ECAR: A Repository for Storing, Retrieving and Analyzing Emergent Component Architecture

David Dreyfus | Bala Iyer

School of Management
Boston University
595 Commonwealth Ave
Boston, MA 02215

Abstract

Organizations making decisions regarding their information system architecture have a distributed cognition challenge: decision makers make local decisions based on local knowledge but those decisions have global, unappreciated consequences. These decisions can be improved through a distributed, shared understanding of the information system architecture and achieving this distributed cognition can be supported by software applications. Information systems are often modular, emergent, complex designs that are amenable to network analysis. In order to capture information system architectures, and analyze them at the organizational level, a repository and associated application are proposed, ECAR (Emergent Component Architecture Repository).

1 Introduction

Every organization has an information system architecture and assumptions about what benefits it will provide [1-3]. It may be computer system assisted, or it may rely solely on written and verbal communication. A key challenge for managers in-charge of information system projects is to cost-effectively build and maintain an information systems architecture that is responsive, supports innovation, and delivers economies of scope while simultaneously making sure that the architecture is aligned with the business strategy [3-6].

One of the difficulties in meeting this challenge is that cognition regarding the information system is distributed [7]. The information system that the IT managers would like to manage is often the product of distributed decision makers, and will often continue to be modified by distributed decision makers. The information system is not the product of a single, rational actor that maintains a coherent and complete understanding of why the system was put into place, and how it will be managed.

The information system may have been designed, or it may have just evolved. It may exist as designed, or it may have diverged from the original design as the result of distributed decisions made by independent decision makers[8, 9]. Every information systems project modifies the architecture in some way. Even if an organization designs an architecture - the espoused architecture, as a result of multiple implementation projects, performed over multiple project cycles, responding to changing business conditions and the organization's products and services trajectory, an architecture emerges that is different than the espoused architecture - the emergent architecture. Such architectural changes, in turn, have such a huge impact on a firm's ability to compete and survive [4, 10], that it would be reckless to make such decisions without understanding the potential consequences.

We posit that a decision support tool that we've named ECAR – Emergent Component Architecture Repository – can provide support that will help decision makers who influence the emergent architecture make more appropriate decisions. Our goal for the tool is to facilitate the collation, storage, analysis, and visualization of information about an information system's components[11], the relationships between them [12], and their time-sequenced emergence in order to enable a more shared understanding of the system that the decision makers will be modifying [7, 13-15]. We anticipate that storing information regarding the emergent information system and using this information to visualize [16] the architecture will improve decision making by replacing incomplete, localized, tacit knowledge with a more global, consistent, explicit version.

This paper follows the emerging precepts of Design Science in which organizational theory is used to drive problem statements, design theory is then applied to the problem statements to create a testable system, and the system is then evaluated to contribute to organizational knowledge, design knowledge, or both [17]. The remainder of the paper outlines the theory that underpins the design, describes the design, and then presents an empirical setting in which preliminary evaluations have been performed. We then conclude the paper with suggestions for further research.

2 Architecture

There are multiple perspectives on what constitutes information system architecture [18]. Architecture has been viewed strategically [2, 4, 18-21], organizationally [6, 9, 22-24], and technologically [25-29]. Previous studies have summarized that an information system architecture includes a group of shared, tangible IT resources that provide a foundation to enable present and future business applications [2, 3, 6, 24]. Architecture, as implemented through its IT infrastructure, should be flexible, reliable, robust, scalable, and adaptable [3, 6, 23]. It should support the reuse of business components within a firm, while supporting firm responsiveness, innovativeness, and economies of scope [3].

Zachman expands the concept of information system architecture to include the various perspectives taken during its design and implementation [1]. Iyer and Gottlieb [9] identified three views through which we can examine architecture. The first view - the espoused - is the outcome of designing the planned dependencies between system modules and is the province of the IS Architect (although there may be many stakeholders). The second view - the emergent - is the outcome of implementing individual projects. It is a descriptive view of the actual dependencies that exist amongst system modules. The third view - the in-use - highlights the dependencies between and among system components and organizational groups that arise from the business of doing the work of the enterprise – selling products, buying supplies, managing employees, etc - as employees, suppliers, and other stakeholders interact with the system [30].

The implementation of the architecture enables certain organizational capabilities, and constrains other capabilities. The tasks that are allocated to specific computers represent the actions and responsibilities of specific users. The allocation of tasks to systems, thus, mirrors existing dependencies between groups, and creates new dependencies between them [31, 32]. As the architecture changes – emerges – over time, so do the dependencies among the systems.

The emergent information system reflects the interdependencies between the tasks and groups and serves as an integration mechanism [33] between groups differentiated to manage complexity [34, 35]. The existence of certain information conduits implemented in the IS system influences communication and routines [36] between the groups both syntactically and semantically [37]. The evolution of the information system also reflects different power relationships and conflicts between stakeholders [38].

