March 2003

Enabling Predictable Assembly

Scott A. Hissam
*Software Engineering Institute*

Gabriel A. Moreno
*Software Engineering Institute*, gmoreno@sei.cmu.edu

Judith Stafford
*Software Engineering Institute*

Kurt C. Wallnau
*Software Engineering Institute*

---

Follow this and additional works at: [http://repository.cmu.edu/sei](http://repository.cmu.edu/sei)

Part of the [Software Engineering Commons](http://repository.cmu.edu/sei)

Published In

Demands for increased functionality, better quality, and faster time-to-market in software products continue to increase. Component-based development is the software industry's response to these demands. The industry has developed technologies such as EJB and CORBA to assemble components that are created in isolation. Component technologies available today allow designers to plug components together, but do little to allow the developer to reason about how well they will play together. Predictable Assembly focuses on issues related to assembling component-based systems that predictably meet their quality attribute requirements. This paper introduces prediction-enabled component technology (PECT) as a means of packaging predictable assembly as a deployable product. A PECT is the integration of a component technology with one or more analysis technologies. Analysis technologies support prediction of assembly properties and also identify required component properties and their

1. Contact Author
certifiable descriptions. This report describes the major structures of a PECT. It then discusses the means of validating the predictive powers of a PECT, which provides measurably bounded trust in design-time predictions. Last, it demonstrates the above concepts in an illustrative model problem: predicting average end-to-end latency of a ‘soft’ real time application built from off-the-shelf software components.
Scott A. Hissam is a senior member of the technical staff for the Software Engineering Institute at Carnegie Mellon University, where he conducts research on component-based software engineering and Open Source Software. He is also an adjunct faculty member of the University of Pittsburgh. His publications include one book, papers published in international journals including IEEE Internet Computing and Journal of Software Maintenance, and numerous technical reports published by CMU. Prior to his position at the SEI, Mr. Hissam held positions at Lockheed Martin, Bell Atlantic, and the US Department of Defense. He has a Bachelor of Science degree in Computer Science from West Virginia University.

Gabriel A. Moreno received the BS degree (honors) in computing systems from University of Mendoza, Argentina, and the Master of Software Engineering degree from Carnegie Mellon University. He is a visiting scientist at Carnegie Mellon University's Software Engineering Institute and a Fulbright Fellow. Previously, he was at ITC Soluciones, Argentina, where he designed and developed multiplatform distributed systems and communication protocols for electronic transactions. His current research interests include predictable assembly, component based software, and software architectures.

Judith Stafford is a senior member of the technical staff at the Software Engineering Institute, Carnegie Mellon University. Dr. Stafford has worked for several years in the area of compositional reasoning and its application to software architectures with an emphasis on the use of software architecture as a foundation for early analysis of software systems. Her current research interests include prediction of the behavior of systems composed of software components, compositional reliability analysis, and software architecture documentation.

Kurt Wallnau is a senior member of the technical staff at the Software Engineering Institute (SEI) at Carnegie Mellon University. Mr. Wallnau currently leads the SEI predictable assembly from certifiable components project. Prior to that he led SEI work in the area of commercial-off-the-shelf software, described in the Addison-Wesley book, "Building Systems from Commercial Components."
1 Introduction

Component-based development is the software industry's response to the demand to build better software faster and cheaper. The software industry has developed technologies such as EJB and CORBA to assemble components that were created in isolation. Traditional software development methods focus on construction from custom components developed and maintained within the control of the organization, however, trends in software development point to increased reliance on pre-existing components over which the developer has little, if any, control. When developers compose systems from such components they must be aware of the assumptions the components make about the environment into which they are to be deployed and assumptions the environment makes about components that are deployed into it. When these assumptions are hidden, unexpected integration difficulties result. Component technologies available today, such as EJB, CORBA, COM, and .NET, allow designers to plug components together, but provide little support for reasoning about how well they will play together. The Predictable Assembly from Certifiable Components (PACC) project at the Software Engineering Institute is exploring various approaches to assembling component-based systems that predictably meet their quality requirements. This paper reports on our experience in creating a prototype prediction-enabled component technology (PECT) that is capable of predicting the latency of a component assembly based on the measured latency of individual constituent components. PECT is both a technology and a method for producing instances of the technology. A PECT instance results from integrating a software component technology with one or more analysis technologies. PECT supports predictable assembly from certifiable components. By predictable assembly, we mean:

- Assemblies of components are known, by construction, to be amenable to one or more analysis methods for predicting their emergent properties.
- The component properties that are required to make these predictions are defined, available, and possibly certified by trusted third parties.

The underlying premise of PECT is that, while it may be impossible to analyze, and thereby predict, the runtime behavior of arbitrary designs, it is possible to restrict designs to a subset that is analyzable. This
premise has already been seen in the use of logical analysis and prediction (Finkbeiner and Kruger, 2001) (Sharygina et al., 2001). It is a further premise of PECT that software component technology is an effective way of packaging the design and implementation restrictions that yield analyzable designs.

This report describes and illustrates the development of a PECT, explores the strengths and limitations of this approach to predictable assembly, and charts a course for further applied research. Section 2 provides background material and pointers to related work. Section 3 presents an overview of the process for developing a PECT. Section 4 describes the component and attribute prediction technologies we used to develop the PECT prototype, COMTEK-λ. Section 5 describes this prototype and its validation. In Section 6, we summarize the key results and questions raised by the prototype.

