor fabrication, software engineering, and board- and equipment-manufacture are routinely outsourced. This is illustrated by Dell Computer, which has focused its successful business model on assembly and sales/fulfillment operations and has consciously minimized board- and chip-level hardware and software design [13]. These trends are encouraged by computer-aided design and manufacturing and by the Internet, which reduces delays and coordination costs. Outsourcing in these forms is not componentization: If supply firms customize their contributions to a finished product, as opposed to supplying the same capability to many firms, then this outsourcing is handcrafted.

Granularity, Scale, and Complexity: With increasing scales of integration and software, both the scale and complexity of our systems has increased dramatically. In this context, scale refers to the replication of similar or identical elements (like transistors on a chip), and complexity refers to many detailed interactions among heterogeneous elements. Scale is the "low hanging fruit" for components. It achieves economies of scale in both design and replication, since designing an element once and then replicating that design many times involves fixed costs that decrease with scale on a unit-cost basis. Components also contribute to a concise design representation of a large-scale system [12]. Thus, we see components like the memory chip that exploit massive replication of a storage cell. Complexity is a different matter. Our conception of components has not kept up with the increasing complexity of our systems. At higher levels of hierarchy, components represent intrinsically more complex (and hence more specialized and context-specific) functionality and are typically replicated fewer times. There is seemingly less pressure to componentize in this context if we stick with outdated rationale. For this reason, components have been largely relegated to the finer grain end of the hierarchy, while larger grain is mostly handcrafting territory.

B. Component Generalization

To fit modern ICT, the concept of the component should be expanded. In the following, some possible directions for generalization are elucidated. Two examples of components are invoked.

1) A communication *link*-as-component is an element of functionality that communicates a two-way stream of data between two geographically separated points. It is described in more depth in Section IV-A.
2) An end-to-end network *connection*-as-component is a composition of communication links together with intervening switches to communicate between any two network access points. See Section IV-B for more details.

Transparent to Physical Realization: A component need not be manufactured as a unit or correspond to a physical package. One package may contain multiple components or just a portion of one component. A component may reside in a single package, room, or it may be distributed geographically (for example, the network connection-as-component).

Element of Design: The component is primarily an element of design; that is, it is informational rather than physical in nature [14]. Of course neither do we rule out cases where a component has a physical reality.

Intermingled Implementation: Component implementations may be intermingled. For example, many components may share the same physical integrated circuit package, two software components may share common execution hardware, and two network connections-as-components may share a common communication link.

Dynamic Replication and Deployment: Components need not be static persistent objects. They may be *fleeting*, meaning they are replicated on demand and destroyed when no longer needed, as is often true of a software component. They may also be dynamically assembled on demand from static persistent subcomponents with that assembly (but not the subcomponents) destroyed when no longer needed. An example is the network connection-as-component that is assembled upon user request. It consists of assembled communication link-as-components, that assembly later destroyed when no longer needed, and where those links may be static and persistent. One implication is that a reference architecture may include specification of how components are dynamically assembled and/or replicated.

Transparent to Implementation: Implementation technologies are fungible to some extent, and their boundaries change with technology advances. For example, an ongoing trend is realization of fixed functions moving from hardware to software. The old concepts of hardware or software components should thus be replaced with a philosophy of "first define functionality, and then worry about implementation later, and allow the implementation to change over time." Different realizations of a component may use different technologies. In some cases, an implementation can incorporate nontechnological elements (for example, the free-space radio channel in a communication link) or functionality realized by humans as well as technology (see Section IV-D for more details).

Fragmented Control: A deployed component may cross multiple operational domains, where each domain has independent administrative responsibility and ownership, as illustrated in Fig. 3. For example, a network connection-as component will typically span two or more network service providers and two user premises. Operational control of the component is thus fragmented among multiple owners and operators. A good way to address this is by hierarchical decomposition into subcomponents, each of which is expected to fall within a single control domain. For example, a single network operator may control each individual link and switch in a connection.

Shared Design Responsibility: Historically, the responsibility for design of each component realization falls to a single firm. However, this is problematic for components with fragmented control, because it implies that all owners have to choose a common supplier. This contradicts the normal prerogatives of ownership, which includes the right to choose among competitive suppliers. Thus, as illustrated in Fig. 3, it is appropriate to allow the design, maintenance, and upgrade of a component to be a shared responsibility; that is, there is no single firm or point of contact with unequivocal responsibility to design the component or provide customer support or track and incorporate changing requirements (see Fig. 3). For example, each of the individual links and switches in a network connection may be independent procurement decisions.

Combining fragmented control and shared responsibility, a component may be a complex entity indeed, combining multiple suppliers with multiple owners. This

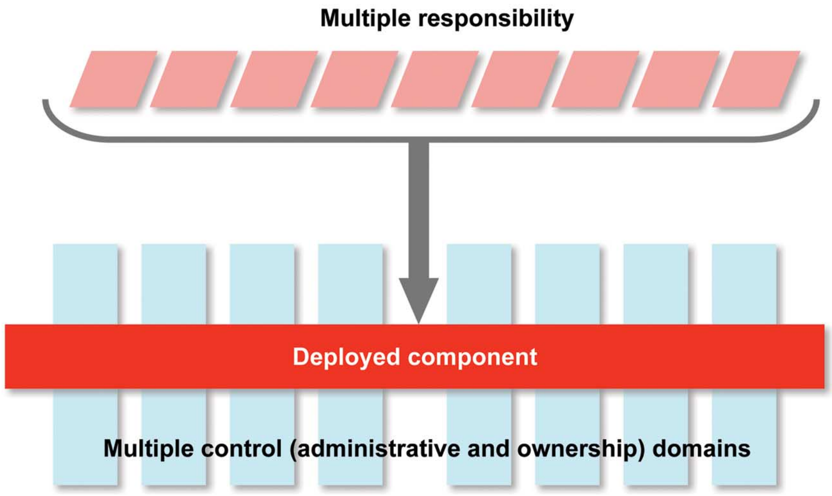


Fig. 3. *Deployed component can cross multiple control domains and encapsulate subcomponents with distinct design responsibility.*

may seem excessively complex and best avoided by narrowing the definition of a component to exclude these complications. Actually, it is an opportunity because the hierarchical component architecture can provide a structured way to coordinate these disparate players, as discussed in Section VI-B4. With the rising importance of facilities and applications shared by many entities, these are increasingly important issues.

C. Opportunities

What are the opportunities for componentization in the modern ICT context? A number of general industry conditions have been identified that drive particular firms and products towards greater modularity and componentization [9] and a number of business advantages and strategies have been identified [15]. Here, engineering and related industrial organizational questions are emphasized. As we see it, the component concept could and should enjoy much more widespread appreciation and adoption, with great benefits. Why?

Managing Complexity: ICT systems are among the most complex of industrial systems. ICT designs are largely unconstrained by physical limitations and thus strain the ability of people and organizations to cope with the increasing scale and heterogeneity of systems. It is telling that the most active component research community exists in software engineering [2], a discipline challenged by complexity. Like modularity, components provide an increasingly abstracted view of system functionality as we move up the design hierarchy. Complex-system theorists have observed the importance of "stable intermediate forms," or modules at a given level of hierarchy that stay fixed through an evolution of the system and often ultimately become building blocks of multiple distinct systems [12]. These stable forms greatly speed up the adaptation or evolution of the system and ultimately allow greater levels of complexity. Examples from biology include the cell (a structure that is replicated throughout most organs and tissues) and organs like eyes and livers (replicated through different animal species). When an ICT component is upgraded, there is strong incentive not to "break" systems already using that component; thus, the component should be open to extension but not to change. Thus, components can be described as precisely the "stable intermediate form" of ICT modularity. The role of componentization in complex systems is discussed further in Section VI-B4.

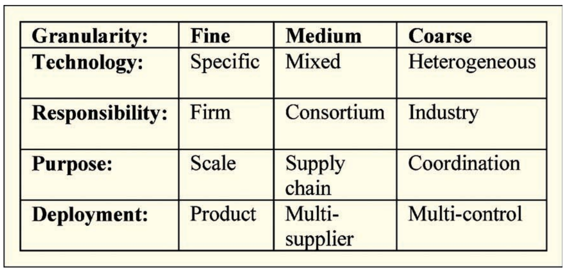
Specialization: Modularity offers a division of labor by system function. Large complex systems are constructed from different functional requirements, such as for example transmission and switching and routing in a network design. Specialization of firms to different functions is essential, since often no single firm (even the largest) can cope with the entire system. Specialization also allows firms to develop and exploit core competencies in specific technical areas, such as communications, databases, and processing. This can be accomplished through handcrafting in conjunction with outsourcing, but componentization allows a more disciplined and structured way to achieve functional division.

Division of Labor by Granularity: A hierarchical definition of components allows specialization by granularity. At the finer granularity, some firms can concentrate on a core technical skill, like software development or semiconductor fabrication and exercise that competency over a wider range of products (these are economies of scope) than would be possible in a vertically integrated firm. Some firms can focus on the coarser granularity, where overall system requirements and how the system meets end-user needs are paramount, and the relevant design skill is composition. Again, this specialization can be achieved with handcrafting, but componentization has other advantages.

Supplier Coordination: Shared design responsibility raises complicated coordination questions. How do we ensure that the products of different firms are compatible? How do we ensure that they work together in ways that achieve desirable system ends? How do we evolve these systems to meet ever more ambitious requirements, while coordinating the efforts of multiple firms? In fragmented control, how are competition and choice preserved simultaneously with compatibility? Answers must take into account complex organizational issues like the appropriate boundary of firms, as well as the appropriate demarcation between industry cooperation and independent firm contributions. A methodology based on hierarchical component design offers a systematic and structured way to deal with these issues, as discussed in Section VI-B.