IT project cost and success, and system flexibility, robustness, adaptability, and performance are the result of an interaction between system and organization that system architecture influences [32]. Project cost and maintenance is directly related to complexity [39], which is a description of interdependency.

From the preceding, it seems clear to us that a system that supports the collective understanding of an organization’s information system architecture must reflect both the software components that constitute the applications that users interact with and the dependencies that link them. ECAR gives prominence to the links between systems following the logic set forth by Brooks when he describes the importance of the systems view [40]. The implementation of software components is no less important than the development of the links between systems; however, individual systems do not suffer from the absence of a distributed understanding. The developers and consumers of individual systems generally understand the particular system quite well. What they don’t understand as well is how an individual application interacts with the other components of the information system.

3 Software components

To describe and understand emergent architecture, we adopt modularity theory [23, 24]. Modularity theory enables us to view an architecture as an interconnected set of modules (software components, in our case). A key concept in modularity theory is the modular component. “A software component is a unit composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” [41] Software libraries, system software (operating systems, database management systems, etc.), applications, and components implemented as EJBs, CORBA objects, and .NET objects are all examples of components. The modules of an enterprise system such as SAP or Oracle

Financials are not components unless they meet the requirement set forth at the beginning of this paragraph. Components differ by their granularity and their interfaces. Enterprise systems are large-grained components, whereas a component that performs a credit card validation may be considered small-grained.

Components become part of the emergent architecture. Each component is linked to the architecture through its interface and dependencies. The interface consists of message passing (loosely or tightly coupled) and data exchange (reports, files, shared database, etc.). The interface may or may not require human interaction. ECAR makes the software component a first-class element in the information system representation. Although any component can be decomposed to any level of abstraction, the real focus in ECAR is the links that bind the components.

4 Dependencies

In the conceptualization of architecture we are developing, software components are linked to other components by the interdependencies between them. Sometimes these interdependencies are manifest as interfaces that one component uses to request service from another. Of those links that are manifest in software, some can be discerned through static analysis: one can read the code of one system and determine that it calls a specific other system. However, in other implementations, the existence of the link may only be manifest at runtime when the first application dynamically determines the other system it must contact. Sometimes, however, the dependency may only exist at a semantic level as when two systems must share a conceptualization of some organizational construct such as the definition of a customer [42].

The links between nodes represent conduits through which information, ideas, and knowledge flow. Within information systems, the most obvious links between systems are those in which two systems exchange data. In this model, two nodes are linked if they share data. As a dependency, we would say that system A depends on system B for data. However, by drawing on the coordination literature [25] and organizational research on interdependence [43], we understand that there are multiple types of dependencies by which we could characterize the tasks represented by systems: resource sharing, producer/consumer, and simultaneity.

A resource dependency exists when system A depends on data managed by system B. For example, applications that share a common Oracle database have a dependency on the shared data resource managed by Oracle. The database management system (DBMS) has many facilities to manage the dependency on this shared resource. A producer/consumer dependency exists when system A depends upon the results of system B. For example, a general ledger application depends upon the data processed in a sales order application. A simultaneity dependency exists when two or more tasks must be completed before a third task can begin. For example, a reporting application may need data from multiple systems before compiling the report.

A consequence of maintaining the direct dependencies between systems is that we can also represent the indirect dependencies between systems – a form of transitivity in which if A depends on B, which depends on C, then A also depends on C. The implication of direct and indirect dependency in the context of nearly decomposable systems [34] is that the influence of one component of the information system on another decreases as the number of components that separate them increases, but does not disappear.

System developers implementing changes to the information system must confront these dependencies when modifying the system. To modify an existing component, the dependencies between the component and its direct dependents must be negotiated. If we make the simplifying assumption that the maximum number of dependencies between any two components is two (bidirectional), and we ignore indirect dependencies, then modifying a component connected to N-1 other components requires a maximum of N*(N-1) negotiations. This includes the negotiations between the target component and its directly connected components, plus the negotiations among those components.

Adding a new component to the network involves the negotiation of two dependencies: the dependency of the new component on its point of connection to the network, and the point of connection's dependency on the new component. By accepting the connection, the point of connection's freedom for subsequent change has been restricted to the extent it will need to negotiate change with the new component. Adding a new component to the network also shifts the relative importance of existing components within the network.

Integration between two existing component increases the number of dependencies on each component by one. It also increases the number of indirect dependencies between components and may shorten the shortest path between pairs of components. As links are added to the network, the balance of influence among components may change.

