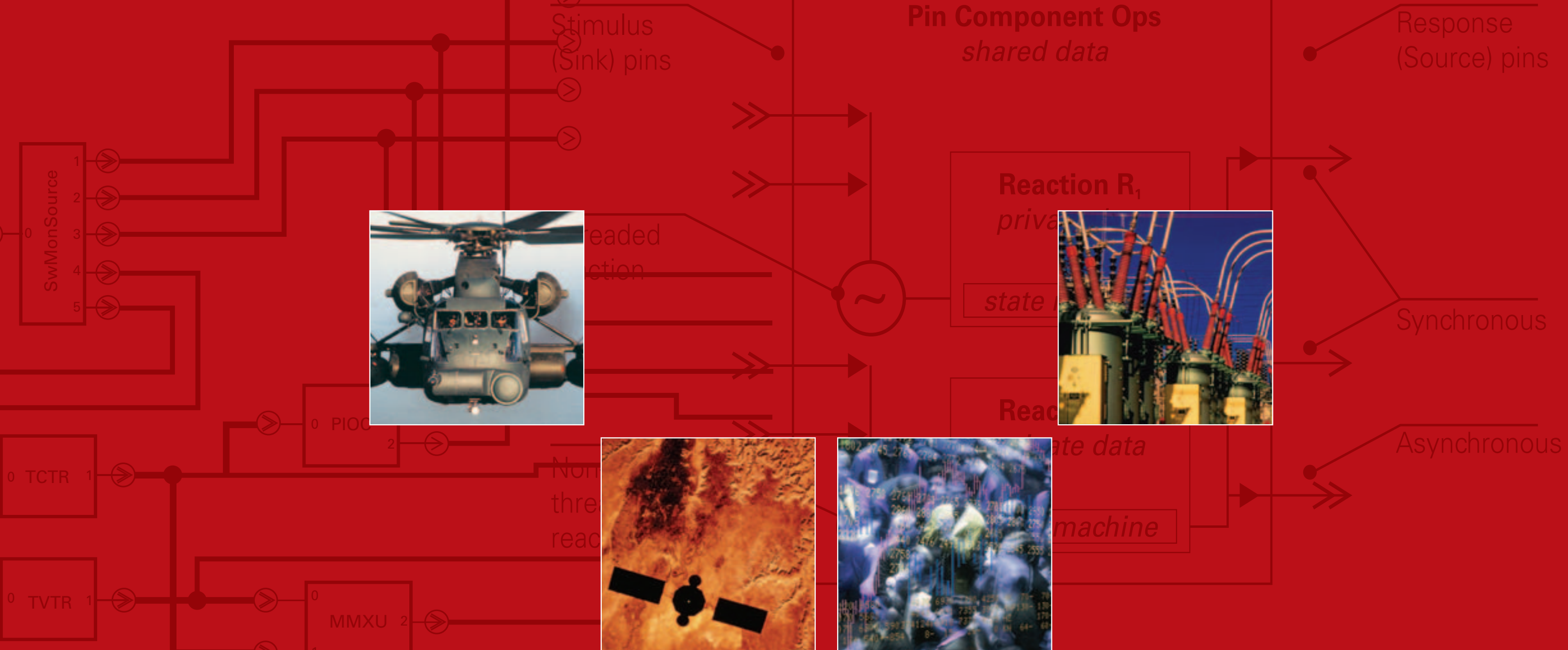
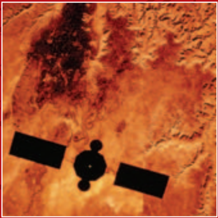
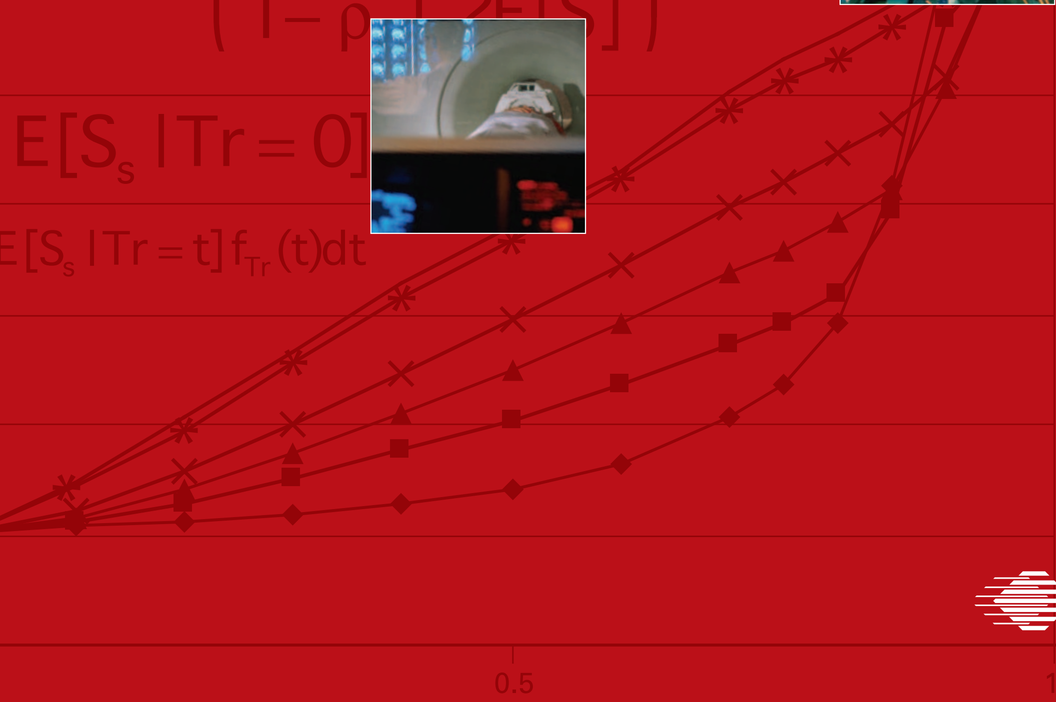


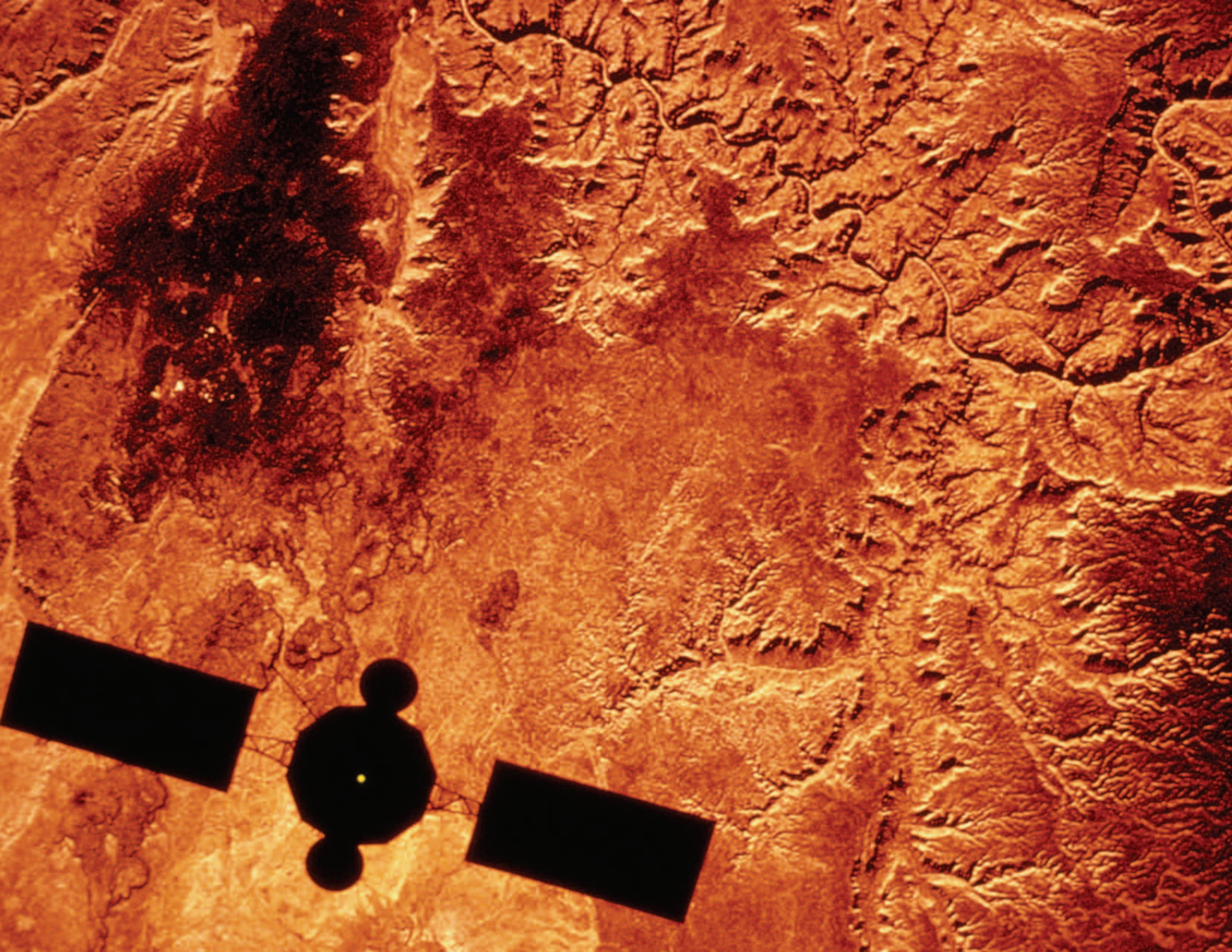




Predictability by Construction

Building high-stakes systems from certified software components

Carnegie Mellon
Software Engineering Institute



Predictability by Construction

3
Predictable Assembly of
High-Stakes Software

7
The Value of Objective Confidence

11
Predictability by
Construction: Where Plug
Implies Play

15
Getting Started with
Predictable Assembly

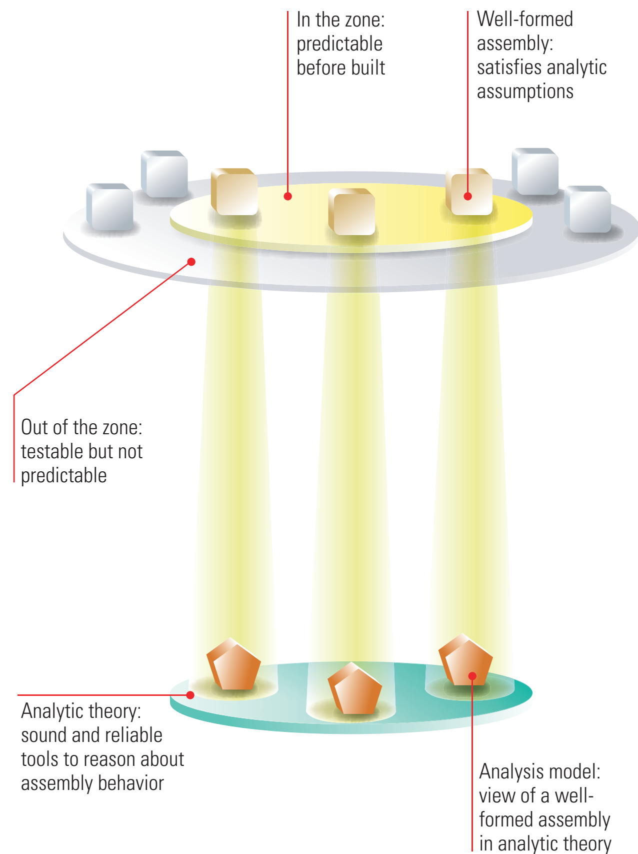
19
Achieving Value
from Day One

23
Competitive Edge for Early Adopters

25
Selected Technical Details

This is the vision that motivates a new research initiative at the Software Engineering Institute: Predictable Assembly from Certifiable Components (PACC).

Imagine the year is 2009. You stake your reputation and your company's bottom line on the innovative features of your software and its unrivaled quality. Your company quickly assembles new and innovative software-intensive systems for high-stakes applications from certified, trusted software components. Timing estimates made at design time are routinely within required tolerance with 99.99% confidence—or better. Critical safety and security properties are formally verified, and you have obtained firm control over the quality of software components developed by suppliers from around the world. These capabilities give you a potent differentiator in your increasingly competitive industry.