The design and business challenges differ greatly from the fine-grain to coarse-grain levels of hierarchy as cataloged in Table 1. Finer grained components usually comprise a more homogeneous technology and thus are amenable to maximum specialization of firms according to technology competency (semiconductors, software, etc.). At coarser granularity, components increasingly mix implementation technologies but can also limit technology expertise by using assembly rather than implementation. As to responsibility, a single firm usually sources a finer grained component, but at the coarser granularity components may have responsibility shared across firms, who organize themselves (for example into a consortium). In the extreme of coarse granularity, responsibility for a component may reside in an industry as a whole (for example through an open standardization process). The motivation for defining a component thus shifts with granularity. Very fine-grain components mainly support concise representation of scale, while very coarse-grain components primarily support industry coordination at the system level. Medium-grain components are primarily

Table 1 Role of Component Granularity



Granularity:	Fine	Medium	Coarse
Technology:	Specific	Mixed	Heterogeneous
Responsibility:	Firm	Consortium	Industry
Purpose:	Scale	Supply chain	Coordination
Deployment:	Product	Multi-supplier	Multi-control

a structured way for multiple firms to contribute to a supply chain.

Product Diversity: Components are designed for multiple uses, and this allows one firm's product variety to increase by mixing and matching assembled components in different combinations and configurations [15]. This strategy also provides design support for versioning (selling different flavors of products with different functionality and quality at different price points), an important pricing strategy [16].

Design Quality: Functional defects, largely borne of complexity, plague ICT. Componentization results in fewer distinct designs, and the incorporation of those widely used designs into more products. This affords more opportunity to identify and remove defects and a greatly expanded opportunity to identify defects in actual usage rather than artificial test scenarios. This can also result in greater scrutiny of security issues.

Time to Market: With shortening product design cycles the time required to complete design, implementation, and testing become important to profitability [17]. By reducing handcrafting and allowing new products or versions of old products based on mixing and matching components, time to market can be reduced [15].

Design Flexibility: A weakness of many current ICT systems is their rigidity or inadequate ability to track and evolve with changing needs. Superficially, handcrafting should be effective at achieving flexibility because it offers unfettered design freedom. In practice, components have considerable potential to improve designs in this dimension. Why? First considerable inflexibility in handcrafting arises from decomposition because module designs become tightly bound to initial system requirements. The shortest route (in terms of time and cost) to realization is to closely tailor the design to those perceived system requirements. The result tends to be a design heavily tailored to present (not future) features and functionality. How can components help? The component design methodology in Section VI-B explicitly generalizes component design requirements, attempting to meet the needs of multiple products or systems. As a result, the component is methodologically less tailored to specific system requirements, and evolution becomes easier.

Economies of Substitution [18]: These occur when changing needs can be satisfied more cheaply and faster (from the design and operational perspectives) by replacing or upgrading some components (rather than the entire system). For example, the makers of enterprise software business applications realize that componentization is necessary to meet the diverse and changing requirements of their customers in a timely fashion [19].

Operational Flexibility: The component is an independent unit of distribution (Section VI-A5), meaning a component can be individually replaced and upgraded in an operational system. This allows a graceful upgrade of a system with less risk and unintended side effects. For example, consider the component home theater system. Individual components (receiver, display, storage, etc.) are routinely replaced, and new products sometimes add fresh new capability with no need to replace the rest of the system.

Design Costs: Increasingly, supplier costs are driven by the people-intensive parts of the business, like design, testing, customer support, etc., rather than replication costs (particularly for software). The significance of design costs also increases with increasing product diversity and redesign rates. Assuming an appropriate set of components is available, a firm can lower these costs by assembly (possibly including proprietary modules) rather than handcrafting.

D. Challenges

We should acknowledge that components have disadvantages and challenges as well.

Design Costs (Again): If a firm is itself creating and marketing a component, there are added costs associated with meeting a more diverse set of requirements and supporting and maintaining its product for a more diverse set of uses. These extra costs must be amortized across an increased number of uses (volume of sales) as well as higher prices enabled by increased value to the customer (see Section VI). Since these increased sources of revenue are uncertain, there can be more risk.

Replication Costs: Components incur overhead, for example in abstracting interfaces and offering configuration options. This can increase hardware complexity, processing cycles, or memory, and more generally replication costs. There may also be adverse performance implications and increased recurring costs such as power consumption. These costs are not an obstacle as long as they are exceeded by the benefits of componentization described earlier.

Innovation: When component upgrade assumes compatibility to existing systems, this can discourage innovation. Fortunately, ICT has rich capabilities such as metalanguages and mobile code, which can partially offset this constraint (Appendix A), and componentization does open up new avenues for innovation at the system level (Section V-B).

Competition: In the face of competition, firms attempt to increase market share and extract higher prices through differentiation from competitor's products. Component suppliers and their customers both face challenges in this regard. Components that have open interfaces (that are documented and can be used or created without intellectual property restrictions [11]) are attractive to customers who gain the opportunity to mix and match components from different suppliers. These components may attract competition, and there are limited opportunities for multiple component suppliers to differentiate their functionality since they must co-exist in the same uses. Competitive advantage will focus instead on quality, price, and other characteristics, such as security and scalability. There may also be the opportunity to offer extended capabilities (but customers will be aware that invoking these capabilities reduces their future options to switch suppliers). Other firms who acquire and adopt components to incorporate into their products also forego one possible source of differentiation, since their competitors have the same components available under similar terms and conditions. On the positive side, for both suppliers and customers componentization focuses attention and effort on improvements and extensions, reducing the temptation to re-implement what already works.

Vulnerability Borne of Homogeneity: To the extent that components result in less diversity of designs, this in itself can expand the scope, subtlety, and impact of defects that remain. For example, AT&T experienced a severe network outage attributed to its widely replicated homogeneous software [20]. Security holes can also be more widely exploited and are more likely to attract the attention and effort of crackers in widely replicated components.

E. What Components are Not

Some other well-known common design concepts overlap components. To understand the component, it is instructive to address how they differ.

Modularity (Defined in Section II-A): Components are an ambitious form of modularity as discussed in Section II-D. In addition, the relaxed assumptions outlined in Section III-B are not generally recognized elements of modularity.

Infrastructure (Elaborated in Section II-F): The component emphasizes composition, while infrastructure emphasizes extension. Infrastructure is available and known at application design time, so the application can take into account full prior knowledge of its interface and capabilities.

An example of infrastructure is a microprocessor. Typically, software is designed for a specific microprocessor, assuming full knowledge of the details of the instruction set. This strong coupling of software and microprocessor argues against any modularity that separates software and hardware. An adaptor approach has been successfully used (see Appendix A). For example, the

Java virtual machine [21] is available and widely deployed for several operating systems, providing a ubiquitous and homogeneous platform for application software running across different platforms.

Although an infrastructure is not in itself a component, infrastructure can and should incorporate components. Think of infrastructure as a distinct large-grain module, the purpose of which is to serve its "users," who are in turn the applications (and their developers). This infrastructure certainly incorporates modularity internally and thus can incorporate components. In this sense, the microprocessor and memory chip are (internal infrastructure) components.

It would be an oversimplification to presume a stable boundary between infrastructure and applications. Like natural systems, which grow complexity by composing new subsystems and systems from existing stable intermediate forms [12], ICT infrastructure grows by adding new layers of hierarchy (usually called layers), some of which started life as applications. For example, the Web began as a simple information access application within a large end-user organization [22] and has been upgraded and extended to become an infrastructure supporting a variety of new distributed applications. This process can also morph an application component into an infrastructure component.

Reuse: Technology reuse attempts, in the context of one project, to design modules that can be reused in other projects. Those other projects are typically executed within the same firm, so reuse is an attempt to achieve economies of scope within a single organization. Significantly, reuse typically allows the design to be modified in the context of subsequent projects. Each project that reuses a module may need to "fork" a new design that for the most part has to be thereafter independently maintained and upgraded, foreclosing future economies of scale and scope.

IV. SYSTEM COMPONENTS

Enthusiastic "component communities" generally fall within the hardware and software subindustries and within application and infrastructure industries. These are, however, artificial distinctions that limit the potential of componentization, particularly at the coarser granularities, which naturally incorporate diverse technologies. To maximize value, component modularity at the design phase should be considered strictly in terms of functionality provided to its system context at the next higher level of hierarchy. Secondary considerations such as implementation (hardware or software or assembly) and physical proximity and the appropriate shared design responsibility or fragmented control should be subjugated to the design considerations of internal modularity and addressed at the next lower level of hierarchy.

Systems components, which abandon the traditional classification into hardware and software, fall into four general categories that illustrate the generalizations offered in Section III-B. These combine hardware and software, or are geographically distributed, or bundle application functionality with supporting infrastructure, or combine technology with social elements. These categories are now described with examples, and component design is considered in Section VI.

A. Hardware–Software Components

A hardware–software distinction, forcing components to be exclusively one or the other, can be deleterious because the system context demands both—hardware is required, at minimum, to execute the software and interface to the physical world. Emphasizing the overall functionality, hardware–software components are very natural.

Historically, computing has been deeply molded by a cost structure that favors the sharing of processing resources among different applications. This statistical multiplexing, also prominent in networking, is particularly efficient when there are varying processing or traffic demands, which have reduced variation when aggregated [23]. A negative implication at deployment is acquiring equipment and software independently and integrating them, and this creates many challenges for both customer and supplier. For example, the supplier wishing to maximize market opportunity has to target multiple platforms and configurations.

As the cost of hardware has decreased dramatically, it is increasingly common to dedicate hardware to a single purpose rather than sharing it. An example is embedded computing [24], in which a product has software embedded within it for purposes of control or interaction, and that software is not separable or sold independently. The increasing ubiquity of broadband networking also encourages equipment-software bundling, through the software as a service (SaaS) model. When software capabilities are accessed over the network, the supplier can target a single uniform platform, and the user is freed of many responsibilities (such as installation and administration). Web services explicitly support the componentization of services, through the so-called service-oriented architecture (SOA) [25]–[27]. Such Web services are hardware–software components.

This identifies at least two distinct ways to treat ICT components. One is to lump hardware and equipment resources as part of an infrastructure and build exclusively software components that extend this infrastructure. Another is to encapsulate supporting hardware resources within the component realization itself, and this is increasingly economic due to declining hardware costs.

