Language-Based Architectural Control

Jonathan Aldrich and others[1]

Carnegie Mellon University and Victoria University of Wellington[1]
{aldrich}@cs.cmu.edu and others@ecs.vuw.ac.nz[1]

Abstract. Software architects design systems to achieve certain quality attributes, such as security, reliability, and performance. Key to achieving these quality attributes are design constraints on communication, resource use, and configuration of components within the system. Unfortunately, in practice it is easy to omit or mis-specify important constraints, and it is difficult to ensure that specified constraints are also enforced.

Architectural control is the ability of software architects to ensure that they have identified, specified, and enforced design constraints that are sufficient for the system's implementation to meet its goals. We argue that programming languages, type systems, and frameworks can help achieve architectural control in practice. The approach we envision leverages frameworks that fix or expose the important architectural constraints of a domain; language support that allows the framework to centralize the specification of those constraints so they are under the architect's intellectual control, and type systems that enforce the constraints specified by the architect. We sketch an approach to architectural control in the context of distributed systems, leveraging capabilities, alias and effect controls, and domain specific languages.

Keywords: software architecture; architectural control; distributed systems; capabilities; layered architectures; alias control; domain specific languages

1 Architectural Control

The central task of a software architect is designing an architecture that enables the designed system's central goals to be achieved [**?**]. Typically many designs can support the intended functionality of a system; what distinguishes a good architecture from a bad one is how well the design achieves *quality attributes* such as security, reliability, and performance.

Quality attribute goals can often be satisfied by imposing architectural constraints on the system. For example, the principle of least privilege is a well-known architectural constraint that, by constraining the privileges of each component to the minimum necessariy to support the component's functionality, enhances the security of a system. Likewise, constraints concerning redundance and independence of failure-prone components can aid in achieving reliability concerns. Broadly speaking, a constraint is architectural in natureif it is essential to achieving critical system-wide quality attributes.

Unfortunately, delivering systems with the desired qualities can be challenging in practice. The possible barriers are many, but in this work we focus on two significant barriers: missed or incorrect constraints, and inadequate constraint enforcement. If an architect is skilled at design but is not an expert in a software system's target domain, the architect may miss constraints that are important to achieving goals in that domain. For example, many architects who were not familiar with the intricacies of Secure Sockets Layer (SSL) configured their SSL libraries to unnecessarily use a heartbeat protocol,[1] and/or neglected to properly enable SSL certificate checking. The result was exposure to the Heartbleed bug in the first case [**?**], and to a man-in-the-middle attack in the second [**?**].

Even if the relevant constraints are identified and specified correctly by the architect, ensuring that they are followed can be quite difficult. A standard defense against SQL injection attacks, for example, is ensuring that prepared statements are used to construct SQL queries. Ensuring that this constraint if followed, however, requires scanning all SQL queries in the entire program; any query that is missed could potentially violate the policy. Similar issues apply to common defenses against other attacks, such as cross-site scripting (XSS).

Defining Architectural Control. The problems above suggest that in practice, architects do not have sufficient control of the architecture of their software systems. ***Architectural control*** is the ability of software architects to ensure that they have identified, specified, and enforced design constraints that are sufficient for the system's implementation to meet its goals. Although tools can aid in achieving architectural control—and in fact, this paper proposes ways of building better tools for doing so—our definition of the term is focused on the practice of software engineering, which can be hanced by tools but not replaced by them.

Today, architects use primarily informal processes to achieve architectural control. To learn about the constraints relevant in a domain, they learn from domain experts and consult documentation of frameworks that capture domain knowledge. To enforce the constraints they specify, they rely on informal communication with the engineers building the system, as well as quality-control practices such as testing, inspection, and static analysis. Unfortunately, testing is good at evaluating functionality but is poorly suited to enforcing many quality attributes; inspection can work well but is limited by the fallibility of the humans carrying it out; and static analysis tools are often too low-level to directly enforce the desired qualities. As a result, the degree of architectural control achieved in practice often falls short of what is needed to produce highly reliable and secure systems.