2 Background

Development of prediction-enabled component technologies leverages development efforts in the areas of software components, software architecture, compositional reasoning, and component certification.

2.1 Component Technology

Naive approaches to system integration are bound to fail. Garlan et al. coined the term “architectural mismatch” (Garlan et al., 1995) to describe the result of integrating components without knowledge of the assumptions the component makes about the environment into which they will be deployed and the assumptions the environment makes about components deployed into it. Component technologies such as EJB, CORBA, COM, and .NET have been developed to support component-based development. Commercial development environments such as WebGain Studio¹, ILOG JConfigurator², and Microsoft Visual Studio³ that are based on these technologies have come on the market to support rapid application development. While, these environments generally provide some support for analysis of assemblies, they do not support prediction of assembly quality before component acquisition.

2.2 Software Architecture

There is a natural affinity between software architecture and software component technology, which is expressed in several ways. First, and most obviously, is the central role of components and connectors as abstraction. While it is true that the levels of abstraction are quite distinct, the kinds of thing being abstracted are quite similar. Second, the correlation of architectural style, as an abstraction of design constraints, and component models and frameworks has been noted elsewhere (Bachmann et al., 2000) (Baggiolini et al., 1998). For example, a component model defines style-specific interfaces that are imposed on components, while a framework provides the run-time mechanisms to implement a style's connectors. Last, as already noted, software architecture and software component technology have, to date, focused on complementary agendas: enabling reasoning about quality attributes, and simplifying component integration, respectively.

2.3 Compositional Reasoning

Compositional reasoning techniques provide a foundation for reasoning about system completeness and correctness early in the development process. To date, research in the area has focused primarily on the use of abstract state models and architecture description languages (ADLs) as a substrate for analysis algorithms. Model checking has been applied to a variety of model types in order to determine whether a system possesses a given property (Fisler et al., 2001) (Sharygina et al., 2001). The analysis algorithms that have been developed for ADLs have, in general, focused on correctness properties, such as liveness and safety (Allen and Garlan, 1997) (Magee et al., 1997) (Naumovich et al., 1997); however, other types of analysis are also appropriate for use at the architecture level and are currently the focus of research projects. Examples include system understanding (Kramer and Magee, 1997) (Stafford and Wolf, 2001) (Zhao, 1997), performance analysis (Balsamo et al., 1998) (Spitznagel and Garlan, 1998) and architecture-based testing (Bachmann et al., 2000) (Vieria et al., 2000). More closely related to our work toward adapting component reliability measures for use to reason compositionally about the reliability of component assemblies (Hamlet et al., 2001). In our work, the relationship between analysis techniques and constraints on interactions is used to co-refine analytic and constructive models to produce a PECT.
2.4 Component Certification

The National Security Agency (NSA) and the National Institute of Standards and Technology (NIST) used the trusted computer security evaluation criteria (TCSEC), a.k.a. “Orange Book\(^1\)” as the basis for the Common Criteria\(^2\), which defines criteria for certifying security features of components. Their effort was not crowned with success, at least in part because it defined no means of composing criteria (features) across classes of component. The Trusted Components Initiative (TCI)\(^3\) is a loose affiliation of researchers with a shared heritage in formal specification of interfaces. Representative of TCI is the use of pre/post conditions on APIs (Meyer, 1997). This approach does support compositional reasoning, but only about a restricted set of behavioral properties of assemblies. Quality attributes, such as security, performance, availability, and so forth, are beyond the reach of these assertion languages. Voas has defined rigorous mathematical models of component reliability based on statistical approaches to testing (Voas and Payne, 2000), but has not defined models of composing reliability measures. Commercial component vendors are not inclined to formally specify their component interfaces, and it is not certain that it would be cost effective for them to do so. Mary Shaw observed that many features of commercial components will be discovered only through use. She proposed component credentials as an open-ended, property-based interface specification (Shaw, 1996). A credential is a triple <attribute, value, knowledge> that asserts that a component has an attribute of a particular value, and that this value is known through some means. Credentials reflect the need to address component complexity, incomplete knowledge, and levels of confidence (or trust) in what is known about component properties, but do not go beyond notational concepts. Therefore, despite many efforts, fundamental questions remain. What does it mean to trust a component? Still more fundamental: what ends are served by certifying (or developing trust) in these properties?

3 Elements of PECT

PECT integrates software component technologies (hereafter, ‘component technologies’) with analysis and prediction technologies (hereafter, ‘analysis technologies’). Component technologies impose design constraints on component suppliers and integrators. These constraints are expressed as a component model that specifies required component interfaces and other development rules (Bachmann et al., 2000) (Heinemann and Councill, 2001). In today’s component technology, component models are designed to simplify many aspects of the integration of components into assemblies. However, the stress is on the syntactic aspects of composition. Behavioral composition is not usually addressed, and where it is addressed, it is usually restricted to rely-guarantee reasoning with pre/post-conditions on operations. While rely-guarantee reasoning can be quite useful for reasoning about correctness, it is not particularly useful for reasoning about other assembly-level properties such as performance, reliability, and security.

Analysis technologies, for example performance (Klein et al., 1993) and reliability (Lyu, 1996), depend on runtime assumptions concerning scheduling policy, process or thread priority, concurrency, resource management policies, and many other factors. A PECT makes these analytic assumptions explicit. We ensure that a component technology satisfies these assumptions through a demonstration of ‘theoretical validity.’ We also ensure that predictions based on an analysis technology are repeatable through a demonstration of ‘empirical validity.’ These validations provide bounded confidence that a collection of design constraints on component suppliers and system integrators will yield systems that are, by design and construction, predictable with respect to one or more critical system properties.

