

# Language-Based Architectural Control

Jonathan Aldrich and others<sup>1</sup>

Carnegie Mellon University and Victoria University of Wellington<sup>1</sup>  
{aldrich}@cs.cmu.edu and others@ecs.vuw.ac.nz<sup>1</sup>

**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 Motivation: Architecture and System Qualities

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; it limits the privileges of each component to the minimum necessary to support the component's functionality, thus enhancing the security of a system. Likewise, constraints concerning the redundancy and independence of failure-prone components can aid in achieving reliability concerns. Broadly speaking, a constraint is architectural in nature if it is essential to achieving critical system-wide quality attributes.

Unfortunately, delivering systems with desired qualities can be challenging in practice. Two significant sources of the challenge include missed or incorrect constraints, and inadequate constraint enforcement. If an architect 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,<sup>1</sup> 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 is 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).

## 2 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 enhanced 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.

**Architectural Control in Java.** To illustrate the challenges with achieving architectural control in the state of the practice, consider the problem of achieving architectural control for a simple distributed system to be illustrated in Java. We will examine a simple sub-problem: understanding what messages are sent over the network, and ensuring the correct protocol is used. Such an understanding is useful for a security analyst to assess the attack surface of the system; for a reliability analysis of what might occur if a network link fails; or for a performance analysis assessing where bottlenecks might lie. In Java, achieving this understanding may be difficult for the following reasons:

---

<sup>1</sup> SSL's heartbeat feature is only needed for long-lasting, possibly idle connections

- There are many parts of the standard Java library that can do network I/O; we must examine how the program uses each of them. Furthermore, we must scan all parts of the program—and all third-party libraries it uses—to find all uses of the network.
- If libraries are loaded at run time, by default they have the same access to the network as does the program that loaded them; thus we must know which libraries are to be loaded and scan them too. Restricting dynamically loaded libraries is possible using Java classloaders and security managers, but the technique used is complex and makes program construction awkward (the loaded code must execute in a new thread, for example). Furthermore, it is easy for developers to implement this technique incorrectly, sacrificing architectural control in the process.
- Once the architect identifies a component that directly accesses the network, she may want to know how that component shares this ability with other parts of the system—and this involves understanding the component’s interface. The combination of aliasing, subtyping, and downcasts supported by Java makes this difficult, however. If the component returns a value of type `Object` to clients, does this give clients the ability to send messages over the network? The `Object` interface itself provides no networking methods, but the value may be downcast to an arbitrary type that may, in general, support network access.
- In practice, systems are built in a layered manner, with high-level communication libraries built on lower ones. All paths through the network stack must be examined for the architect to get a full picture of the messages sent over the network and the protocols used. The aliasing and casting problem in the previous bullet makes this more difficult.
- A security analyst might want to ensure that SSL is being used—and might conclude, upon seeing that `Socket` objects are obtained from code such as `SSLSocketFactory.getDefault().createSocket(host, port)`, that all is secure. Unfortunately, the Java SSL libraries, like many others [?], are *insecure* by default; they do not validate the server’s certificate, opening the door to a man-in-the-middle attack. This insecurity, and the need to configure SSL Socket Factories to secure them, is not mentioned in the API documentation;<sup>2</sup> it can only be found deep in the JSSE reference guide.<sup>3</sup>

**Achieving Architectural Control.** How can architects do better at controlling 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) what is the basis for choosing among and configuring those constraints. For example, an SSL library or component should make its configuration parameters obvious (e.g. used by the main classes in the library), provide documentation on

<sup>2</sup> <http://docs.oracle.com/javase/7/docs/api/javax/net/ssl/SSLSocketFactory.html>

<sup>3</sup> <http://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERefGuide.html>

how to choose them, ensure that the default configuration is secure, and support reusable configurations (e.g. so an organization can easily standardize a configuration appropriate to its 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—and, to fit modern agile processes, we would like to see this specification manifest in code. 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. In our distributed system example, we envision that the architect could look at the system’s entry point, together with the interfaces of the components mentioned there, and immediately determine (A) all of the components that can directly access the network, (B) which components might dynamically load code, and whether they give that loaded code network access; (C) component interfaces that are complete in the sense of showing all methods that can be invoked by clients; and (D) where to look for architecturally-relevant configuration information or higher-level layers of the architecture.
- **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. Fully automated tool-based mechanisms may not be feasible for enforcing many architectural constraints. However, we outline a vision below in which the type system and run-time semantics of an *architecture-exposing programming language* will semi-automatically enforce a variety of important architectural constraints, given input from developers in the form of partial, type-like specifications.

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]

### 3 Prior Work

[TODO: a lot of missing text here. What follows are notes and fragments.]

Haskell’s monads provide a potential technical foundation for architectural control [?]. In practice, however, monads impose an overly strict discipline, requiring every operation that might transitively perform IO to use the IO monad; for many architectural control purposes, simply describing which components perform IO might be sufficient. The syntactic overhead of IO monads also leads to an escape hatch, `unsafePerformIO`, which when used subverts architectural control. Finally, in

```

1  def package feedback
2
3  import FeedbackClient, FeedbackServer
4  import resource org.wyvern.network
5  import resource org.wyvern.distributed.SSLSOAPConnector(network)
6
7  language org.wyvern.distributed.ADL
8
9  architecture feedback
10     component client = FeedbackClient()
11     component server = FeedbackServer()
12     connect client.pServer, server.pClient with SSLSOAPConnector

```

Fig. 1: Client-Server Architecture

```

1  package feedback
2
3  import FeedbackInterface
4  import org.wyvern.distributed.IPClient
5  import org.wyvern.distributed.IPAddress
6
7  class FeedbackClient
8     val pServer = IPClient<FeedbackInterface>()
9     def run(String address, String feedback)
10         val server = pServer.connect(IPAddress(address))
11         server.provideFeedback(feedback)

```

Fig. 2: Client code

Haskell’s design all IO operations are in one monad, whereas we believe many architects will find it useful to distinguish different forms of IO.

Dynamic capability systems. Mark Miller’s E and Bracha’s Newspeak module system

Ownership - useful but insufficient due to downcasts.

## 4 Architecture-Exposing Languages and Frameworks

[TODO: Write me! See separate outline document]

### 4.1 Extending Languages with Architecture

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

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].

```
1 package feedback
2
3 import FeedbackInterface
4 import org.wyvern.distributed.IPServer
5 import org.wyvern.distributed.IPAddress
6 #import resource org.wyvern.logging.stdlog
7
8 class FeedbackServer
9     val pClient = IPServer<FeedbackInterface>()
10     def run()
11         pClient.listen(this.callback)
12     def callback() : FeedbackInterface
13         new FeedbackInterface
14         def provideFeedback(feedback:String)
15             stdlog(feedback)
```

Fig. 3: Server code