

# Designing What You Can't

**Steven P. Reiss**  
**Brown University**  
**Providence, RI 02912**  
**401-863-7641, spr@cs.brown.edu**

The traditional notion of a software system is disappearing. Along with it, the window in which we might have been able to design complex software systems is vanishing, even before we understand the techniques and tools that should be used in such designs.

Software is becoming more and more integrated. The use of web services, software components, grid computing, peer-to-peer networks, global file systems, cooperative tools, shared source, web access, etc. all point to a world in which all applications are interrelated. Applications will actively share files, share data, share components, share user interfaces, and even share computations. Every software system will actually be a part of every other software system. We will not be able to look at, understand, or even design a software system by itself. Instead, we will have to look at all software systems at once.

We are moving to a world where software systems will be built mainly from components designed, developed, maintained, and modified by different people who are not under the control of the software developers or designers. The components themselves will evolve outside of the control of the software system. Such a world has analogs. The evolution of the Internet itself was not designed or planned in detail, but is loosely controlled by a set of rules that purport to give it stability. Our electrical distribution system has developed from a large number of independent networks to a global, interacting network that is dynamically changing.

One cannot expect to design such a system in the traditional sense. No one person or group of persons has control over the whole system. The system is going to have to deal with unforeseen evolution and changes of its various components. Moreover, as we can see from the analogous systems that have evolved this way, unconnected local effects can have large global consequences. What is really needed then, looking toward the future, is not a science of design that addresses today's large software systems, but rather a way of managing the complexity of software where there is only one program and it is everything. Here the primary concern should be ensuring that the software or component can achieve its goals without breaking anything and do it even if other things change.

We should be thinking now about the science that will let us support large numbers of interacting and evolving components in what is effectively a single software system. We have to assume that we can't control or even trust many of these components. Components are going to change, become available, and become unavailable at arbitrary times. Yet we should and will be able to get real work done in a reliable and effective manner. The world is moving rapidly in this direction, and we have to be prepared.

While our particular research hasn't developed such a science yet, we have begun the development of a framework to support such a software environment and thus provide the basis upon which a science can be built. Our approach involves the definition of an interface to a software component that can have multiple, independent instantiations. The interface pro-

vides the syntax for calling the component, the semantics of the component, security and privacy properties that the component must provide, an auction-based cost model for choosing implementations, and support for handling failure, recovery and general unreliability. Implementations for components can be provided by arbitrary users, are checked before use, and are automatically and dynamically chosen by the underlying system. Implementations can run locally as libraries or shared code, as web services at one or more URLs, as services at a fixed address, or, using grid technology, run on an appropriate idle machine. The framework provides global data stores both with a Internet-wide file system and with shared tuple spaces. The framework also includes a common user interface model for use by arbitrary distributed services.

We are currently starting experiments within the context of this framework that are aimed at developing the appropriate tools and techniques for understanding how such systems work under various circumstances and attempting to determine how to achieve the levels of stability and reliability that will be needed in the future. Whether you want to use our framework or some other framework that addresses the future, these future problems are the ones we should be looking at, not yesterdays.

**Biography.** Dr. Reiss is Professor of Computer Science at Brown University. He has been a member of the Brown faculty since 1977. Dr. Reiss's research interests and expertise lie in the area of software engineering, programming environments and software visualization. He has developed several programming environment and introduced new programming concepts such as message-based (control) integration. He has also developed several innovative software visualization tools and integrated these with programming environments. He is the author of a monograph on the FIELD system and a text book on software design. His current research involves using constraints to control the evolution of software systems, static analysis tools for checking dynamic software properties, efficient dynamic visualization of software, and the TAIGA pervasive programming environment described above.