A PECT is an association of a component technology with one or more analysis technologies. ‘Association validity’ stipulates that each such association is validated. ‘Assumption’ and ‘interpretation’, taken together, demonstrate theoretical validity. Each of these forms of validity is described in Section 3.4 and discussed in detail in (Hissam et al., 2001). In principle, component technology and prediction technology can each be treated as separately packaged entities—we will not yet go so far as to call them “components” in their own right\(^1\). Where both are separately packaged, an N:M association between component and anal-
ysis technologies would be reasonable. However, our perspective is centered on component technology, and how these can be extended to predict emergent properties.

3.1 Component Technology

While there is no iron-clad definition of ‘component technology’ any more than there is for ‘component,’ a consensus has emerged regarding the essential elements of a component technology (Bachmann et al., 2000) (Heineman and Councill, 2001) (Szyperski, 1997):

- A **component model** defines one or more required component interfaces, allowable patterns of interactions among components, interactional behaviors among components and between components and the component runtime, and, possibly, a programming model for component developers.

- A **component runtime environment** provides runtime enforcement of the component model. The runtime plays a role analogous to that of an operating system\(^1\) only at a much higher level of abstraction, one that is usually tailored to an application domain or required assembly properties (e.g., performance or security).

- An **assembly environment** provides services for component development, deployment, and application assembly. The assembly environment may also provide assembly-time enforcement of the component model.

Each of the elements listed above plays a role in PECT. The component model is the locus of the integration of component and analysis technologies; it specifies the design and implementation constraints that are required to enable predictable assembly. The runtime and assembly environments are important insofar as they enforce at least some of these constraints. The runtime environment is itself a target of certification, as it may always be treated as a component in its own right, with properties that contribute to the prediction of emergent properties.

---

1. There is, however, an interesting analog between the context dependencies that are included in Szyperski’s base definition of software component (Szyperski, 1997), and the assumptions of analysis technologies.

1. In fact, the various COM-based Microsoft component models are an integral part of the Microsoft operating systems.
In addition to the models described above we introduce the concept of an *assembly model*, as a refinement of the traditional component model. The assembly model plays the same role as component model, and may indeed describe many of the same things. There are two reasons for introducing this refinement. First, a single component technology may be restricted or generalized in different ways for different analysis technologies. It therefore makes sense to isolate those changes that are particular to a prediction technology. More fundamental, though, the refinement allows us to distinguish between constructive and analytic interfaces.

The *constructive* interface includes those properties of a component that permit it to interact with other components. It also includes such things as the traditional application programming interface (API). The constructive interface corresponds closely to the typical component model. The *analytic interface* includes those component properties that are required by an analysis technology, including things such as performance measures, state transition models, and process equations. This interface does not correspond to any existing component model.

### 3.2 Analysis Technology

There are many analysis technologies available to software engineers. However, these technologies have not been developed with the objective of being integrated with software component technology. As a result it is, in many cases, difficult to distinguish between an analysis technology and an underlying strategy for optimizing a system with respect to a particular (analyzed) attribute. That is, analysis technology and architectural design pattern (sometimes called a “style” (Shaw and Clements, 1997)) are often conflated. For example, Simplex is an architectural design pattern that optimizes for fault tolerant system behavior during replacement of critical control functions (Sha et al., 1995). A formal definition of Simplex has been used to prove (the strongest form of prediction) a number of properties (Rivera et al., 1996). However, the link between these proofs and the design pattern is at best implicit, and there is no generalization of these predictions over structurally related patterns. The work of Klein and others on quality attribute design patterns...
(Klein and Kazman, 1999) (Bass et al., 2000) offers some clues as to how analysis models and their contingent design patterns may be disentangled; this is one starting point for PECT research.

An ‘analytic model’ is a distillate of an analysis technology. It defines the property theory that underlies the analysis technology. It also defines the parameters of this theory, e.g., the properties that components must possess. For example, a property theory that can be used to predict assembly-level deadlock or safety properties might require component-level process equations to describe their concurrent behavior in an assembly. Such equations would be constituents of the analytic interface of a component.

It is customary to think of component types as being defined by one or more interfaces. In Enterprise JavaBeans, SessionBean and EntityBean are component types defined by distinct interfaces. Since these interfaces define properties that govern how components are integrated, we consider them as part of the constructive interface. Naturally, components may implement several constructive interfaces. Such components are therefore polymorphic in that they satisfy more than one constructive type definition. Components that satisfy a constructive interface are called constructive components. The analytic model may also introduce one or more analysis-specific component types, and components may likewise be polymorphic with respect to these type definitions. Such components are called analytic components.

An assembly of constructive components is called a constructive assembly. Analogously, an assembly of analytic components is an analytic assembly. The mapping from a constructive assembly to an analytic assembly is called the analytic interpretation of that constructive assembly. In effect, we consider that the analysis model defines an analysis-specific view of an assembly. The analytic interpretation defines how these views are instantiated for any given assembly.

### 3.3 Integration Co-Refinement

There are many available component and analysis technologies in research and in the commercial marketplace. An important practical consideration for our research is to demonstrate that existing technologies can be integrated into viable PECT instances. However, since component and analysis technologies have developed independently, and to satisfy different objectives, their integration may not always be straight-
forward due to mismatched assumptions. Where mismatches arise, either or both must be adjusted, as illustrated in Figure 1.