A communication link-as-component example is shown in Fig. 4. The goal is a self-contained link that composes without modification into a variety of system contexts and

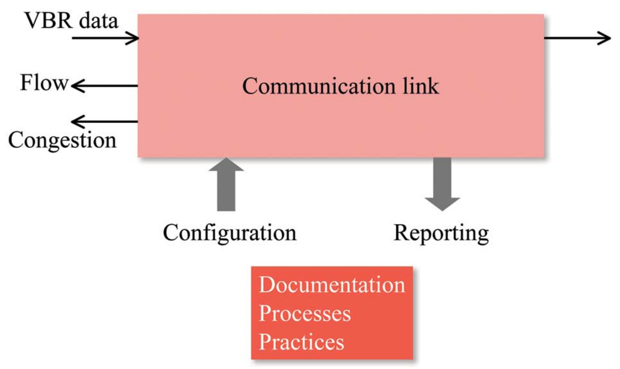


Fig. 4. ***Elements of communication link-as-component.***

can be substituted freely without regard to underlying technology. To accommodate different data rates, and also physical channels whose rate depends on traffic conditions (such as a wireless media access layer), the link accepts variable bit-rate (VBR) source data and has flow control to adjust the input rate to available resources and also signals congestion status to the sources (roughly indicating the impact of this link's traffic on other traffic sharing common resources). At the assembly phase, a link can conduct a negotiation of quality-of-service (QoS) parameters (such as delay-reliability tradeoffs) and establish a service level agreement (SLA). The link also incorporates reporting functions, such as malfunctions. Note that this component's implementation encapsulates not only hardware and software but also a physical communication medium.

Functions normally thought of as network system functions (like flow control, buffering, and congestion control) are included within the link, illustrating that considerations of functionality in the interest of component composition may result in nontraditional forms of modularity. The inclusion of transceivers at both ends of the medium, and the medium itself, is also different from most current practice. At the time a link is replicated, this may necessitate an internal dynamic assembly of subcomponents, such as the transceivers participating in the link. In that composition, these subcomponents may need to negotiate QoS and other parameters. The link also has a social aspect (Section IV-D) in the form of documentation and various standard processes and practices that surround its deployment and operation.

B. Distributed Components

A component should not be constrained in physical centricity—it need not be contained in one chip, one package, within one rack, or be physically constrained to one locale.

An example of a distributed component with shared responsibility and fragmented control is an end-to-end network connection-as-component, which is dynamically replicated upon demand with the aid of separate routing capabilities. It offers services appropriate for different media and different quality and performance parameters. The connection has a similar interface to the communication link-as-component in Fig. 4 but requires an internal composition of multiple links and "switching" components. A connection may overlap control domains and thus must consist of a dynamically created assembly of subcomponents with fragmented control.

Why create a connection-as-component and thus have to deal with the complexities of fragmented control in this component's design? The connection is a coherent body of functionality with strong internal coupling; for example, constituent links share common traffic demands and must jointly participate in achieving QoS parameters and provisioning an SLA. Useful services may require opportunistic composition among a connection and other components. For example, a videoconferencing application requires a video codec component at each end. Finally, whether or not the connection is treated as a component, some industry coordination process must be responsible for a reference architecture and protocols for connection establishment and takedown; the hierarchical component architecture offers a structured framework to accomplish this (see Section VI-B).

How does the connection-as-component differ from an internet TCP/IP connection [23], which has some superficial similarity? TCP as a protocol rather than interface focuses on implementation. The interface is called an internet socket, an entity closer to one end of the connection-as-component. Unlike a socket, which requires specification of implementation details such as the protocol, a connection-as-component offers an abstracted interface focusing on communication needs (quality of service parameters) and satisfies those needs transparently to composed components (like codec's) through access to a whole suite of transport services that can be improved and expanded over time. A connection encompasses both end points and everything between, unlike the socket. Internally, care is taken to modularize a connection both vertically (for example transport and congestion control are treated as separate but interacting components) and horizontally (communication links and switching functions). This modularity is defined hierarchically, with no consideration of industrial or technology issues until the lower levels. Each module is generalized; for example, congestion control's interface does not presume any specific mechanism (such as TCP) but attempts to capture many or all possibilities. How is componentization advantageous? Besides the general opportunities listed in Section III-C, the hierarchical discipline affords a structured framework for dealing with industry coordination issues in a multi-vendor environment (Section VI-B4).

C. Application-Infrastructure Components

As in the case of hardware–software, a component need not fall strictly within the application or infrastructure space. Rather than consider infrastructure interoperability as part of a component interface, it can be advantageous to logically encapsulate supporting infrastructure within a component. This allows the component designer complete control over the infrastructure support and frees the component user from integration of application and infrastructure. Of course, the component may be internally assembled from separate infrastructure and application components. Web service components are an example of this, in which the application-infrastructure demarcation is invisible at the service interface but may appear in internal modularity at lower levels of hierarchy. The application-infrastructure component is related to the information appliance [28] (for example, the portable music player), the difference being that the information appliance is a packaged application while an application-infrastructure component is intended to be composed into an application.

D. Social-Technology Components

At the highest level of technological hierarchy, the system context is often social. That is, at the top of the hierarchy the interfaces often are to humans and organizations [14]. Just as the economy is embedded within a social context with personal and organizational relationships [29], so are many technological systems, not unlike how software is often embedded within an equipment context. The larger system design issue becomes to assign appropriate roles to the social elements (such as decision making and exception resolution) and to the technological system (such as information management) in accordance with their respective strengths. Social systems are subject to a similar analysis of modularity [30] and separation of state and process [12] as technological systems. Even professional businesses that emphasize judgment and expertise (such as legal services [31]) have more routine aspects subject to systematic or repetitive processes that can be automated. Thus, there is the opportunity to use similar methods of analysis and synthesis for the social as for the technological system elements.

A social-technology component definition specifically incorporates presumed processes executed by humans. The component implementation includes human and organizational interfaces, presumed processes within the social context that are coordinated with and easily compose with the technology, and human-targeted documentation and training materials. These elements can fruitfully be considered not only a part of the component design, but also a part of the component itself. That is to say, the component realization includes social as well as technological processes, with similar options for configuration and extension.

An example of a social-technology component is an insurance policy-as-component. An expansive definition could include (among other things) the sales and underwriting processes, where human participation is important. The sales aspect captures information about the customer's needs, risk factors and ability to pay, with explicit support for collection of the needed data from the customer (often using a sales agent as intermediary) as well as other sources. The underwriting process relies on human judgment, informed by a process for efficiently capturing and presenting relevant information to an underwriter and capturing his or her insights and conclusions. The component includes a definition of the organizational interfaces and processes, including explicit flexibility and configuration options, as well as bundled (and often online) documentation and training aids. In essence, the component incorporates human (or more complex organizational) entities into its definition.

Suppliers of software applications serving business processes, driven by customer needs and expectations, are increasingly recognizing the need for componentization (an example is enterprise application suppliers [19]). Although they make implicit assumptions about the human elements of the processes, they have not yet taken the step of explicitly incorporating people (and supporting elements like documentation and training) within the component scope.

V. HISTORICAL PERSPECTIVE

In this and the following sections, we address the organizational challenges of invigorating componentization in the ICT industries. The component is an old idea, with its genesis in an even older idea, the interchangeable part. The latter was a key innovation in mass production for the munitions and automobile industries of the 19th and 20th centuries. A brief historical review serves to illuminate some of the challenges faced in establishing a component design culture.

A. Interchangeable Parts

By the 1720s, Swedish inventor Christopher Polhem was manufacturing clocks with interchangeable parts, although not on an industrial scale [32]. Gunsmith Honoré Blanc demonstrated the idea in Paris, France, in 1790 and proposed that the French armament industry widely adopt this method for arms production. Blanc's immediate objective was a limited one: battlefield repairs could make the Army more effective. The implications turned out to be far wider, following decades of further development and complementary innovations.

Despite the enthusiasm and direction of Napoleon (himself an artillery engineer), France failed to establish the interchangeable part in armament production largely due to resistance or inaction by existing munitions suppliers [33]. A major reformulation of an industrial process is difficult to achieve, for numerous reasons. Everyone throughout the process, from design and manufacture through procurement, is accustomed to the previous way of doing things, and change requires a change in mindset. Reorganization of the whole industry, including the organization of the factory and the introduction of whole new professions and processes, such as inventory management and accounting, is daunting. Among firms, there are winners and losers, and the potential losers are resistant. In an industrial supply chain, an optimization of overall efficiency requires some to increase costs (for example, the higher design costs of interchangeable parts) in order for others to reduce costs (for example, the manufacturing assembly).

Thomas Jefferson (third President of the U.S.) encountered the interchangeable part idea in France and encouraged the U.S. Army to adopt it. Rather than impose it on suppliers, for approximately six decades the idea was the subject of research and development and factory floor experimentation at the U.S. Armory in Springfield, MA [34]. A series of incremental advancements and inventions accumulated and gradually moved the idea toward widespread practical realization. By 1860, new armament vendors were organized around this approach, such as Colt Manufacturing and Smith & Wesson. This illustrates several points. First, disruptive innovations (that require major reorganization of industry and changes to business plans) often require considerable precommercialization research and development before they are ready for commercial use. It is difficult for profit-maximizing enterprises to make such an investment, so it is more likely to be government funded. Second, it reinforces the role of new companies in bringing such innovation to market. Third, there are winners and losers, and potential winners must take affirmative action to overcome inevitable inaction or resistance by losers, or new players need to displace the losers.

All these issues apply to ICT, but a fourth lesson is especially relevant. The culture of engineering design methodology (perpetrated in part through the educational system) interjects considerable inertia. As one historian commenting on France's experience asserts [33] "...engineering rationality is not a set of timeless abstractions, but a set of social practices which have emerged historically." It is worth reflecting on the degree to which current ICT design methodologies have a "cultural imprint," as opposed to being based on fundamental principles or unavoidable physical constraints. A handcrafting design methodology in ICT may be preordained by the difficulties of componentization or may simply have evolved historically as constrained by older generations of technology or as the path of least resistance. Our thesis is the latter.

