Lorado: Architecture based system creation

The goal of Lorado is to reduce the time and effort needed to build and change large information systems.

Lorado is a model schema and toolset based on that schema to maximize automation in large information system construction, and enable them to be designed, built, and managed by smaller more agile teams.

If enterprise system architectures can be documented in a consistent schematic way, then the schematic can be used as the foundation for tools that will result in large efficiency improvements in the overall time and cost to deliver and enhance the systems. Efficiencies will result from improved code generation, early resolution of integration issues, narrow modularization of testing scope which will simplify distribution of development to widely dispersed teams, vendor neutrality in virtualized deployment resulting in competitive pricing, more streamlined architectures, faster system development and redevelopment due to pattern re-use, and higher leverage of business domain expertise resulting in faster system evolution to a complete business vision.

Tools exist to help with the aforementioned benefits, but lack a unifying vision for how they could be aligned to a cohesive architecture.
Further, when a large enterprise system has been achieved and evolves over time, it is difficult to communicate even if some body fully understands it. Even in cases where it becomes sufficiently communicated for some large proposed change, it takes many verbal and visual communication efforts to pass along understanding to the teams that attend to the vision in

enough detail to change it or recreate a portion and deploy a new working system. [Houston to Milton number]

Many high quality runnable [open source] tools tools exist to store, serve, and process information, and skilled IT practitioners can assemble them into large well integrated systems, and even systems of systems. Bringing these large systems to realization currently requires a number of teams and incremental delivery efforts.

In addition to runnable tools to store, serve and process information, there are great languages to create the unique capabilities and even specification methods for service APIs to explain how one system can be interacted with. Missing from the technologyscape however is an efficient method to *specify* which functions to store and process information belong to which systems behind the service APIs and why. [why alludes to the higher level mapping of capabilities to IT functions] [efficient means we can generate system stubs and containers from it because it has sufficient detail to do so] [need to address composition by function and information store instead of entrenched team]

In the current state of massive availability of virtualized infrastructure, the time to bring systems to realization is not at all limited limited by hardware acquisition cycles as in the past. Time to realization is however limited by two developmental phases: the time to *decide* what to build, and the time to *describe* what to build. [show later that *efficacy in describing what to build* greatly reduces the effort in actually building it] [standards, automation and architectural pattern reuse will greatly reduce the time required to actually build systems, but it is *dependant on efficacy in describing what to build*.]

Literature about innovation and business startup published in the last n years [i think n= 10] centers around the idea of '*fail fast*'. The fail fast approach can be expedited by improvements in both the time to decide and in the time to describe.

With an efficient way to describe and visualize the system, it will be possible to evaluate directions faster. And with less time to fulfillment, decision on business direction can occur in minimum viable product trials that accelerate learning in the business place.

IT System Modeling and the Pure Information Organization (Difference From EA Tools)

Enterprise Architecture Tools generally accommodate an existing system of people, and cater to their needs, including 'People' as resources. [ArchiMate 3.0 in Practice (Part 2 Capability-based Planning) - The Open Group] By contrast, the approach described here addresses an IT system. By extension, it is believed that many businesses can be modeled as pure information organizations where by, given an information state of the business, customer wishes can be expressed in terms of information, typically a request, which will update the state of the business, and generate outputs, typically returning information state to the customer also. It is also believed that the information architecture does not have an edge at the information boundary with the customer, but that such boundaries are and extrapolation of Conway's Law.

Comparing to Togaf, this approach is all about information lifecycle ad movement. Togaf by contrast barely even mentions information manipulation. This approach may be considered first, what Gartner called Mode 2 development, and second, a way to tie mode 2 systems to mode 1 systems.

Lorado and the intended tool implementation extend to agile principles to large system construction with short times to working production software.

As contrasted to ADLs, such as Wright, Lorado is a low abstraction way to describe first a set of information and functions on that information, and second the implementation requirements agnostic of what a deployable implementation. Deployed environments of different scales can be created from the same functional architecture. A small QA environment with the same functionality as a large production environment can be built from the same architectural specification because environment specific parameters are left until deployment details are known.