The effect of making a component technology more specific is to make assemblies more uniform in structure, but at the cost of further constraining the freedom of component developers and system assemblers. Analogously, making an analysis technology more detailed might make its predictions more accurate, but might also increase the cost of applying the technology. Figure 1 depicts three (non-exhaustive) alternative ways of integrating a component technology with an analysis technology; each alternative reflects the above trade-off:

1. PECT-1 shows an integration of component and analysis technologies that require a weakening of constraints on both. The effect of this trade-off might be to increase the population of designs that admit analysis and prediction, but at the cost of making the analysis and hence predictions more abstract and less accurate. In general, there is no way to know whether a generalization or restriction of an analysis technology will result in an enlarged or diminished scope, or enhanced or degraded accuracy. We can say, however, that a restriction on the component technology results in a smaller population of allowable designs.

2. PECT-2 shows an integration where the component technology remains unaffected but the prediction technology is made more specific. This tradeoff might reflect the specialization of a prediction technology to an existing component technology, or to the need for increased accuracy of predictions.

3. PECT-3 shows an integration where both technologies are constrained. The net effect in this case is to restrict the population of designs that admit analysis and prediction and, possibly, to improve the accuracy of the resulting predictions.

We refer to the integration process implied by Figure 1 as ‘co-refinement’ since either or both technologies may be refined (we include generalization and abstraction in our admittedly colloquial use of this term) to enable the integration of component and analysis technologies.
3.4 PECT Validation

The consumers of a PECT will want to know, in advance of using it, how much confidence to place in the predictive powers of the technology. That is, can the PECT be trusted? A technology that purports to enable predictable assembly would be meaningless if its predictions could not be validated. To paraphrase the wisdom of Wittgenstein for use in our own context: *A nothing will do as well as a something (that is, a prediction) about which nothing can be said.*

Theoretical and empirical validity must be established to engender bounded, quantifiable trust in a PECT:

- **Empirical validity** establishes measurable (and most likely statistical) evidence of the reliability of predictions made using the analysis technology. All analysis technologies must be falsifiable with respect to their predictions. This is a strong condition that rules out a variety of “soft” attributes that are defined using subjective and non-repeatable measures.

- **Assumption (theoretical) validity** establishes that the analytic model is sound, and that all of the assumptions that underlie it are satisfied either by the component technology in part or as a whole, or by engineering practices external to the component technology.

- **Interpretation (theoretical) validity** establishes that each constructive assembly has at least one counterpart analytic assembly, and that if more than one such counterpart exists, the set of such counterparts is an equivalence class with respect to predictions.

All three forms of validation are essential, but we place special emphasis on empirical validity. Like Simon, we accept the utility of predictive models even if their assumptions are falsifiable with respect to the inner workings of the systems under scrutiny, so long as the predictions are consistently accurate and useful with respect to observed phenomena (Simon, 1996). We also observe that software engineering literature is notoriously weak with respect to empirical validation of design theories. With PECT, we stake a position that opposes this continuing trend.
4 COMTEK and Latency Prediction

The PECT prototype combines the COMTEK component technology with a property theory for latency prediction. We briefly describe both as background to the PECT prototype.

4.1 COMTEK Component Technology

COMTEK\textsuperscript{1} was developed by the SEI for the U.S. Environmental Protection Agency (EPA) Department of Water Quality. Water quality analysis is computationally expensive, and in many cases, requires the use of simulation and iterative equation solvers. COMTEK was a proof of feasibility that third-party simulation components could be fully compositional, and could produce reliable and scalable water quality simulations.

COMTEK has the following high-level characteristics:

- It enforces a typed pipe-and-filter architectural style.
- A fixed round-robin schedule is calculated from component input/output dependencies.
- The execution of an assembly is sequential, single-threaded, and non-preemptive.
- It runs under the Microsoft Windows family of operating systems.
- Components are packaged and deployed as Microsoft Dynamic Link Libraries (DLLs).

Despite its simplicity, the generality of COMTEK was demonstrated in both the audio and the hydraulic application domains. Components are chosen from one or more component families, depending on the application.

Figure 2 presents a screenshot of the COMTEK assembly environment. The graphic depicts an assembly built from components of the Wave family. This and similar assemblies are the subject of the PECT demonstration. These assemblies implement audio signal sampling, manipulation, and playback functionality.

\footnote{COMTEK was originally called ‘WaterBeans.’ We have elected to rename WaterBeans because our scope is far broader than the domain of water quality modeling, and because we wish to avoid confusion between the original work in simulating water quality and our current work in predictable assembly.}
We chose to develop a PECT for composing audio playback applications because of the simplicity of the COMTEK scheduler, which only required using a simple performance analysis model.

4.2 Predicting the Latency of COMTEK Assemblies

The audio playback application lies in the domain of what is sometimes referred to as ‘soft real-time’ applications. In soft real-time applications, timely handling of events or other data is a critical element of the application, but an occasionally missed deadline is tolerable. In the audio playback application, audio signals received from an internal CD player must be sampled at regular intervals—approximately every 46 milliseconds for each 1,024 bytes of audio data. A failure to sample the input buffer, or to feed the output buffer (i.e., the speakers) within this time interval will result in a lost signal. Too many lost signals will disrupt the quality of the audio playback; however, a few lost signals will not be noticeable to the untrained ear. Thus, audio playback has ‘soft’ real-time requirements.

