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 quality attributes like security, reliability, and performance. Key to achieving these quality attributes are design constraints governing how components of the system are configured, communicate and access resources. Unfortunately, identifying, specifying, communicating and enforcing important design constraints – achieving *architectural control* – can be difficult, particularly in large software systems.
We argue for the development of architectural frameworks, built atop language mechanisms that provide for domain-specific syntax, editor services and explicit control over capabilities, that help increase architectural control. In particular, we argue for concise, centralized architectural descriptions which are responsible for specifying constraints and passing a minimal set of capabilities to downstream system components, or explicitly entrusting them to individuals within a team. By integrating these architectural descriptions directly into the language, the type system can help enforce technical constraints and editor services can help enforce social constraints. We sketch our approach in the context of distributed systems.

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 redundance 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,[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).

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:

- 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.

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

- 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;[2] it can only be found deep in the JSSE reference guide.[3]

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) 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 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).

[2] http://docs.oracle.com/javase/7/docs/api/javax/net/ssl/SSLSocketFactory.html
[3] http://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERefGuide.html

- **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 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

```
define package feedback

import resource wyvern.logging.stdlog
import feedback.FeedbackClient
import resource feedback.FeedbackServer(stdlog)
import resource wyvern.network
import resource wyvern.distributed.SSLSOAPConnector(network)
import extension wyvern.distributed.architecture

architecture feedback
    component client : FeedbackClient
    component server : FeedbackServer
    connect client.out, server.in
        with SSLSOAPConnector
            certificateAuthority = <verisigninc.com>
```

Fig. 1: Client-Server Architecture

Ownership - useful but insufficient due to downcasts.

4 Architecture-Exposing Languages and Frameworks

We propose *architecture-exposing programming languages and frameworks* as a solution to providing better support for architectural control. An architecture-exposing language is a programming language that provides primitives to facilitate exposing architecture and enforcing constraints in a centralized way. In addition to languages, we leverage software frameworks, because frameworks are already used to capture domain-specific architectures and to impose architectural constraints on applications that extend them [**?**]. An architecture-exposing framework is one that is specifically designed to encapsulate and enforce architectural constraints, while making important architectural choices more visible to architects that design their applications on top of the framework.

To make these ideas concrete, we sketch the design of a distributed system application and framework in a future version of Wyvern, a programming language we are currently designing. We show how four technical characteristics—extensible languages, capability-based module systems, architectural layering in frameworks, and typed interfaces—may serve to provide a practical initial step towards providing architectural control in this domain. Our concrete goal will be to show how the specific problem from the previous section—understanding the messages and protocols used in a distributed system—can be rendered not just possible but easy.

4.1 Making Architecture Explicit in an Extensible Language

Figure 1 shows one way of making the architecture of a simple client-server system explicit, so that it is easy for the architect to observe the messages exchanged and protocols used over the network. The approach is inspired by ArchJava's custom connector support [**?**], although we envision supporting architecture syntax via Wyvern's library-based language extension mechanism [**?**] rather than making it a core part of the language.

The code shows a client-server application for gathering feedback. The architecture is simple: there are clients, servers, and a connection between the client's `out` and server's `in` ports that sends SOAP messages over SSL. Here SSLSOAPConnector is

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

Fig. 2: Client code

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

Fig. 3: Server code

a connection library that (unlike the standard Java library) checks server certificates unless otherwise specified, with a default set of widely-accepted certificate authorities that has been overridden here to specify only the VeriSign CA.

Looking at the client code in Figure 2, we can see that the client declares the `out` port as an `IPClient` object that is parameterized by the high-level interface used for communication with the server. The `FeedbackInterface` is not shown, but it consists of a single `provideFeedback` method that accepts a string. When the client program is invoked—we envision that our Wyvern VM would support a command line syntax of the form `wyvern feedback client server.feedback.com 'my feedback here'`—the `run` method is called with the command line arguments. The client program executes by connecting to the server—yielding an object of type `FeedbackInterface`—and then the `provideFeedback` method is invoked on this object.

Similarly, when the server code in Figure 3 runs, it initializes its `in` port to listen for incoming connections, passing a callback to be invovked when a connection is received. The callback function returns an object of type `FeedbackInterface`, and the implementation responds to client messages by printing feedback to a log.

The implementation details of the approach are beyond the scope of this paper, but would follow ideas from [**?**,**?**]. While using metaprogramming and/or reflection techniques to implement the domain-specific language for architecture may be a challenging task, they are not necessarily more challenging than the reflective techniques used in

existing framework implementations such as Ruby on Rails. Complex implementations are often acceptable in frameworks if they provide corresponding simplicity or other advantages to framework users.

4.2 Using Capabilities to Control Resources

The integration of architectural specifications into the code supports centralized reasoning about architecture. However, a critical question remains: how do we know that the architecture given is a correct and complete description of what the implementation does? How, for example, can we be sure that the client or server is not using a networking library to communicate with another entity that is not shown in the architectural description in Figure 1?

We propose to use capabilities [**?**] to control the way that various parts of the program use architecturally significant resources such as the network. In our design, libraries such as `wyvern.network` are identified as *resource libraries*, and must be imported with an `import resource` construct rather than a normal `import` construct. Each package in our design is defined (`define package`) in a single top-level file (Figure 1), and that file must explicitly import all of the resources used in the entire package. A resource may only be used by other parts of the system—including imported libraries—if a capability to the resource is explicitly passed on when the top-level file imports the relevant code. For example, on line 5 of Figure 1, the `feedback` program passes the `wyvern.network` resource to the `SSLSOAPConnector` library, so that library can be used for network communication.

Now we can clearly see that the `FeedbackClient` and `FeedbackServer` cannot possibly communicate over the network, except via the `SSLSOAPConnector` as specified in the architecture—the `SSLSOAPConnector` is the only component in the entire program that has access to the network resource.

The concept of a resource generalizes naturally to other forms of I/O, recalling Haskell's monads, but without the overhead of distinguishing effectful code from functional code at the expression level. Following Miller [**?**], our intended design will require that ordinary modules have no global state; modules that need global state must be marked as resources. Furthermore, developers who feel that the functionality provided by a module should be governed architecturally can enforce this by marking the module as a resource. In our design resource-ness is sticky; for example, although the `FeedbackServer` cannot access the network, it does import a resource in order to write to a log, and therefore it becomes a resource itself.

We are not the first to propose a module system in which a module's parameters are explicitly bound; notable prior examples include Units [**?**] and Newspeak [**?**]. Our design differs from these in that we do not require all module imports to be explicitly linked; instead, we require this only of resource modules, with the result that resources are controlled but "harmless" modules can be imported anywhere with a minimum of fuss.

4.3 Discussion

[TODO: Write the rest of me! See separate outline document]

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.)