Title Text [*]

Subtitle Text, if any [†]

Name1 [‡]
Affiliation1
Email1

Name2 Name3 [§]
Affiliation2/3
Email2/3

Abstract

This is the text of the abstract.

CCS Concepts • **Software and its engineering** → **General programming languages**; • **Theory of computation** → *Program analysis*

Keywords keyword1, keyword2

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

* with optional title note
† with optional subtitle note
‡ with optional author note
§ with optional author note

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.

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.

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...