An architect interested in managing the evolution of the network, its impact on the organization, and the organization’s impact on the system may be interested in how these modifications to the network affect certain network characteristics. In our research, we start with the problem of collecting and visualizing the network so that decision makers can better understand both why the current system may be hard to modify and the potential impact of potential changes to the information system.

5 Networks, Analyses, and Visualizations

Researchers have demonstrated the value of exploring the connectedness of everything to everything else through the use of network diagrams [44]. In our research, we extend his work to incorporate information system architecture. Rather than view architecture through the prism of various software diagramming techniques (ER, OO, UML), we explicitly study the connections between objects and the resulting, emergent network. Instead of focusing on the nodes, we focus on the links.

Networks provide two benefits to understanding architecture. First, they provide a good metaphor to communicate architectural issues. Networks are both simple in their symbols and rich in describing complexity. One of the challenges architects face is that architectural models and specifications are often useful only to other architects. Enterprise architects, chartered with a long-term, global architecture strategy, are often challenged to convey how they contribute to the day-to-day business. This includes conveying what the key issues are, and why projects get out of control (fail, run over budget, ship late, fail service level agreements, etc.) to their business colleagues.

Second, networks are amenable to certain analytics that can help the architect compute relevant metrics. Primarily, the analytics are used to identify the most important nodes.

Historically, researchers have assumed that nodes are connected to one another randomly; thus, no node is more important than any other node. However, Barabasi [44] found that in many real-world networks the number of links incident to particular nodes followed a power-law distribution. This meant that these networks had a great majority of nodes with very few connections and few nodes with a great many connections. Barabasi listed two conditions for making any network scale-free [44]. First, the network grows with new nodes appearing at random times. Second, a new node preferentially attaches to an existing node based on the number of connections that the existing node already has.

Watts [45] described another type of network pattern – Small Worlds – in which a network is composed of multiple clusters of nodes in which the nodes within a cluster have many links with a few links between clusters. The average number of links between nodes in this type of network is much smaller than a random network with the same number of links. These networks are more robust to perturbations than random networks.

Researchers of organizations, people, and power-grids (to name a few) have been interested in the structural and flow properties of networks [46]. These researchers have developed a number of measures that describe a nodes importance, generally classified as centrality [47]. Centrality and the way links are formed over time has implications for control and risk. When nodes occupy a central position they have the ability to influence the entire network. As a result of this, they gain importance. Within the architectural context, such nodes have to be managed closely or they could adversely impact the network as a whole. If nodes connect to other nodes randomly, complexity is not managed and the network represents a random network. If nodes preferentially attach to other nodes, a few nodes become very central – critical – to the network as a whole. These nodes may be very hard to modify, and could expose the firm to undue risk if they are not controlled or are uncontrollable.

Within the architecture domain, a few network-related requirements stand out. Most enterprise architects are interested in identifying what they call the shared core. These are the set of components (systems, services, applications) that are used by most other components. Visualization packages such as Pajek, a freeware program [48], can be used to draw network diagrams and generate meaningful and replicable visualizations of very large networks (networks having hundreds of thousands of vertices).

A second requirement is identifying the architectural control points (ACPs) [49]. Architects interested in risk management and the evolution of the architecture may be interested in the group of systems that represent those systems that must be maintained collectively to minimize disruption to the network or maximize influence over the network. The ACPs represent those nodes that, collectively, can have either the greatest positive or negative impact on the information system architecture. Although the ACPs may be the most important nodes, collectively, the ACPs do not necessarily include those nodes identified as having the highest centrality measure.

Identifying the critical set of systems is a greater challenge than identifying a single node. The Key Player KPP-NEG test can be used to identify the nodes that have the potential to maximally disrupt the network. The Key Player KPP-POS test can be used to identify the nodes that have the potential to maximally influence the network [50, 51].

The final requirement (at least in this paper), is to identify the best decomposition of a set of components that minimizes dependencies across clusters while maximizing dependency within a cluster. Enterprise architects are familiar with the notion of tight and loose coupling. Our challenge is to use network analysis techniques to create and analyze useful clusters. The problem of finding a grouping of nodes, such that for all nodes in each group they are powerfully connected within and weakly connected to others, remains a very difficult problem [52].

6 ECAR Architecture

The preceding sections outline the objectives of the ECAR system. It should 1) support the localized collection of information regarding software components and their interdependencies, 2) import existing architecture repositories from spreadsheets to presentation files, 3) remotely monitor and analyze network packets to infer runtime dependencies, and 4) support the development of a shared, distributed understanding of the architecture. Given these broad objectives, we made certain design decisions.

We developed the core of the application around a relational database management system (RDBMS) (Microsoft Access). The RDBMS supports distributed access so that multiple people can enter and examine information concurrently. The RDBMS schema imposes minimal semantic requirements making it easy to understand across a wide audience.