Functional Approach

So much of what is done in IT organizations on a daily basis is thought of on the fly in reaction to problems as they emerge that it may seem like there is plan that could possibly be known to get from a business goal that has been dreamed up without any concrete notion of what systems are needed to achieve it. And while talent, expertise and flexibility are the chief ingredients in successfully defining and bringing systems to life, much of expertise draws on recognition of applicable patterns that have been used before. There are large efficiency gains possible by identifying, and capturing for re-use the patterns being used in large systems just as single applications components now benefit from reuse of repeatable software patterns.
To capture that benefit, a language of architecture is needed to express the solutions, and put them forth as candidates for reuse.
No language for architecture to date is in wide use. This paper will establish conceptual foundations for a language of IT architecture in hopes that it can tie many areas involved in IT processes together into a standard ecosystem of progress.

Where major architectural tools such as TOGAF or the Zachman Framework focus on an analytical process to be tuned to the needs of an organization, this paper will take an applied approach that is in many ways less ambitious and less general, but is instead more pragmatic and specific. This architecture specification approach will be validated by the successful implementation of tools that IT practitioners will use to dramatically reduce the time and effort it takes to build beneficial business impacting systems.

The centerpiece of the architectural specification is a layer of specification that first enumerates the logic components that house the information manipulation operations required to fulfil the business objectives and are actually deployable, and second identifies the information that must be conveyed from any of those deployable components to others for further manipulation. The full architecture documents events that cause information to initially enter some component and all the information that gets created as a result of chains of events and conveyances of information amongst components.

The aforementioned centerpiece is what a developer and an architect would call a ***functional*** architecture in the context of a business system because it specifies information operations in a way that demonstrate that the business functions will be fulfilled if the system can be built and run.   The functional architecture definition has further specification that must be added to indicate what choices and configuration must be made to describe a valid deployment of the functional architecture. This additional part of the architecture supports some of the ***nonfunctional*** requirements of the system, especially scalability, availability, and cost.

Through the business objectives achieved by the functional architecture, there is an opportunity to relate progressions in the architecture to planned projects identifying further business goals, and project the effort level and cost of the architectural progression as an input to project prioritization processes.

The architectural model relates through parameter definition in the nonfunctional elements to the various deployed environments such as the environments run the business, and also any customer trial, or QA environments needed to prove out business and technical integrations. The defined parameters, with values provided to bind the architecture to infrastructure resources are sufficient to automatically generate scaled out and configured runnable infrastructure as software environments that are implicitly valid against the architecture and transitively therefore against the business objectives.   The generated ***environment description*** serves as a deployment spec that can be used to support IT operational controls and provide cost traceability to business functions of the architecture.

Prior to it’s realization in runnable environments, the functional architecture is used to support development through service and message specification tools and processes which in turn generate implementation stub code.   The stub code represents separately a an interface contract for the producer of information in a data relationship and a complimentary consumer view of the same contract in the relationship.   Unit tests within components involved in the interface allow for early detection of problems that might otherwise not be found until deployment in integration testing environments. Further with stub code for all inputs and outputs, the work to implement in information manipulation functions of the components is well defined and easily testable.

The pieces of the Model

The Functional Architecture

With minor exceptions, a program has to run for an IT system to do anything. Because of this, the logical building blocks of IT systems are the program artifacts that can be deployed and run.

To begin putting architectural specification into concrete terms the first foundational object type in the data model is the ‘deployable component’: the application code and configuration bundle that can be deployed to run a set of information manipulation operations. Those operations will require data inputs from other ‘up stream’ component types in the architecture and provide data for other ‘down stream’ component type in the architecture.   To execute, code runs on a host or in a runnable container, so the deployable components also define how the the functional architecture binds to infrastructure in deployment.

