Trinity: A Language for Multi-View Architecture Description and Control

1. Multiple Views of Software Architecture

The software architecture of a system is the set of structures needed to reason about the system, each consisting of elements of the system, relations among them, and properties of both [3]. Software architecture enables the system designers to analyze the system's ability to meet quality objectives like performance, reliability, etc. Thus, it facilitates the early detection of design errors and leads to improved software quality. It also provides a blueprint for system implementation and evolution.

To facilitate the different uses of software architecture in system development activities, a number of architecture description languages (ADLs) have been developed [1]. ADLs are formal languages that can be used to represent the architecture of a software system in an unambiguous way. Thus, they provide a rigorous basis for the analysis of system designs. ADLs also enable the various stakeholders to understand the system better and thus aid in system implementation, maintenance and evolution.

System designers often organize the architecture description of a software system as a set of views. An architectural view is a representation of those elements of the system that are needed to show how the architecture addresses a concern held by one or more stakeholders. For example, to devise an implementation plan, the system designers might create a (module) view that shows the module decomposition of the system and the dependencies among the modules (i.e., implementation units). Moreover, this view would not include the system's runtime entities or elements from the system's deployment environment. The separation of concerns achieved by splitting the architectural description into different views helps keep the architecture description cognitively tractable.

The vast majority of existing ADLs, however, lack support for multiple views. This hampers the use of ADLs for the specification of software architecture in practical settings. A survey conducted by Malavolta et al. [5] found that better support for multiple views is one of the features most desired by industrial practitioners in ADLs.

2. Architectural Control

Software architects design systems to meet quality attribute requirements like performance, reliability, security, etc. To guarantee that the implemented system exhibits the desired quality attributes, it is necessary to ensure that the implementation adheres to the design principles and constraints prescribed by the architecture. Architectural control is the ability of software architects to enforce architectural constraints on the implementation so that the system meets its design goals.

Luckham and Vera have identified *communication integrity* as a key aspect of maintaining consistency between the architecture and implementation of a software system [4]. Communication integrity is the property that each component in the implementation may communicate directly only with the components to which it is connected in the architecture.

In prior work, ArchJava used a custom type system to verify communication integrity statically [2]. A limitation of ArchJava is its inability to ensure that an application communicates over the network using only the connections shown in the architecture. Since programs in ArchJava have unrestricted access to system libraries, a component can use the network library directly to communicate in ways not specified in the architecture.

3. Trinity

We are developing Trinity, an ADL that would allow software architects to describe the software architecture of a system using multiple views and to guarantee communication integrity even for applications that involve communication across processes, possibly over a network.

Multi-View Architecture Description in Trinity. Clements et al. [3] identify three basic types of views: a) module view, b) component-and-connector view, and c) allocation view. Trinity provides support for these three views.

Module view. The module view is used to document the principal implementation units of a system, together with the relations among these units.

We have designed Trinity to describe the architecture of systems implemented in the Wyvern programming language. Since Wyvern provides first-class modules [6] which can be instantiated, passed as arguments and returned from a function, we use module definitions in Wyvern to implicitly describe the module view by specifying the Wyvern modules

```
resource type CSIface
  def getVal(key: String): String
module def Client(cPlumbing: CSIface): ClientIface
  def startClient(): Unit
    ...
module def Server(sPlumb: SPlumbing): ServerIface
  module def sendInfo(): CSIface
    def getVal(key: String): String
      ...
  def startServer(): Unit
    sPlumb.setFn(sendInfo().getVal)
    ...
```

Listing 1. Module view of a client/server system

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

Listing 2. C&C view of the client/server system

that would be used for implementing the system and the dependencies among them.

For example, Listing 1 shows the module view of a client-server system. It consists of a `Client` module and a `Server` module. The `Client` module depends on a module instance of type `CSIface` and the `Server` module depends on a module instance of type `SPlumbing`.

Component-and-connector view. The component-and-connector (C&C) view expresses runtime behavior. Listing 2 shows the C&C view of our example client-server system. It consists of two *components* (i.e. processes): a `client` component of component type `Client` and a `server` component of component type `Client`.

Components have *ports* which are interfaces through which a component interacts with the components it is connected to. A `provides` port consists of methods that are implemented by the component, while a `requires` port consists of methods that must be implemented by a component connected to this port.

For components to interact with each other, they must be connected together with a *connector*. In Listing 2, the `client` and `server` components interact via the `jsonCtr` connector. The `jsonCtr` connector enables the `client` and `server` components to communicate by passing JSON data between each other.

Allocation view. Allocation view describes the mapping of software units to elements of an environment in which the software executes. In particular, software elements from the

```
deployment CSDeployment extends ClientServer
  jsonCtr.host = 192.168.1.1
  jsonCtr.prt = 9090
```

Listing 3. Allocation view of the client/server system

```
1 module def cPlumbing(tcpClient: TCPClient): CSIface
2   def getVal(key: String): String
3     tcpClient.connect(IPAddress("192.168.1.1"),9090)
4     ...
5
6 val tcpClient = TCPClient()
7 val plumbing = cPlumbing(tcpClient)
8 val client = Client(plumbing)
9 c.startClient()
```

Listing 4. Plumbing code generated for the client

C&C view are allocated to the hardware of the computing platform on which the software executes. For example, Listing 3 shows the allocation view for our client-server system. It specifies the IP address and port of the `server` component connected by the `jsonCtr` connector.

Architectural Control in Trinity. Communication integrity is guaranteed in Trinity by means of the non-transitive authority property of Wyvern's capability-based module system [6]. For example, Listing 4 shows the plumbing code generated for the `client` component from the architecture description to facilitate its communication with the `server` component using the `jsonCtr` connector. Note in lines 6 to 8 that the `Client` module does not have a direct reference to the `TCPClient` module. Since authority is non-transitive in Wyvern, therefore the `Client` module would not be able to call the `connect` method of the `TCPClient` module. Thus, it would not be able to make connections to arbitrary hosts over the network.

References

[1] Architectural Languages Today. URL `http://www.di.univaq.it/malavolta/al/`. Accessed August 2, 2017.

[2] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 187–197, 2002.

[3] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Second edition, October 2010.

[4] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[5] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, 39(6):869–891, June 2013.

[6] D. Melicher, Y. Shi, A. Potanin, and J. Aldrich. A Capability-Based Module System for Authority Control. In *Proceedings of the 31st European Conference on Object-Oriented Programming (ECOOP)*, 2017.