Component assemblies are "in the zone" if their runtime behavior is analytically predictable. Each assembly (system assembled from components) in the zone has a corresponding model in that zone's analytic theory. Using sound analytic theories to understand assembly behavior early in the development process has many advantages when compared to conventional testing techniques.

Predictable Assembly from Certifiable Components (PACC): The Science and Technology of Predictable Assembly

Engineering predictability—beyond testing

The vision for 2009 implies nothing more—or less—than engineering predictability. The Carnegie Mellon® Software Engineering Institute (SEI) has been on the forefront of research in the use of software architecture to predictably satisfy the quality requirements of software systems. The PACC vision takes architectural predictability to the extreme ranges of rigor and objective confidence, and incorporates architecture design directly into the basic units of software construction.

The PACC vision represents a fundamental advance on current practice. Today, predictability is achieved on a system-by-system basis in a way that typically involves rigorous testing. This notion of predictability relies on the premise that if we observe enough past executions, we can predict future behavior. However, the limitations of testing are well known. After all, we don't build bridges and then watch for the parts that collapse! A far better approach is to develop analytic theories that predict the behavior of entire classes of systems. In this approach, the emphasis is on confirming the validity of theories. Once confirmed, we can be sure that all systems satisfying the assumptions of the theory will have behavior that is predictable in that theory.

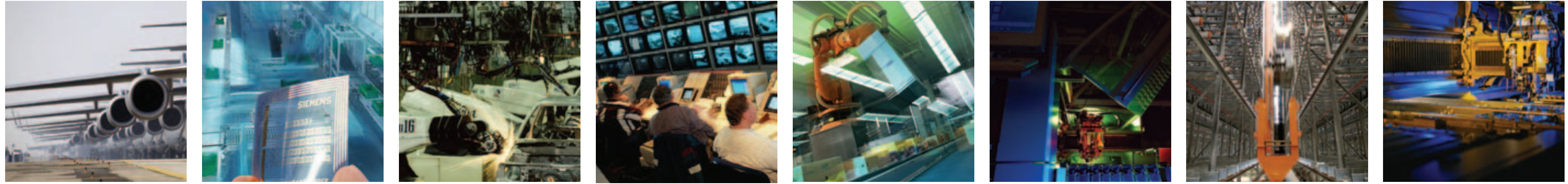
Zones of predictability

PACC divides the world into systems whose behaviors are predictable by analytic means and those whose are not. Analytically predictable systems are said to lie within the set, or *zone*, of predictable assemblies (systems assembled from components).

The behavior of assemblies that are in the zone can be determined at design time with objective confidence. Assemblies that lie outside the zone are not analytically predictable. Their behavior can only be observed after they are built.

Assemblies may be predictable using any number of analytic theories (for example, timing, security, safety, fault tolerance, and power consumption) and may therefore lie within any number of zones of predictability.

The PACC approach is to ensure that we build only systems that lie within the required zones of predictability. Our focus is on critical qualities that are likely to be of significant business value, such as performance, safety, and security.



Predictable assembly

Engineering predictability is more than just a good idea. It is achievable today. The SEI's PACC Initiative has made significant progress to this end by establishing two axioms:

1. Objective confidence. Predictions must be trustworthy. Objective confidence, through rigorous empirical and formal validation, is the acid test PACC applies to predictions. This distinguishes PACC from other model-based and generative approaches to software development.

The predicted behaviors of all assemblies have associated measures of confidence. This confidence is the bankable outcome of predictable assembly—leading to shorter development cycles, decreased development and testing costs, and higher quality systems.

2. Predictability by construction. Our aim is to build only those systems that have predictable behavior, rather than to predict the behavior of any systems we build. PACC technology ensures that the assumptions of analytic theories are established—and enforced—as invariants during system construction.

In PACC, analytic theories provide the means for predicting critical system behaviors. Analytic theories impose design constraints—the systems that satisfy these constraints are "in the zone" of this theory. PACC uses a new class of component technology to enforce—by construction—these constraints.

- We know our systems are predictable because the component technology guarantees that assemblies always stay in their required zones.
- We know which properties of components are important and must be trusted since they are the same properties required by the analytic theories.

Predictable Assembly = Objective Confidence + Predictability by Construction

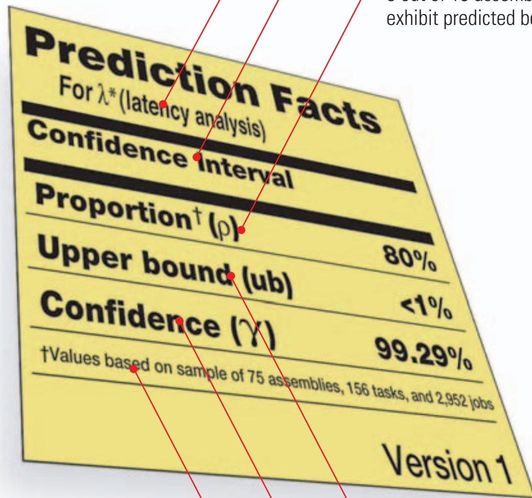
"With predictability by construction Siemens can maintain high-quality standards in our embedded software while simultaneously using a growing worldwide source of software component developers."

Matthew Bass
SEI Resident Affiliate
Siemens Corporate Research, Inc.

"There is a wide gulf between claiming predictability and trusting the predictions. Trust requires objective evidence. Objective evidence mandates systematic rigor in engineering analysis and measurement. Empirical and formal validation are part of the fiber of PACC to provide objective evidence, instill objective confidence, and engender trust."

Linda Northrop

Linda Northrop has over 35 years of experience in the software development field as a practitioner, researcher, manager, consultant, and educator. She is currently director of the Product Line Systems Program at the SEI where she leads the work in software architecture, software product lines, and predictable component engineering. Under her leadership, the SEI has developed software architecture and product line methods that are used worldwide, a series of five highly acclaimed books, and Software Architecture and Software Product Line curricula that include a total of 11 courses and 6 certificate programs. Linda is a recipient of the Carnegie Science Award of Excellence for Information Technology and the New York State Chancellor's Award for Excellence in Teaching. She is coauthor of *Software Product Lines: Practices and Patterns* and chaired both the first and second international Software Product Line Conferences (SPLC1 and SPLC2). In addition, she is the current Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) Steering Committee Chair and was the OOPSLA 2001 Conference Chair.



Objective confidence in predictions comes from reliable, quantifiable evidence. The predictions of an analytic theory are validated against sample assemblies representative of those in the zone of that theory. The SEI is charting the future of engineering confidence for software systems.

The Value of Objective Confidence

Trusted predictions

Making predictions is easy; making predictions that are consistently and quantitatively good is much more difficult. For one thing, it requires that we understand precisely what we mean by “good.” For another, it requires that we provide objective evidence of “goodness.” Despite the difficulties, the effort is well worth our while:

- Without objective evidence, engineers cannot, and should not, trust a prediction. Without this trust, a prediction has little, if any, value.
- With quantifiable trust, significant optimizations can be made to designs, families of designs, and engineering processes.

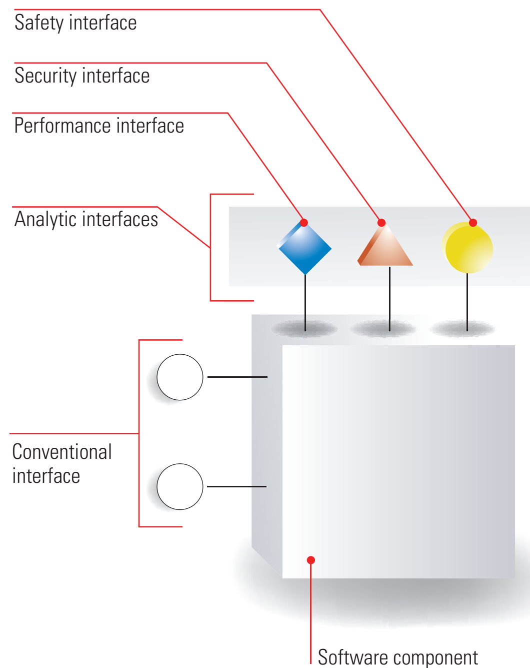
PACC already assigns a statistical confidence label to analytic theories, such as the label shown at left. The day will come when the form and content of these labels will be industry standards.

Objective confidence puts bounds on uncertainty. This is as good as money in the bank. Why? Because uncertainty is the root of all risk, and there is a cost to mitigate risk. For performance-critical systems, one mitigation might be overly conservative designs that lack useful and distinctive features or longer development cycles with extra testing and quality assurance effort. Objective confidence will allow you to avoid unnecessary risk mitigation; conversely, it will help you avoid the need to carry unnecessary risk.

Combining objective confidence with state-of-the-art analytic theories will allow us to continuously reduce uncertainty. In this way, PACC aims to effectively eliminate whole classes of design risk from your software systems. The result will be better quality software, better resource allocation, and less liability.

More concretely, quantification begs optimization. For example, perhaps a 1% upper bound is not acceptable, or the population parameter needs to be 99 out of 100 or 999 out of 1,000 assemblies. PACC provides a repeatable process for systematically improving engineering predictability for specific critical system qualities. This engineering predictability will be a core asset in maintaining your competitive position in the marketplace.

Most concretely, objective confidence will lead to substantial reductions in testing effort. For example, assuming that a 1% difference between predicted and actual latency is acceptable, testing might be limited to determine that an assembly lies within the “8 out of 10” population. Once that is established, only spot checks of latency might be required—a substantial savings in time and effort over conventional development approaches.



Today's components typically expose minimal information—often only a conventional, syntactic interface and a brief API description. Trusted components must provide more insight into their inner workings. Analytic theories require additional quality-specific properties such as task execution time to make predictions. These quality-specific properties are exposed as analytic interfaces of components.

Trusted components

PACC establishes a rich foundation for trusted components, one that goes beyond functional correctness to encompass other "as built" qualities that components must possess. Analytic theories tell us which qualities are needed to support predictability.

All analytic theories make assumptions. Some of these assumptions dictate what we need to know—and trust—about components. For example, a performance theory might require that we know the maximum non-blocking execution time for all component operations, while a safety theory might require that we possess a state machine for each component operation. In these examples, analytic theories require visibility into some aspect of the behavior or implementation of components.

These new visibilities make up the analytic interface of a component. Different analytic theories will, in general, require different analytic interfaces; each such interface defines a distinct, analysis-specific view of the component.

To have objective confidence in the predictions of analytic theories, we must also have objective confidence in the parameters to these theories—components' analytic interfaces. PACC therefore establishes an objective, measurable, and predictive foundation for component trust and certification.

By making analytic interfaces first-class elements of component technology, we extend the benefits that components bring to quick integration to reliable predictability.

Today's Component Technology: Only a First Step

Why component technology is a good first step

Component-based development is not a new idea. Components are implementations of software functionality. The boundaries of components are not set by technical considerations alone, but also by economic considerations, chief among which are the potential to apply the component in multiple markets and the need to implement nontrivial capabilities so the component has substantial value.

The logic of these considerations leads to components that are inherently complex. This complexity is packaged for consumption in what is known as software components. Whether components are developed in-house or by third-party suppliers, the inner workings of components are intentionally hidden by this packaging. Today, this packaging is a major part of the value of component technology. Current component technology also packages another form of complexity by providing standards that enable components to fit together—providing "plug" compatibility.

