PREDICTIABILITY BY CONSTRUCTION

Sagar Chaki, Scott Hissam, Gabriel Moreno, Linda Northrop, Kurt Wallnau

December 2016

Overview

Predictability by construction (PBC) makes the behavior of a component-based system predictable prior to implementation, based on known properties of components. The PBC vision is for software components to have certified properties (for example, performance) and for the behavior of systems assembled from components to be predictable.

Benefits

PBC enables you to

- establish design and implementation standards that lead to software systems with predictable runtime quality
- use automation to enforce these standards, leading to systems that are predictable by construction
- define objective standards and measures for trusted components, developed internally or by third parties
- incrementally and systematically introduce state-of-the-art prediction for new or more general classes of systems and properties
- provide a sound and objective basis to manage design risk and optimize design features

Getting Started

Across the software industry, projects face the challenge of satisfying ubiquitous demands for increased functionality, better quality, and reduced cycle time (time to market or field) while simultaneously ensuring that delivered software and software systems satisfy security, survivability, availability, and interoperability requirements.

While the use of component-based development promises to address the first set of demands, current processes and technologies fail to help developers predict the qualities of a system of components, resulting in expensive integration and testing efforts. Exploiting the full potential of component-based development to meet these demands requires the software industry to develop improved, enhanced, or new processes, methods, and tools for determining the properties of software systems before they are built and for confirming their "as-built" properties.

The software industry has developed numerous technologies such as .NET, Enterprise JavaBeans (EJB), and the Common Object Request Broker Architecture (CORBA) to assemble systems from components that are created in isolation. Significant economic and technical benefits from component approaches have accrued. However, component approaches miss the mark in being able to substantively address some of the real issues, namely, the inability to predict individual component behavior, the behavior of assembled components, and hence, the quality of the system. Component technologies available today allow system builders to plug components together, but do little to allow the builder to ensure how well they will play together. As a result, there are rampant failures with component assemblies that receive inadequate testing and expensive integration and test cycles on those that are made to succeed. Consequently, it is not surprising that there is a lack of consumer trust in the quality of software components and in the quality of assemblies that have not received extensive and expensive testing.

In short, though the component approach has helped and does hold promise, it does not now address the real challenges; software components are critical to the software industry, but the behavior of component assemblies is unpredictable. The impetus for today's component technology was inarguably the exponential growth of information technology (IT) industry needs. However, software component technology has not yet demonstrated an ability to predictably meet the requirements for scale, robustness, and performance that information-technology-bound organizations have. The result is unpredictable and costly development, decreased assurance of how the delivered system will behave, and ultimately, slowed adoption of component technology.

The fundamental technical reasons behind these inadequacies are

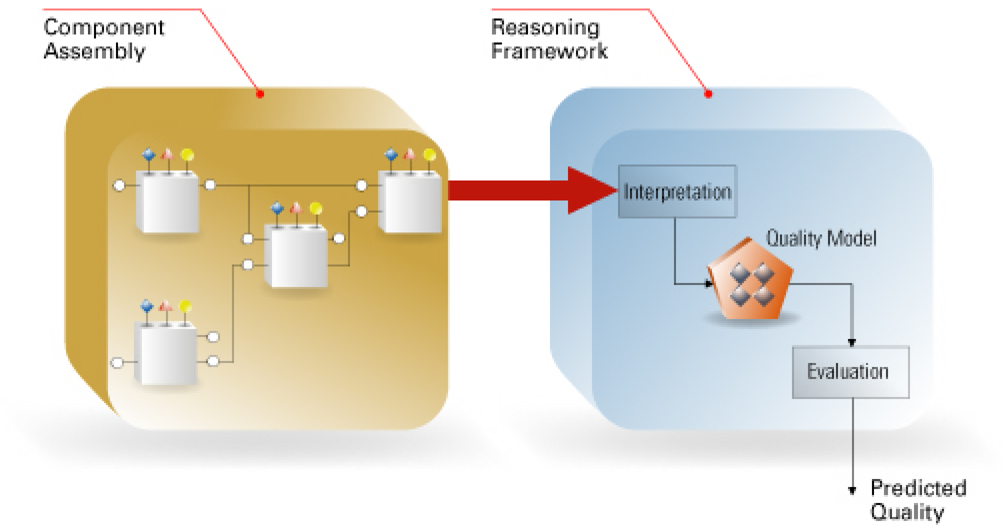
- Component interfaces are not sufficiently descriptive.
- The behavior of components is, in part, an *a priori* unknown.
- The behavior of component assemblies consequently must be discovered.

