

# Position Summary: Towards Zero-Code Service Composition

Emre Kıcıman, Laurence Melloul, Armando Fox  
{emrek, melloul, fox}@cs.stanford.edu  
Stanford University

**Zero-Code Composition** For many years, people have been trying to develop systems from modular, reusable components[2]. The ideal is *zero-code composition*: building applications out of components without writing any new code. By investigating zero-code composition, our goal is to make composition easy enough to be of practical use to systems researchers and developers. We are focusing on identifying and removing systemic impediments to composition, and on exploiting composition to achieve system-wide properties, such as performance, scalability, and reliability.

**Impediments to Composition** Today, even when components are designed to be reused, software developers have difficulties composing them into larger systems. We believe the problem lies with the methods and fundamental abstractions used to package and compose components. For example, abstractions such as function calls work well when building small systems, however, they actually enforce properties on components that significantly impede composition and reuse generally. These impediments can be classified into two categories:

- Control flow impediments relate to the ordering of execution of components [1]. For example, two components cannot be used together when they make different assumptions about the sequencing of computation and passing of control between them.
- Interface impediments occur when components contain statically bound information about other components' interfaces, such as method names, data types and orderings, and communication protocols. However, this information will be invalid in different contexts, and will prevent the component from being reused in an arbitrary composition.

**A Data Flow Composition Model** To avoid these impediments, we advocate that compositions be built of autonomous *services* connected together in a data flow network. Autonomous services avoid control model mismatches by keeping their own locus of control. Interface

impediments are avoided by allowing services to only name their own input and output ports. The data flow model is defined by the data dependencies between services, and provide an explicit description of the composition. A generic run-time system handles passing data from one component's output port to another's input port according to the data flow description of the composition.

Explicitly exposing the structure of applications enables systematic inspection, manipulation, and augmentation of applications. We can inspect the data flow composition for bottlenecks in performance, and strategically move, replicate or replace parts of a composition which are performing poorly. For example, one simplistic strategy is to dynamically place caches around strings of expensive services in a composition to improve performance. We can similarly manipulate a composition to increase its fault-tolerance, scalability and reliability.

**Current Status** We have implemented a prototype composition architecture [3], and are beginning to implement dynamic manipulations of compositions, and explore the relationships between these manipulations, system-wide properties and various service attributes such as determinism or idempotency.

## References

- [1] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings of International Conference on Software Engineering '95*, Seattle, April 1995.
- [2] P. W. Gio Wiederhold and S. Ceri. Towards megaprogramming: A paradigm for component-based programming. *Communications of the ACM*, (11):89–99, 1992.
- [3] E. Kıcıman and A. Fox. Using dynamic mediation to integrate cots entities in a ubiquitous computing environment. In *Handheld and Ubiquitous Computing (HUC 2000)*, *Second International Symposium*, Sept. 2000.