Why component technology is not enough

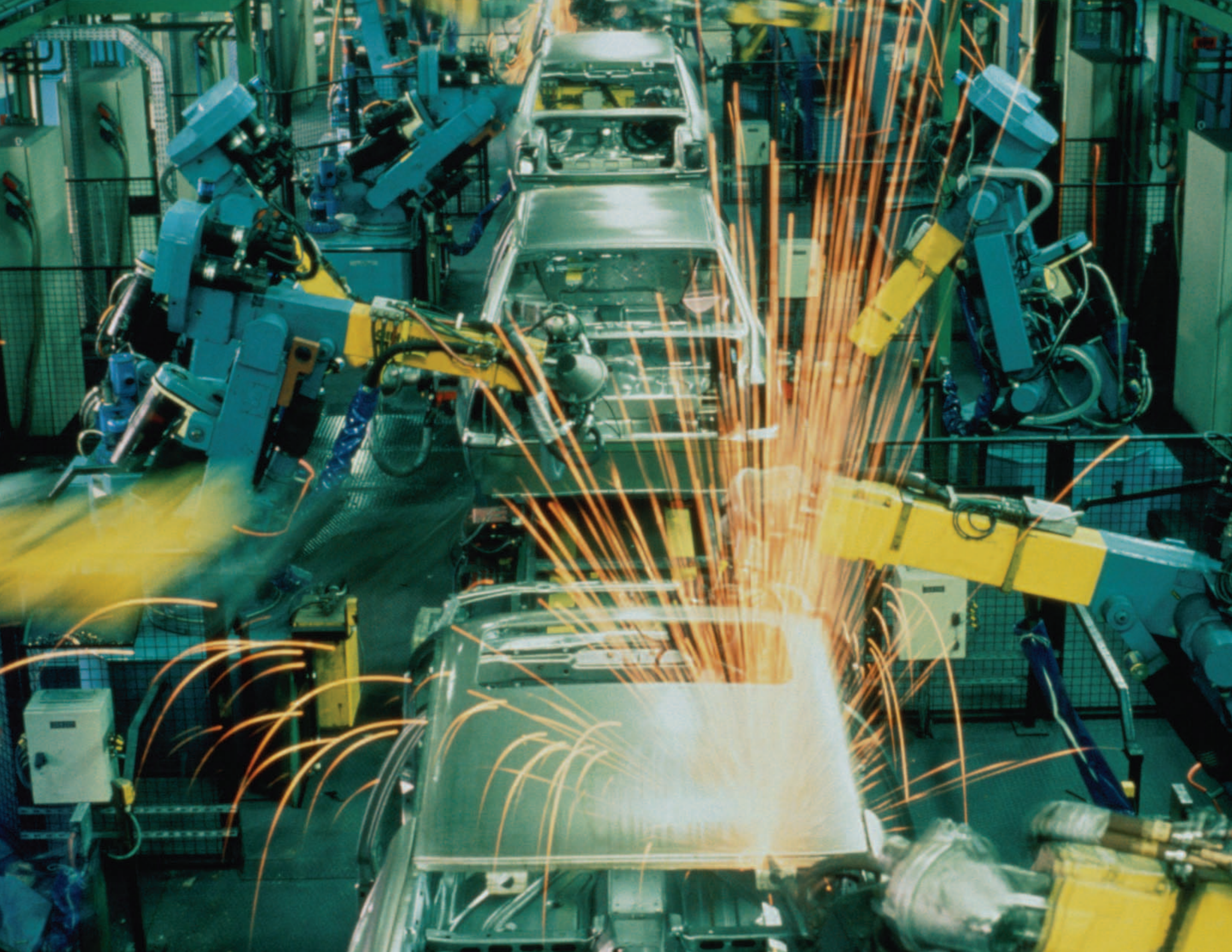
For all of its promise, today's component technology falls short of what is needed for systems with tight tolerance on runtime behavior. In today's software component technology, little is known about components beyond their plug interfaces—critical component behavior is, by design, hidden. This opaqueness makes today's component technology a risky proposition for high-stakes software systems.

When design tolerance is extremely tight, system developers mitigate risk by playing it safe. They employ conservative designs and build in long cycles for testing and rework. Development time is long, costs are high, and computing resources are not fully exploited.

But it doesn't have to be this way. The SEI and its corporate partners are developing technology to bring the promise of component-based development to software systems with tight design tolerance for timing, safety, and reliability. To do this, we must go beyond today's concept of software components by adding what is needed for the development of performance-critical systems. Our aim is to guarantee that components do more than just "plug": that they also "play"—securely, reliably, and exactly as system developers intend.

"We need a new paradigm for software engineering that moves engineering analysis to the forefront. As opposed to building software line by line, we need a system of fabrication from reusable components (not unlike how we leaped forward in the electronics industry years ago), and that results in zero defects being introduced into the software during its design and development. There is already very interesting and important work going on in this regard, known as Predictable Assembly from Certifiable Components (PACC)."

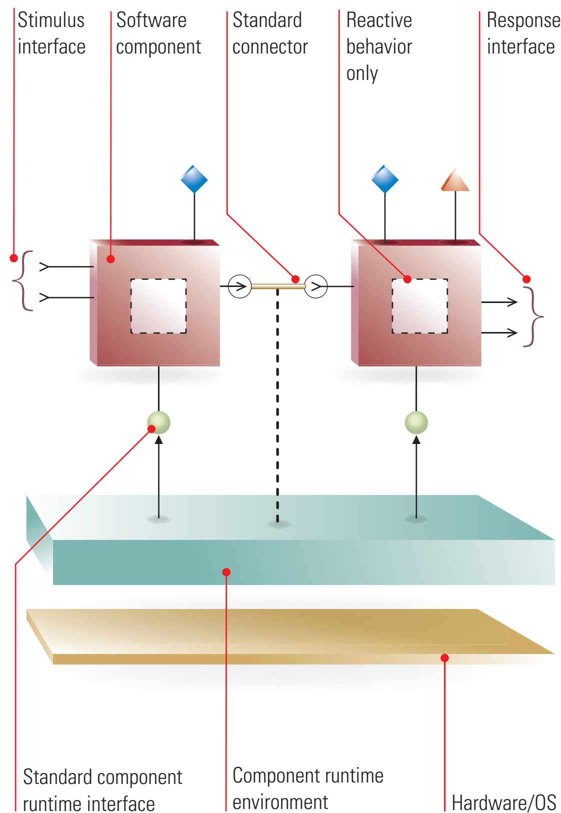
Congressional Testimony of Former Congressman Dave McCurdy, March 19, 2002, *The Current and Future Ability of the U.S. Industrial Base to Effectively and Affordably Meet Our National Security Requirements*



"A component technology is nothing more than a way of packaging design constraints. If we also include design constraints that lead to predictable behavior, then we have, in effect, packaged predictability. This is the motivation for prediction-enabled component technology."

Kurt Wallnau

Kurt Wallnau is a senior member of the technical staff at the SEI. He has 20 years of experience in software engineering research as a defense contractor and with the SEI.
A recurring theme in this research has been the use of software components. Kurt's most recent research focuses on formalizing the connections between software architecture and software component technology with the goals of making software architecture more concrete, software components more abstract, and systems composed from components more predictable at runtime. Kurt is currently the technical lead of the SEI PACC Initiative; previously he led the design and engineering work in the SEI COTS-Based Systems Initiative.
Kurt's earliest research was in application-specific languages, program generation techniques, and the use of knowledge representation techniques for semiautomated assembly of systems from components. He is coauthor of *Building Systems from Commercial Components* and numerous papers and articles on component-based software engineering.



Predictability cannot be achieved with an arbitrary component technology. To support predictability, a component technology must exhibit the minimal, ideal design pattern shown above. In this ideal pattern, all interactions among components are exposed and use standard connection mechanisms. Additionally, resource management and coordination policies are defined and enforced by a standard component runtime environment. This ideal pattern is a crucial first step toward predictability by construction.

Predictability by Construction: Where Plug Implies Play

A higher order component technology

The SEI and its research partners are developing a next generation, or higher order, component technology. By “higher order,” we mean a component technology that is parameterized by analytic theories. Filling in these parameters with specific analysis theories, for performance and security for example, results in a prediction-enabled component technology (PECT). Prediction enabling means that systems built using the component technology will be guaranteed—by construction—to be in the zone of predictability for critical system properties.

To support predictability, we need more of some things found in today’s component technologies and less of others. We need more visibility into the behavior of components and their runtime environment and more control over the design constraints imposed by the component technology. Conversely, we need less freedom for programmers to invent new and untested software architectures and fewer complex technical features that have marginal utility.

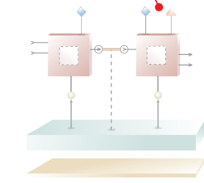
A strict foundation

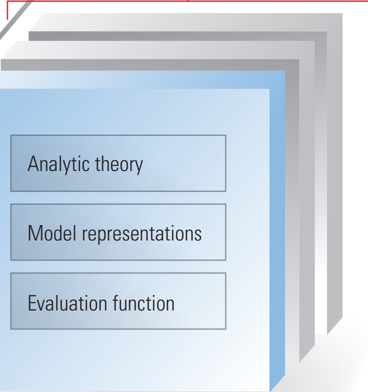
A PECT requires a strict and well-defined component technology at its core.

The component technology we use is a result of close examination of the best features and patterns found in today’s component technologies. Our design pattern (shown at left) differs from those patterns in its combination of features and the strictness with which it is enforced. It includes only the features needed to support predictability by construction.

There are many ways to implement this design pattern. The important point is the strict enforcement of the design pattern, not its particular implementation. In any case, the implementation need not be packed with features. Indeed, for systems with tight bounds on runtime behavior, less is more.

The SEI has developed the Pin component technology as an exemplar of this pattern. Pin is a small-footprint, real-time component technology that has been designed to also be simple to program, extend, and rehost.





A prediction-enabled component technology (PECT) moves a step beyond the ideal component technology pattern by introducing reasoning frameworks. Reasoning frameworks package state-of-the-art analytic theories and enable nonexperts to predict critical runtime qualities such as performance, safety, and security.

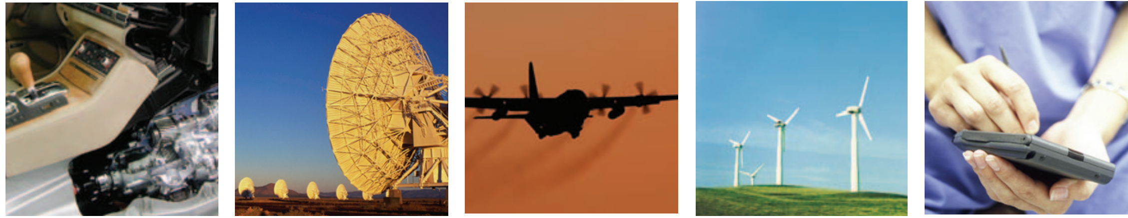
The real breakthrough: reasoning frameworks

The real breakthrough in our research, however, is not component technology but rather the extension of component technology with reasoning frameworks.

Reasoning frameworks make state-of-the-art analytic theories accessible to system integrators and component developers rather than just to experts in security, performance, or safety. In effect, the reasoning frameworks treat this expertise as a new kind of component in its own right. Reasoning frameworks encapsulate, and package for use, the engineering competence required to construct systems that satisfy critical runtime requirements with specified design tolerance.

