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

Subtitle Text, if any *

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

This is the text of the abstract.

1. Introduction

Understanding how a software system works is important in designing, implementing, and using it. Software architecture can be defined as the fundamental organization of a system embodied in its components, their relations to each other, and the environment. Any system, including a living organism can be understood from different perspectives and views; a human being can be analyzed in terms of his or her comprising systems as in the cardiovascular system or nervous system, or as a part of a larger structure, like a friendship, society, or species.

A single software system can similarly be interpreted as a composite whole or a part of a larger system. The software architecture of a system, or its fundamental organization embodied in its components, their relationship to each other, and to the environment, and the principles guiding its design and evolution, is comprised of three basic levels. Software is comprised of three general architecture views: Software can be understood as sets of implementation units and sets of runtime components that have behavior and interact with one another, as well as in terms of its interactions with non-software elements, like hardware. These views are concisely called the Module, Component-and-Connector, and Allocation views respectively. Understanding a software system in terms of code-implementation (how it works), runtime entities (how it interacts with itself), and non-software interactions (how it interacts with non-software parts of a system) is crucial for its initial design and implementation, later adaptation, and use.

While architecture is essential to developing, adapting, and understanding a software system, current software architecture practices provide few and weaker-than-desirable guarantees that architecture views reflect what happens when a system is in use. When a developer initially designs her software system, she might draw up a graph representing various components and their relationships. Whether the components are set of programmatic code pieces, runtime elements, or system parts depends on what stage of development and from what architecture view she is approaching the design. While the logical relationships between these theoretical elements are sound, their realization may present some issues.

For example, when designing a ¡SYSTEM¿, one might draw up a graph representing the intended Component-and-Connector (CnC) view, including the components, or the ¡LIST OF COMPONENTS¿ and the connectors, or the ¡LIST OF CONNECTORS¿. However a conflict between the ¡CONNECTOR TYPE¿ connector and the ¡COMPONENT¿ forces the developer to change the architecture during implementation. Depending on the effects of this change and others like it, the realization may not uphold the guarantees of the intended architecture. In the aforementioned example, the planned ¡CONNECTOR TYPE¿ connector ensured that the ¡COMPONENT¿ could only communicate with the ¡OTHER COMPONENT¿ in a specific way. While implementation adjustments were required for completion, the assumptions from the original architecture about how COMPONENT1 is able to communicate with other runtime components may no longer hold. Here, communication integrity is no longer guaranteed.

Software system designs must adapt as the problem or understanding of it changes. This type of adaptability is essential for building software, but it also makes verifying intended software guarantees, such as communication integrity, difficult. Understanding how a software system works in practice, as opposed to theoretically, is necessary when developing, adapting, and using it.

* with optional subtitle note

2. Early Design

As a solution to the issue of carrying intended architecture guarantees through to the final implementation, we present Trinity, an extension to the Wyvern programming language that aims to make software architecture views a presecriptive, rather than simply descriptive, aspect of a software system in Wyvern. Trinity incorporates architecture in a unique way, making it a live component: software systems can have hard-coded architectures for all three architecture views. Instead of providing a descriptive architecture that attempts to explain how a system functions, Trinity supports architecture as a language construct in Wyvern. In the case of the CnC architectural view, what would conventionally be only logical organizations and distinctions between components and connectors, can now be actually implemented components and connectors. Since Trinity supports prescriptive software architecture views it allows developers to catch architecture-related errors at compile time, rather than runtime. This novel ability enables safer inspection for vulnerabilities originating in architecture flaws.

Trinity is designed to safeguard communication integrity, which is a top priority in regards to desired guarantees to preserve in end-product implementations. To enforce communication integrity is to ensure that each component in [an] implementation may [communicate] directly with components to which it is connected in the architecture. In our early implementation of Trinity, we have provided support for a hard-coded Component-and-Connector view of a Wyvern software system. Our design translates concepts from software architecture into implementation capabilities.

Entities in the Cnc architecture view are keywords/types/objects in Trinity. An entire software architecture, comprised of components and connectors, is contained within an architecture object/type/keyword. Trinity components are runtime components such as servers, clients, and databases. Ports are fields of a component that act as access points to interact with other components; these include interfaces that provide details about the interface another component must fulfill to communicate with said component through said port. Connectors, such as JDBC connectors, connect multiple ports of components. Attachments allow the programmer to actually connect components using ports of components and provided connectors. An entryPoint is an access point for a program using the specified architecture. It might be a function of a particular component in the architecture. For example, an architecture describing a three-tier web application might have an entryPoint specified by the start() function of a client component. Bindings allow the programmer to alias functions of components, to reduce code verbosity.

We have implemented a simple three-tier web application example of Trinitys encoded architecture, including server and client components. The server is connected to a database, which is external component or a component only accessible by [INSERT HERE]. The client can only interact with the database through its own getInfo port, which is a client method, that corresponds to the servers sendInfo method port. The servers JSON connector gives it access to the database, which provides the clients indirect channel of communication with the database.

While the programmer provides client and server interface code, Trinity generates code for the servers connector, whose type is specified by the programmer. In our example, JSON connector code is generated. Referring to the architectures attachments the Trinity parser type checks the connect expression and the types of the requires and provides ports of the client and server respectively are matched. Then the metadata associated with the JSON connector specifies the code that must be emitted for coordinating between the requires and provides ports. The code for the clients getInfo port, the servers sendInfo port, and serializing and deserializing to and from JSON is generated. By allotting communication-integrity sensitive code to Trinity automatic code generation, we ensure that communication integrity between the client and database is enforced. Further, code generation makes producing correct code easier for the programmer, as well as makes port matching and type checking errors apparent before run time.

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

P. Q. Smith, and X. Y. Jones. ...reference text...