B. Combinatorial Innovation

Once a set of interchangeable parts is available, there are many opportunities for innovation in assembling them in novel and unexpected ways, a phenomenon that has been termed *combinatorial innovation* [35]. Industrial history is ripe with examples, such as the first successful flying machine constructed by the Wright brothers largely from parts drawn from the bicycle (wheel), kite (lightweight structures), automotive (engine), and marine (propeller) industries. Significantly, the availability of parts (or components) significantly reduces the barriers to systems-level innovation, reducing the skill level and needed financial and organizational resources. Combinatorial innovation is and can be even more beneficial to ICT which, as mentioned in Section I, is experiencing notable barriers to systems-level innovation.

C. Fitting

Today's vision of a straightforward assembly of interchangeable parts did not reach fruition until 1913. Earlier parts were remachined in order to match and assemble with other parts; this is called *fitting*. Fitting prevented realization of the full potential. The division of labor was not fully realized—assembly required skilled craftsmen and was itself partially a handcrafting enterprise. Both the skills and tools required for fitting were largely absent in the field.

Fitting is pervasive in ICT today, often arising under terms such as "programmability" or "configuration," or "integration." Programmability extends functionality to fit a part (such as a microprocessor) into its system context. Configuration entails less work than programming but also confers less flexibility since useful configuration options have to be anticipated in the design. Integration of subsystems frequently involves not only configuration and assembly, but also modification and extension. Another example of fitting is software reuse, which usually allows modification of old modules to fit new uses (Section III-E).

The assembly line was the major turning point. It structured the assembly of parts into a sequential process in which each assembly step remained stationary while the partially assembled product moved down the line (the "pipelining" of digital-system architecture.) Elemental forms of the assembly line go back to the early 19th century England, but the concept was perfected by Ford Motor Company in 1913 [34] and in 1926 Henry Ford famously announced to the world [36] "In mass production there are no fitters." The efficiency of the assembly line required a consistent and nearly equal time for each assembly step and was thus incompatible with fitting. This, in turn, required much greater discipline in the design and production of parts, making them truly "interchangeable without modification" for the first time.

History suggests that fitting is difficult to stamp out and takes research, commitment, time, and investment. The rewards can be profound, as evidenced by the historical examples of combinatorial innovation and by the absence of fitting in today's mechanical industries.

The assembly line introduced a new challenge, which was production volume easily outstripping customer demand. The early General Motors Corporation through marketing innovations like product variety and differentiation and frequent model changes addressed this most effectively. These marketing innovations required considerable production flexibility, and the interchangeable part played a crucial role in reducing design and retooling costs by increasing commonality among a set of superficially distinctive designs. Flexibility in design is certainly a big issue for ICT too, particularly in applications serving business, and componentization should assist as long as a certain vision of future needs is incorporated into a component design.

D. Standardization

A later major innovation in the automobile industry was parts interchangeable across different manufacturers, creating a supply chain. This moved the industry to a division of labor in which firms (not individual craftsperson's or divisions of a factory) specialized in the production of particular types of parts. A related innovation was a "platform" or common framework within which interchangeable parts articulated with one another, requiring industry consensus on general design issues (like how the steer the car). This platform is analogous to a reference architecture in ICT. Industry standards provided the coordination among assemblers necessary to share common parts suppliers. The first successful industry standards in automobiles were not promulgated by firms or by consortia of firms, but by a professional organization, the Society of Automotive Engineers (SAE). (There are earlier examples of industrial standardization, such as the railroad gauge and screw thread [37], [38].) The largest companies were lukewarm, but the smaller companies encouraged and supported standardization because this allowed them access to technology and designs that they could not otherwise afford. Similarly, componentization in ICT would likely encourage more small firms. Standardization in autos was promulgated by an outside engineering professional organization, whose members had more of an industry perspective than the managers of firms in the industry. This suggests that activism on the part of government and professional organizations like DARPA, NSF, IEEE, and ACM would be fruitful in ICT.

One might conclude that an industry like automobiles moves inevitably from handcrafting to interchangeable parts and then to a supply chain with increasing specialization, but history does not bear this out [37]. We have already observed that componentization has languished in the ICT industry. In the automotive industry there were 88 manufacturers in the U.S. by 1921, and this proliferation was encouraged by the availability of interchangeable parts from independent suppliers. However, the industry experienced consolidation, probably not as a result of design and production efficiencies *per se*, but as a result of economies of scale, branding, and the strength of dealer and service networks. While industry standards in areas like lubricants and fuels are prominent to this day, standards for parts like hinges and steering wheels have largely disappeared. While outsourcing of parts suppliers have consistently played an important role, the few remaining auto manufacturers have alternated between greater vertical integration and divestment of their internal parts production (Delphi from General Motors and Visteon from Ford Motor are recent examples).

The boundary of the firm is explained in economic terms by the relative transaction costs for internal coordination as opposed to market purchases [39]. Componentization eliminates most coordination costs among component suppliers and thus tends to move an industry toward smaller firms and a supply chain, all else being equal. On the other hand, other factors can favor the consolidation of an industry into fewer large firms, and if this occurs it discourages componentization, in part because componentization encourages competition from smaller firms (as discussed in Sections III and IV). Our view is that the benefits of componentization in ICT, such as dealing with large-scale system complexity and the opportunities for combinatorial innovation, as well as pressure applied by customers and users, can make componentization compelling to the largest of firms.

VI. DESIGN FOR COMPONENTIZATION

The previous discussion of the opportunities for componentization begs the question of where components come from. This is the issue of component design. The discussion surrounding Table 1 concludes that the component can play an important role over a range of granularities. However, the design objectives and methodologies are quite distinctive over this range. Fine-, coarse-, and medium-grain designs are thus discussed separately.

A. Finer Grain Components

In the categorization of Table 1, finer grain components are distinctive in that they are the responsibility of a single firm, which typically faces competition from other firms offering a similar component. Rather than discuss component technology (such as tools and platforms), the emphasis here is on the sources of value, based in part on [11]. Component designers should be acutely aware of value as the major driver when design decisions and tradeoffs are faced. When evaluating any module for component-like characteristics, these sources of value can also offer insight into shortcomings and possible improvements.

Since the reference architecture that defines the system context of a finer grain component is the coarser grain component at the next higher level of hierarchy, discussion of architectural design is deferred to Section VI-B.

1) Multiple Uses: Components are designed and sold for multiple uses. What sources of value does this impart?

Economies of Scale in Design: The cost of designing a component is considerably larger than an element designed for a single specific use, principally because understanding and anticipating multiple uses takes more time and effort, not to mention the additional costs of marketing and sales and support associated with multiple customers for the design. Fortunately, those larger costs can be amortized across a larger number of unit sales. This balance only makes economic sense in cases where the number of uses is large enough to more than offset the higher costs. This is economies of scale, lower unit costs as unit volume increases.

Economies of Scale in Replication: If a component is fabricated or manufactured, multiple uses results in larger production volume, spreading various costs over larger volume. However, as with design, the generality of a component may increase its unit fabrication or manufacturing cost. Thus, to make economic sense the customer willingness to pay must account for this higher unit cost. In modern ICT, there is another opportunity that is subtler: a component design can be specialized as it is replicated. That is, a component design can embody great generality, but as that design is translated (in automated or semi-automated form) into a concrete realization, extraneous features that might otherwise lead to higher replication costs can be eliminated. Thus, no value is lost through greater context independence in the design than the realization, especially if the translation is automated.

Lower Barriers to Entry: A component supplier seeks and benefits from the larger market and sales volume inherent in multiple uses. The consumers (buyers) of a component seek lower design or manufacturing costs than they could achieve by internally designing and manufacturing. The more they can rely on components in a new design, the more their barriers to market entry (such as the lengthy development time and cost) are reduced, but also the lesser their opportunity to differentiate products from competitors. Shortened time to market is a particularly strong motivator, since much of the profitability of a product accrues before competition is able to arise.

Other sources of value include flexibility, quality, and lowered vulnerability (Section III-C).

2) Composition: Composition achieves a system's higher purpose by assembling two or more components that successfully interact and their functionality is mutually reinforcing. Opportunistic composition achieves this goal in spite of the components being designed without specific knowledge of one another. What are some sources of value?

Mixing and Matching: Opportunistic composition leads to many opportunities to mix and match components, achieving different system functionality, cost, or performance characteristics.

Competition: Mixing and matching offers more supply options to the customer and greater competition among suppliers. It shifts the granularity of competition from the system level to the component level.

Component Diversity: The greater the diversity of available components, the more opportunities there are for composition and the greater the potential value.

Other sources of value include division of labor, lowered barriers to entry, reduced time to market, and greater system flexibility and diversity (Section III-C).

3) Context Agnostic: A component design is context agnostic when no specific system context is used to define its design requirements and parameters. This is a matter of degree, since only very fine-grained components (like transistors) can be almost completely agnostic.

Agnosticism is superficially similar to multiple uses but more powerful because the multiple intended uses of a component could still be quite context specific. For example, there might be several systems suppliers of GSM cellular network equipment and a number of service providers who purchase this equipment and deploy GSM cellular networks. A component designed to support those multiple deployed systems is multiple use but relatively context specific because it targets a voice telephony application. A component designed to be incorporated into any network (GSM, internet, video distribution or other) is relatively context agnostic.

System Experimentation and Diversity: Context agnosticism is attractive to a component supplier because of the larger market opportunity. But to the larger society and end users, it is especially attractive because it lowers the barriers to deploying a greater diversity of systems. For example, a more componentized networking technology in which the components were relatively context agnostic would encourage greater market experimentation with different networking system concepts and presumably better and more diverse (and yet interoperable) networks as a result.

4) Encapsulated: A component is encapsulated when it is designed to be used "as is," without modification. This is the elimination of fitting (Section V-C). Generally, this is achieved by presenting a component interface to the outside world, an interface that offers a specific well-documented set of services and capabilities, but which is designed to hide internal implementation details. For a physical component, encapsulation may be enforced through impenetrable packaging, but often encapsulation is enforced by licensing terms and conditions or intellectual property rights (trade secrets and copyright). How is encapsulation valuable?

Customer Costs: Pure assembly eliminates the time and cost of design or design modifications.

Multiple Uses: Encapsulation enforces a design discipline that enhances the multiple use property. Where an encapsulated component does not fully meet the needs of a specific use, the customer communicates new features and requirements back to the component supplier. As the supplier incorporates these additional features and requirements, all current and future users can benefit from these enhancements. Thereby, the effort of a single user, together with the supplier, provides value to the entire community of adopters and further expands the use opportunities.