The problem we posed for PECT was to predict the end-to-end latency of an assembly of COMTEK components, where latency is defined as the time interval beginning with the execution of the ‘first’ component executed in an assembly and ending with the return from the ‘last’ component in that assembly (in a round-robin schedule, the notions of ‘first’ and ‘last’ are relative, but we assume there is some designated ‘first’ and ‘last’). This will allow engineers to predict whether a particular assembly will satisfy its performance requirements prior to its integration, and possibly, prior to acquiring the components. Such predictions must be made despite the fact that the Windows platforms we used make no performance guarantees.

It is not possible to know, thus to model, all environmental factors that have potential to affect latency analysis. What is important is to quantify what is required and determine the degree to which that requirement can be satisfied. Our emphasis in this work was on developing a model for PECTs rather than in solving the latency prediction problem. Thus we would be satisfied to achieve prediction to within 10% of observed assembly latency—good enough to demonstrate the PECT concept, even if insufficient for real engineering practice. As will be seen, however, we did much better than a 10% margin of error, although this was never a goal of the prototype.
5 Illustration

This section describes the enabling of COMTEK to support latency analysis and prediction. We refer to the resulting integration as COMTEK-λ.

5.1 Assembly Model

As explained in Section 3.1, an assembly model defines the set of (constructive and analytic) component types recognized by a component technology, and also specifies rules for their composition into assemblies that can be analyzed with one or more analytic models. We adapted the original COMTEK specification to reflect the additional requirements of latency analysis of COMTEK-λ, and to explore how the requisite aspects of the constructive and analytic interfaces might be specified.

A specification of COMTEK can be found in (Plakosh et al., 1999). This specifies, among other things, the interface that components must implement, the types of properties allowed, the data structures used to support introspection, and the ways to transfer data between components. We suggest three views of the assembly model: the component metatype, interaction rules, and dynamic behavior of COMTEK-λ assemblies. Each is described briefly below. Detailed descriptions can be found in (Hissam et al., 2001).

5.1.1 Component Metatype Specification

The component metatype defines interfaces and other rules that components must satisfy, that is, what it means to be a COMTEK component. This includes interface, pre- and post-conditions, invariants, and packaging. Note that we include packaging in this list because this specification must describe the component types as deployable units. Therefore, the specification of the binary form a component must take, for example a dynamic link library (DLL) or a Java archive (JAR), is also part of the constructive model.

---

1. We modify the name of the base component technology with an attribute designator that indicates the types of analyses that are enabled by the PECT. The form of a PECT name is as follows:

\[
\text{component technology designator-attribute theory designator}^* \\
\]

More than one attribute theory might be used with the same component technology simultaneously, and thus a string of one or more attribute theory designators is used. We use Greek letters to designate attribute theories, and \(\lambda\) for the latency theory.
COMTEK-λ deployable units are component factories—they provide runtime instances of themselves via their `getNewInstance()` method. For example, let us assume that we have component type whose name is ‘WaveView.’ Then, ‘WaveView’ is a component that can be deployed, and can provide instances of itself. The metatype distinguishes between constructive and analytic interfaces. Input and output ports and a set of properties constitute the constructive interface. The analytic interface for COMTEK-λ components consists of two mandatory analytic properties, ‘p’ and ‘e,’ whose meanings are described in Section 5.2.2.

5.1.2 Interaction Rules
The interaction model describes rules for composing components into assemblies. It defines the way that components can be connected, and specifies invariants for the assembly: how components are linked together and the constraints that must be observed when composing components. A component cannot connect to itself and the data types of the output and input ports must be the same. The COMTEK-λ runtime environment will not execute the application unless it is fully connected.

5.1.3 Assembly Behavior
The assembly model may also detail runtime aspects of the component technology, such as when and how instances are created and initialized, scheduling, and data transferred among components. This information might be vital not only for implementing the component runtime environment, but also for constructing a PECT. For latency prediction, there are several questions that must be answered:

- How are components scheduled?
- Can components be preempted?
- Do components block on resources?
- Can components have different priorities?
- Can components be multi-threaded?
The above list is not exhaustive and will, of course, vary from analysis model to analysis model. This kind of information is also important for component developers and application assemblers. For instance, a component technology that executes components in a multi-threaded environment might require the component developer to synchronize accesses to shared resources. Figure 3 shows a runtime view of the assembly model. In this model, ‘A’ is an assembly of components; ‘p(c)’, where c is a component such that \( c \in A \), is the number of times c has been executed; ‘\( \text{getTime()} \)’ is a function that returns the current value of the system clock. The model shows that components are executed once per cycle. There are other runtime details that we have abstracted, such as the order in which components are executed. Due to constraints imposed by the COMTEK scheduler, execution order is not required to predict end-to-end latency of a COMTEK-\( \lambda \) assembly in steady state. However, this information might be needed in other circumstances.

Also defined in Figure 3 are the runtime interpretations of assembly latency (‘A.latency’) and component latency (\( c_j . \text{latency} \)). These definitions serve two purposes. First, they give (reasonably) unambiguous model definitions of latency in the context of a particular component technology. Second, they describe how the properties will be measured for empirical validation.

### 5.2 Analytic Model

The analytic model defines a property theory used to predict the latency of an assembly of components based on their measured properties. In addition, the analytic model also exposes the assumptions that must be satisfied for the analysis to be valid.