To compound these nontrivial technical reasons, there is, in general, a lack of consistency in how software industry speaks about components and consequently much misunderstanding as to what a component really is, what current component technologies provide, and where COTS components fit into the component discussion. Some equate components exclusively with COTS, while others equate component technologies with a specific vendor's offering, for example, EJBs. Of course, such inconsistencies and misunderstandings lead people to make poor decisions.

In order to effect the promised potential of component technology and to make component technology effective for the software industry, there must be focused technical leadership to crystallize component concepts and channel researchers to address their current inadequacies, to find ways to predict the behavior of an assembly of components before their development, purchase, installation, and integration and, finally, to improve the level of trust that can be associated with software components.

PBC Concepts

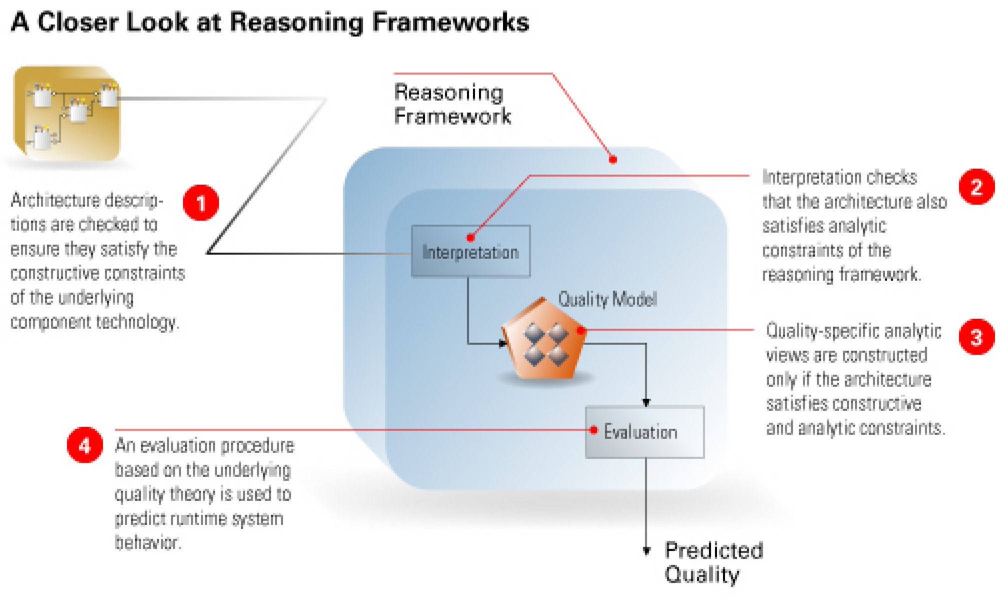
PBC builds on software architecture technology, software component technology, and a growing body of theory for predicting the quality attributes of software systems (for example performance, security, safety). Architectural design constraints that satisfy the assumptions of quality attribute theories ("smart constraints") are enforced at construction time and run time by software component technology. Analysis is automated by automatic generation of predictive models from assembly specifications. The complexity of this interpretation, and of the underlying analytic theories, is packaged in a reusable form called a reasoning framework. The resulting predictions have an established and verifiable statistical or formal basis for objective confidence.



Predictability By Construction Concepts

Reasoning Frameworks

A reasoning framework is a way to package, as a fully-automated tool, the expertise needed to understand and predict the runtime qualities of software systems. A key objective of a reasoning framework is to make this expertise available to engineers who are not expert in the quality attribute. As such, the reasoning framework includes all that is necessary to generate and analyze quality-specific views of an architectural specification. The reasoning framework also exposes the assumptions underlying an analytic theory, and ensures that systems satisfy these assumptions so that the user has confidence that all predictions are not only sound, but valid.