Many different types of applications can access the database: one application can import spreadsheets, another can monitor network traffic, and a third can generate visualizations. With this design choice the coupling between the RDBMS and the supporting applications is loose. The restrictions imposed by the RDBMS are minimal and numerous standard interfaces can access the data (e.g., ODBC, DAO, and ADO).

The second decision was to treat the data-entry component of the application as just another application. If different groups within an organization want to develop custom data-entry applications, they are able to do so. Our data-entry application was written in Microsoft Access and contains modules for editing the various reference tables as well as modules for editing elements and links. Using such a tool and its associated language opens the tool to development by a large number of developers. It also facilitates rapid software development [53] due to its interpretive code execution and runtime type checking.

The third decision was to make the network visualization capability distinct from the data-entry application. Many different visualization packages (commercial and freeware) can utilize the same database. Each visualization package may have its own special features.

The fourth decision was to make the network analytics distinct from the visualization and data-entry applications. Here too, existing network analysis packages can be used. Collectively, these design choices provide the highest levels of code reuse [54, 55].

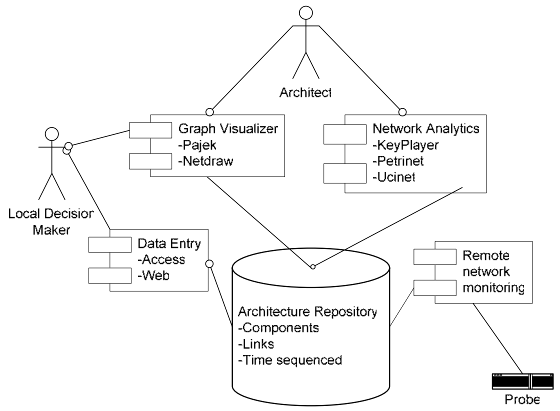

Figure 1: ECAR Architecture

In the design of the RDBMS schema we aimed for maximum flexibility, even it resulted in a design that is more complicated than required for a specific application in which the requirements are well defined [56]. As a case in point, during a visit to one of our research sites the system developers were interested in storing a great many attributes of their system that we didn’t anticipate. However, our design anticipated our ignorance of the requirements for any particular site and has support for an unlimited number of user-defined attributes.

We intend to analyze the emergent architecture of information systems. The modular component and its dependencies are our focal point, but the system can incorporate many node types, such as users, databases, specifications, protocols, etc. Our analysis focuses on dependencies between components using network analysis. Software components, users, databases, specifications, etc. are treated as nodes while the dependencies between them are edges. Dependencies can be categorized into multiple classes: call interface, state change (database records), action sequence, exception handling, etc. The analysis used in the case study captures call interface dependencies between software components.

There is no existing system designed to capture, analyze and present emergent architectures. Other component repositories [57] support search and retrieval. Therefore, we designed ECAR to support our research efforts. Our repository supports research into the emergent network of in-use components. The following sections describe the database schema and associated application. Of course, the application we've described here is not the only possible producer or consumer of data in the repository

7 Repository schema

The schema supports drill-down, navigation, and network analysis. All designs are assumed interconnected and fractal. As the user drills down into the details of a design, she sees similar patterns at each level. Architectures are built from components, which in turn can be architected/decomposed. This design and the associated processing logic assume that non-cyclic designs will be entered into the system.

8 Table Descriptions

The following sections describe selected tables. Space limitations restrict us from fully describing all the attributes of each table (see Figure 2).

The **Element** table contains the name and description for each module (component) within the design (architecture). Examples of elements include software components, databases, organizations and users (indicated by lngObjTypeId). An element also has measures of size and complexity.

Associated with each element is the organization responsible for maintaining the element (lngRespID) and the organization that owns it (lngGroupID). The community that owns the element may not be the same as the community that builds it [1].

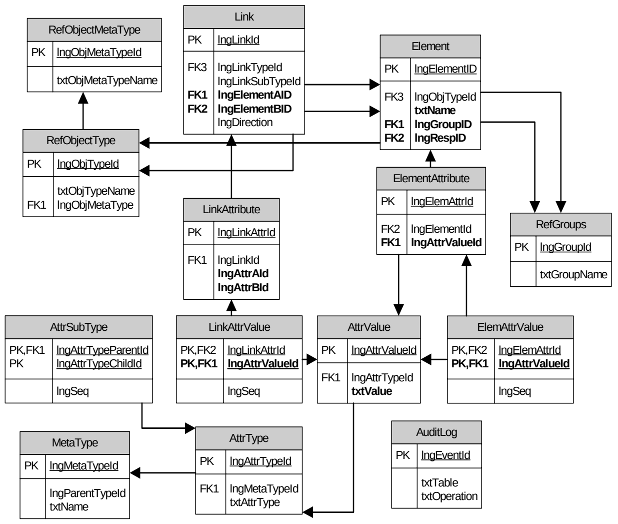

Figure 2: Repository Schema

Elements have many attributes. The attributes include such information as specific design parameters (instruction length, instructions, etc.), design specifications (pointers to design documents), and domains (Workgroup, Department, Division, etc.). Elements of different types (e.g., software modules) will have attributes of a corresponding type (e.g., interfaces).