Economies of Scale in Upgrade and Support: When users modify a component for their own purposes, a new design for that component has been "forked." Thereafter, that new design cannot benefit easily or fully from future maintenance and upgrades, because there are now two inconsistent designs to deal with. Also, the component supplier will subsequently find it difficult or impossible to separate features or defects introduced in the original design (as opposed to the user modifications). Thus, encapsulation adds value through efficiencies in user support and through the lowered friction in the propagation of upgrades and fixes through all deployed components.

Separation of Responsibility: Encapsulation enforces a separation of responsibilities between supplier and user of a component, and more importantly among the suppliers of distinct components. The supplier is fully responsible for the design and the implementation and can be held accountable by the user. Absent encapsulation, suppliers of components will generally not offer user support, and as a practical matter cannot offer a warranty that guarantees capabilities or performance parameters. Encapsulation provides value to users through this shift of responsibility to the supplier.

Separation of Uses From Technology: Since the user of an encapsulated component is freed from awareness of its implementation, the adopter can focus completely on the uses of the component rather than the technologies incorporated into its implementation. The user is freed from the need for technological expertise, and the supplier of the component is freer to change technologies and implementations in the upgrade process transparently to the uses.

5) *Independent Unit of Deployment:* A component is an independent unit of deployment if it is acquired, provisioned, and installed independently of the other components and elements of the system. This is valuable for several reasons.

Extendibility: If a component can be added to an already operational system, the functions or capabilities of that system are extended.

Independent Upgrade: If a deployed component can be replaced by an upgraded version without adversely interrupting or impacting its system context, the system can be upgraded incrementally without wholesale upgrade of the entire system. This may extend the functions or capabilities of the system, or it may repair defects. It also may entail lower risk than wholesale upgrade and can be reversed more easily and quickly if problems arise.

Serviceability: Particularly applicable to a physical component, repair of a system in the field is facilitated if a single malfunctioning component can be replaced.

6) *Design Tradeoffs:* There are inevitable disparities among design objectives, and inevitably they must be traded one against another. Any existing or proposed component can be compared against the value metrics, revealing what tradeoffs have been chosen and how its value may have been affected.

For example, composition and context agnosticism are in opposition. Composition requires complementarity of function and purpose, but complementarity is distinctly contextual. For example, in a network, routing and switching are complementary (routing configures the switching), but this complementarity occurs within a specific context (the network). Thus, for these specific functions "multiple uses" should be qualified to read "uses in multiple networks."

Another tradeoff occurs between multiple uses and encapsulation. The discipline of not allowing modification of the design in a multiple use context implies that the needs of the multiple uses have to be more fully anticipated. This is certainly desirable, but pragmatically the ability to modify or extend the design will expand the feasible uses. As well, expanding the capabilities of the component in order to specialize it to particular uses may require choosing among a set of configuration options, a relatively benign form of fitting.

B. Coarser Grain Components

Coarser grain components can serve industry and firm coordination purposes as cataloged in the discussion surrounding Table 1. From an industry perspective, these components may serve a coordination function to deal with multiple responsibility and fragmented control (Section III-B). From a design perspective, these components define the system context that motivates a reference architecture for the finer grain components that are encapsulated within. The design of coarser grain components is thus more about the design of componentized architectures and systems than about the design of components *per se*. Thus, this section emphasizes the design of reference architectures and industry processes that might surround the component as an industry coordination mechanism.

Many of the sources of value for finer grain components discussed in Section VI-A are less relevant to coarser grain components and should therefore be approached with caution. For example, the system at the top

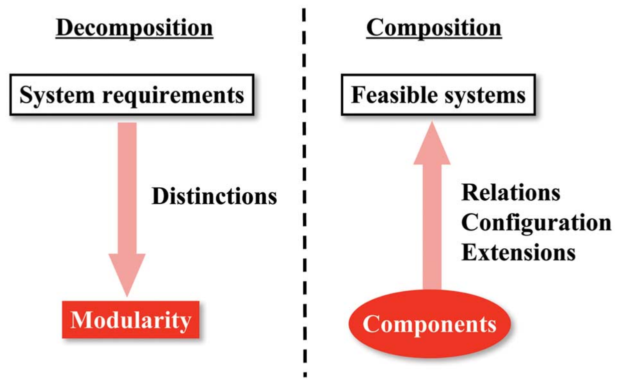


Fig. 5. ***Two diametrically opposed system design methodologies: decomposition and composition.***

level of hierarchy need not compose with anything. Toward the top of the hierarchy, components tend to become more context specific. A coarse-grain component whose design is the responsibility of an industry standardization process is certainly not encapsulated, and many of the value generators of encapsulation (like warranty and customer service) are irrelevant.

1) Top-Down and Bottom-Up: Understanding of componentized system design is aided by comparing two design styles as illustrated in Fig. 5. When a coarse-grain component or a system is designed by component assembly as on the right, those components are presumed to already exist and their design cannot be modified. The starting point for design is thus a set of the system requirements and a hopefully extensive and rich catalog of components that are available. The emphasis is upon uncovering relations among available components and possibly choosing among configuration options and adding new extension modules. In composition, the range of feasible systems is constrained by the available components, but that range is greater if components are available in variety and are themselves flexible and configurable.

In contrast, in design by decomposition on the left side of Fig. 5 the module definition and design flow directly from the system requirements. Those requirements are systematically decomposed into a supporting modularity by recognizing distinctions among functions (loose coupling) as well as natural grouping of functions (cohesion). (The "relations" and "distinctions" terminology is drawn from [14].) That modularity, and the capabilities of the respective modules, follows directly from system requirements.

2) Deconstruction: Where does a reference architecture come from? In this section, this is addressed from a design perspective and in Section VI-B3 from a process perspective. The architecture is the outcome of a special form of decomposition called *deconstruction* (terminology used by [27]). Deconstruction followed by component realization followed by composition is necessary [14] to achieve a concrete system realization as illustrated in Fig. 6. The starting point for deconstruction is a set of system requirements, but these requirements characterize an abstracted class or category of systems rather than a single concrete system, a general context rather than a specific set of requirements. Decomposition to define an appropriate modularity is followed by generalization and reasoning about each of the modules, which attempt to define these functions in the most general way possible. In the case of each module, the reference architecture defines functions and interactions, anticipating that the goal is to later turn each module into a component.

The reference architecture has some similarity to the successful concept of a design pattern in software engineering. This is a reusable architecture that solves a class of design situations and can relate to a software component in an analogous way [40].

Again, a communication network can be used as an example. Digital cellular, the Internet, and hybrid fiber-copper television distribution are recent examples of network architectures developed by decomposition from narrower requirements. The componentization alternative develops a reference architecture appropriate for a wider set of system requirements—the wider, the better. Deconstruction begins by identifying "the basic functions of any network," perhaps including connections, flow control, congestion control, authentication, billing, and so forth. Some carefully chosen limiting assumptions are probably made at this stage to make this task feasible, such as variable bit-rate connection-oriented transport. Then, we ask which of these functions relate and interact, and how?

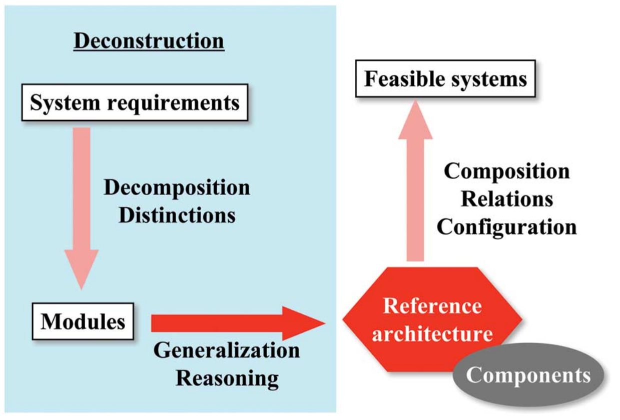


Fig. 6. ***Component methodology follows a hybrid of deconstruction (a disciplined form of decomposition), followed by composition.***

The next step is to generalize and reason about all of these functions, yielding functional definitions that satisfy a wide range of system requirements. For example, what is the most expansive abstract view of congestion control, one that captures a significant range of the known possibilities (if not all)? The goal is to identify a set of congestion control options and attributes that can satisfy a wide range of system requirements.

A reference architecture at the top level is developed from overall system objectives and requirements. As a conceptual framework, it is not constrained to align with existing technologies, responsibility, control, or physical proximity. A hierarchy is created by iteratively deconstructing each component using the next higher level of hierarchy as the system context. At lower levels of hierarchy, the deconstructions will begin to take account of concrete technological, physical, and market realities. For example, a deconstruction of a network connection-as-component into communicating link-as-components and switching functions can identify subcomponents that not only enable opportunistic composition, but also are in addition reasonably aligned with the boundaries of existing firms interested in those respective business opportunities.

3) Industry Processes: What industry processes might yield a reference architecture? There are numerous possibilities. One observed in practice is an evolutionary process emanating from a proprietary system in which one or more suppliers choose (or market forces dictate) an "unbundling" of the modularity, selling the modules separately to other system suppliers, choosing to outsource some modules to other suppliers, or in the extreme choosing to become strictly component suppliers [10]. In each of these cases, the prior system architecture, derived via a proprietary decomposition from system requirements, becomes the basis for the reference architecture. Legacies and anonymous market forces drive this approach and for this reason usually do not incorporate to a significant degree the "generalization and reasoning" step in the deconstruction of Fig. 6.

Deconstruction has the explicit goal of a reference architecture with as large a nonspecificity of system requirements as is practical. This is usually not something that a single supplier firm, which is focused on near-term product opportunity and profits, is likely to undertake. Rather, there are two main possibilities individually or in combination: supplier industry cooperation and end-user industry cooperation.