Achieving Architectural Control. How can architects do better at controling the architecture of their systems? We believe there are three key elements to achieving architectural control in practice:

– **Accessible Guidance.** Because it is difficult for architects to be expert in every domain and with all component software used, it is essential that architects be able to leverage effective and accessible guidance concerning (A) what are the important potential constraints to consider with respect to a domain or a component, and (B)

[1] SSL's heartbeat feature is only needed for long-lasting, possibly idle connections

what is the basis for choosing among and configuring those constraints. For example, a library or component should make its configuration parameters explicit, provide documentation on how to choose them, and either force the architect to make particular choices or choose safe and secure defaults. Software frameworks are often used to capture domain knowledge in a reusable library, and as such should include not just functionality but also explicit guidance concerning the architectural constraints that are relevant in the target domain.

– **Centralized Specification.** The reality of team-based development in the large is that it is not possible for a single person to review and understand all the project artifacts. For an architect to achieve architectural control, therefore, requires that the specification of architectural constraints be centralized. An architecture specification document could fulfill the role in processes that produce such a document, but in agile organizations where a lot of the architecture is distributed and embedded as documentation in the code, this becomes challenging. Centralized specification is also challenging with respect to constraint enforcement mechanisms such as types; in a type system, the specification is spread throughout the program, easing enforcement but also making it difficult for the architect to understand whether the types correctly specify the desired architectural constraints. An ideal scenario of centralized specification would place all architectural constraints in a small set of files that is under source control and where all revisions are personally reviewed and approved by the architect.
– **Semi-automated Enforcement.** Finally, once the proper architectural constraints have been identified and specified in a central way, the architect must be confident that they will be followed in the implementation of the software system. Process-based mechanisms are important, but are also as fallible as the humans carrying out that process. Where feasible, therefore, the ideal is to provide semi-automated tools that can enforce the system's architectural constraints whenever the system's code is modified or executed. In practice, full automation may not be feasible, e.g. type-like specifications may be required for tools based on type systems. Furthermore, in practice tools may not be able to enforce exactly the constraint that the architect cares about, but the closer they can come, the more assistance the tools can provide to a process-based enforcement approach.

Finally, while architectural control is clearly desirable, in practice any mechanisms used to achieve it must be cost-effective. We would like to realize the benefits of architectural control while minimizing sacrificed productivity. In the ideal case, we would like to explore whether we can build tools that move the productivity-control curve outward, providing better architectural control while at the same time actually enhancing the productivity of a development team.

[TODO: Overview and outline of the rest of the paper]

2 Architecture-Exposing Languages and Frameworks

[TODO: Write me! See separate outline document]

2.1 Extending Languages with Architecture

Goals: ensure that SSL is used in a simple client-server application. be able to audit the messages exchanged.

```
def package feedback

import FeedbackClient, FeedbackServer
import resource org.wyvern.network
import resource org.wyvern.distributed.SSLSOAPConnector(network)

language org.wyvern.distributed.ADL

architecture feedback
    component client = FeedbackClient()
    component server = FeedbackServer()
    connect client.pServer, server.pClient with SSLSOAPConnector
```

Fig. 1: Client-Server Architecture

```
package feedback

import FeedbackInterface
import org.wyvern.distributed.IPClient
import org.wyvern.distributed.IPAddress

class FeedbackClient
    val pServer = IPClient<FeedbackInterface>()
    def run(String address, String feedback)
        val server = pServer.connect(IPAddress(address))
        server.provideFeedback(feedback)
```

Fig. 2: Client code

Related goals (think of some and discuss them. Maybe use of an authentication architecture? Maybe a temporal constraint that authentication succeeds before some resource is accessed.)

Would not build this into the language (as ArchJava did, cite it), but rather use it as a library-defined extension language [cite ECOOP 2014 paper].

```
package feedback

import FeedbackInterface
import org.wyvern.distributed.IPServer
import org.wyvern.distributed.IPAddress
#import resource org.wyvern.logging.stdlog

class FeedbackServer
    val pClient = IPServer<FeedbackInterface>()
    def run()
        pClient.listen(this.callback)
    def callback() : FeedbackInterface
        new FeedbackInterface
            def provideFeedback(feedback:String)
                stdlog(feedback)
```

Fig. 3: Server code