The **ElementAtttribute** table provides the mechanism for associating an unlimited number of attributes with an element. The attributes include design parameters, documentation, domain information, and interface definitions. Each attribute contains an attribute type and an attribute value – a name/value pair. Moreover, attribute types can be grouped into MetaTypes. For example, the Meta Type might be domain and could include values such as collective, enterprise, subsidiary, division, department, or group. Other MetaTypes could include documentation, technical representation, upper systems model, etc.

Many element attributes are compound objects. For example, an interface to a software module may be a function and require a function name and ordered list of parameters. Or, it may be a message placed onto a queue. In this case, the attribute may be described by a combination of message name, format, protocol, queue name, port, etc. The cost of changing an attribute (design decision) is also captured and used for subsequent network analysis.

The **Link** table provides the mechanism for linking elements. Link types include decomposition, interface, or hyperlink (lngLinkTypeId). A decomposition link connects a parent element to a child. An interface link connects two modules by virtue of software, contract (legal relationship), or set of expectations. A hyperlink links two modules according to some other association of interest to the designer. The types of links in any implementation are up to the user. Each link also has a subtype (lngLinkSubtypeId). For example, the user could divide interface into function, message, data, etc.

Each Link represents a dependency between two elements. A module, A, is dependent upon another module, B, if a change to B requires a change to A. If and how modules A and B communicate is an attribute of the Link or of the interface definitions for A and B.

Some dependencies are more important than others. Not every change in the independent module will break the dependent module. The design provides for some measure of the cost of the dependency and for a measure of transaction volume between component pairs. Both cost and volume information factor into network analysis.

The **LinkAttribute** table specifies which attributes of the dependent element are dependent upon which attributes of the independent element. The design assumes that dependencies are unidirectional. Bidirectional dependencies imply interdependency and a lack of modularity.

9 Design Discussion

A user of this system needs to think about how much detail to capture. Using this system forces the designer to document dependencies. One challenge will be developing tools to test the completeness of the designer's model with automation. In addition, not all design dependencies can be captured. Not only are ElementAttribute, LinkAttribute, or AttrSubType not recursive, but some dependencies between modules are subtle and can only be found when trying to make a modification. It is the hidden interdependencies that lead to the greatest problems, and these interdependencies are often unknown [12].

The design is intended to be general purpose. It can be used for mapping organizations, hardware systems, software systems, and other systems. It supports layered networks. One network can describe an organization, another can describe the information system, and a third can describe the computer systems that support the information system. Links exist between modules in each layer, and between modules on different layers.

10 Case Example

In order to more fully motivate ECAR, and test some of the claims we make for it, we present selective information regarding one of the organizations we are working with as we develop our tool. FinServ is an organization that provides processing services to the investment management industry. FinServ has over a trillion dollars in assets under management. They provide global full service transfer agency and accounting services.

Over the last couple of decades, FinServ has grown rapidly primarily through mergers and acquisitions, and also by launching new services and entering new regions. As companies were acquired, each line-of-business (LOB) maintained much of the decision rights to control most aspects of their (line-of) business, including design, sales, back-end processing and IT services. The resulting enterprise architecture has duplication of functionality, limited integration, and a wide variety of user interfaces. Each LOB was empowered to focus on, and independently respond to, specific opportunities and needs. In the language of this paper, there was local cognition but no distributed, shared understanding of the information system.

The FinServ architecture had several characteristics that prevented FinServ from quickly responding to changing market conditions. These characteristics included tight coupling between systems, monolithic solutions (often not well documented), and the duplication of functionality (as noted above). This was sustainable during the economic growth phase of the 90's, but not when their efforts turned towards cost reduction and information system rationalization.

In response to this predicament, FinServ hired a new CIO and created a new corporate initiative to address the problems. The IT organization was changed, creating an enterprise architecture team, a management oversight committee, and an organization to provide company-wide shared application services. A FinServ enterprise architecture strategy was created and a Global Platform, SOA-based, enterprise architecture was defined. Their core mandate was to reduce IT spending though the elimination of redundant systems, componentization and exposure of core functionality through accessible services, redesign of strategic shared applications and components and the provision of mutually consistent customer access channels (web, voice, wireless, etc.). What FinServ needed to develop was a shared understanding of the architecture.