The elements of a reasoning framework are

- **an analytic theory, based on a solid foundation such as queuing theory, rate monotonic scheduling theory, finite-state automata, or temporal logic**
- **analytic constraints that mirror the enforceable design and implementation assumptions of the analytic theory**
- **an interpretation that checks assemblies for well-formedness in the reasoning framework and generates analysis models**
- **model representations in the theory, defining behavioral “views” or “semantics” of assemblies of components, for example, in terms of time**
- **a computable evaluation function, which, when given an analysis model, constructs a prediction of assembly behavior that can be objectively (observably) validated**

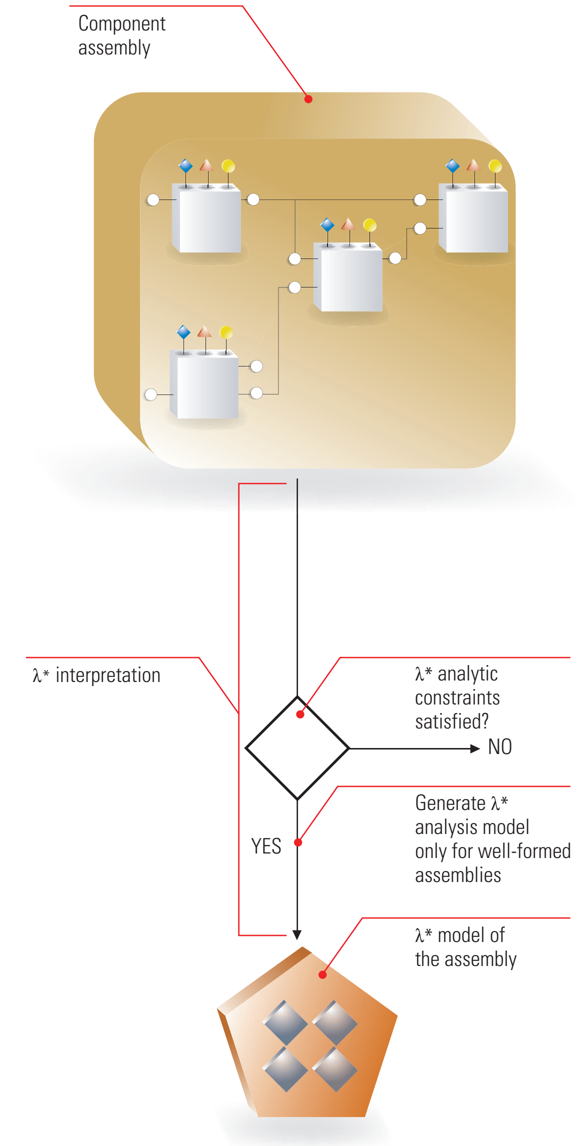


Although there is quite a bit of computer science in the structure and implementation of reasoning frameworks, the core idea is simple and straightforward. Every prediction is based on assumptions about the subject of the prediction (the assembly). Reasoning frameworks ensure that assumptions are established as invariants in assemblies, much as type checkers in modern programming languages establish representation invariants in computer programs.

The illustration at right shows how the idea works using the PACC technology developed at the SEI. The reasoning framework called "λ*" packages a variant of rate monotonic scheduling theory that has been specialized to a range of characteristic assemblies for some application area. The λ* interpretation checks whether component assemblies satisfy the assumptions of this specialized theory (formalized as analytic constraints). If they do, the interpretation generates a corresponding analysis model and applies the evaluator to "solve" the model.

The discipline of constructing reasoning frameworks generates unique value in and of itself by codifying the engineering competence on which your organization depends and by making this competence a first-class asset in its own right.

Of course, few systems only have performance, security, or safety requirements. To deal with multiple quality requirements, multiple reasoning frameworks must be applied. Each reasoning framework's interpretation defines a projection of the system with respect to that theory. In any well-constructed PECT, the zones of predictability for multiple reasoning frameworks will overlap significantly, ensuring that systems are predictable with respect to multiple qualities.



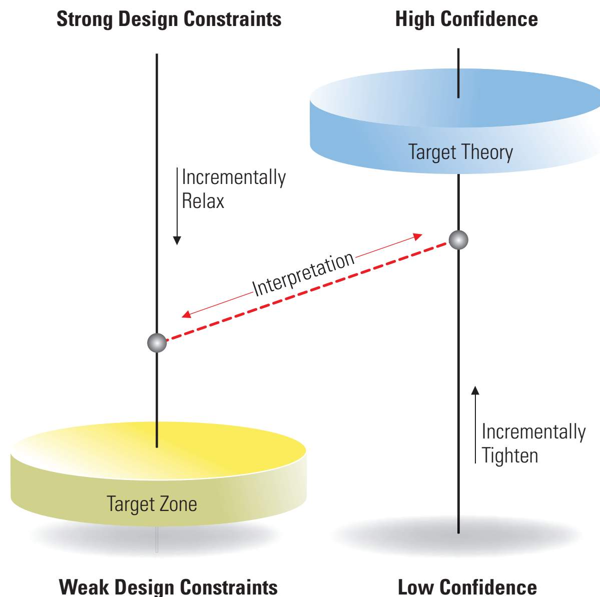
Each component assembly that satisfies a reasoning framework's analytic constraints is in that reasoning framework's zone of predictability. For such an assembly, the reasoning framework's interpretation can be used to automatically generate an analysis model that describes a view of the assembly in terms that are relevant to the reasoning framework. The reasoning framework uses this analysis model to predict the runtime behavior of the assembly.

"Utmost reliability of the software in our products and systems is crucial for ABB. The safety of an electric grid or the operation of a whole factory may depend on that. We follow with deep interest the PACC approach and welcome further initiatives to improve software engineering."

Markus Bayegan
Chief Technology Officer
Group R&D and Technology
ABB Asea Brown Boveri Ltd

Dr. Markus Bayegan began his career in 1973 as a system engineer with Integrated System Development in Oslo, Norway. Within a couple of years, he moved to the Center for Industrial Research, also in Norway, where he worked as a research scientist for two years and later as R&D manager. His work has focused on computer-aided design, databases, and microelectronics. During this time, he was also a visiting research scientist at Stanford University in the U.S. In 1981, he took the position of senior vice president of R&D at the EB Corporation, a telecom company, which today is part of Ericsson. Since 1985, he has held the position of professor of electronics manufacturing at the Norwegian Institute of Technology. For 10 years, he held dual positions of president of ABB Corporate Research Norway and senior vice president for ABB Asea Brown Boveri Ltd. Since January 1998, he has worked in the Group R&D and Technology division in two capacities: as senior corporate officer, and since January 2001, as chief technology officer. Dr. Bayegan is the coauthor of several books on microelectronics and information technology.

Getting Started with Predictable Assembly



The co-refinement process is a systematic process for developing reasoning frameworks. The process begins with a base step: a reasoning framework that is as constrained as necessary to be thoroughly understood. The “zone” of this base step may be a small subset of the target zone and fail to deliver the required confidence. The process proceeds by incrementally relaxing design constraints and improving confidence (co-refinement), while preserving the ability to apply the reasoning framework’s analytic theory. It does so by ensuring that a valid interpretation exists at each step. The process continues until the reasoning framework can be applied to the target systems and produce predictions with the target level of confidence.

Co-refinement: a method for systematic refinement of predictable assembly

The SEI has developed the co-refinement process for trading off restrictiveness of design constraints, required design tolerance, and analysis tractability. Co-refinement has proven to be an effective tool for capturing the design competence needed to deliver predictability by construction. It identifies the weakest design and implementation constraints needed to achieve required prediction accuracy for your systems. With this process, your organization will obtain real control of the critical runtime qualities of your systems and the components that make up your systems.

The co-refinement process starts with the largest set of assemblies for which developers can make design predictions with confidence and that meet required tolerances. The idea is to start with a base case that is thoroughly understood and for which a sound interpretation to a reasoning framework can be constructed.

Co-refinement proceeds through a series of controlled relaxations of design restrictions. Each relaxation increases the size of the zone of predictability. Each step also preserves interpretability so that design constraints and predictability evolve hand in hand (or co-evolve).

The process stops when the set of predictable assemblies is large enough to contain the target zone; that is, when it describes the set of applications of interest to your business. The resulting design restrictions are sufficiently tight to enable the reasoning framework to predict qualities with the required confidence and tolerance, and sufficiently loose to permit the exploration of various system designs and product variants that are relevant to your application domain.

But the process need not stop there. The exceptional focus on objective confidence introduced by PACC will enable your organization to systematically improve its engineering competence over time by tightening confidence bounds or relaxing design constraints and by introducing new reasoning frameworks. Engineering predictability is not an end state—it is a mind-set.

Getting Started with PECT

A basic but complete PECT that includes performance and model checking reasoning frameworks that can be used to predict the behavior of time- and safety-critical embedded systems. The Starter Kit was designed to jump-start the PECT development process, allowing extensions to other reasoning frameworks or product domains.

A specialization of the Starter Kit for a particular product domain, set of qualities, or product line. The SEI works with partner organizations to develop these embodiments of an organization's engineering competence.

Building your PECT

A prediction-enabled component technology, or PECT, is an expression of your organization's engineering competence. Since this competence is your competitive distinction in the marketplace, don't expect to buy it off the shelf. Instead, to be effective, a PECT must be targeted to the class of systems your organization builds, the critical behaviors that your systems must exhibit, and the distinctive way you build these systems.

Starter Kit

Developing a PECT may seem like a daunting task, but the SEI has been working on a way to jump-start the process with the Starter Kit. The Starter Kit is a basic but complete PECT that can be used "as is" for time- and safety-critical embedded systems. From this basis, you can construct your own business-specific PECT.

The elements of the Starter Kit are in themselves state of the art, but move well beyond that when combined in the Starter Kit.

Today's Starter Kit provides

the Pin component technology, a small-footprint, real-time component technology that includes the minimal features needed to support predictability. Pin operates on Windows CE and Windows 2K platforms and can be readily rehosted to "bare" devices.

a family of performance reasoning frameworks (λ*) for predicting average- and worst-case latencies in a variety of design settings (e.g., periodic and stochastic event arrivals)

a model checking reasoning framework (ComFoRT) for verifying safety and liveness conditions of components and assemblies (e.g., ensuring that communication protocols route information correctly)

an instrumentation and observation infrastructure for observing runtime behaviors of assemblies and for performing statistical analysis on the differences between observed and predicted behavior

a construction and composition language (CCL) and compilation tools that perform static checking of design constraints against CCL specifications and generate analysis models and component implementations (code) for well-formed specifications

Today's Starter Kit is a basic but complete PECT that can be used "as is" for time- and safety-critical embedded systems. The elements of the Starter Kit can also be used to jump-start the development of new PECTs, through the addition of new reasoning frameworks or the extension or replacement of existing elements.

Assembly Specification

Code Generator

Deployed Assembly Implementation

Pin Runtime Environment

Instrumentation and Observation Environment

WinCE Version

Win2K Version

Other

Intel x86

Other

Model Checking Reasoning Framework

Model Checking Interpretation

Temporal Logic and Finite-State Automata Theory

Labeled Kripke Structures

Model Checking Engine and Theorem Prover

Performance Interpretation

Queuing Theory and Rate Monotonic Analysis

Tasks, Subtasks, and Timelines

Simulation and Equation Solvers

Performance Reasoning Framework

"The SEI works closely with organizations to provide support throughout the development life cycle. We strive to create mission success and competitive advantage through our training programs, technical service offerings, and advanced research initiatives. We look forward to leveraging the potential of PACC with organizations looking to win the future."

Achieving Value from Day One

The SEI excels at bringing state-of-the-art software methods and technology to bear on industrial-strength problems. The SEI's PACC Initiative has pioneered a method for technical collaboration that focuses predictable assembly directly on those system qualities for which predictability is paramount. This method produces objective proofs of feasibility for PACC that are situated in your critical business area, while at the same time transitioning to your organization the capability to scale these proofs to industrial use.

The method is centered on the collaborative specification and solution of model problems. A model problem specifies a base design pattern and one or more operational scenarios, with each scenario distinguishing system stimuli from system responses. Prediction is expressed in terms of response measures and confidence bounds—both establish the norms for objective confidence. A model problem reduces a predictability problem to its essence; its solution is clearly applicable to the system it models.

The process of specifying model problems is of significant value to collaborators. System-specific analysis problems can often be generalized in model problems to a class of systems—and the solution to those problems is therefore applicable to that class. Defining a model problem is not simple, but it is a critical first step in achieving engineering predictability. As with co-refinement, the social processes involved in specifying model problems can be as important as the technical results of these processes.

Each model problem initiates a co-refinement process, each step of the co-refinement process yields a prediction solution to some class of systems, and each successive step produces solutions that better approximate the class of systems represented by the model problem. The value of this process is not limited to the technical by-products it creates; the process also imparts a depth of understanding to its participants about the class of systems under development and about the tradeoff between design freedom and predictability. At each step, your organization will be accruing value.

Benefits for SEI research partners

SEI research partners actively participate in a research activity and receive numerous benefits:

- Early access to research results provide the partner with the opportunity to adopt cutting edge technologies well before their competitors can.
- We tailor our research agendas around the applications and model problems provided by research partners. Doing so often provides direct and tangible benefits to the partner in the application area.
- In the case of PECTs, partners will help us set the standards for a new class of component technologies.