#### 5.2.1 The Property Theory

The COMTEK-\( \lambda \) latency theory, denoted as \( A_\lambda \), is summarized by Equation 1. \( A . \text{latency} \) is the end-to-end latency of an assembly. We denote an assembly as the set of components A, and the \( k^{\text{th}} \) component of A is denoted as either or \( \Theta_k \) or \( \Phi_k \). \( \Theta \) and \( \Phi \) correspond to one of two analytic component types: \( \Theta \) refers to components that only have dependencies that are internal to A, while \( \Phi \) refers to components that also exhibit dependencies on external periodic events. These component types only have meaning within an \( A_\lambda \).
interpretation, i.e., within an analytic assembly and their symbols were chosen for their graphical mnemonic value.

\[
A\text{. latency} = \max \left( \left( \sum_{\Phi_j \in A} \Phi_j.e + \sum_{\Theta_j \in A} \Theta_j.e \right), \max\left( \{ \Phi_j \in A \mid \Phi_j.p \} \right) \right)
\]

Eq. 1

A property of a component or assembly is denoted using ‘dot’ notation. Each Φ component has two required properties that describe its latency information: ‘Φ.e’ and ‘Φ.p,’ while each Θ component has only the required property Θ.e, where e and p are defined as follows (also, refer to Figure 3):

- e: is the execution time of a component, exclusive of component blocking time.
- p: is the period of the external event on which a Φ depends and may block.

The function \( \max \) returns the largest of its arguments.

### 5.2.2 Adapting COMTEK for Latency Prediction

For reasons of expediency, we chose to derive \( A_{\Lambda} \) from COMTEK rather than begin with a more robust performance theory such as rate monotonic analysis (RMA). One of our goals was to modify COMTEK as little as possible, and derivation was a straightforward way of achieving this goal. We were not interested in re-engineering or extending COMTEK to address a realistic spectrum of soft real-time analysis issues.

Nonetheless, we found it necessary to impose new constraints on the use of COMTEK to support latency prediction. These constraints are above and beyond those that are imposed by COMTEK itself. In particular:

1. The value of ‘e’ for all components is constant over all executions of that component.
2. The value of ‘p’ is likewise constant for each external periodic event.

One question posed by these constraints is whether they should be documented as part of the constructive interface or analytic interface. On the one hand, they are meaningful only within the context of \( A_{\Lambda} \), and are therefore arguably part of the analytic interface of PECT-\( \lambda \). On the other hand, it is conceivable that such
constraints might be enforced by a component technology, which would argue that they belong to the constructive interface.

In this illustration, neither constraint can easily be enforced by the design or runtime environment; they must be enforced by engineering processes such as code inspection. We therefore assign them to the analytic interface. This is, perhaps, a minor point whose resolution will surface with greater experience with PECT.

5.3 Association Validity

PECT will be of little value unless its users (application designers and engineers) trust the predictions. Trust is a complex social phenomenon that balances many factors, only some of which can be addressed by technology. Nonetheless, we must provide a technical foundation for trust so that other non-technical factors can be addressed. This is done by validating the associations between a component technology and one or more prediction technologies. We refer to this in as association validity.

Association validity takes two forms: theoretical and empirical validity. Theoretical validity is concerned with the soundness of the analysis model and the way this model is integrated with a component technology. Empirical validity is concerned with the reliability and accuracy of the predictions made using the PECT. The following discussion provides a thumbnail sketch of how association validity was established for COMTEK-λ. A detailed exposition of both empirical and theoretical validity is provided in (Hissam et al., 2001).

5.3.1 Theoretical Validity

How does one go about establishing the validity of a scientific theory? In an important sense, the ultimate arbiter of theoretical validity lies in predicting phenomena that are observed under experimental conditions; this validation corresponds to what we refer to as empirical validity.

Often, though, empirical validity is not of itself sufficient, and it is usually not the starting point for establishing the validity of a theory. This is certainly true of the established physical sciences, where a theory
will undergo extensive scrutiny before it is tested experimentally. We believe this should also be true of the theories underlying software engineering practice.

There are two key questions that must be asked prior to investing the time and effort required to empirically validate a PECT:

1. Is the property theory sound?
2. Can the theory be falsified?

We consider the first question to lie in the province of theoretical validation, while the second question lies in the province of empirical validation, although in the strict sense no theory can be validated but can rather only be falsified. Assumption validity establishes that the mathematics used to describe the theory are sound, and that if the theory purports to describe causality (not strictly necessary in a property theory), there is a clear link between theory elements and mechanisms in the underlying software system. Interpretation validity establishes that each constructive assembly has an interpretation in the theory. This validation addresses the two key questions by demonstrating the relationship between constructive assemblies and model theoretic assemblies, and by providing a basis for theory falsification. We briefly describe how we demonstrated these forms of validity before turning to the question of empirical validation.

**Assumption Validity**

As we noted earlier, $A_{\Lambda}$ was derived from COMTEK. That is, the latency theory $A_{\Lambda}$ emerged from a detailed understanding of the mechanisms that manifest the property—the COMTEK runtime environment. Demonstrating association validity therefore reduced to demonstrating the validity of this derivation. A demonstration of association validity has the dual effect of demonstrating the mathematical soundness of the derivation, and highlighting those qualities of COMTEK that $A_{\Lambda}$ depends upon, i.e., its assumptions.