A Closer Look at Reasoning Frameworks
Reasoning Framework
1
Architecture descriptions are checked to ensure they satisfy the constructive constraints of the underlying component technology.
Interpretation
2
Interpretation checks that the architecture also satisfies analytic constraints of the reasoning framework.
Quality Model
3
Quality-specific analytic views are constructed only if the architecture satisfies constructive and analytic constraints.
4
An evaluation procedure based on the underlying quality theory is used to predict runtime system behavior.
Evaluation
Predicted Quality

Reasoning Frameworks

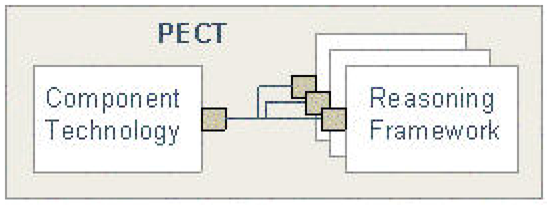
Prediction-Enabled Component Technology

Our approach to achieving the PBC objectives is to use prediction-enabled component technology (PECT). As its name suggests, a PECT is an enhanced component technology. What is a component technology? There is no answer to this question that won't provoke an argument, any more than there is a universally agreed-upon answer to the question "what is a component?" Nonetheless, there is growing agreement on the following rough definitions:

- A software component is an implementation, ready to execute on some (possibly virtual) machine, with well-defined interfaces that enable third-party composition (roughly, integration with other components).
- A component technology is a component model and runtime environment where
  - the component model specifies what interfaces a component must provide, and how components are allowed to interact with one another and their runtime environment
  - the runtime environment is a container in which component behavior executes and in which components interact. The runtime environment may also provide useful services—persistence, transactions, etc.

There is a similarity between this definition of component model and the usual definition of architectural style (or pattern) as a collection of *component types* and their allowable *patterns of interaction*. Even though they may differ in many respects, a component model and architectural style both specify invariants that must be satisfied by any instance of that model/style. These invariants are exactly those “well-formedness” rules that we impose on component assemblies to ensure that they can be analyzed, and therefore to ensure their predictability.

Seen in this light, a component technology can be thought of as an infrastructure for designing, developing, and deploying applications that adhere to a particular architectural style. The infrastructure does restrict the freedom of developers and designers, but in compensation it enforces design and implementation invariants that, in this case, ensure predictability. The tradeoff between restricted freedom and predictability has been seen before—in the development of strongly typed programming languages, now considered an essential element of modern software engineering practice. The long-awaited shift to a higher level of abstraction—from functions and classes to components—is underway.



PECT as a Component Technology with Validated Reasoning Frameworks and their Interpretations

In this figure, abstract component technology specifies the invariants imposed on by a specific component and a collection of reasoning frameworks (each which may impose their own additional invariants). A PECT can and generally will support several analysis models, each of which is “packaged” in its own automated reasoning framework. An interpretation defines an automated translation from assemblies of components specified in the construction language to the reasoning framework.

Glossary

abstract component technology
a vocabulary and notation for specifying components, assemblies, and their runtime environments in a component-technology-independent way, and for specifying the constraints, imposed by reasoning frameworks, that must be satisfied for predictions to be valid

analytic constraints
constraints imposed by one or more reasoning frameworks on an abstract component technology

annotation
a property P associated with a referent R, meaning that "R has property P," denoted as R.P

assembly
a set of components and their enabled interactions

assembly constraints
behavioral and topological rules of well-formedness imposed on components and assemblies by one or more (real) component technologies, and one or more reasoning frameworks

automated reasoning procedure
a decision procedure and interpretation, each susceptible to full automation. See also *property theory*

binding label
a linking mechanism embedded in components to enable their interaction with other components. See also *pin*.

component
an implementation in final form, modulo bound labels, that provides an interface for third-party composition and is a unit of independent deployment

component technology
a component technology imposes fabrication standards for assembling software from large-scale building blocks. A component technology consists of a component model and a runtime environment. The component model specifies fabrication standards governing such issues as a component's life cycle and allowable forms of interaction. A runtime environment is an execution environment that enforces aspects of the component model and provides standard interaction mechanisms and services.

compose
to enable component interaction through connectors

composition
a set of interactions among components enabled through connectors. See also *assembly*.

