Section 3: Week 7: Immortal Software Systems

Nate Bachmeier

TIM-8110: Programming Languages and Algorithms

May 19, 2019

North Central University

# Immortal Software Systems

## Background

The Building Resource Adaptive Software Systems (DARPA BRASS) has presented an open challenge to come up with innovative solutions to increase the product lifecycle of complex systems. Many applications already offer support up to a decade but there are scenarios which require significantly longer periods, such as a century or more.

There are several key scenarios that warrant the need for these long support lives and many challenges that need to be overcome. Today businesses address these extended life scenarios by simply throwing money at the problem. This is not an economically efficient nor does it allow for the businesses to reap long term returns from their investment in the code base.

To address these challenges requires a multi-faceted approach starting with the languages used to build the software through encompassing the runtime behavior.

1. Domain Specific Languages (DSL) enable the development team to decouple the ‘how’ from the ‘why’ and create forward compatible software. These new languages allow for a decoupling of the system libraries from the communication and hosting model.

2. Aspects of the larger environment need to be injected at design and runtime. Often the algorithms can be designed to operate on common data structures through standardized interfaces. Consider a queue that is in-process or hosted by a cloud provider, the mechanics of pushing and popping are still consistently the same.

3. The use of metaprogramming to extend existing General-Purpose languages to express additional sematic validations. Having expressive capabilities and requirements forces the code to be self-documenting and enables tooling to catch many compatibility errors during the design phase.

## Problem Statement

The Mechanization of Contract Administration Services (MOCAS) was launched in 1958 as an advanced system that operates on punch cards. Today it is still operational and managing roughly 1.3T$ in obligations and government contracts. There have been several efforts over the years to replace it, however the criticality of the system prevents any modification (Verma, 2017).

The same scenario is likely to happen for many future distributed systems such as health care networks, utility control systems, and arbitrary backend services across the business community. Perhaps the scenarios will not be as extreme, however the costs to maintain these legacy systems will be proportionally high.

As the manufacture of these systems drops support the businesses are left with the choice of paying an uncompetitively priced third parties or relying on proprietary solutions. In either scenario the legacy technology is not capable of adapting modern frameworks and platforms; which can increase performance, reliability, and security. This can lead to critical systems left vulnerable to attack, operations that are slow to response or costly failures that nearly impossible to fix due to the regression risk.

The scenario is partially mitigated using virtualization technologies, such as virtual machines; containers; and system call emulation. While each of these technologies allows for the continued longevity of the legacy system it does not address any of the shortcomings. To fully address making the system continuously upgradable, new extensions are required during the compilation phase. One method for implementing these extensions is with a recursive build approach, where DSL languages express different aspects of the system and emit code which is targeted to modern platforms. As the definition of modern changes, the compilation can be centrally modified to emit the desired behavior.

## Goal

Creating a paradigm that makes future software future compatible only addresses half the problem. Most businesses also have existing systems that need a mechanism for promoting their source code into this model. A study is first required to determine a prioritized list of language constructs that need to be promoted forward.

Based on the survey DSL languages will be created to allow the expressing dependencies between components. The language would then be capable of emitting distributed call graphs and performing static analysis across disconnected systems. This information would allow for optimizers to understand which parts could be repackaged such that they provide the most reliability. It would also be possible to tell from the graph which aspects of the system could be rewritten to inject modern platforms using annotations instead of complex manual edits.

For instance, Contoso is an educational software company with numerous systems written in C# for Windows 2008. Their goal is to migrate from private data centers into a public cloud such as Amazon Web Services. The DSL language could express the endpoints of each micro service, their capabilities and service requirements.

At compilation time the DSL emits Infrastructure as Code (IaC) scripts to create serverless API gateways and a rehosts the code using modern Function as a Service (FaaS) technologies. The DSL might also express the need for a work item queue which is presently implemented with in process objects. Instead the code is rewritten such that the cloud native managed service is injected. This further increases the reliability and scalability of the system with minimal changes to the business logic.

Perhaps there is scalability challenges when a certain operation is performed. The DSL language could express the need for auto scaling only that specific class. At compilation time data contracts could be constructed and the calls translated into RPC. With the complexity decoupled from the primary application it then becomes possible to farm out the operation across hundreds of cores.

## Relevance and Significance