November, 2001

Packaging Predictable Assembly with Prediction-Enabled Component Technology

Scott A. Hissam, *Software Engineering Institute*
Gabriel A. Moreno, *Software Engineering Institute*
Judith Stafford, *Software Engineering Institute*
Kurt C. Wallnau, *Software Engineering Institute*

Packaging Predictable Assembly with Prediction-Enabled Component Technology

Scott A. Hissam
Gabriel A. Moreno
Judith Stafford
Kurt C. Wallnau (corresponding author)

November 2001

TECHNICAL REPORT
CMU/SEI-2001-TR-024
ESC-TR-2001-024

Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

Packaging Predictable Assembly with Prediction-Enabled Component Technology

CMU/SEI-2001-TR-024
ESC-TR-2001-024

Scott A. Hissam
Gabriel A. Moreno
Judith Stafford
Kurt C. Wallnau (corresponding author)

November 2001

Product Line Systems

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Norton L. Compton

Norton L. Compton, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Table of Contents

List of Figures

List of Tables

Abstract

This report describes the use of prediction-enabled component technology (PECT) as a means of packaging predictable assembly as a deployable product. A PECT results from integrating a component technology with one or more analysis technologies. Analysis technologies allow analysis and prediction of assembly-level properties prior to component assembly, and, presumably, prior to component acquisition. Analysis technologies also identify required component properties and their certifiable descriptions. This report describes the major structures of a PECT. It then discusses the means of validating the predictive powers of a PECT so that consumers may obtain measurably bounded trust in design-time predictions. Last, it demonstrates the above concepts in a simple but illustrative model problem: predicting average end-to-end latency of a 'soft' real-time application built from off-the-shelf software components.

1 Introduction

This report describes a prototype prediction-enabled component technology (PECT). 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 allows 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 our designs to a subset that is analyzable. This premise has already been seen in the use of logical (formal) analysis and prediction [Finkbeiner+01] [Sharygina+01], and it can also be applied to empirical analysis and prediction. 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 structure of a PECT, explores the strengths and limitations of this approach to predictable assembly, and charts a course for further applied research.

1.1 Background

This report presents results of an ongoing internal research and development activity at the Software Engineering Institute (SEI). The objective of this research is to accelerate the industrial adoption of technologies that can reliably predict the runtime behavior of systems. A premise of this research is that software component technology provides an effective vector for packaging analysis and prediction technologies. Prediction-enabled component technology is one form this packaging can take.

While this research draws upon an array of published research results, here we describe only how this report is related to other SEI reports stemming from this internal research effort. Bass et. al. provide a market assessment of component-based software in [Bass+01]. Bachmann et.

al. discuss the technical elements of software component technology in [Bachmann+00]. Wallnau and Stafford outline the key conceptual distinctions of predictable assembly in *The Philosophy of Predictable Assembly.*[1]

1.2 Organization of this Report

Chapter 2 presents an overview of the major elements of PECT. Chapter 3 describes the component and attribute prediction technologies used in the PECT prototype, COMTEK-λ. Chapter 4 describes this prototype and its validation. We describe only enough to illustrate the structure of a PECT and how, in general, a PECT must be validated. In Chapter 5, we summarize the key results and questions raised by the prototype. Appendix A and Appendix B provide the details of how we established the theoretical and empirical validity of the PECT prototype. Appendix C shows the assemblies used in empirical validation.

1. Wallnau, K. & Stafford, J. *The Philosophy of Predictable Assembly*, (CMU/SEI-2000-TR-023), Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, (to be published in December 2001).

2 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 software component suppliers and integrators. These constraints are expressed as a component model that specifies required component interfaces and other development rules [Bachmann+00] [Heineman+01]. In today’s component technology, component models are designed to simplify many aspects of the integration (composition) 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 for performance [Klein+93] and reliability [Lyu+96], 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.

The structure of a PECT is depicted using the Unified Modeling Language (UML) in Figure 1. As already noted, a PECT is an association of a component technology (‘Component Technology’) with one or more analysis technologies (‘Analysis Technology’). The UML association class ‘Association Validity’ stipulates that each such association is validated. ‘Assumption’ and ‘Interpretation’ validity, taken together, demonstrate theoretical validity. Each of these forms of validity is discussed later, and are elaborated in detail in the appendices.