To demonstrate COMTEK-λ association validity, we began itemizing aspects of COMTEK that the assumptions of $A_{\Lambda}$ would likely depend upon, (e.g., scheduling policy, concurrency policy, how components and assemblies are defined, what it means for an assembly to be in steady state), and then assigned
names to and carefully defined these aspects of COMTEK, and then from these definitions we proceeded in stepwise fashion to derive $A_{\Lambda}$. The assembly model description in Section 5.1 contains this set of names and definitions.

Of itself, assumption validity is not particularly useful for generating trust. In fact, our first latency theory was also derived from COMTEK, but proved to be inadequate during empirical validation. We had, in effect, missed a key phenomenon of COMTEK assemblies: that components might block on periodic events external to COMTEK and its assemblies. However, assumption validity can significantly enhance trust when used in conjunction with empirical validation. In that situation, effective predictions are combined with an explanation of why the prediction theory holds.

**Interpretation Validity**

Recall that an analytic assembly is an interpretation, or mapping, of a constructive assembly under some property theory. For COMTEK-$\lambda$, interpretation validity results from demonstrating that this mapping is both complete and consistent:

- By complete, we mean that all constructive assemblies can be interpreted under the property theory. Note that this does not refer to the completeness of a property theory with respect to its assumptions about a component technology. It is the task of empirical validation to ferret out such missing assumptions.
- By consistent, we mean that all interpretations of a particular constructive assembly will result in the same prediction. Consistency is only an issue if there are several valid interpretations of a constructive assembly under a property theory. Consistency means that all such interpretations form an equivalence class with respect to the property theory.

Demonstrating the completeness of $A_{\Lambda}$ was trivial, since there was a complete mapping of constructive component types to analytic component types, and since interactions among constructive components were not parameters of $A_{\Lambda}$ (see Eq. 1 in Section 5.2.1 on page 18), and hence need not be considered. Demonstrating the consistency of $A_{\Lambda}$ was likewise trivial. However, had $A_{\Lambda}$ not been specialized to deal only with
steady state latency, execution order would have been significant (in non steady-state). In this case, demonstrating completeness and consistency would have been more involved.

5.3.2 Empirical Validity

Establishing empirical validity consists of demonstrating that the predictions made using an analytic model conform to observations. Thus, our confidence in the quality of a PECT is limited by the stability of measured assembly-level properties and their comparisons to predicted assembly-level properties. Our confidence in predictions is also bounded by our confidence in the measurements of component properties that parameterize property theories. It should not be surprising, then, that empirical validity rests on a foundation of measures and measurement. Nor should it be surprising that statistical analysis plays an important role in establishing empirical validity. All measurement processes introduce error, and the abstraction of complex phenomena into property theories invariably introduces additional error. In addition, the intervals in which component properties are reported introduce uncertainty that affects the precision with which we can predict assembly properties. We must use statistics to quantify each of these.

Empirical Validation and Formal Property Theories

Before continuing with a discussion of the use of statistics in establishing the empirical validity of COMTEK-λ, we digress to discuss the relevance of empirical validation of formal property theories. We might argue that there is little (if any) use in empirically validating theories that are established by proof theoretic means. To examine this argument, we focus only on the proof of component properties. We consider the case where a particular component behavior has been specified, and we must establish that the implementation conforms to, or satisfies, this specification, using model checking (Clark, et al., 1999).

Suppose a model checking proof of satisfaction is constructed. This would be sufficient only if we had a similar proof that the component environment possesses and satisfies its own formal specification. This argument will regress from the dependencies of that environment to some other environment, etc., but it will not regress forever. The ultimate environment is provided by a physical device such as a microprocessor. Notwithstanding formal verification of microprocessor logic, all physical devices are subject to manu-
facturing defects and wear that can only be detected by empirical means. Thus, all proof demonstrations ultimately rely upon empirical demonstration.

Although this reasoning may be a bit pedantic, it does demonstrate the tenuous nature of formal demonstrations of software behavior. As observed by Messerschmitt and Szyperski, the boundary between software and hardware is virtually non-existent, as software can always simulate hardware, and hardware can always realize software (Messerschmitt and Clements, 2001); the choice of which to use (hardware or software) is an economic rather than theoretical question. From this perspective it may be more reasonable to consider the selection of logical (software) or empirical (hardware) theories likewise to be a matter of practicality.

On a practical note, proving satisfaction is difficult and costly, and researchers are investigating how to use empirical methods to obtain a statistical proof of satisfaction of a formal specification (Giannakopoulou and Havelund, 2001). Such approaches rely upon comparing traces of program execution to traces produced by symbolic execution of a specification. In this way, demonstrating satisfaction is reduced to demonstrating complete test coverage, or some statistical percentage of coverage.

We do not argue against the utility of formal property theories. It is clear that where logical property theories exist and can be practically used, the burden of empirical validation may be significantly reduced. We are merely suggesting that empirical validation can probably never be eliminated in practice, even if software analysis and prediction become fully formal. Still, it remains a question for further research to demonstrate a seamless way of integrating and packaging a mix of formal and empirical property theories.

**Statistical Approach to Validating COMTEK-\(\lambda\)**

Empirical validity consists in quantifying the accuracy and repeatability of predictions made using an analytic model by statistically comparing those predictions with actual measurements of assembly properties. The process of empirically validating a PECT can be summarized in the following steps:

1. Obtain analytic properties of components (for example, through measurement).
2. Design validation assemblies and predict the assembly property of interest.