As a result of an initial inventory of the systems at FinServ and the connections between those systems, we were able to store and then represent their emergent enterprise architecture. As shown in Figure 3, FinServ's systems are not isolated: FinServ has 158 interconnected systems (nodes) in their enterprise architecture (network). It has a few highly integrated systems and many sparsely integrated systems.

Figure 3 represents the simplest visualization and is intended only to show the interconnectedness of the applications that constitute the information system. However, because the repository contains many different application and link attributes, the visualizations can become far richer. Nodes and links can be sized and colored according to attributes such as hardware platform, operating system, investment, business unit, age, or transaction volume, to name a few. By changing the size, color, shape, and line-thickness, the simple visualization can convey a lot of important information. Furthermore, database driven visualizations can be changed with a few keystrokes.

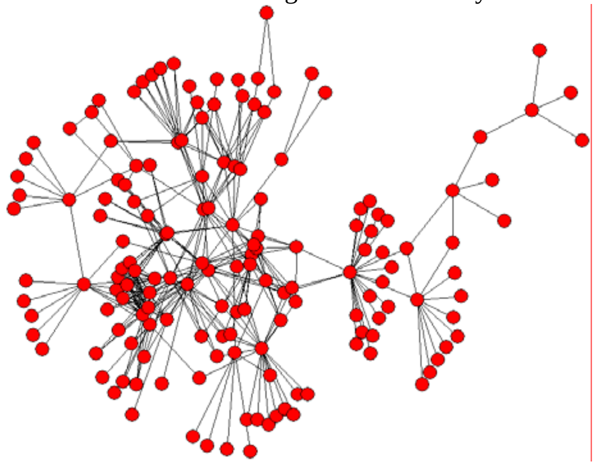

Figure 3: FinServ Architecture Visualization

11 Conclusions, limitations and future work

As mentioned in the introduction, architecture is largely emergent. As individual projects are implemented, they affect the dependencies between systems and, thereby, the emergent architecture. The ECAR system allows the individual project stakeholders to enter component and dependency data about their systems into the repository. In addition, this system provides a view of the emergent architecture (see Figure 3). Nodes and links within this diagram can be used to derive practical implications for the organization.

According to the chief architect at FinServ, the network visualization and associated simulation is “a powerful communication tool to gain the support of people to do the right thing.” The big issue is being able to visualize what is happening to the architecture and then being able to show others without “wading through reams of spreadsheet data.” “Communications with decision makers and stakeholders at a senior level is notoriously difficult. They live in an old world where things were not complicated.” The decision makers take an attitude that, “IT is difficult. So, I will buy it from outside. There is still very much the silo model.”

The silo model is one in which decision makers are either ignorant of the dependencies between systems or choose to ignore them. By taking a more holistic approach to system architecture the organization can develop a distributed understanding of their system and make different decisions regarding its evolution. Outsourcing application silos does not eliminate the need to integrate them; it only makes it potentially more difficult as the number of organizations involved with the architecture increases.

This paper describes ECAR (Emergent Component Architecture Repository), a tool we are developing that captures the time-sequenced evolution of architectures. Through it, we capture information about information systems, including its components, interfaces and dependencies. This paper describes the initial database schema, supporting data-entry application, and preliminary analysis tools.

ECAR supports a variety of navigational and analytical tools. Users can decompose large-grained components, they can segment the information system by project or sponsoring organization, they can examine the architecture based upon dependency type or other attributes. Users can interactively navigate the links of the architecture’s graphical representation. In addition, users can analyze the level of modularity in their architecture through static and dynamic network analysis. The links between nodes vary by dependency type and can be weighted by complexity and transaction volume.

As part of future work, we will be using a combination of ECAR and more intensive field study to better understand the organizational impact of disjoint cognition and the role and consequences of ECAR facilitating the development of a shared cognition of the entire system in contrast to the individual cognitions of individual components. We are interested in how the tool is used to enter data, how the visualizations facilitate understanding, and the managerial insights the tool supports. We are also conducting investigations regarding the relationship between network typology, a nodes position in the network, and outcome measures such as IT flexibility, innovation, and project predictability. Finally, we are exploring financial models that link real-option investment considerations to enterprise architecture.

