Component Based Software Engineering

The complexity of software systems sometimes lingers on the border of human comprehension and often hinders their timely development, effective deployment, proper maintenance, and incremental evolution. On the other hand, because complex software systems involve heavy investments of human and financial resources, they are likely to be critical to the missions of the institutions where they are deployed for very long periods of time. The longevity and significance of such software as mission-critical systems makes their change through maintenance and evolution their inescapable fate, and this intensifies the problem of coping with their complexity throughout their long lives.

A sensible approach to reducing this complexity to manageable levels is the application of a technique that has already been successfully deployed in other, older production environments: building systems out of components. Component-Based Software Engineering (CBSE) advocates this sensible approach. Unlike other programming abstractions, such as objects, software components are expected to accommodate certain non-functional requirements, such as commercial viability, and must be built 'defensively' with certain minimum assumptions about their deployment environments. They are expected to be deployed and used only in binary form in a language-independent (and sometimes even platform-independent) manner.

Component-based software architectures rationally postulate two types of code: the code that comprises individual software components, i.e., component code; and the code that composes a set of individual components into a coherent system, i.e., the glue code. Variants of the so-called scripting languages have been proposed and used with some success as glue-code languages. Regardless of what (programming, scripting, etc.) language is used to write the glue code, the success of the component-based methodology requires the semantics of the glue code to be independent of the internal semantics of the components it composes into a system. This semantic independence requires an interface layer where the externally observable semantics of a component can be precisely described independently of its irrelevant internal semantics.

The existing technologies that offer component-based software engineering, are generally restricted to syntactic interfaces for components (e.g., CORBA and COM) and furthermore, some (e.g., Jini) are language specific. The goal of our ongoing experimental work is to forge out a practically useful language called Ρεω (rheo means stream). Ρεω is a language for composition of mobile components using channels as their primitive connectors. Simpler connectors can be combined using the compositional operators in Ρεω to build more complex ones. The semantics of connectors in Ρεω are independent of the semantics of the components they connect. This enhances modularity of software systems, allows separate, independent verification of connectors and components, and facilitates reuse of connectors as well as components.

The work done on CBSE at CWI builds upon and extends our extensive earlier work on Coordination models and languages. It emphasizes the semantic independence of the components and the glue code, and focuses on compositional models and languages for developing the glue code. We have already developed a formal model for component-based systems, together with a formal-logic-based component interface description language that conveys the observable semantics of components. We have presented a formal system for deriving the semantics of a composite system out of the semantics of its constituent components, and the conditions under which this this derivation system is sound and complete.