Multi-View Architecture Description and Enforcement

Amanda Liu
Columbia University
New York, NY
al3623@columbia.edu

ABSTRACT

In software development, implementation strays from the intended architecture in what is known as architectural drift as programmers introduce communication pathways between components that shouldn't be interacting. In our approach, we have embedded an architecture description language (ADL) into a programming language that restricts resource access with capabilities and generates connections via metaprogramming. Not only does this ensure that the system semantics reflect its architectural principles, but this also facilitates editing compatible with rapid software evolution.

CCS CONCEPTS

• **Software and its engineering → Software organization and properties**; *Software system structures*;

KEYWORDS

Software architecture, architecture description language

ACM Reference Format:
Amanda Liu. 2018. Multi-View Architecture Description and Enforcement. In *Proceedings of 2018 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (Splash Companion '18).* ACM, New York, NY, USA, Article 4, 3 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Software architecture allows programmers to reason about systems and create coherent designs consistent with project demands [3]. Complex architectures are often described with multiple views. Common architectural views include a module view of relationships between units of functionality, a component-and-connector (CnC) view of runtime entities and interactions, and a deployment view of component configurations and allocations [2]. The separation of concerns afforded by views can also ease software evolution. In today's languages, an architectural change entails modifying multiple files. However, integrating architecture and semantics could localize such changes and streamline software evolution.

As software evolves, it is common for programmers to violate the intended architecture by introducing unintentional interactions between components. These dependencies are generally created when programmers can make arbitrary connections between components and access libraries which provide unrestricted use of system resources. Take, for instance, a three-tier system; it is not uncommon for programmers to make direct connections between the client and database, bypassing the server. After several iterations of such changes, the architectural drift results in software that bears little to no relation to the conceived architecture [11].

In this work, we propose an architecture description language (ADL) that prevents structural violations of the intended architecture by incorporating the architectural specification into the software implementation.

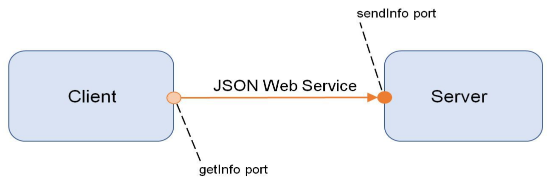

Figure 1: A simple client-server network architecture.

2 APPROACH

We implemented this ADL in the context of the Wyvern programming language. Wyvern was designed to enable programmers to express and enforce design constraints [10, 12]. In the CnC view of our ADL, systems are described using standard architectural notions such as components, connectors, ports, etc [2]. Consider the client-server system whose architecture is shown in Figure 1. This system would be described as follows in our ADL:

Splash Companion '18, November 2018, Boston, Massachusetts
© 2018 Association for Computing Machinery.
ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

```
component Client
    port getInfo: requires CSIface
component Server
    port sendInfo: provides CSIface
connector JSONWebCtr
    val host: String
    val prt: Int
architecture ClientServer
    components
        Client client
        Server server
    connectors
        JSONWebCtr jsonCtr
    attachments
        connect client.getInfo and server.sendInfo with jsonCtr
```

To implement a system using our ADL, programmers must supply implementations of the components by providing a corresponding module definition for each component type. The component ports reflect the dependencies of the modules. Since the `Client` component declares a `getInfo` port that requires the `CSIface` interface, the `Client` module takes in a corresponding argument. Likewise, since the `Server` component `sendInfo` port provides the `CSIface` interface, the `Server` module exposes a `CSIface` field.

```
module def Client(getInfo: CSIface): TCPClient
    def start(): Unit
        getInfo.getVal("key")
```

```
module def Server(): TCPServer
    val database = hashmap.make()
    val sendInfo: CSIface = new
        def getVal(key: String): String
            database.get(key)
```

To prevent unintended component interactions, our system does not leave the creation of connections up to the programmer. Instead, they are assembled at runtime with their implementations derived from the architecture. This is done via metaprogramming, wherein Wyvern types can contain metadata declaring methods that are run at compile time. Our ADL uses methods in the connector type's metadata to create component connections at compile time.

```
type JSONWebCtr
    val host: String
    val prt: String
    metadata new
        // methods for generating connector implementation
```

