Obtaining the Benefits of Predictable Assembly from Certifiable Components (PACC)

Kurt C. Wallnau

PACC combines the complementary features of software architecture technology and software component technology to improve both engineering productivity and product quality in the design and implementation of quality-critical software systems. With PACC, your organization can develop software systems that are predictable by design and by construction and that are composed from objectively trustworthy software components. With PACC, you can put into the hands of practicing software engineers the power of state-of-the-art analysis tools for performance, safety, reliability (and soon, security). You can use these tools to optimize the quality of your systems and manage design risk as your systems push the envelope of what can be achieved in software.

Component models fill the gap between the quality attribute concerns of software architecture and the low-level concerns of programming.

With PACC you can establish ***design and implementation standards*** *that lead to software systems with predictable runtime behavior.*

Design decisions that influence the software architecture of systems are strongly correlated to the quality attributes of those software systems—performance, modifiability, reliability, and so forth. However, there has always been a gap, and indeed a conceptual mismatch, between the high-level abstractions of software architecture and the detailed abstractions of computer programming. Software component technology bridges this gap.

A software component is not just software with an interface: it is a unit of independent deployment and substitution, with behavior characterized in terms of stimulus/response *interfaces*. A component is deployed to a *container* that implements a well-defined coordination model for interacting with other components in systems. A component has a well-defined life cycle that is fully managed by a runtime environment—an environment that is, itself, a container for *assemblies* of components that can interact with other assembly containers.

Conventional programming languages, such as Java and C/C++, are used to implement software components. From the programmer's point of view, a component model is a higher level, abstract, organizing scheme for software development. From the architect's point of view, it is a lower level, concrete organizing scheme for translating architectural decisions—especially those motivated by quality attribute requirements—into design and implementation standards that are guaranteed to respect these decisions as invariants in deployed software.

Design standards can be mapped to component technology to achieve predictability by construction.

With PACC you can use automation to enforce these standards, leading to systems that are ***predictable by construction.***

There used to be debate about the merits of strongly typed programming languages, but no longer. Modern programming languages are equipped with type systems that impose certain *syntactic restrictions* on the way computer programs are written. In return for restrictions on freedom of expression, type checkers give programmers *guarantees* about the abstraction safety of well-typed programs. Errors are caught early, and guarantees as strong as formal proofs can be produced automatically—by construction. The debate over the virtue of type restriction is over; the emphasis now is on making type systems more expressive.

PACC is on the forefront of a movement that is kindling an analogous debate at the intermediate level of abstraction that lies between programming languages and software architecture—the level of software components. In addition to typical concerns that motivate classical type systems, a new set of concerns emerges—one that pertains to system quality attributes. While we are not yet in possession of a type theory that is suitably robust and expressive for these new concerns, we are making steady and practical progress by defining *reasoning frameworks* and their associated *smart constraints*.

A smart constraint is a restriction imposed on component implementations or assemblies of components. Each smart constraint corresponds to one or more analytic assumptions, made by a system designer or architect, that provide a basis for making a quality attribute claim about a system. In PACC, smart constraints are introduced by *reasoning frameworks*, which are packaged analytic theories for reasoning about a system's runtime quality attributes. The constraints imposed by a reasoning framework are "smart" because they capture both architecture and quality attribute expertise. Systems that satisfy the smart constraints of a reasoning framework have behavior that is predictable by that framework.

A performance reasoning framework might assume a particular scheduling discipline, and a reliability reasoning framework might assume a particular failover discipline. These assumptions must be satisfied and, hence, become the smart constraints of these frameworks. Using PACC technology, smart constraints can be satisfied (or discharged) by the component runtime environment or by prefabricated component or assembly containers. Alternatively, responsibility might be delegated to component developers or system assemblers. In this case, specification checking is required. PACC uses the Construction and Composition Language (CCL) for this purpose, although other notations are of course possible.

Predictive quality measures are an important step towards objectively trustworthy software components.

*With PACC you can define objective standards and measures for **trusted components**, developed internally or by third parties.*

Smart constraints sometimes correspond to something that we need to know about the component implementation. For example, a performance reasoning framework may need to know the service time distribution for a particular component operation, and a security reasoning framework may need an abstract state machine description of the patterns of interaction that a component exhibits on a specified stimulus. In PACC, we refer to what we need to know about a component to predict its behavior, alone or in an assembly, as the *analytic interface* of that component.