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 in Figure 1 between component and analysis technologies would be reasonable. However, our perspective is centered on component technology, and how these can be extended to predict emergent (assembly-level) attributes. We also expect that it will, in general, be much easier to adapt existing prediction technologies to work with existing component technologies rather than the converse. Thus the 1:N association between component technologies and prediction technologies in Figure 1.

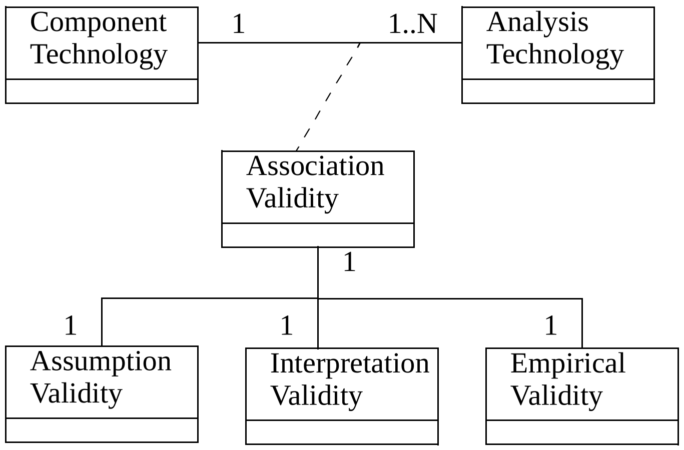


Figure 1: PECT Conceptual Structure (UML)

2.1 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+00] [Heineman+01] [Szyperski+97]:

- 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[2] only at a much

1. There is, however, an interesting analog between the context dependencies that are included in Szyperski’s base definition of software component [Szyperski+97], and the assumptions of analysis technologies.

2. In fact, the various COM-based Microsoft component models are an integral part of the Microsoft operating systems.

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 (assembly-level) attributes.

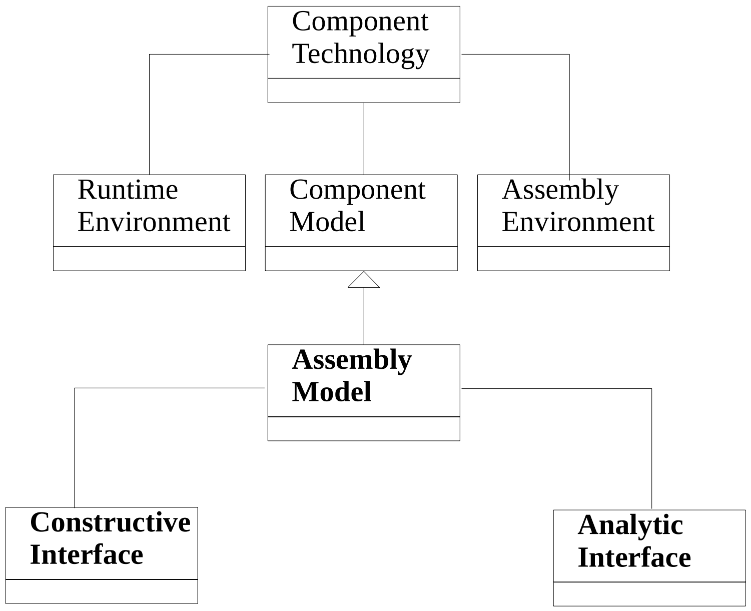


Figure 2: Component Technology and the Constructive Model

Figure 2 introduces the new concept, ‘Assembly Model,’ as a refinement of the traditional component model. By this we suggest that 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.

Although Figure 2 shows the analytic interface to be a constituent of the assembly model, the analytic interface described later is, in fact, derived from the component technology. The conceptual model in Figure 2 reflects our assertion that component models must be extended to include design-time analysis and prediction of emergent (assembly-level) properties.

2.2 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+97]) 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+95]. A formal definition of Simplex has been used to prove (the strongest form of prediction) a number of properties [Rivera+96]. 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 [KIein+99][Bass+00] offers some clues as to how analysis models and their contingent design patterns may be disentangled; this is one starting point for PECT research.

In Figure 3, we introduce the new concept, ‘Analytic Model.’ The 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* component. 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.