Communicating Sequential Processes (CSP)
a specification language and formal notation for describing concurrency behavior in systems.

connector
a mechanism provided by the runtime environment that enforces an interaction protocol, or discipline, on the components that are participants in an interaction

construction framework
an abstract component technology, tools to enforce assembly constraints, and other tools used to automate the specification, development, and deployment of components and their assemblies

construction language
a language for specifying abstract component technologies (ACTs) and their well-formed components and assemblies

counter example
a counterexample is an execution trace that results in the violation of a behavioral assertion Counterexamples provide diagnostic feedback that help engineers understand and reproduce failures

contain
to restrict the visibility of interactions on pins

co-refinement
a process for developing reasoning frameworks, and in particular, for finding an acceptable tradeoff among various qualities of a reasoning framework, such as generality, complexity, and stability

decision procedure
a function that evaluates claims made on assemblies, described in the property theory, to the values “true” or “false”

deploy
defines where (in which instance of a runtime environment, and, ultimately, on which physical computing device) component behavior is executed

empirical evidence
evidence acquired through direct observation, preferably under controlled circumstances, with results reported in well-defined units of measure. Empirical evidence is therefore provisional, as any other observation might have been different. See also *formal evidence*.

final form
a software specification that is ready for execution on a physical or virtual machine. See also *component*.

formal evidence
evidence acquired through mathematical proof. Formal evidence is therefore irrefutable, as all such proofs are tautological. See also *empirical evidence*.

interaction
a composition of two or more reactions, from distinct components, using a runtime-environment-provided connector

interpretation
a mapping from assemblies specified in a construction language to specifications in the language of a reasoning framework

in the zone
within trusted and predictable parameters. Components are "in the zone" are predictable before they are built, and component assemblies are "in the zone" if their runtime behavior is analytically predictable.

partial assembly
a (recursively defined) abstraction that aggregates a set of components and their enabled interactions and exposes selected component pins. Logically, a partial assembly is a component implemented entirely in terms of other components. See also assembly.

prediction-enabled component technology (PECT)
a component technology that has been extended with one or more predication-enabling technologies

property
an n-tuple <*name, value, ...* >, where name and value refer to the name of some property and the value it takes, respectively. See also *annotation*.

property theory
a calculus and logic that provides an objective, rigorous, and verifiable or falsifiable basis for predicting the properties of assemblies

reaction
specification of the behavior of a unit of concurrency within a component (e.g., a thread) and the behavioral dependencies between sink pins and source pins of a component

reasoning framework
a combination of a property theory, an automated reasoning procedure, and a validation procedure that is used to predict assembly properties

pin
a binding label in the construction and composition language (CCL). See also *source pin*, *sink pin*, *connector*.

runtime environment
environment that provides runtime services that may be used by components in an assembly, provides an implementation for one or more connectors, and enforces assembly constraints

sink pin
a pin that **accepts** interactions with the environment of a component (i.e., from other components or the runtime environment). See also *pin*, *source pin*.

source pin
a pin that **initiates** interactions with the environment of a component (i.e., to other components or the runtime environment). See also *pin*, *sink pin*.

unit of independent deployment
a component is independently deployable if all its dependencies on external resources are clearly specified (e.g., as pins), and if it can be substituted for, or substituted by, some other component. See also *deployment*.

validation procedure
provides an objective basis for trusting the validity and soundness of a reasoning framework, and defines its required component properties with sufficient rigor to provide an objective basis for trust in assertions of component behavior

Bibliography

Bass, Len; Ivers, James; Klein, Mark; & Merson, Paulo. *Reasoning Frameworks.* CMU/SEI-2005-TR-007. Software Engineering Institute, Carnegie Mellon University. 2005. http://re-sources.sei.cmu.edu/library/asset-view.cfm?AssetID=7637

Predictability by Construction. . Software Engineering Institute, Carnegie Mellon University. 2009. http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=28450

Wallnau, Kurt. *Obtaining the Benefits of Predictable Assembly from Certifiable Components (PACC).* Software Engineering Institute, Carnegie Mellon University. 2005. http://csauth-techxfer.sei.cmu.edu/library/asset-view.cfm?assetID=29339

Wallnau, Kurt. *Volume III: A Technology for Predictable Assembly from Certifiable Components.* CMU/SEI-2003-TR-009 . Software Engineering Institute, Carnegie Mellon University. 2003. http://csauth-techxfer.sei.cmu.edu/library/asset-view.cfm?AssetID=6633

Contact Us

Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA 15213-2612

Phone: 412/268.5800 | 888.201.4479
Web: www.sei.cmu.edu | www.cert.org
Email: info@sei.cmu.edu