Photos: ABB

ABB collaboration

ABB Ltd., a leader in power and automation technologies that enable utility and industry customers to improve performance while lowering the environmental impact, became an SEI research partner in 2001. This collaboration has produced significant results for both ABB and the SEI.

The initial application of predictable assembly was in the electric power grid domain. The question was whether operator stations and primary equipment controllers could be assembled such that average latency could be predicted for primary control functions such as closing a high-speed breaker and for interactions between human operators and primary controllers. It was also important that the resulting assemblies satisfy the IEC-61850 functional standard for substation automation systems. The ABB Corporate Research Center provided funding and a full-time resident affiliate to explore this question.

The results were promising:

- A PECT for soft real-time operator stations and a PECT for hard real-time primary equipment controllers were built using the SEI Pin component technology. These controllers were integrated using proprietary ABB messaging software and conformed to the IEC-61850 standard.
- The λ_{ABA} reasoning framework was developed for predicting average latency in primary equipment controllers. λ_{ABA} supports very general topologies of components interacting both synchronously and asynchronously.
- The general method and supporting automation for validating the objective quality of performance theories were developed and applied to λ_{ABA}. The results were foundational for PACC and also earned the ABB resident affiliate his PhD.

ABB and the SEI later applied the results of the substation automation research to the domain of industrial robot control. Controllers in this domain perform many functions and manage a large variety of concurrent tasks, only some of which are hard real-time and safety critical. This domain presented several challenges to the research team. Again, the results were promising:

- An assortment of safety properties of the controller's interprocess communication mechanism were formally verified using model checking. Verifying properties of this nontrivial subsystem (exceeding 10^{1932} discrete states) provided results of great interest to the controller software engineers. In one case, the model checker found a rare anomaly (only recently discovered by ABB) that had eluded detection for 7 years.
- The λ_{ABA} reasoning framework was extended to handle events with aperiodic arrivals and introduced novel queuing-theory extensions to classical rate monotonic analysis (RMA). The extended reasoning framework allows third-party extensions to be plugged into a new generation of open robot controllers in a way that guarantees bounded invasiveness of the plug-in component on hard real-time control operations and allows accurate prediction of average-case latency of plug-in operations.

The four years of collaboration between the SEI and ABB have been of tremendous value to both parties. ABB benefits from working with the SEI on PACC because it will be among the first to reap the benefits of predictable assembly. PACC solutions to challenging ABB problems are directly applicable to the class of systems built by ABB.

The SEI also benefits from working with industrial collaborators such as ABB. The PACC solutions to ABB problems can be applied to a vastly large class of analogous problems found throughout the commercial and defense segments of the software industry. Collaboration is the surest way for the SEI to achieve its mission of large-scale technology transition to improve the state of the art, and the state of the practice, of software engineering.

The SEI Seeks Innovative Collaborators

Collaboration opportunities

During 2002-2004, the SEI developed PECT and applied it, and its associated methods, in increasingly demanding industry settings. The result is state-of-the-art technology—the Starter Kit.

The SEI seeks innovative research collaborators to apply PACC technology to industry systems that will benefit the most from predictability by construction—systems that exhibit three or more of the following characteristics: embedded, mobile, distributed, safety critical, time critical, and security critical.

Representative application areas include industrial robotics, automotive, aerospace, power generation and transmission, medical diagnostics, and command and control. Qualified collaborators will have significant control over the design and implementation of these systems, although major portions of them may be developed by third parties.

The following kinds of collaborations are sought (and are not mutually exclusive):

1. trial use of PACC to produce scaled-up advanced technology demonstrations, proofs of feasibility, and return on investment (ROI) data

2. refinement of Starter Kit reasoning frameworks and development of new reasoning frameworks. Near-term opportunities exist in the areas of security, safety and reliability, and timely behavior.

3. applying PACC technology to software product lines, resulting in an industry- or corporate-standard Starter Kit that combines the functional variants of a product line with design restriction for predictability

4. developing technology for automated, high-assurance certification of software components and reasoning frameworks, including self-certifying components, model-code conformance, and standard component and prediction labels

5. advanced research in automated design assistance, including design transformations to optimize performance within the predictability zone for specific runtime behaviors and tradeoffs among behaviors in different zones

As part of their relationship with the SEI, research partners provide the following:

- model problems or sample applications that can be used as test cases or focal points for our research
- funding, which is used to support research activities agreed to in advance with the partner and for which the partner will receive direct and indirect benefits
- optionally, personnel. The partner organization may choose to have its own staff actively participate in research activities.

Are your systems embedded, mobile, distributed, safety critical, time critical, and/or security critical?

“There are several opposing ‘forces’ at work in creating a PECT. For example, practitioners advocate enlarging the construction framework while reasoning framework developers advocate constraining it to ensure interpretability and solvability. Co-refinement can be thought of as a managed, incremental negotiation between these opposing forces.”

Mark Klein

Mark Klein is a senior member of the technical staff at the SEI. He has over 20 years of experience in research on various facets of software engineering, dependable real-time systems, and numerical methods. Mark's most recent work focuses on the analysis of software architectures, architecture tradeoff analysis, attribute-driven architectural design, and scheduling theory. Mark's work in real-time systems involved the development of rate monotonic analysis (RMA), the extension of the theoretical basis for RMA, and RMA's application to realistic systems. Mark's earliest work involved research in high-order finite element methods for solving fluid flow equations arising in oil reservoir simulation. He is the co-author of *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems* and *Evaluating Software Architecture: Methods and Case Studies*.

Competitive Edge for Early Adopters

Our objectives apply to your business

The mission of the SEI is to advance the practice of software engineering.

The SEI aims to

- accelerate the introduction and widespread use of high-payoff software engineering practices and technology by identifying, evaluating, and maturing promising or underused technology and practices
- maintain a long-term competency in software engineering and technology transition
- enable industry and government organizations to make measured improvements in their software engineering practices by working with them directly
- foster the adoption and sustained use of standards of excellence for software engineering practice

Effective technology transition is paramount to achieving our mission. Each research agenda we undertake is pursued with the practitioner in mind.

Our research collaborators become early adopters of technology and methods that we are dedicated to transitioning into widespread practice. Our collaborators are in a unique position to influence the direction and the packaging of that technology. They ultimately become community leaders in using high-payoff practices and technology that give them a competitive edge.

The PACC pedigree

The PACC work is the newest arm of the broader technical agenda of the SEI Product Line Systems Program. PACC builds on more established but complementary research in software architecture and software product lines. Both areas of research have exemplary track records in following the SEI model of applied research and widespread transition. Both areas of research have afforded early collaborators unique technical and business advantages. PACC will provide these same advantages.

Software architecture technology has been a key research area of the SEI since 1990. The SEI has developed and applied methods and techniques so that practitioners can use effective architecture-centric practices in software development. The SEI Architecture Tradeoff Analysis Method® (ATAM®) is now an internationally used software architecture evaluation method. The SEI "Views and Beyond" documentation approach for software architecture has given practitioners a way to create relevant and useful documentation for software architectures. The SEI has published four books on SEI software architecture technology that have sold more than 50,000 copies to date and are used by major corporate technical training centers and university computer science departments for software architecture instruction. Individuals from more than 160 companies have taken software architecture courses at the SEI in 2004.

The SEI is also a leader in software product lines. The SEI provides the seminal product line reference model, the SEI *Framework for Software Product Line Practice* ℠, as well as proven product line methods and patterns that have been successfully used to achieve tenfold time-to-market decreases and cost decreases of 50 percent.

Robert Bosch, the world's second leading supplier of automotive technology, was an early collaborator with the SEI on software architecture and software product lines. This relationship has been fruitful for both the SEI and Bosch. SEI ATAM training was made available to the public in 2003. By this time, Bosch had already trained evaluators and was using the ATAM routinely to surface architectural risks early in the development life cycle. Likewise, Bosch has already experienced substantial advantages in piloting the *Framework for Software Product Line Practice*, such as one-third savings in memory resource consumption and higher reuse potential than with their former reuse approach.

The National Reconnaissance Office, another early SEI product line collaborator, had already achieved a 50-percent reduction in its cost and schedule with its pioneer product line effort before the SEI began to transition SEI product line practices to the broader community.

PACC is representative of the most forward-leaning work at the SEI. We are looking for technology innovators to help us develop the potential of predictable assembly.

Selected Technical Details

26
Overview of a Model Checking Reasoning Framework

28
Overview of a Performance Reasoning Framework

30
Validating and Labeling the Objective Confidence of a Reasoning Framework

32
Overview of the Pin Component Technology

34
Selected Publications

Overview of a Model Checking Reasoning Framework

**Sagar Chaki, James Ivers,
Natasha Sharygina, Kurt Wallnau**

1. Introduction

Model checking is a powerful error detection technology that has been successfully applied in the hardware industry. Model checking works by checking a model of a system against behavioral claims (requirements) that should be true of the model. Examples include claims that an actuator cannot move while the system is in a safe mode and that messages are never delivered to the wrong recipient.

Model checking goes far beyond conventional testing procedures in terms of coverage because it uses an algorithmic approach to *exhaustively* check every possible path through a model of a system. All combinations of every thread interleaving, every point at which events could be received, and every parameter value that an event or invocation could have is examined.

Moreover, while conventional testing only indicates whether a system passes or fails a test, model checking provides feedback to help solve the problem. Whenever a claim does not hold, a counterexample is reported detailing the execution path leading to the problem. This information can be used to understand or localize the cause(s) of the problem or to generate specific test cases to reproduce the problem.

While model checking is used extensively in the hardware industry, to date, successful applications to software have been limited to small examples. Two obstacles to wider use are scalability and the degree of expertise needed to successfully apply model checking.

We address these issues by incorporating state-of-the-art model checking research into a reasoning framework called ComFoRT that allows nonexperts to gain the benefits of model checking for component-based software.

2. State-of-the-Art Research

Model checking's scaling issues arise from its strength—exhaustive analysis. The size of the complete state space for a concurrent system is exponential in the number of states and concurrent units; this is known as the state space explosion problem.

Consequently, most model checking research is focused on reducing the number of states that must be searched or performing searches more efficiently.

The two principle approaches to dealing with state space explosion are

1. abstraction: constructing smaller abstract models such that a claim is guaranteed to hold for the concrete model if it also holds for the abstract model
2. compositional reasoning: partitioning a model into a number of smaller models that can be checked independently and proving system correctness based on the model checking results for the submodels

Our model checking reasoning framework draws on current research from both approaches, combining them to address real software systems. A brief overview of some of the specific algorithms used is provided below.

2.1 Automated Predicate Abstraction

Predicate abstraction is used to systematically generate a smaller, abstract model from the original, complete model. It maps concrete data types to abstract data types through predicates over the concrete data. For example, reducing a 32-bit integer to 3 predicates: $i < 0$, $i == 0$, and $i > 0$.

The automated abstraction techniques used in ComFoRT make use of the predicates found within the original model (e.g., loop and branch conditions) to determine the set of predicates to be applied.

Model checking such abstract models is considerably easier and is a safe operation because the abstract model preserves the complete model's behavior. Each abstraction is proven to be a conservative abstraction (using automated decision procedures), meaning that if a claim holds for the abstract model, it must also hold for the complete model.

2.2 Compositional CEGAR Framework

However, if a claim does not hold for the abstract model, that does not necessarily mean that it does not hold for the complete system. Consequently, each counterexample must be checked to determine whether it is spurious (a result of the abstraction), and if so, model checking must be repeated with a less abstract model.

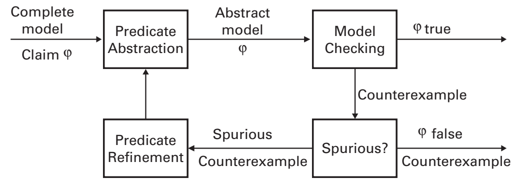


Figure 1: CEGAR Framework

We use predicate abstraction in conjunction with a counterexample-guided abstraction refinement (CEGAR) framework, shown in Figure 1, to discharge spurious counterexamples. In the CEGAR framework, abstract models are constructed and refined as needed, until model checking is completed.