It is not enough for a reasoning framework to impose a requirement for information about component behavior; it must also specify how this information can be reliably obtained, and, if necessary, independently certified. The predictions of each PACC reasoning framework must be susceptible to rigorous formal and/or empirical validation; the same is true of the component properties on which the framework depends. This strong condition on reasoning frameworks is not always easy to satisfy, but the result is a rigorous approach to component specification that combines the best features of functional interface specification with the latest techniques for nonfunctional, or quality attribute, specification of components.

Predictability is a strategic asset; it is a capability that can be honed and used for competitive advantage.

With PACC you can incrementally and systematically introduce ***state-of-the-art prediction*** *for new or more general classes of systems and properties.*

Predictability is not a "yes/no" phenomenon; it is a capability that exists in degrees. An architect may have sufficient experience with a class of system to make performance predictions early in the design phase. An organization with a product line architecture may reasonably predict that the performance of successive products will vary by some amount. A first-of-a-kind system might have demanding performance requirements but not be heavily resource constrained, and designers might confidently predict that high-end processors and memory will enable the system to meet worst-case performance requirements.

In each of these cases, predictability is evident, more or less. However, each case is lacking in one or more qualities. One complex quality falls under the broad heading of "objective confidence." Under this quality rubric, we want to make statements *about* predictions, such as confidence or tolerance intervals, and how these intervals were constructed. Similarly, predictions about behavior based on fully formal means, such as model checking, also pose questions about objective confidence. How can we be certain that the model checking tool is implemented correctly or that the model it checked is consistent with the actual software? Although we do not have answers to all such questions, PACC is making good progress in generating statistical labels and proof certificates to provide an objective, quantifiable basis for using predictability as a tool in decision making.

Another quality lacking in the above cases is a clear and explicit understanding of the design assumptions that must be satisfied for a prediction to be valid or of the effect of relaxing or strengthening these assumptions. Indeed, it is our experience that there is a strong correlation between the strength of the smart constraints imposed on developers and the tightness of the required confidence intervals on predictions; tighter intervals impose stronger constraints. Freedom and predictability are often in tension, and there is no unique solution to the implied tradeoff between them. In any given system, tradeoffs must be made not only between the freedom and predictability of a quality attribute but also among quality attributes; for example, performance and security. Resolving these tradeoffs often involves a controlled relaxation of constraints that requires an understanding of how these decisions influence predicted system performance with respect to each quality attribute and of the confidence we can have in these predictions.

PACC attacks a root cause of risk—uncertainty—and provides an objective and systematic basis for satisfying high-risk quality requirements.

With PACC you can provide a sound and objective basis to ***manage design risk*** *and* ***optimize design features.***

The root of *all risk* is uncertainty—or, to use a word closer to home, unpredictability. Improvements in predictability will diminish uncertainty and the risks that arise from it.

Risks impose real and quantifiable costs—the cost of the risk mitigation strategy. In the world of software development, the risk of failing to satisfy a worst-case latency requirement can be mitigated by the use of a more powerful processor. The cost here is extra manufacturing cost, extra heat generated at runtime, a diminished product lifespan, and, ultimately, loss of market share or product effectiveness. The risk of failing to expose a software fault during the test phase can be mitigated by lengthening the test phase itself, introducing independent quality assurance processes, and using N-version redundancy. The cost here is an extra development cost, longer schedules, more costly manufacturing, greater system complexity, and so forth.

The promise of PACC is not that it will eliminate all risk. Rather, PACC promises—for specific, critical quality requirements—to provide an objective basis for managing uncertainty and, therefore, design risk. In the worst-case latency example above, predictable performance will give designers the confidence they need to design to maximum processor usage before imposing the costly mitigation of using a higher end processor. In the software fault example, verification of critical functional requirements by model checking can shorten testing and remove the need for redundancy.

More important than the specific reasoning frameworks or other technologies used at the Carnegie Mellon Institute® Software Engineering Institute, PACC promises to provide the tools your organization needs to identify its critical quality requirements. In addition, PACC helps you put into place a means to systematically and predictably satisfy these quality requirements—by construction—in your environment using tools, theories, and standards that are appropriate for your domain. PACC is not about a particular constellation of technologies; it is about a software engineering capability—a capability within the reach of many software and system development shops.

For More Information

Linda Northrop
Director
Product Line Systems Program
Telephone: 412-268-7638
Email: lmn@sei.cmu.edu

U.S. Mail:
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

World Wide Web:
http://www.sei.cmu.edu/pacc

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.