In the first option, a group of suppliers may undertake the definition of the reference architecture as part of a cooperative industry effort. This may include considerable research, prototyping, and experimentation. The Internet Engineering Task Force (IETF) is an example of a long-term cooperative effort among supplier firms and network researchers that incorporates research and experimentation into an architectural and protocol standardization effort [10], [41]. Fig. 7 illustrates the stages of an open standards process of this nature. Although the outcome is intended to be a set of approved standards, the intermediate stages make up essentially an industry-cooperative architectural design process, including intermediate milestones with specifications, prototype implementations, and experimentation.

The second option is a reference architecture initiated and driven by end-user organizations. End-user industry cooperation is driven by the *information asymmetry* between suppliers and end users: Suppliers are better able to make technological innovations, but end users possess domain expertise and more thoroughly understand their own needs, organizations, and processes [42]. One

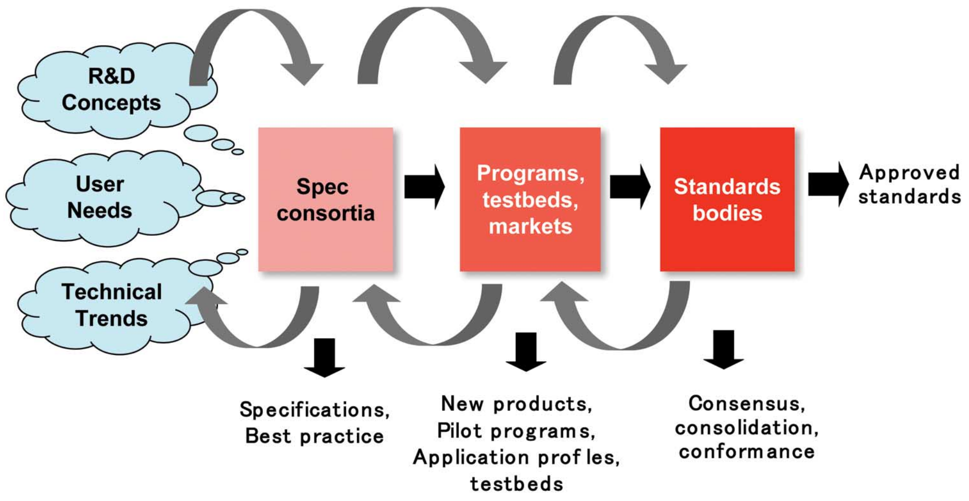

R&D Concepts
User Needs
Technical Trends
Spec consortia
Programs, testbeds, markets
Standards bodies
Approved standards
Specifications, Best practice
New products, Pilot programs, Application profles, testbeds
Consensus, consolidation, conformance

Fig. 7. ***Phases of open standards process (Source: Ed Walker of IMS Global Learning Consortium and used with permission).***

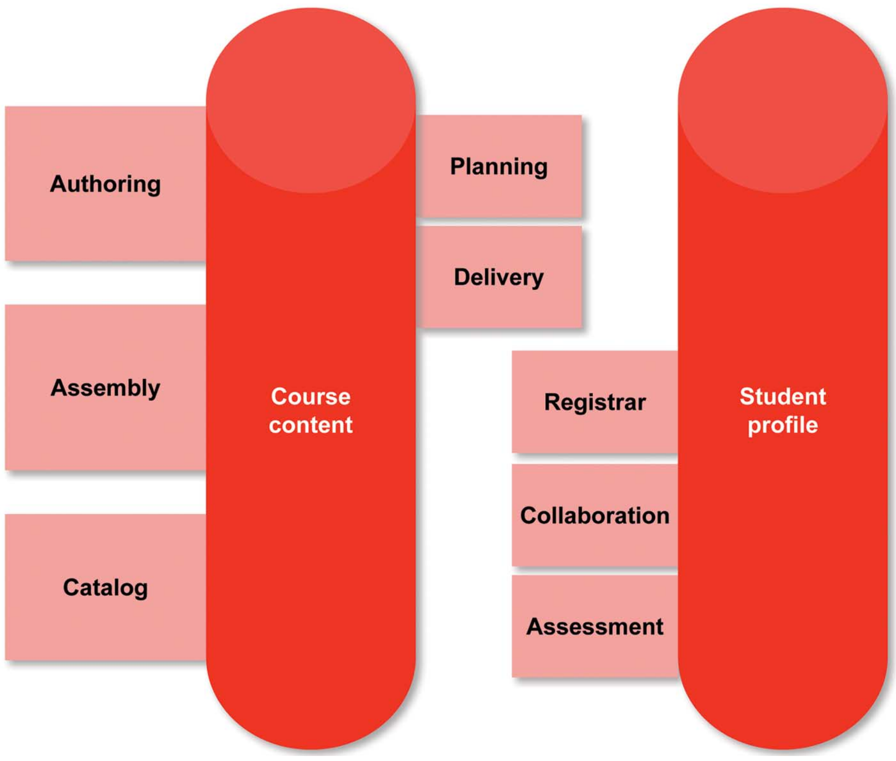


Fig. 8. *Modularity of a learning management system at top level (Source: Geoff Collier of Eduworks Corp. and used with permission).*

response could be end-user organization participation in open standards processes such as Fig. 7. The IETF illustrates this and was even initiated by ARPA acting on behalf of the U.S. military, itself an end user. Even more interesting is a process driven by end-user cooperation in the interest of ultimately obtaining better solutions from their suppliers. This type of end-user innovation and cooperation is "frequent, pervasive, and important" across many industries [42].

An example of end-user architectural control is the learning management system, which provides various capabilities to instructors and students in a university course with basic top-level modularity shown in Fig. 8. Universities dissatisfied with monolithic commercial solutions have tried internal development of these capabilities, but the large recurring costs of maintenance and upgrade and duplication of effort are unattractive to universities and the proliferation of content management standards is unattractive to academic publishers. Therefore, a group of universities joined an open standards and community software development process called the Sakai Project [43] with the goal of establishing a reference architecture and a set of component prototypes consistent with content representation standards. Universities are contributing development resources to prototyping efforts and using their own courses to gain experience with students and instructors and refine requirements. The participation of commercial firms in contributing components is welcomed, either in the prototyping and experimentation stages or later.

4) *Complex Systems:* The introduction asserted that componentization has a role to play in the evolution of large complex systems like a communication network or enterprise business application. How? Many of these complex systems have to be designed and operated in a complex environment like that of Fig. 3. The industry players face difficult strategic questions like the boundary of products, what complementarities are needed and who will supply them, and when to standardize or offer open interfaces inviting extensions or seek proprietary advantage. The appropriate degree and scope of standardization and related tradeoffs between industry versus firm-level activity abound. Viewed abstractly, these are all questions of responsibility and control (Section III-B) and how they can be successfully partitioned within a dynamic market economy. Who is responsible for what, and in particular how does responsibility partition among individual firms or multiple firms or an industry? Some industry and customer coordination is essential, but how much and where? Hierarchical component architectures offer a structure within which responsibility can be systematically partitioned, while the market resolves remaining ambiguities.

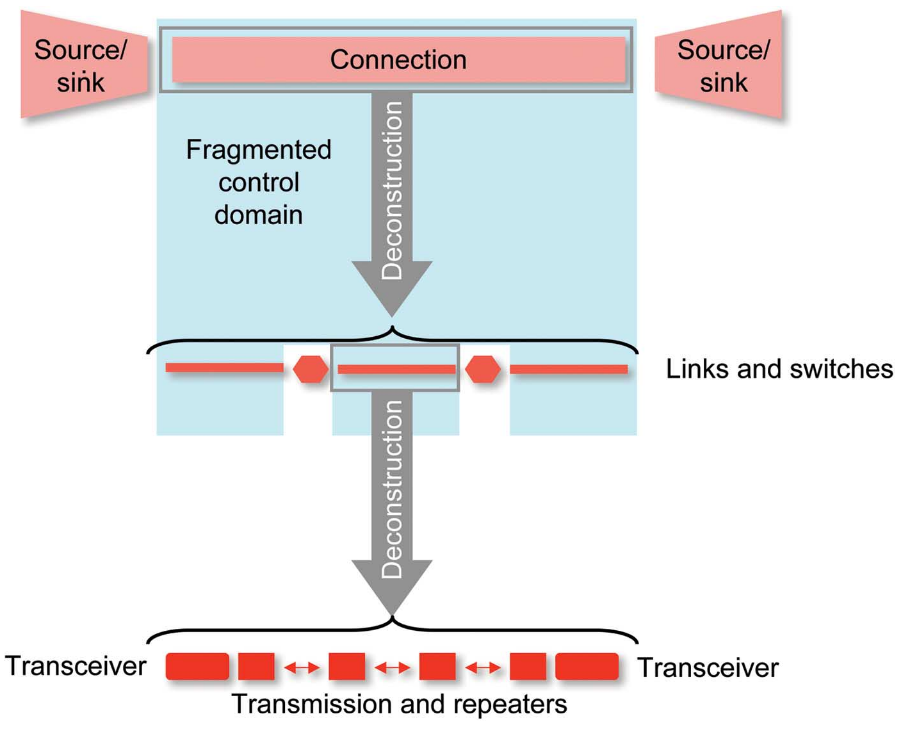


Fig. 9. ***Deconstruction of connection-as-component illustrating domain of fragmented control.***

Consider a component at a given level of hierarchy. At each and every level of hierarchy, the initial focus during deconstruction at that level should be on functionality and reasonable ways to modularize that functionality without regard to responsibility, control, or implementation issues. Once a principled and reasoned modularity is established, the focus can turn to responsibility, taking into account the current industrial organization, business models, core competencies, etc. At any hierarchical level where the deployed component may reasonably have fragmented control due to market structure, the internal architecture of that component is clearly the responsibility of the industry collectively (or more likely a subset of interested firms in the industry) rather than individual firms. There will have to be multiple firms contributing subcomponents, because otherwise a single firm claiming responsibility would be in a market monopoly position (all co-owners of the deployed replica would be forced to source from that same firm). In this case, industry cooperative mechanisms (like a standards organization or consortia or task force) can be established to assume long-term responsibility for this component's internal reference architecture and manage its evolution. The definition of an internal reference architecture would proceed on the same basis, with deconstruction based on principled and reasoned decompositions (to the extent that politics can be removed from the process). At some point in the hierarchy, subcomponents will revert to the simpler single-control case. Architectural standardization is not an obstacle to (and indeed enhances) direct competition among suppliers and customer choice at the subcomponent level.