Specifically, for a complete model C consisting of concurrent units $C_1 \ldots C_n$, the algorithms that check whether a claim φ holds for C use the following iterative process:

1. abstract. Create a conservative abstract model $M = M_1 || \ldots || M_n$.
2. verify. Check if a claim φ holds for *M*. If so, report success and exit. Otherwise, let *CE* be a counterexample that indicates where φ fails in *M*.
3. refine. Check if *CE* is a valid counterexample with respect to *C*. If *CE* corresponds to a real behavior, report the counterexample showing why φ does not hold for *C*. If *CE* is spurious, refine *M* using *CE* to obtain a more precise abstract model and repeat from Step 1.

Two important benefits of this approach include the following:

1. By starting with a small abstraction and incrementally adding details, state space explosion is delayed as long as possible.
2. Steps 1 and 3 are both performed compositionally (i.e., one concurrent unit at a time). Only Step 2 requires the construction of a composed state space, and this composition always involves only the current abstract models.

2.3 State/Event-Based Formalism

In model checking, behavioral claims are typically expressed using temporal logics over state information (e.g., variable values or the current state) or communication information (e.g., event arrival or function invocation). But limiting claims to one form or the other is usually a bad fit for concurrent software in which both concepts are significant and often related.

For example, to check whether a controller takes the right actions when a sensor reports (an event) a value that exceeds a threshold (state), both concepts are needed.

ComFoRT uses a state/event-based formalism that allows such claims to be naturally expressed and efficiently verified. A labeled Kripke structure (LKS) is used to represent models. An LKS is a directed graph in which states are labeled with atomic propositions and transitions are labeled with events. A state/event derivative of linear temporal logic is used to express claims over the LKS.

Experiments have shown that this formalism not only simplifies writing claims, but shows significant gains in efficiency. Without a state/event formalism, to check a claim like our example, users have to annotate their models with additional information to capture the concepts that are not expressible in a state- or event-based formalism, exacerbating the state space explosion problem.

3. Packaged as a Reasoning Framework

Successful application of model checking often involves

- learning the specialized languages used by a particular model checker, many of which are better suited to hardware description
- simplifying the model by hand to improve tractability, often in clever or unrealistic ways
- finding the right combination of algorithms to permit successful model checking for a particular model and claim

Much of this work can be automated and hidden from users, particularly when incorporated in a reasoning framework and restricted to a particular problem domain (embedded controllers).

Figure 2 shows how the ComFoRT reasoning framework is applied. Each step is described below.

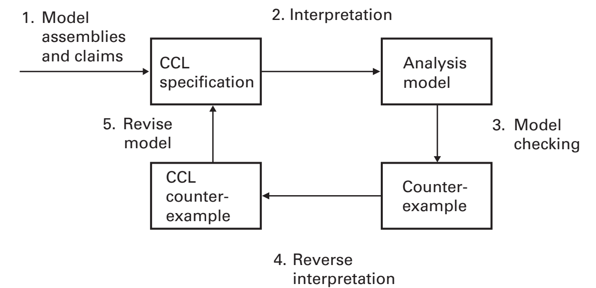


Figure 2: Applying the ComFoRT Reasoning Framework

1. Users design their components and assemblies in a design language (e.g., CCL) and specify the claims to be checked.
2. The interpretation generates an analysis model from the assembly.
3. Model checking is applied, determining whether each claim holds. When a claim does not hold, a counterexample is generated.
4. Reverse interpretation is applied to any counterexamples to express them in terms of the assembly, rather than the analysis model.
5. Users examine any counterexamples and determine how to modify their designs to solve any problems. Any modifications are then checked by returning to Step 2.

Steps 2-4 are fully automated, restricting the user's attention to where it belongs—designing the assembly, specifying the behavioral claims that should hold, and deciding how to solve any problems that are detected.

Overview of a Performance Reasoning Framework

Scott Hissam, Mark Klein,
Paulo Merson, Gabriel Moreno

1. Introduction

Performance, or specifically timing behavior, is important to all systems. For some systems, meeting hard deadlines is critical. For other systems, missing deadlines occasionally might be fine, but only if you can guarantee that the miss rate will not exceed a specified threshold. In many cases, average latency is the performance measure of merit. Our ultimate goal is to have reasoning frameworks for each of these performance measures.

This overview provides a high-level summary of a reasoning framework that is suitable for predicting the average latency in control systems comprising a mix of periodic and aperiodic events. This reasoning framework is motivated by systems that must handle aperiodic events without sacrificing the hard real-time deadlines of periodic processes.

2. Background

We began with a performance reasoning framework called λ_{ABA}. λ is short for latency, and ABA is short for Average case, with Blocking and allowing for Asynchrony.

λ_{ABA} is built on a body of work known as Generalized Rate Monotonic Analysis (RMA), which offers the ability to predict worst-case latency to ensure that hard deadlines are met. λ_{ABA} extends RMA to predict average latency. λ_{ABA} was constrained to a set of component assemblies, the interpretation of which reduces to sequences of tasks (units of concurrency) initiated periodically using both synchronous (e.g., call/return) and asynchronous (e.g., message passing) communication. Recently, we relaxed the analytic constraints on assemblies to also include those tasks initiated stochastically (or aperiodically—the terms are used synonymously) as well as periodically. This work entails developing a new analytic theory for predicting the average latency of aperiodic events in the context of a collection of real-time periodic events. This new theory also imposes analytic constraints—in particular, assemblies must manage aperiodic events with sporadic servers.

The sporadic server scheduling algorithm protects periodic events with hard deadlines from bursts of high-priority stochastic events that also require high-priority processing. The hallmark of a sporadic server is that it provides a periodic "virtual processor" within which aperiodic events can be processed and analyzed. For the analytic theory described below, the two key parameters of a sporadic server are its execution budget and replenishment period. We assume that the execution time needed to handle an aperiodic event is equal to the sporadic server's execution budget.

3. Empirical Analysis

Our performance reasoning framework combines empirical analysis via simulation with theoretical results. Some of our early simulations revealed an interesting pattern of average latency as we varied some key parameters.

We simulated and analyzed a very simple situation that gave us insight into a much more general class of problems. We studied a single aperiodic stream of events with interarrival times governed by an exponential distribution and constant services. This is known as an M/D/1 queuing problem.

The aperiodic events were processed under the control of a sporadic server. The assembly also had a single periodic stream of events. The sporadic server executed at the highest priority and allowed the aperiodic events to execute whenever it had an available budget. Aperiodic events also exploited any available idle time left over from the periodics, that is, background time.

We examined how the average aperiodic latency (E[W]) varied as a function of a period of the periodic task (T_p) and the utilization of the periodic task (U_p). The results are shown in Figure 1.

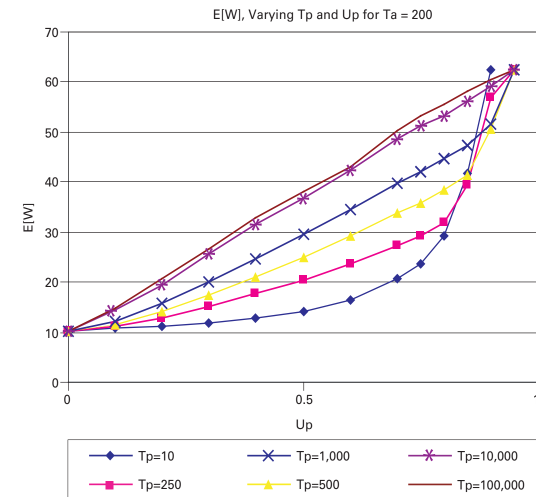


Figure 1: Average Aperiodic Latency

Each curve represents a suite of simulations for which the periodic task's period was held constant and periodic utilization, U_p, varied from 0 to .95. The lowest curve had a periodic period, T_p, of 10 ms, and the highest curve had a periodic period of 100,000 ms.

Note that

- The curve for the smallest periodic period, $T_P = 10$, looks like a standard queuing curve; it became the target of our queuing analysis.
- Curves for the smallest and largest periodic periods are bounds for the other curves.
- All the curves nearly meet at the lowest and highest levels of periodic utilization.

4. Queuing Theoretic Analysis

The question was, "can we coerce" this problem into a standard queuing theoretic framework? The key queuing result that we draw on is the following formula:

$$E[W]=\left(\frac{\rho}{1-\rho}\right)\left(\frac{E[S^2]}{2E[S]}\right)+E[S]$$

The first term of the equation is the mean queuing time, which we will denote by E[Q]. E[S] is the mean aperiodic execution time, and ρ is the mean service time divided by the mean aperiodic interarrival time. Therefore, the equation basically says that the mean latency is the mean queuing time plus the mean service time: E[W] = E[Q] + E[S].

4.1 Special Case of No Periodics

When the periodic utilization equals zero, it is as if there are no periodics. In this case, the above formula is directly applicable.

4.2 Special Case of No Background

When the periodic utilization is at its highest level, there is no background available for the aperiodic events to exploit. In this case, all aperiodic processing must be performed within the budget of the sporadic server. To aperiodic events in the queue, it appears as if each event's service time is equal to the sporadic server's replenishment period. However, once an item starts executing, it only takes its actual execution time to complete.

This dichotomy between the queuing and executing perspectives leads us to our first refinement of the standard queuing formula. To reflect this dichotomy in the queuing formula, we denote the service time from the queuing perspective as S_q and from the server perspective as S_s.

This more general formula is

$$E[W]=\left(\frac{\rho_q}{1-\rho_q}\right)\left(\frac{E[S_q^2]}{2E[S_q]}\right)+E[S_s]$$

4.3 Special Case of Continuous Background

Our next challenge was to understand the "curves" between these two extremes. We chose to study what we have been calling the continuous background case. When the sporadic server has exhausted its budget, the only way for aperiodics to execute is in background. For a given periodic utilization, the smaller the period, the more continuously background is available. An infinitesimal period results in a continuous background.

This has given us a great deal of insight into the queuing theoretic structure of this problem. In this case, background processing is equivalent to being processed in a slower processor. For example, when $U_P = .5$, background processing in the continuous case is equivalent to executing in a CPU that is half the speed of a full processor. If S_a denotes the execution time of the aperiodic, the effective execution when executing in background is $S_a / (1-U_P)$.

This equivalence to a degraded processor is what makes this an interesting and illuminating special case. Even more interesting is the result that, from a queuing perspective, $S_a / (1-U_P)$ is the effective execution whether the aperiodic executes in the background or at a high priority (within the sporadic server). This means S_q in the previous formula is equal to $S_a / (1-U_P)$. This observation allows us to analytically predict queuing time in the continuous background case.

The benefit of the sporadic server comes from its impact on $E[S_s]$. Calculating this term requires a much more detailed analysis. The key insight for calculating $E[S_s]$ is that sometimes an aperiodic event is executed within the sporadic server, sometimes in the background and sometimes a combination of both.

To account for how much of each type of processing occurs, we needed to derive the probability density function for the "time to next replenishment," which determines how often each type contributes to $E[S_s]$. It turns out that the time to replenishment is almost uniformly distributed over the interval $[0, T_{ss}]$, where T_{ss} is the replenishment period. Given the density function for time to replenish, T_r, the formula below can be used to calculate $E[S_s]$.

$$E[S_s]=E[S_s \mid Tr=0]\Pr(Tr=0)+\int_0^{T_{ss}} E[S_s \mid Tr=t]\, f_{Tr}(t)dt$$

The first term handles when an aperiodic event is executed within the sporadic server; the second term handles the remaining cases. Figure 2 compares our analytic prediction with simulation results for time to replenishment.

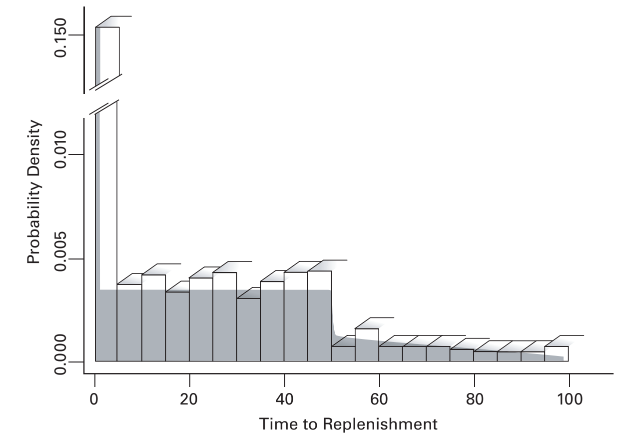


Figure 2: Analytic Predictions Versus Simulation Results

Qualitatively, the prediction appears to be quite good. The shaded bars are the predictions, and the 3-D bars are the simulation results.