The model specifying functional architectures includes a ‘relationship’ type. The relationship type is a model type that provides a high level explanation of what data objects are produced by one ‘deployable component’ type for use by another. The relationship has a related object type called an ‘architectural message’ representing a set of related messages to carry data about and describe operations on a high level ‘business data object’, or set of related objects. It is in these messages that a data architecture relates to the functional architecture. A section to follow will describe how the relationship is also used to specify what nonfunctional components may be configured within the specified functional architecture, but which may vary across deployed environments, depending on environment specific requirements.

To be descriptive, flexible, and support complex relationships involving multiple components, especially multicast relationships, the model includes concept of a ‘participation’ in the relationship by the deployable component. The participation includes a role indicating whether the component is producing or consuming data from the relationship, and a participation pattern indicating how groups of the component type relate to each other when they are participating on the same relationship. Participation patterns enumerate the possible ways that groups of data producers may share transmission media to pass data to groups of consumers, and will be described in more detail in the deployment section below.

Layers needed for diagramming:

1. Business data
2. Message (encoding)
3. Transport type (protocol stack)

The Deployment Characteristics of the Non-functional Architecture

Even in cloud environments or completely virtualized datacenter deployment environments, the host that runs the program code remains an important piece of the model that connects functional architecture pieces to each other and to the infrastructure

- Show how DNS names or IP addressesa serve as proxy for hosts in the scale out of the environment.
    - Need new section on cloud scaling styles
        - Application managed scaling
        - Name based scaling is a location broker pattern vs on the fly IP provisioning
    - Business partitioned scaling, or transports such as MD channels.
    - Enumerated scaling of hosts such as market segments.
    -

the application to be capability and ties together any technologies u logic s are programs, so w
-->
Though there are lower (more detailed) levels of understanding, The building blocks of IT systems are programs. Programs take data in, do something with it, and send data out. Later patterns will be shown to classify be enumeration what kinds of things programs do. First however it is useful to describe what a program is, and how information can be used by a program.

Model and Model Vocabulary

A program is a sequence of logic written in code that starts up as a running process. Then a program becomes a running process, it exists in memory with typically 2 pieces, its program text, and some data that it operates on using a CPU.   In the general sense that is all! A program has no meaningful effect on any real or virtual world until its does something to data, and the sends that data out. While there are some things one might claim violate this simplicity, we can relax our definitions and assert some truths to end up with a powerful way to talk about systems, and no harm is done by relaxing the terms.
Projects are collections of Business Capabilities to be delivered together. Projects can be refactored to group business capabilities differently.
Business Capabilities are built from Information Features.

The above objects represent the Roadmap Model

Information Features are collections of Chains of Functions triggered by Events that produce a business meaningful result.

It is only necessary to document the Edge Events that kick off Information Chains, and assume that other events in the chain are implied when the output of one Function is the Input event that triggers the next Function in the chain.
Functions use Information Inputs to either change the state within a container or produce output or both. [The patterns of Functions can be enumerated in a list of a dozen or two]
Information Inputs can be thought of as ‘arguments’ to Functions.
Operations are the units of processing that are only of internal concern to the Function.
Deployable Components contain collections of Functions. Architectures can be re-factored to move Functions so they are grouped differently in Deployable Components. Relations represent movement of events or data from one Deployable Component to another.

The above objects represent the Functional Model.

Relation Implementations Describe the Protocol Stacks and Scaling Units will be used to move information from one Deployable Component to another when a system is deployed into an environment. … Host… Platform…

The above objects represent the Implementation Architecture.

Scaling units values named object business partitions are enumerated scaling units.

The above values provide parameters to generate an Environment Description from an implementation Architecture.
...

Fundamental Component patterns

Combine data inputs to make new data
Store data
Collect human input (UI)
Sense data

High level Component Patterns

Proxy
Cache
Change transport (Bridge)
Change Application Protocol (Gateway)
Service location broker
Queue
Store
Direct messages to transport based on content (Router)

Referential UI.
Messaging Component FT
Web load balancing
Global Server Load balancing