At hierarchical levels where single control of all deployed realizations can be assumed (as in the components of a insurance policy application of Section IV-D), two options are available, both operating with a common context defined at the next higher level of hierarchy. One firm (typically a large one) may choose to take responsibility for this component, creating the best implementation it can (such as doing a handcrafted implementation or assembling acquired components) and offering the component for sale. Alternatively, a set of firms commercially interested in this component (typically smaller ones) may choose to cooperate (such as through an industry consortium) in taking responsibility for their own internal reference architecture, with different firms contributing subcomponents. The beauty is that both these options can be pursued in parallel and compete with one another in the market. In this fashion, groups of smaller firms are able to complete with larger firms by joining forces, with overall coordination of responsibility occurring through maintenance and upgrade of the hierarchical reference architecture. Other benefits of componentization discussed elsewhere (such as flexibility and evolution) also apply.

A simple example is shown in Fig. 9. A network connection-as-component is deconstructed into a composition of concatenated communication link-as-component's (and switches, etc.), and each link is further deconstructed into a composition of transceiver's, repeaters, transmission, etc. The domain of fragmented control includes the connection (because it may span multiple service providers) and the links (because some may terminate in different service providers). On the other hand, switching-like components may be assumed to fall within a single control domain. To preserve the independence of the control domains in choosing their respective suppliers, responsibility for the connection and link components are assigned to industry processes (Section VI-B3), and these components are constrained to be a pure assembly of subcomponents with opportunistic composition of these subcomponents the primary architectural goal. Subcomponents of the link can in turn each be assumed to fall within a single control domain, so responsibility for their design is left to individual firms and market forces.

In what ways does this approach improve on current industry practice? First, it is not a radical departure since hierarchical modularity is at least implicit in present design practice, so getting there from here is not out of the question. By making hierarchical component architecture explicit and universal, with one goal the creation of generic "stable intermediate forms," it holds out hope of containing perceived system complexity. By explicitly defining the scope of industry versus individual firm responsibility in a structured and principled way, it maximizes the scope of competition and market forces while preserving customer choice in a fragmented control environment. In other words, it provides a systematic framework for splitting responsibility between standardization processes and individual firms. By defining open component architectures down to the level of individual firm responsibility, it preserves the ability of large and small firms or industry consortia to compete with one another in the market while contributing complementary technologies in a structured and coordinated way. Finally, it captures all the opportunities for componentization discussed in Section III-C.

Of course, this presents an idealized view, and reality will inevitably be far messier. Getting all industry players on the same page is never feasible, and commercial interests will sometimes trump the best of intentions. Nevertheless, a hierarchical component architecture could provide a principled framework around which to organize industry-level processes like standardization and community development, one that would offer significant advantages.

C. Medium-Grain Components

In the taxonomy of Table 1, the primary role of medium-grain components is architectural support for a supply chain. Thus, the dominant design methodology is neither deconstruction nor component implementation, but rather internal assembly of components acquired from other companies. Suppliers of medium-grain components are primarily "integrators," and their viability depends on a vibrant variety of components available for purchase [2], [44]. For this reason, component markets are discussed next.

VII. COMPONENT MARKETS

Finer grain components that can be sourced by a single firm are a natural to be bought and sold in the market, rather than used as an internal design and development strategy within a single firm.

Technology reuse is a strategy typically followed within a development organization to make more efficient use of development assets (Section III-E). There exist notable reuse successes, such as Toshiba in software engineering [45]. In contrast, components developed for proprietary use within a firm are difficult to achieve in practice. Managerial incentives emphasize delivering projects under budget and on time, and components are more expensive and take longer to develop. Future cost savings or other benefits are difficult to quantify, and incentive systems that take these factors into account are difficult to establish. More fundamentally, the limited uses of a component available within a given firm are significantly less likely to justify the higher costs and longer development times, even compared to reuse. Notable exceptions can occur when the benefits of componentization are so compelling that these obstacles can be overcome. For example, suppliers of enterprise applications must meet the needs of multiple customers with differentiated needs evolving over time, a challenge for which componentization is ideally suited [19].

On the other hand, the component value measures of Section VI-A are entirely consistent with a component sold and bought. Componentization reduces coordination costs among suppliers, thereby favoring industry fragmentation into smaller firms as predicted by the economic theory of the firm [39]. A component seller seeks larger revenues, which is consistent with making its products as context agnostic as possible and seeking multiple uses. The larger revenues that result can justify the higher development and maintenance costs [18]. Encapsulation is compelling for any product or design that is sold, as outlined in Section VI-A4. Composition by assembly reduces design costs and more importantly reduces the time to market and the ability to track user needs more rapidly. When new uses define new requirements, those are fed back to suppliers who upgrade components over time, and other uses and customers benefit from those enhancements. There is a strong incentive for the supplier to avoid "breaking" existing uses with those enhancements, which likely reduces its market or upsets its customers, and this contributes to the "stable intermediate form" characteristic. The component as an independent unit of deployment avoids many coordination and transaction costs among suppliers, since they can act relatively independently and customers can deal with suppliers independently.

An important qualification is that the market is an anonymous force that is unlikely to perform the principled deconstruction that is necessary for a well-functioning larger grain system solution. For this purpose, industry cooperation in some form is advantageous as discussed in Section VI-B3.

There are other advantages accruing from the interactions of components and a market. The market has proven to be a great tamer of complexity [46]. Markets favor designers and vendors who make their products easier to adopt and use and otherwise have characteristics that encourage widespread adoption. Among these characteristics are successful hiding of internal complexity, credible roadmaps for future enhancements, and a track record of seamless enhancement without breaking existing uses.

While economics offers insights into industrial organization [39], the impact of new technologies is a complex issue involving many considerations [47]. In a component market, component interfaces must coincide with the boundary of firms. This raises the question of whether firms define component boundaries, or component boundaries define the boundaries of the firms. Like natural complex systems, the answer is likely to be complicated. Components in a market will evolve chaotically, with new hierarchical layers built on existing layers. Firms experiment with new components, often constructed from existing proprietary modules that benefit from test in actual use. The market applies selection criteria that favor some solutions over others [12], in this case based on feedback from customers and economic metrics like revenues or profits. Successful components can build on their success through greater investment in maintenance, customer service, and upgrade, further enhancing their future prospects.

The nascent condition of today's market in components is a significant barrier to the component methodology. Industrial history suggests that establishing these markets is not easy, and sometimes there can be backtracking (Section V). Existing technology suppliers may not jump on the component bandwagon as enthusiastically as hoped, as this is a major change to established business models and puts more power in the hands of customers, so the activism of startup companies may be beneficial. A long-term investment in research into components is necessary, as there is certainly much to be learned regarding technology, tools, and methodologies. The involvement and activism of professional organizations would be beneficial. Most importantly, the ultimate customers of components are the ones who stand to benefit the most from a viable component market. They (or governments acting as their proxies) need to become activists, through supporting research and setting up industry-level processes. Such activism on the part of users has been commonplace and effective in similar situations [42], just as the active involvement of the U.S. Army was instrumental in establishing the interchangeable part as an important fixture in industrial production (Section V-A).

VIII. CONCLUSION

Recall the assertion [33] that "...engineering rationality is not a set of timeless abstractions, but a set of social practices which have emerged historically." To what extent is the atrophy of componentization in ICT a result of inattention or under-appreciation, as opposed to technical difficulty or economic obstacles? Certainly, componentization will require a major shift in design culture within the ICT industries, driven in part through changes to engineering education. It will also follow from (or more likely drive) changes in industry organization to create a component supply chain and require additional actions by industry as a whole in defining context-specific reference architectures. These changes will happen more quickly with the activism of end-user organizations or government research funding acting as their proxy.

We do not claim that componentization is a panacea. Details such as specific modularity design and industry processes and component markets matter, and some market participants may actively fight change. Components are nevertheless a significant opportunity. Hopefully, this paper will stimulate a dialog that leads ultimately to more successful ways of constructing large complex ICT systems that take into account the realities of industry competition, cooperation, and market forces.

There are many unanswered questions regarding componentization, many of them nontechnical as well as technical in nature. Does componentization apply equally well to emerging as well as mature product categories? How does a more vibrant market in components arise, or how can it be stimulated? How well can both suppliers and customers differentiate their products and services when they rely more on components? What is the role of community development methodologies (often called "open source") in componentization? These and other questions will hopefully be answered through both conceptual research and market experimentation. ■

APPENDIX

The richness of ICT technology makes it much more flexible than other technologies. What follows are some capabilities that can be exploited to make ICT components more functional and flexible.

Separation of Manufacture From Functionality: Arguably the most powerful idea to emerge in ICT is software, which allows functionality to be deferred at the time of manufacture. Software can be distributed over the network, allowing functionality to be added or modified or extended later. This capability can be used to maintain or upgrade deployed components.

Translation of Design to Realization: A component design resides in a computer-aided design (CAD) system. It is important to distinguish this design from a component realization (a deployable representation of the component). An automatic translation step can convert design to realization, and many configuration options can be built into the design and implemented in the translation. This concretely addresses one criticism of the component, which is increased cost or decreased performance in its replication (Section III-D). Since a component design will typically incorporate the union of the capabilities needed across multiple uses, in any specific use some component capabilities may be unneeded. These can be eliminated during translation.

Technical Capabilities: ICT components can be complex, self-aware, and communicate over a nearly ubiquitous Internet. This offers opportunities not available in any other technology, as for example automatic upgrades and remote usage monitoring and license enforcement [46].

Incorporating Physical Connections: The communication link-as-component of Section IV-A encapsulates a physical medium. Similar situations occur often, as for example the computer universal serial bus (USB). Such a medium should typically be encapsulated within a component rather than appear as the component interface. In practice, this is achieved by positioning a component's software and hardware on both sides of the medium. For example, a printer will provide a software "device driver" for installation on the computer it is connected to. That device driver embeds the low-level details of the physical connection within the logical printer functionality and simultaneously presents an abstracted interface freed of these low-level details to the application.

