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 [2]. 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. [6] 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 [5]. 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 [3]. 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 ...

Multi-View Architecture Description in Trinity. Clements et al. [4] identify three basic types of views: ...

Trinity provides support for these three views.

Module view. Module view is used to document the principal implementation units of a system, together with the relations among these units. The primary elements in module view are implementation units that provide a coherent set of responsibilities. In Trinity, implementation units take the form of capability-based modules.

Modules are related to one another using the depends on relation.

What is a module view useful for? The module view can be used to determine how a system's source code is decomposed into units, what assumptions each unit can make about services provided by other units, and how these units are ag-

```
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

  entryPoints
    client: startClient
    server: startServer
```

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

gregated into larger ensembles. The module view can thus be used to create an implementation plan and also to determine how changes to one part of a system might affect other parts and thus reason about the system's modifiability, portability and reuse.

Component-and-connector view. Component-and-connector view expresses runtime behavior. It is described in terms of components and connectors. A component is one of the principal processing units of the executing system. Components might be services, processes, threads, filters, repositories, peers, or clients and servers, to name a few. A connector is the interaction mechanism among components. Connectors include pipes, queues, request/reply protocols, direct invocation, event-driven invocation, etc. Components and connectors can be decomposed into other components and connectors. The decomposition of a component may include connectors and vice versa.

```
deployment CSDeployment extends ClientServer
  jsonCtr.host = 127.0.0.1
  jsonCtr.port = 9090
```

Listing 3. Allocation view of the client/server system

```
module def cPlumbing(tcpClient: TCPClient): CSIface
  def getVal(key: String): String
    tcpClient.connect(IPAddress("127.0.0.1"), 9090)
    ...

val tcpClient = TCPClient()
val plumbing = cPlumbing(tcpClient)
val c = client(plumbing)
c.start()
```

Listing 4. Plumbing code generated for the client

Allocation view. Allocation view describes the mapping of software units to elements of an environment in which the software is developed or executes. The environment might be the hardware, the file systems supporting development or deployment, or the development organization(s).

Listing 2 [8] [7]

Architectural Control in Trinity.

References

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

[2] Modern Software Architecture Definitions. URL `http://www.sei.cmu.edu/architecture/start/glossary/moderndefs.cfm`. Accessed August 2, 2017.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] L. Nistor, D. Kurilova, S. Balzer, B. Chung, A. Potanin, and J. Aldrich. Wyvern: A Simple, Typed, and Pure Object-Oriented Language. In *Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance (MASPEGHI)*, pages 9–16, 2013.