3. Construct the assemblies and observe their properties.

4. Statistically analyze the difference between predicted and observed properties.

We constructed a component benchmarking environment for (1) and instrumented component runtime for (3). The first turned out to be non-trivial since it was required to simulate but not re-implement the COMTEK runtime. We constructed a component benchmarking environment for (1) and instrumented component runtime for (3). The first turned out to be non-trivial since it was required to simulate but not re-implement the COMTEK runtime. We used statistical methods for two different purposes: latency measurement (1) (3), and quantification of accuracy and repeatability of the predictions (4). The sources of our statistical approach are (Kemerer, 1987) (NIST, 2001) (Walpole and Myers, 1989). We consider a large sample of measured latencies and use their mean as the value to be used as inputs to the model.

For statistical analysis of the predicted latency (4) we used both descriptive and inferential statistics, namely correlation analysis, and confidence and tolerance intervals of the magnitude of relative error (MRE). We used thirty sample assemblies as the basis for statistical analysis of our latency theory (Eq. 1). Thus, in the summary in Table 2, \((N)\) refers to the number of distinct assemblies that we tested, i.e., \((N = 30)\).

Correlation analysis allows us to assess the strength of the linear relation between two variables, in our case, predicted and observed latency. The result of this analysis is the coefficient of determination \(R^2\), whose value ranges from 0 to 1; 0 meaning no relation at all, and 1 meaning perfect linear relation. In a perfect prediction model, one would expect to have all the predictions equal to the observed latency, therefore the goal is a linear relation. The results of the correlation analysis are shown in Table 1, and can be interpreted as the prediction model accounting for 99.99% of the variation in the observed latency. The significance level means that there is only a 1% probability of having obtained that correlation by chance.

For the statistical inference about the latency property theory, we are interested in the magnitude of relative error (MRE) between the predicted and the observed latency. To validate a property theory and draw statis-
tical conclusions, we need a sample of MREs, based on a set of possible, and distinct, analytic assemblies. That is, for each assembly in the sample, we compute the MRE, obtaining in that way a sample of MREs. In doing this, we considered the mean of a sample of 15,000 measured assembly latencies to be the observed latency for each assembly.

We use tolerance intervals for statistical inference. Three types of questions are addressed by tolerance intervals (NIST, 2002):

1. What interval will contain \((p)\) percent of the population?

2. What interval guarantees that \((p)\) percent of the population will not fall below a lower limit?

3. What interval guarantees that \((p)\) percent of the population will not exceed an upper limit?

The first question applies to situations in which we want to control either the center or both tails of a distribution. In the case of MRE, because we are using the absolute value of the error, the predictions with MRE falling in the left tail of the distribution are even better than those in the center of the distribution. Therefore, it is better to use a one-sided tolerance interval, as in the case of the third question.

Table 2 is interpreted as saying that the MRE for 90\% \((p = 0.90)\) of assembly latency predictions will not exceed \((6.33\%)\); moreover, we have a confidence of 0.95 that the upper bound is correct. As can be seen, we achieved our goal of predicting with MREs no larger than 10%.

### 6 Key Results, Open Questions

We described the elements of a PECT and their relationships. We also described several ways to validate a PECT. These validations are essential to generate trust in a PECT and the components used in a PECT. We also described a number of concepts that are pertinent to documenting and packaging predictable assembly. Ultimately, we hope to provide guidelines for a standard approach to labeling components and prediction-enabled component technologies. We have, in most cases, demonstrated our ideas with the COMTEK-\(\lambda\) prototype.
This work has evoked a number of questions:

- How will more than one prediction technology be integrated in PECT? The constraints placed on a component technology by two prediction technologies may be incompatible or may interact in such a way as to perturb their individual or combined predictions.

- Assuming that more than one prediction technology can co-exist, can they be based in radically different theories? For example, can one theory rest in formal verification while another rests in a more empirical approach? Will the “seams” between these be visible?

- The PECT exemplar described in this report emphasized empirical measurement of resource consumption—time, in this case. This was well suited to empirical validation. How are non-resource attributes such as security to be empirically validated?

- What effect does PECT have on the feasibility of industrial certification of components? Would the same certification approach work for both resource and non-resource related component properties?

- At what time should component properties be evaluated and, possibly, certified? Properties can be evaluated in a component vendor environment, third-party environment, deployment environment, assembly environment, and end-execution environment.

These and many other questions remain open and provide fertile ground for future research.

Acknowledgements

The authors’ work is supported by the Software Engineering Institute, which is a federally-funded research and development center sponsored by the U.S. Department of Defense.

References


Figure 1: PECT Integration Co-Refinement
Figure 2: COMTEK Assembly Environment
Figure 3: Assembly Behavior and Definition of Assembly Latency
Table 1: Correlation Analysis Results

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R^2 = 0.9999053$</td>
<td>Coefficient of determination</td>
</tr>
<tr>
<td>$\alpha = 0.01$</td>
<td>Significance level</td>
</tr>
</tbody>
</table>
### Table 2: Second MRE Tolerance Interval

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>sample size</td>
<td>30</td>
</tr>
<tr>
<td>γ</td>
<td>confidence level</td>
<td>0.95</td>
</tr>
<tr>
<td>p</td>
<td>proportion</td>
<td>0.90</td>
</tr>
<tr>
<td>µ_{MRE}</td>
<td>over 30 assemblies</td>
<td>1.99%</td>
</tr>
<tr>
<td>UB</td>
<td>upper bound</td>
<td>6.33 %</td>
</tr>
</tbody>
</table>