12 References

1. Zachman, J.A., *A framework for information systems architecture.*, in *IBM Systems Journal*. 1999 [1987], IBM Corporation/IBM Journals. p. 454.
2. McKay, D. and D. Brockway, *Building IT Infrastructure for the 1990s*, Nolan, Editor. 1989, Norton & Company: New York. p. 1--11.
3. Kayworth, T.R., D. Chatterjee, and V. Sambamurthy, *Theoretical Justification for IT Infrastructure Investments.* Information Resources Management Journal, 2001. **14**(3): p. 5--14.
4. Henderson, J.C. and N. Venkatraman, *Strategic alignment: Leveraging information technology for transforming organizations.* IBM Systems Journal, 1993. **32**(1): p. 4-16.
5. Perry, D.E. and A.L. Wolf, *Foundations for the Study of Software Architecture.* ACM SIGSOFT Software Engineering Notes, 1992. **17**(4): p. 50-52.
6. Duncan, N.B., *Capturing flexibility of information technology infrastructure: A study of resource characteristics and their measure*, in *Journal of Management Information Systems*. 1995, M.E. Sharpe Inc. p. 37.
7. Boland, R.J.J., R.V. Tenkasi, and D. Te eni, *Designing information technology to support distributed cognition.* Organization Science, 1994. **5**(3): p. 456.
8. Orlikowski, W.J., *Improvising Organizational Transformation Over Time: A Situated Change Perspective*. Information Systems Research, 1996. **7**(1): p. 63-92.
9. Iyer, B. and R.M. Gottlieb, *The Four-Domain Architecture: An approach to support enterprise architecture design.*, in *IBM Systems Journal*. 2004, IBM Corporation/IBM Journals. p. 587-597.
10. Henderson, R.M. and K.B. Clark, *Architectural Innovation: The Reconfiguration of Existing Product Technologies and the Failure of Established Firms.* Administrative Science Quarterly, 1990. **35**(1): p. 9-30.
11. Hopkins, J., *Component Primer.* Communications of the ACM, 2000. **43**(10): p. 27-30.
12. Simon, H.A., *The Sciences of the Artificial.* Third ed. 1996 [1969], Cambridge, MA: The MIT Press. 231.
13. Nonaka, I., *A Dynamic Theory of Organizational Knowledge Creation.* Organization Science, 1994. **5**(1): p. 14-37.
14. Cook, S.D.N. and J.S. Brown, *Bridging Epistemologies: the generative dance between organizational knowledge and organizational knowing.* Organization Science, 1999. **10**(4): p. 381-400.
15. Carlile, P.R. and E.S. Rebentisch, *Into the black box: The knowledge transformation cycle.* Management Science, 2003. **49**(9): p. 1180-1195.
16. Ware, C., *Information visualization: perception for design*. 2000, San Francisco: Morgan Kaufman. xxiii 438 p.
17. Hevner, A.R., et al., *Design science in Information Systems Research.* MIS Quarterly, 2004. **28**(1): p. 75-105.
18. Ross, J.W., *Creating a Strategic IT Architecture Competency: Learning in Stages.* MIS Quarterly Executive, 2003. **2**(1): p. 31-43.
19. Morris, C. and C. Ferguson, *How Architecture Wins Technology Wars.* Harvard Business Review, 1993. **71**(2): p. 86-97.
20. Sauer, C. and L.P. Willcocks, *The Evolution of the Organizational Architect.* Sloan Management Review, 2002(Spring): p. 41-49.
21. West, J. and J. Dedrick, *Innovation and Control in Standards Architectures: The Rise and Fall of Japan's PC-98.* Information Systems Research, 2000. **11**(2): p. 197-216.
22. Richardson, G., B. Jackson, and G. Dickson, *A Principle-Based Enterprise Architecture: Lessons from Texaco and Star Enterprise.* MIS Quarterly, 1990. **14**(4): p. 385-403.
23. Byrd, T. and D. Turner, *Measuring the Flexibility of Information Technology Infrastructure: Exploratory Analysis of a Construct.* Journal of Management Information Systems, 2000. **17**(1): p. 167-208.
24. Weill, P. and M. Broadbent, *Leveraging the New Infrastructure: How Market Leaders Capitalize on Information Technology*. 1998, Boston: Harvard Business School Press.
25. Malone, T.W. and K. Crowston, *The Interdisciplinary Study of Coordination.* ACM Computing Surveys, 1994. **26**(1): p. 87-119.
26. Messerschmitt, D. and C. Szyperski, *Software Ecosystems: Understanding an Indispensable Technology and Industry*. 2003, Cambridge, MA: The MIT Press. 424.
27. Nezlek, G.S., H.K. Jain, and D.L. Nazareth, *An integrated approach to enterprise computing architectures.* Communications of the Acm, 1999. **42**(11): p. 82-90.
28. Parnas, D.L., *On the Criteria To Be Used in Decomposing Systems Into Modules.* Communications of the ACM, 1972. **15**(12): p. 1053-1058.
29. Larman, C., *Applying UML And Patterns*. 3 ed. 2005, Upper Saddle River, NJ: Pearson Education, Inc. 703.
30. Orlikowski, W.J. and C.S. Iacono, *Research commentary: Desperately seeking the "IT" in IT research - A call to theorizing the IT artifact.* Information Systems Research, 2001. **12**(2): p. 121-134.
31. Tillquist, J., J.L. King, and C. Woo, *A representational scheme for analyzing information technology and organizational dependency.* MIS Quarterly, 2002. **26**(2): p. 91.