5. Where Are We Going?

While our theoretical analysis is for a very special case, the combination of theoretical and empirical analysis gives us the capability of analyzing a broad class of assemblies. Our ongoing work involves generalizing the types of aperiodic streams we can analyze and exploiting a very recent development in queuing theory known as real-time queuing theory (RTQT). RTQT offers the ability to predict the probability of missing deadlines, which, in turn, allows for reasoning about deadlines in a stochastic setting. Our current capability augmented with the aforementioned enhancements will allow us to develop PECTs for an extremely broad spectrum of assemblies and a broad set of performance requirements.

Validating and Labeling the Objective Confidence of a Reasoning Framework

Gabriel Moreno, Paulo Merson,
Scott Hissam, Kurt Wallnau

1. Empirical Validation

We establish objective and quantified confidence in the quality of a reasoning framework's analytic theory through empirical validation. Validation does not yield a simple pass/fail result, however. Instead, it produces statistical evidence that can be used to assess the likelihood that predicted assembly behavior will correspond to observed assembly behavior. Our goal is to establish a uniform set of statistical models and labels for describing objective confidence in reasoning framework predictions.

This overview describes the empirical validation approach used on the λ^* reasoning frameworks. Section 2 gives an overview of statistical labels for components and reasoning frameworks. Section 3 describes an automated validation process that uses constrained random sample (assembly) generation.

2. Statistical Labels

Analytic theories (of reasoning frameworks) that predict quantifiable phenomena, such as time, space, and power, are called empirical theories. Validating such theories requires controlled observation and measurement.

Even the most accurate measurement instrumentation will introduce error that must be accounted for. Empirical theories may also abstract details, for example low-level caching mechanisms, that introduce marginal but observable (apparent) nondeterministic behavior. For these reasons, the results of component and assembly measurements are described not as discrete values, but in a statistical form, as probability density functions over a range of values. The effectiveness of empirical theories is likewise statistically described. We refer to such descriptors as *statistical labels.*

2.1 Component Labels

Empirical theories may depend on empirical properties of components. Two forms of statistical labels are used for components: property estimators and property intervals.

2.1.1 Estimator Labels

Ideally, the goal would be to know the *true value* of component execution time (for some operation of a component). However, true value is just a theoretical value defined as *the mean* (μ) *that would result from an infinite number of measurements of the same measurand carried out under repeatable conditions, assuming no systematic error.* Obviously, it is not possible to obtain true value. We must therefore use a statistical estimator of true value.

For example, we can take sample observations of X and use their average $\bar{x}$ as the estimator of μ. The uncertainty associated with this estimator is expressed as the standard deviation s such that the true value of μ will fall within some interval ($\bar{x} \pm k \cdot s$) with some specified confidence. The factor k is known as the coverage factor. Typically, we compute the 0.95 confidence interval (i.e., k = 2).

A confidence interval of component *execution time* is a fundamental measure for λ^* theories.

2.1.2 Interval Labels

Other kinds of component intervals are useful. A tolerance interval for component execution time contains a specified proportion ρ of all executions of a component. For instance, a tolerance interval with $\rho = 0.95$ might state that 95% of a component's traces will have execution time X ± 17 ms, where ±17 ms is the computed tolerance interval.

Also useful is the confidence interval on the probability of satisfying a specified property requirement. In this case, we specify the execution time interval and use it to compute the probability that any trace will fall within this interval. For example, we might specify the interval X ± 20 ms and use it to calculate a $\rho = 0.64$ probability that a given component trace will fall within this interval.

2.2 Analytic Theory Labels

Two forms of statistical labels are used for empirical theories: inferential and descriptive.

2.2.1 Inferential Labels

For empirical theories, we are usually interested in the magnitude of relative error (MRE) between predicted and observed values. For latency, we use $MRE = |\lambda - \lambda'| / \lambda'$, where λ is the predicted assembly latency and λ' is the measured latency.

We use confidence and tolerance intervals to characterize how effective an analytic theory is likely to be for future predictions. We use one-tail intervals (or bounds) rather than two-tail intervals, since we are generally only interested in knowing how frequently predictions are worse than the mean MRE for some sample.

Analogues to the confidence/tolerance intervals used for component properties are also used for analytic theories. A confidence interval describes the probability that the MRE for a particular prediction will lie within a specified MRE interval. Table 1 shows the tolerance interval for the λ_{ABA} member of λ^*.

Part of Interval	Description
N = 156	sample size
$\gamma = 0.9929$	confidence
$\rho = 0.80$	proportion
UB = 0.01	upper bound

Table 1: Statistical Label for λ_{ABA}

The interpretation of the λ_{ABA} label is that 80% of latency predictions have less than 1% MRE, and the interval is constructed in a way that gives 99% confidence in this upper bound.

2.2.2 Descriptive Labels

Linear correlation is a descriptive statistic that describes the strength of correlation between two data sets—for λ^*, predicted and observed assembly latency. Linear analysis yields the coefficient of determination R^2, $0 \leq R^2 \leq 1$, where 0 represents no relation and 1 represents a perfect linear relation. In a perfect prediction model, predicted and observed latency would be identical; therefore, the goal for the model builder is a linear relation.

For λ_{ABA}, we used Spearman rank correlation, which is a distribution-free linear correlation. The resulting $R^2 = 0.998$ means that the theory accounts for 99.8% of the variability in the actual latency.

3. Constrained Random Sampling

To construct a label for a property theory such as λ_{ABA}, we require a representative set of assemblies. But what is the precise definition of this set? And how do we choose a sample from this set?

The representative set of assemblies S_R of analytic theory T is *defined* as the set of assemblies that satisfies the constraints $X_T \cup X_C$, with analytic constraints X_T imposed by T and constructive constraints X_C imposed by component technology C. In practice, we are interested in *discovering* one of two subsets of S_R:

- $S_L \subseteq S_R$ of *limit* assemblies
- $S_A \subseteq S_L$ of *application* assemblies

We might discover S_L by subjecting a theory to destructive testing. Implementation details that have been abstracted by a theory might have negligible impact until certain extremes are encountered. For example, context-switching overhead may have negligible impact on predictions until the execution time of tasks approaches the scheduling quantum. These limits are made explicit as constraints X_L, and assemblies S_L will be made to satisfy $X_L \cup X_T \cup X_C$ by construction.

We are likely to be interested in only $S_A \subset S_L$ that is representative of the applications for which the theory will be used. For example, the controllers in an industrial robot may be a small subset of S_L, and this subset may be different from the controllers found in electric grid substation automation. We use judgment sampling to define these additional constraints X_A; S_A satisfies $X_A \cup X_L \cup X_T \cup X_C$. The constraints X_A are used for validation only; they are not enforced constructively.

For λ_{ABA}, domain experts in substation automation defined X_A to include the maximum number of event sources, components, and connections per component; the minimum and maximum load factors; the minimum and maximum periods; the ratio of synchronous to asynchronous interactions; and an indication of whether harmonic periods are allowed.

We use a library of components for synthetic execution from which we automatically generate a sample $S'_A \subset S_A$. A suite of tools controls how this sample is selected and generated. The tool suite and its associated measurement infrastructure are described in publicly available SEI reports.

Overview of the Pin Component Technology

Daniel Plakosh, Scott Hissam,
James Ivers, Kurt Wallnau

1. Introduction

At the heart of every prediction-enabled component technology (PECT) lies a kernel component technology. This kernel must be designed expressly to support predictability. In practice, this means strictly enforced patterns of component interaction and strictly defined runtime services and resource management policies.

A key element of the Starter Kit is Pin, a canonical, kernel, and adaptable component technology:

- canonical: Pin is a minimal component technology for predictable assembly. Removing any feature will reduce its effectiveness for embedded, safe, secure, time-critical systems or make reliable prediction difficult or impractical.
- kernel: The Pin runtime environment provides real-time services for concurrency, scheduling, and inter-component communication. It can execute as the highest priority task in Windows CE/2K and can be rehosted easily to “bare” hardware.
- adaptable: Pin has plug-in interfaces for alternative inter-component communication protocols and scheduling disciplines.

These characteristics combine with the minimal nature of Pin to enable a variety of application-specific customizations of Pin. We briefly describe the Pin component model in Section 2 and its runtime environment in Section 3.

2. The Pin Component Model

A component model defines the structure of components and their assemblies, and the coordination policies they use to interact. The Pin component model was designed with the following goals in mind:

- simple programming model. The programmer’s view of components and assemblies is kept simple to facilitate code generation and conformance checking.
- strict encapsulation. Implicit and hidden interactions are the mortal enemy of predictability. Pin reduces the chances that these interactions can be introduced.
- pure composition. A declarative style of component composition allows alternative coordination policies to be specified and enforced in a transparent and uniform way.

2.1 Simple Programming Model

Pin provides a standard component core that handles much of the complexity typically associated with component development. Among other things, the core manages the interaction of components with their environment. The unique functionality of a component (“Component Ops” in Figure 1) is a plug-in to the core that is invoked in response to messages delivered on the core’s external interface. A basic introspection mechanism is also provided (not shown) to allow components and their controllers (discussed in Section 3) a limited form of runtime configurability.

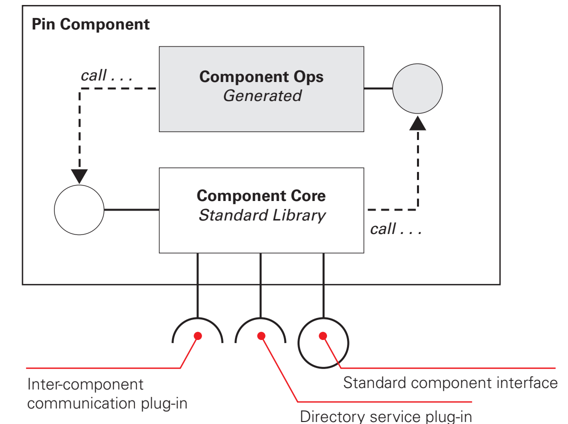


Figure 1: Module View of a Pin Component

Each interface depicted in Figure 1 is standard, and the implementation of Ops is sufficiently stereotypical that automated code generation from design specifications is straightforward. In fact, code generation is the usual and preferred way we develop Pin components, although, on occasion, we combine code generation and manual development. The code generator is designed with a mixed style of development in mind.

2.2 Strict Encapsulation

Pin imposes strict forms of encapsulation on component developers. Components may only interact with their environments through their Pin interface. As a first approximation (but only that), the Pin interface comprises a set of inbound (stimulus) and outbound (response) communication channels. There are no hidden runtime communication paths; all interactions of a component with the external world are expressed as Pin interactions.

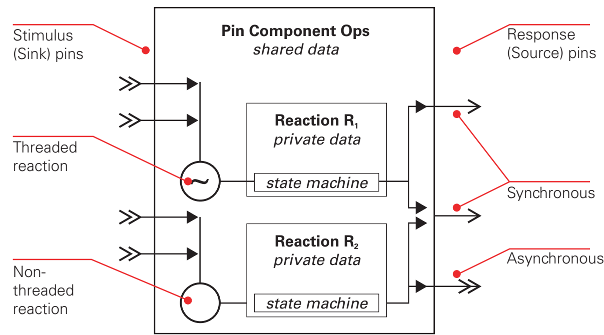


Figure 2: A Closer Look at Component Ops

Internally (see Figure 2), a component is structured as one or more reactions, each of which might or might not have its own thread of execution. The term *reaction* reflects the rule that components may not initiate activity on their own: they can only respond (or *react*) to environment-provided stimuli. Reactions accept stimuli from one or more *sink* pins; each sink pin serves exactly one reaction. Reactions produce responses on zero or more *source* pins; each source pin serves at least one reaction. Reactions may share component-scope data and can have their own reaction-scope data. Components may not share data (again, there are no communications beyond those expressed as interactions on pins).