The modules generated to make connections must be given the capabilities to do so. Due to non-transitive authority in Wyvern, these capabilities are inaccessible to the components [8]. The client-server ports must be given network capability. The generated client port restricts network access to a socket connection to a specific host and port. The generated server port only listens for connections on the specified port.

```
module def reqPort(prop: JSONWebCtr, net: Network): CSIface
    def getVal(key: String): String
        val conn = net.connect(
          prop.host, prop.prt)
        // invoke getVal of Server
```

```
module def provPort(prt: CSIface, net: Network)
    def run(): Unit
        // accept incoming connections
```

Since our approach groups relevant concerns into multiple architectural views, it is easier to make certain modifications. For example, if a connector is changed, only the CnC view must be modified. No change has to be made to the implementation since the connections are generated based on the architecture description. Take for instance changing a client-server connection using a JSON protocol to one using a Simple Object Access Protocol (SOAP). This would normally involve changes across multiple files. Furthermore, it is difficult to ensure that all relevant changes were addressed. The same modification in our system would be limited to changes in the CnC view alone. Thus, by automatically handling connector creation, we definitively construct component interactions in terms of the ADL specification and preserve architectural integrity across system modifications.

3 RELATED WORK

Many existing formal ADLs have sufficient complexity to describe software systems and their structural constraints but lack an integrated method of enforcing these properties [4–7]. A few ADLs such as ArchJava can check and enforce architecture in implementation but do not address communication through the runtime system [1]. Techniques such as reflexion models support conformance checking between the static architecture and the implementation [9]. However, reflexion models require the programmer to map elements of a high-level model to the source code. This process can be tedious and error-prone, especially for large systems.

4 CONCLUSION

Consistency between software semantics and architecture is important in software development—not only for protection against hazardous dependencies and structural vulnerabilities, but also for traceability in large systems. We can protect against the introduction of component interactions that violate the architectural specification by constructing a system that derives its connector implementation from the architecture in a capability-safe language. Moreover, our use of multiple architectural views allows high-level changes to be made to software in a simple, concise way. Our implementation of a multi-view ADL incorporating the capability safety of Wyvern and its application on a client-server network system demonstrates its potential for the development of larger, complex software systems that are architecturally sound.

ACKNOWLEDGMENTS

This project was supervised by Jonathan Aldrich and Selva Samuel at Carnegie Mellon University. It was supported by a National Science Foundation REU grant.

REFERENCES

[1] Jonathan Aldrich, Craig Chambers, and David Notkin. 2009. ArchJava: Connecting Software Architecture to Implementation. *International Conference of Software Engineering* (2009).

[2] Paul Clements, Felix Bachmann, Lenn Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. 2011. *Documenting Software Architectures: Views and Beyond* (2nd ed.). Addison-Wesley Professional.

[3] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. 2002. A Practical Method for Documenting Software Architectures. (2002).

[4] Philippe Cuenot, Patrick Frey, Rolf Johansson, Henrik Lonn, Yiannis Papadopoulos, Mark-Oliver Reiser, Anders Sandberg, David Servat, Ramin Tavakoli Kolagari, Martin Torngren, and Matthis Weber. 2007. *The EAST-ADL Architecture Description Language for Automotive Embedded Software*.

[5] Peter H. Feiler and Bruce A. Lewis. 2006. The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems. *2006 IEEE Conference on Computer Aided Control System Design* (2006).

[6] Marc M. Lankhorst, Henderik Alex Proper, and Henk Jonkers. 2010. The Anatomy of the ArchMate Language. *International Journal of Information System Modeling and Design* (2010).

[7] David C. Luckham. 1996. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. (1996).

[8] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A Capability-Based Module System for Authority Control. *31st European Conference on Object-Oriented Programming (ECOOP 2017)* (2017).

[9] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. 2001. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Transactions on Software Engineering* 27, 4 (2001).

[10] Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. 2013. Wyvern: A Simple, Typed, and Pure Object-Oriented Language. *Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance* (2013).

[11] Lakshitha de Silva and Dharini Balasubramaniam. 2011. Controlling software architecture erosion: A survey. *The Journal of Systems and Software* (2011).

[12] Esther Wang and Jonathan Aldrich. 2016. Capability Safe Reflection for the Wyvern Language. *ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (2016).