Metalanguages: Rather than defining a static predefined set of interface capabilities, in ICT special languages can be developed for describing specialized capabilities. Sharing this language, components can define new functionality dynamically by describing what they want or what they can provide. Standardization can then focus on language expressiveness rather than specific functionality. An example is Visual Basic for Applications supported by Microsoft Office, which is used to develop customized applications building on Office as a platform.

Mobile Code: One component can dynamically share code with another component, that code available to be executed internally to the other component. In this way, one component can directly add enhance the capabilities of another component in ways that contribute to complementarity. An example is JavaScript, which is used by web servers to dynamically add functionality to web browsers.

Infrastructure Intermediation: Forming direct interfaces between pairs of components leads to a combinatorial explosion of interfaces. Designing components to interact through an appropriate infrastructure and embedding generic intermediation capabilities within the infrastructure can limit the burden of this on the reference architecture. For example, a network connection-as-component can be abstracted by encapsulating that connection within an infrastructure that presents a friendlier interface (for example, with "actions" consisting of "functions" and "arguments," such as is supported by "distributed object management" systems like CORBA and Java [23]). The generic capabilities needed to map interface to physical connection can be implemented once and only once within that infrastructure.

Adapter Intermediation: Infrastructure provides generic intermediation capabilities. Although best to avoid if possible, an adapter can intermediate between two components possessing complementarity but lacking interoperability. Even better, each component can be equipped with its own adapter to convert to a common intermediate form or to a new infrastructure, separating responsibility among the respective suppliers and reducing the overall number of adapters to maintain. This is particularly appropriate for legacy components that may have been designed outside a reference architecture, which emerges later, and/or making incompatible infrastructure assumptions. Of course, an adaptor is a blatant form of fitting.

Acknowledgment

The authors appreciate the valuable discussions on these topics with M. Katz and H. Varian of the University of California, Berkeley, J.-P. Hubaux of Ecole Polytechnique Fédérale de Lausanne, and S. Verdu of Princeton University. H. Varian and F. Beshears of the University of California, Berkeley, and N. Oza of the Helsinki University of Technology provided valuable comments on this paper.

REFERENCES

[1] G. Rostky, "A tale of the unstoppable electronic kit," *EE Times*, Oct. 2, 2000.

[2] C. Szyperski, D. Gruntz, and S. Mauer, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Boston, MA: Addison-Wesley, 2003.

[3] A. Tanenbaum and A. Woodhull, *Operating Systems: Design and Implementation*, 3rd ed. Upper Saddle River, NJ: Pearson/Prentice Hall, 2006.

[4] M. Sauter, *Communication Systems for the Mobile Information Society*. Hoboken, NJ: Wiley, 2006.

[5] M. Langberg, "Windows Vista delay is explained in 1975 book: Too much manpower," *San Jose Mercury News*, Mar. 26, 2006.

[6] J. Ottensmeyer, "IP version 6—An analysis of the long way from concept to large-scale deployment," in *Proc. 28th Euromicro Conf. IEEE Comput.*, 2002, pp. 229–232.

[7] D. Talbot, "The internet is broken," *MIT Technol. Rev.*, vol. 108, pp. 62–69, 2006.

[8] D. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, pp. 1053–1058, Dec. 1972.

[9] M. Schilling, "Toward a general modular systems theory and its application to interfirm product modularity," *Acad. Manage. Rev.*, vol. 25, pp. 312–334, Apr. 2000.

[10] L. Wilson, A. Weiss, and G. John, "Unbundling of industrial systems," *J. Market. Res.*, vol. 27, pp. 123–138, May 1990.

[11] D. Messerschmit and C. Syperski, *Software Ecosystem: Understanding an Indispensable Technology and Industry*. Cambridge, MA: MIT Press, 2003.

[12] H. Simon, "The architecture of complexity," *Proc. Amer. Philos. Soc.*, vol. 106, pp. 467–482, Dec. 12, 1962.

[13] D. Angel and J. Engstrom, "Manufacturing systems and technological change: The U.S. personal computer industry," *Economic Geography*, vol. 71, pp. 79–102, Jan. 1995.

[14] L. Richards and S. Gupta, "The systems approach in an information society: A reconsideration," *J. Oper. Res. Soc.*, vol. 36, pp. 833–843, Sep. 1985.

[15] R. Sanchez, "Modular architectures in the marketing process," *J. Market.*, vol. 63, pp. 92–111, 1999.

[16] C. Shapiro and H. Varian, *Information Rules: A Strategic Guide to the Network Economy*. Boston, MA: Harvard Business School Press, 1998.

[17] M. J. Cohen and T. Ho, "New product development: The performance and time-to-market tradeoff," *Manage. Sci.*, vol. 42, pp. 173–186, Feb. 1996.

[18] R. Garud and A. Kumaraswamy, "Technological and organizational designs for realizing economies of substitution," *Strategic Manage. J.*, vol. 16, pp. 93–109, Summer, 1995.

[19] D. Sprott, "Componentizing the enterprise: Application packages," *Commun. ACM*, vol. 43, pp. 63–69, Apr. 2000.

[20] T. Nolle, "What we should learn from the AT&T outage," *Network World*, vol. 15, p. 73, May 4, 1998.

[21] F. Lindholm, *The Java Virtual Machine Specification*. Reading, MA: Addison-Wesley, 1997.

[22] T. Berners-Lee, *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. San Francisco, CA: Harper, 1999.

[23] D. Messerschmitt, *Understanding Networked Applications: A First Course*. San Mateo: Morgan Kaufmann, 2000.

[24] W. Wolf, *Computers as Components: Principles of Embedded Computing System Design*. San Mateo, CA: Morgan Kaufmann, 2001.

[25] E. Marks and M. Bell, *Service-Oriented Architecture: A Planning and Implementation Guide for Business and Technology*. Hoboken, NJ: Wiley, 2006.

[26] J. Yang, "Web service componentization," *Commun. ACM*, vol. 46, pp. 35–40, Oct. 2003.

[27] L. Cherbakov, G. Galambos, R. Harishankar, S. Kalyana, and G. Rackham, "Impact of service orientation at the business level," *IBM Syst. J.*, vol. 44, pp. 653–668, 2005.

[28] E. Bergman, *Information Appliances and Beyond Interaction Design for Consumer Products*. San Mateo, CA: Morgan Kaufman, 2000.

[29] J. Lie, "Sociology of markets," *Annu. Rev. Sociol.*, vol. 23, pp. 341–360, 1997.

[30] R. Langlois, "Modularity in technology and organization," *J. Economic Behavior Org.*, vol. 49, p. 19, 09, 2002.

[31] D. Engel, "The standardization of lawyers' services," *Amer. Bar Found. Res. J.*, vol. 2, pp. 817–844, 1977.

[32] R. Woodbury, "The legend of Eli Whitney and interchangeable parts," *Technol. Culture*, vol. 1, pp. 235–253, summer, 1960.

[33] K. Alder, "Innovation and amnesia: Engineering rationality and the fate of interchangeable parts manufacturing in France," *Technol. Culture*, vol. 38, pp. 273–311, Apr. 1997.

[34] D. Hounshell, *From the American System to Mass Production, 1800–1932: The Development of Manufacturing Technology in the United States*. Baltimore, MD: Johns Hopkins Univ. Press, 1984.

[35] H. Varian. (2001, Nov.). *The Economics of Information Technology*. [Online]. Available: http://www.ischool.berkeley.edu/~hal/Papers/mattioli/mattioli.pdf

[36] Ford Henry, "Mass production," in *Encyclopedia Britannica, a Dictionary of Arts, Sciences, Literature General Information*, vol. 30. London, U.K. 1926.

[37] G. Thompson, "Intercompany technical standardization in the early american automobile industry," *J. Economic History*, vol. 14, pp. 1–20, Winter, 1954.

[38] E. Wertheimer, "New screw thread standards," *Science*, vol. 110, pp. 155–159, Aug. 12, 1949.

[39] R. Coase, "The nature of the firm," *Economica*, vol. 4, pp. 386–405, Nov. 1937.

[40] B. Meyer and K. Arnout, "Componentization: The Visitor example," *IEEE Comput.*, vol. 39, pp. 23–30, Jul. 2006.

[41] O. Hanseth, E. Monteiro, and M. Hatling, "Developing information infrastructure: The tension between standardization and flexibility," *Science, Technol. Human Values*, vol. 21, pp. 407–426, Autumn, 1996.

[42] E. von Hippel, *Democratizing Innovation*. Cambridge, MA: MIT Press, 2005.

[43] J. Young, "Sakai project offers an alternative to commercial course-management programs," *Chronicle of Higher Education*, Sep. 24, 2004.

[44] C. Szyperski and D. Messerschmitt, "The flexible factory," *Software Development Mag.*, Nov. 12, 2003.

[45] M. Cusumano, *Japan's Software Factories: A Challenge to U.S. Management*. New York: Oxford Univ. Press, 1991.

[46] B. Cox, *Superdistribution: Objects as Property on the Electronic Fontiier*. Reading, MA: Addison-Wesley, 1996.

[47] H. Varian, "If there was a new economy, why wasn't there a new economics?" New York Times, Jan. 17, 2002.

ABOUT THE AUTHOR

David G. Messerschmitt (Fellow, IEEE) received the Ph.D. degree in computer, information, and control engineering from the University of Michigan.

Currently, he is the Roger Strauch Professor Emeritus of Electrical Engineering and Computer Sciences (EECS), University of California, Berkeley, and also a Visiting Professor in the Department of Computer Science and Engineering at the Helsinki University of Technology (HUT), Helsinki, Finland. At Berkeley, he has served as the Interim Dean of the School of Information and Chair of EECS. His recent research focuses on improvement of software and communications technology by applying business and economics insights, and he is the coauthor of five books, the most recent *Software Ecosystem: Understanding an Indispensable Technology and Industry* (MIT Press, 2003). At HUT he is conducting research in the Software Business Laboratory.

Dr. Messerschmitt served on the NSF Blue Ribbon Panel on Cyberinfrastructure and co-chaired a National Research Council (NRC) study on the future of information technology research. He is a member of the National Academy of Engineering and a recipient of the IEEE Alexander Graham Bell Medal recognizing "exceptional contributions to the advancement of communication sciences and engineering."