2.3 Pure Composition

An intuition about pure composition in Pin may be obtained from Figure 3, taken from an application of Pin to an electric grid substation controller problem. The innermost boxes depict Pin components, while the outermost box depicts an assembly of components. Note that Pin imposes coordination and encapsulation schemes but does not impose a functional decomposition scheme. For example, the components depicted in Figure 3 conform to the Pin component model and also to the functional (domain) model defined by the IEC-61850 standard for substation automation systems.

Only components that have been composed may interact with one another. We say that composition is pure (or strict) if it can be achieved declaratively and without recourse to custom "glue" code. The form of declarative mechanism used in Pin is sometimes referred to as *connectors*, and composed components are said to be *connected*. Interaction among Pin components is enabled when the source pin of one component is connected to the sink pin of some other component. Connectors may impose topological constraints and pin-to-pin conformance constraints.

The value of pure composition is not iconic system integration, although that has its advantages. Even with Pin, custom "glue" code may sometimes be needed, but it will be encapsulated as a Pin component; this is an extra development step in comparison with non-pure composition.

Instead, the value of pure composition lies in predictability: Pin completely exposes the runtime interactions among components and those interactions between the components and the runtime environment. As a result, no interactions are hidden from reasoning frameworks. Moreover, exposing interaction as a first-class element in system specification provides analytic leverage. For example, it permits greater formalization of interaction policies that can be exploited by reasoning frameworks. Alternatively, reasoning frameworks can impose their own coordination policies—for example, priority queuing semantics which can then be uniformly implemented and enforced. These aspects of pure composition all serve the end goal of predictability by construction.

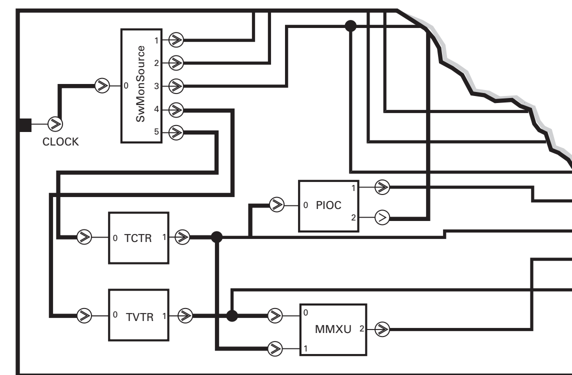


Figure 3: A Portion of a Pin IEC-61850 Switch Assembly

3. The Pin Runtime Environment

A component runtime environment provides an implementation of interaction mechanisms and other aspects of the component model. The Pin runtime has two main elements:

1. The component runtime implements the Pin component model.
2. The kernel runtime implements elements of a small-footprint real-time operating system (OS).

The component runtime provides a container for deployed assemblies: each deployed assembly executes in its own container called its *controller*. The controller is responsible for managing the life cycle of components in an assembly and for managing interactions with components in other, possibly distributed, assemblies. The controller provides support for error reporting and handling, and provides a primitive but customizable (through code generation) fault model. Lastly, the controller provides support for dynamic loading and unloading of components, although the current generation of Starter Kit reasoning frameworks assumes static topologies.

The kernel runtime provides the necessary primitives of a real-time OS for embedded applications. Unlike general OS kernels, the Pin kernel is tailor-made for component-based development. The kernel runtime supports real-time timers, threads, and synchronization; it also supports real-time message queues that have been specifically adapted to closely mirror the dynamic semantics of UML statecharts. As a result, the gap between component-level behavior specifications and their realization in the runtime environment is drastically reduced, making analysis and code generation much simpler than would otherwise be the case.

Selected Publications

Starting points

Hissam, Scott; Hudak, John; Ivers, James; Klein, Mark; Larsson, Magnus; Moreno, Gabriel; Northrop, Linda; Plakosh, Daniel; Stafford, Judith; Wallnau, Kurt; & Wood, William. *Predictable Assembly of Substation Automation Systems: An Experiment Report, Second Edition* (CMU/SEI-2002-TR-031, ADA418441). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. www.sei.cmu.edu/publications/documents/02.reports/02tr031.html

Hissam, S.; Klein, M.; Lehoczky, J.; Merson, P.; Moreno, G.; & Wallnau, K. *Performance Property Theories for Predictable Assembly from Certifiable Components (PACC)* (CMU/SEI-2004-TR-017). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. To be published.

Ivers, James & Sharygina, Natasha. *Overview of ComFoRT: A Model Checking Reasoning Framework* (CMU/SEI-2004-TN-018). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. www.sei.cmu.edu/publications/documents/04.reports/04tn018.html

Wallnau, K. *Volume III: A Technology for Predictable Assembly from Certifiable Components* (CMU/SEI-2003-TR-009, ADA413574). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. www.sei.cmu.edu/publications/documents/03.reports/03tr009.html

General material

Bachmann, F.; Bass, L.; Buhman, C.; Comella-Dorda, S.; Long, F.; Robert, J.; Seacord, R.; & Wallnau, K. *Volume II: Technical Concepts of Component-Based Software Engineering, Second Edition* (CMU/SEI-2000-TR-008, ADA379930). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. www.sei.cmu.edu/publications/documents/00.reports/00tr008.html

Hissam, S. & Ivers, J. *PECT Infrastructure: A Rough Sketch* (CMU/SEI-2002-TN-033, ADA413548). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. www.sei.cmu.edu/publications/documents/02.reports/02tn033.html

Hissam, S.; Moreno, G.; Stafford, J.; & Wallnau, K. "Packaging Predictable Assembly," 108-124. *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment (CD 2002)* (in Lecture Notes in Computer Science [LNCS] volume 2370). Berlin, Germany, June 20-21, 2002. Berlin, Germany: Springer-Verlag, 2002.

Moreno, G.; Hissam, S.; & Wallnau, K. "Statistical Models for Empirical Component Properties and Assembly-Level Property Predictions: Toward Standard Labeling." *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering.* Orlando, Florida, May 19-20, 2002. www.sei.cmu.edu/pacc/CBSE5/Moreno-cbse5-final.pdf

Stafford, J. & Hissam, S. *The Software Engineering Institute's Second Workshop on Predictable Assembly: Landscape of Compositional Predictability* (CMU/SEI-2003-TN-029, ADA418396). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. www.sei.cmu.edu/publications/documents/03.reports/03tn029.html

Stafford, J. & Wallnau, K. "Is Third Party Certification Necessary?" 13-17. *Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering.* Toronto, Canada, May 14-15, 2001. Los Alamitos, CA: IEEE Computer Society, 2001.

Wallnau, K. & Ivers, J. *Snapshot of CCL: A Language for Predictable Assembly* (CMU/SEI-2003-TN-025, ADA418453). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. www.sei.cmu.edu/publications/documents/03.reports/03tn025.html

Model checking reasoning framework material

Chaki, S.; Clarke, E.; Groce, A.; Ouaknine, J.; Strichman, D.; & Yorav, K. "Efficient Verification of Sequential and Concurrent C Programs." To be published in the journal *Formal Methods in System Design.*

Chaki, S.; Clarke, E.; Ouaknine, J.; & Sharygina, N. "Automated, Compositional and Iterative Deadlock Detection," 201-210. *Proceedings of Formal Methods and Models for Codesign (MEMOCODE '04).* San Diego, CA, June 23-25, 2004. Madison, WI: Omnipress, 2004.

Chaki, S.; Clarke, E.; Ouaknine, J.; Sharygina, N.; & Sinha, N. "State/Event-Based Software Model Checking," 128-147. *Integrated Formal Methods 4th International Conference (IFM 2004)* (in Lecture Notes in Computer Science [LNCS] volume 2999). Canterbury, UK, April 4-7, 2004. New York, NY: Springer-Verlag, 2004.

Clarke, E.; Grumberg, O.; & Peled, D. *Model Checking.* Cambridge, MA: MIT Press, 1999.

Ivers, J. & Wallnau, K. "Preserving Real Concurrency," 15-22. *Correctness of Model-Based Software Composition (CMC) Proceedings,* done in association with the 17th European Conference on Object-Oriented Programming (in Lecture Notes in Computer Science [LNCS] volume 2743). Darmstadt, Germany, July 21-25, 2003. New York, NY: Springer-Verlag, 2003. www.ubka.uni-karlsruhe.de/vvv/ira/2003/13/13.pdf

Performance reasoning framework material

Doytchinov, B.; Lehoczky, J.; & Shreve, S. "Real-Time Queues in Heavy Traffic with Earliest-Deadline-First Queue Discipline." *Annals of Applied Probability 11,* 2 (2001): 332-378.

Klein, M.; Ralya, T.; Pollak, B.; Obenza, R.; & Gonzalez Harbour, M. *A Practitioner's Handbook for Real-Time Analysis.* Boston, MA: Kluwer Academic Publishers, 1993.

Kleinrock, Leonard. *Queuing Systems, Volume 1: Theory.* New York, NY: John Wiley and Sons, 1975.

Sprunt, B. *Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System* (CMU/SEI-89-TR-011, ADA211344). Pittsburgh, PA. Software Engineering Institute, Carnegie Mellon University, 1989. www.sei.cmu.edu/publications/documents/89.reports/89.tr.011.html

For more information about PACC, go to
www.sei.cmu.edu/pacc/pacc_init.html

For more information about becoming a
PACC research partner, please contact

Linda Northrop
Director, Product Line Systems Program
Software Engineering Institute
4500 Fifth Avenue
Pittsburgh, PA 15213

Phone: 412-268-7638
Fax: 412-268-5758
Email: lmn@sei.cmu.edu

For more information about the
Software Engineering Institute, please contact

Customer Relations
Software Engineering Institute
4500 Fifth Avenue
Pittsburgh, PA 15213

Phone: 412-268-5800
Fax: 412-268-5758
Email: customer-relations@sei.cmu.edu
Web: www.sei.cmu.edu

The Software Engineering Institute (SEI) is operated by Carnegie Mellon University for the Department of Defense. As such, the following conditions apply:

Copyrights
Carnegie Mellon University SEI-authored documents are sponsored by the U.S. Department of Defense under Contract F19628-00-C-0003. Carnegie Mellon University retains a non-exclusive, royalty-free license to publish or reproduce these documents, or allow others to do so, for U.S. government purposes only pursuant to the copyright license under the contract clause at 252.227-7013.

Disclaimer of Endorsement
References in this publication to specific commercial products, processes, or services by trade name, trademark, manufacturer, or otherwise, do not necessarily constitute or imply their endorsement, recommendation, or favoring by Carnegie Mellon University or the U.S. government. The ideas and findings of authors expressed in any reports or other material should not be construed as an official Carnegie Mellon University or Department of Defense position and shall not be used for advertising or product-endorsement purposes. Information contained in this document is published in the interest of scientific and technical information exchange.

No Warranty
Any material furnished by Carnegie Mellon University and the Software Engineering Institute is furnished on an "as-is" basis. Carnegie Mellon University makes no warranties of any kind, either expressed or implied, as to any matter including, but not limited to, warranty of fitness for purpose or merchantability, exclusivity, or results obtained from use of the material. Carnegie Mellon University does not make any warranty of any kind with respect to freedom from patent, trademark, or copyright infringement.

Trademarks and Service Marks
Carnegie Mellon Software Engineering Institute (stylized), Carnegie Mellon Software Engineering Institute (and design), and the stylized hexagon are trademarks of Carnegie Mellon University.

® Architecture Tradeoff Analysis Method, ATAM, and Carnegie Mellon are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

SM Framework for Software Product Line Practice is a service mark of Carnegie Mellon University.

The Software Engineering Institute (SEI) is a federally funded research and development center sponsored by the U.S. Department of Defense and operated by Carnegie Mellon University.

Copyright 2004 by Carnegie Mellon University.

Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213-3890
Phone: 412-268-5800
Fax: 412-268-5758
www.sei.cmu.edu

Other Offices
Washington, DC Area
NRECA Building, Suite 902
4301 Wilson Boulevard
Arlington, VA 22203

SEI-Europe
An der Welle 4
60322 Frankfurt
Germany

U.S. Army Aviation
and Missile Command
AMSAM-BA Bldg 6263
Redstone Arsenal, AL 35898