32. Leifer, R., *Matching Computer-Based Information Systems with Organizational Structures.* MIS Quarterly, 1988. **12**(1): p. 63-73.
33. Lawrence, P.R. and J.W. Lorsch, *Organization and Environment; Managing Differentiation and Integration*. 1967, Boston,: Division of Research Graduate School of Business Administration Harvard University. xv, 279.
34. Simon, H., *The architecture of complexity*, R. Garud, A. Kumaraswamy, and R.N. Langlois, Editors. 2003, Blackwell: Malden, MA. p. viii, 411 p.
35. Boland, R.J.J. and R.V. Tenkasi, *Perspective Making and Perspective-Taking in Communities of Knowing.* Organization Science, 1995. **6**(4): p. 350-372.
36. Nelson, R. and S. Winter, *An Evolutionary Theory of Economic Change*. 1982, Cambridge (MA): Harvard University Press.
37. Argyres, N.S., *Technology strategy, governance structure and interdivisional coordination.* Journal of Economic Behavior and Organization, 1995. **28**: p. 337-358.
38. Kling, R., *Social Analyses of Computing: Theoretical Perspectives in Recent Empirical Research.* ACM Computing Surveys, 1980. **12**(1): p. 61-110.
39. Banker, R.D., et al., *Software complexity and maintenance costs.* Communications of the ACM, 1993. **36**(11): p. 81.
40. Brooks, F.P., *The mythical man-month: essays on software engineering*. 1975, Reading, Mass.: Addison-Wesley Pub. Co. xi, 195 p.
41. Szyperski, C., D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming*. Second ed. Component Software, ed. C. Szyperski. 2002, New York: Addison-Wesley.
42. Bowker, G.C., S. Timmermans, and S.L. Star, *Infrastructure and Organizational Transformation: Classifying Nurse's Work*, W.J. Orlikowski, et al., Editors. 1996, Chapman & Hall: London. p. 344-370.
43. Thompson, J.D., *Organizations in action: social science bases of administrative theory*. 1967, New York: McGraw-Hill. xi, 192.
44. Barabasi, A.-L., *Linked: The New Science of Networks*. 2002, Cambridge, MA: Perseus Publishing. 280.
45. Watts, D.J., *Networks, dynamics, and the small-world phenomenon.* American Journal of Sociology, 1999. **105**(2): p. 493-527.
46. Borgatti, S.P. and P.C. Foster, *The Network Paradigm in Organizational Research: A Review and Typology.* Journal of Management, 2003. **29**(6): p. 991.
47. Wasserman, S. and K. Faust, *Social Network Analysis: Methods and Applications*, ed. M. Granovetter. 1994, Cambridge: Cambridge University Press.
48. Batagelj, V. and A. Mrvar. *Pajek Program for Large Network Analysis*. 2004 [cited 2004; Available from: http://vlado.fmf.uni-lj.si/pub/networks/pajek/.
49. Dreyfus, D. and B. Iyer. *Enterprise Architecture: A Social Network Perspective*. in *HICSS-39*. 2006. Kauai, Hawaii.
50. Borgatti, S.P., *The Key Player Problem*, in *Dynamic Social Network Modeling and Analysis: workshop summary and papers*, R.L. Breiger, et al., Editors. 2003, National Academy of Sciences Press: Washington, D.C.
51. Borgatti, S.P. and D. Dreyfus, *Key Player*. 2005: Harvard, MA.
52. Karp, R.M., *Reducibility among combinatorial problems*, in *Complexity of Computer Computations*, M.a. Thatcher, Editor. 1972, Plenum Press. p. 85-103.
53. Ousterhout, J.K., *Scripting: Higher-Level Programming for the 21st Century.* IEEE Computer, 1998. **31**(3): p. 23-30.
54. Krueger, C.W., *Software Reuse.* ACM Computing Surveys, 1992. **24**(2): p. 131-183.
55. Brooks, F.P., *No Silver Bullet - Essence and Accidents of Software Engineering (Reprinted from Information-Processing 86, 1986).* Computer, 1987. **20**(4): p. 10-19.
56. Markus, M.L., A. Majchrzak, and L. Gasser, *A design theory for systems that support emergent knowledge processes.* MIS Quarterly, 2002. **26**(3): p. 179-212.
57. Padmal, V., Z. Fatemeh "Mariam, and J. Hemant, *Knowledge-Based Repository Scheme for Storing and Retrieving Business Components: A Theoretical Design and an Empirical Analysis.* IEEE Transactions on Software Engineering, 2003. **29**(7): p. 649-664.