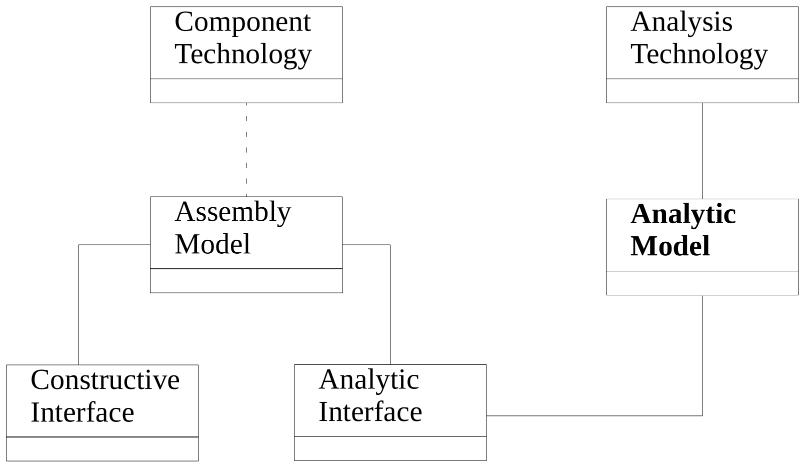


Figure 3: Analytic Model and Analytic Interface

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.

2.3 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 straightforward due to mismatched assumptions. Where mismatches arise, either or both must be adjusted, as illustrated in Figure 4.

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 may make its predictions more accurate, but may also increase the cost of applying the technology. Figure 4 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 tradeoff might be[1] 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.

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.

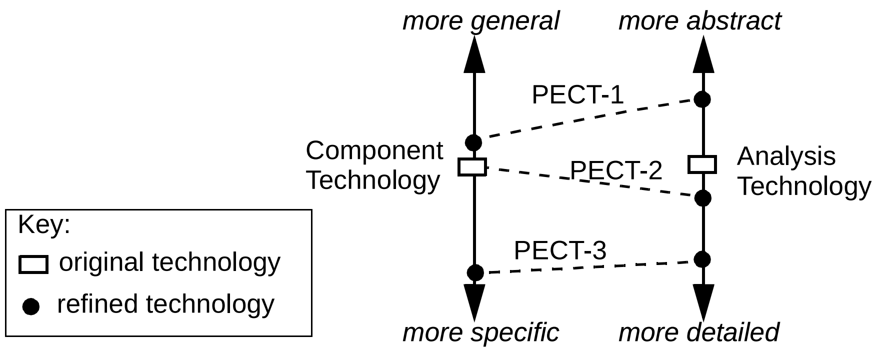


Figure 4: PECT Integration Co-Refinement

We refer to the integration process implied by Figure 4 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.

1. 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.

2.4 PECT Validation

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.*

The consumers of PECT will want to know ahead of time how much confidence to place in the predictive powers of the technology. That is, can the PECT be *trusted*?

We believe that theoretical *and* empirical validity must be established to engender *bounded, quantifiable trust* in a PECT:

- *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.
- *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.

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+96]. 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.

3 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.

3.1 COMTEK Component Technology

COMTEK[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:

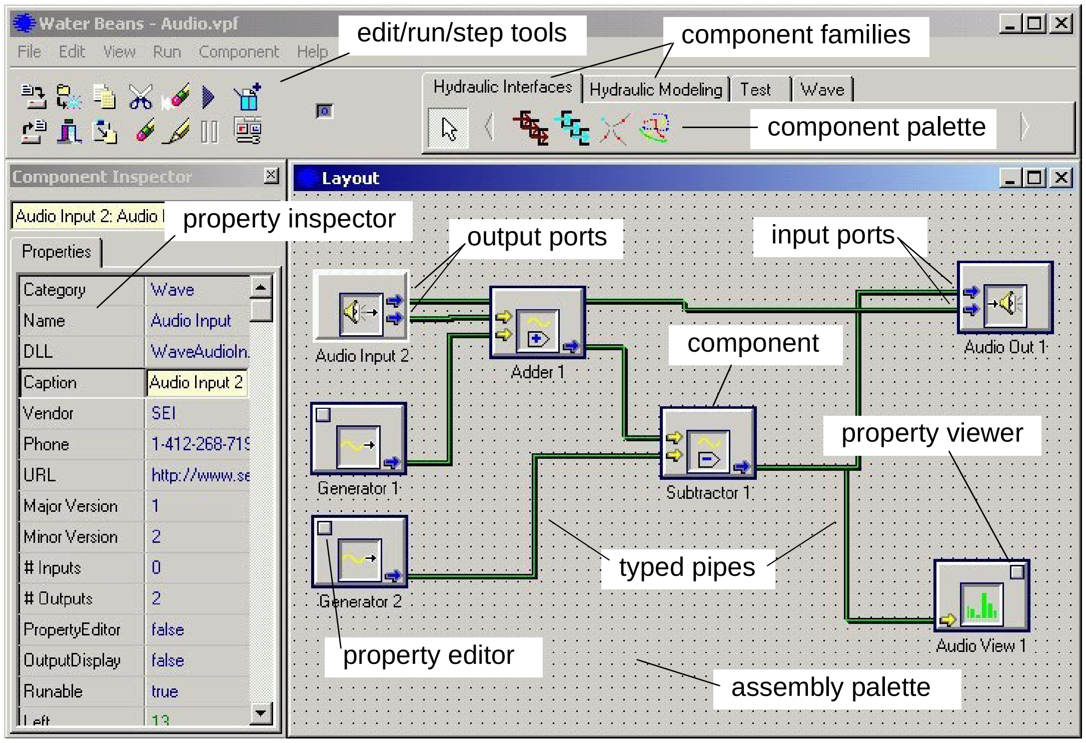
- It enforces a typed pipe-and-filter architectural style.
- A fixed round-robin schedule is calculated from component input/output dependencies.
- The execution of an assembly is sequential, single-threaded, and non-preemptive.
- It runs under the Microsoft Windows family of operating systems.
- Components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

Despite its simplicity, the generality of COMTEK was demonstrated in several application domains. The menu tabs above the assembly canvas in Figure 5 display four families of components: Hydraulic Interfaces, Hydraulic Models, Wave Interfaces, and Test Interfaces. Components are chosen from one or more component families, depending on the application.

Figure 5 presents a screenshot of the COMTEK assembly environment. The graphic depicts an assembly built from components of the Wave Interface family. This and similar assemblies are the subject of the PECT demonstration. These assemblies implement audio signal sampling,

1. 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.

manipulation, and playback functionality. We chose to develop a PECT for assembling audio playback applications since we could develop (we thought) a simple performance analysis model to accommodate the relative simplicity of COMTEK.



Audio Input 2: Audio I

Properties

Category	Wave
Name	Audio Input
DLL	WaveAudioIn.
Caption	Audio Input 2
Vendor	SEI
Phone	1-412-268-71
URL	http://www.se
Major Version	1
Minor Version	2
# Inputs	0
# Outputs	2
PropertyEditor	false
OutputDisplay	false
Runable	true
Left	13

Figure 5: COMTEK Assembly Environment

3.2 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.

Our emphasis in this work was on understanding PECT rather than in solving the latency prediction problem. We were hoping 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.

4 Illustration

This chapter describes the packaging of latency analysis and prediction with COMTEK. We refer to the resulting integration as COMTEK-λ.[1]

4.1 Assembly Model

As explained earlier, 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. A specification of COMTEK can be found in [Plakosh+99]. 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 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. Figures 6, 7, and 8 show UML specifications of three views of the assembly model. In these views we use UML stereotypes to distinguish the constructive (‘`<<constructive>>`’) and analytic (‘`<<analytic>>`’) interfaces.

At this stage of our research, we are uncertain about the best way to document the assembly model, and indeed are uncertain about precisely what lies within the scope of the assembly model. What follows should be interpreted as suggestive rather than normative. The three views of the assembly model illustrated below define the component metatype, interaction rules, and dynamic behavior of COMTEK-λ assemblies, respectively.

4.1.1 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-condi-

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:

   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 λ for the latency theory.

tions, 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.

Figure 6 shows a UML model of the metatype for COMTEK-λ components. Many things have been abstracted away in this model—but it gives an example of the things that have to be specified, such as required properties, methods, and invariants. Instances of 'ComponentType' are deployable units that are component factories—they provide runtime instances of themselves via their '`getNewInstance()`' method. For example, let us assume that we have an instance of 'ComponentType', whose 'componentTypeName' property is set to 'WaveView.' Then, 'WaveView' is a component that can be deployed, and can provide instances of itself via '`getNewInstance()`.'

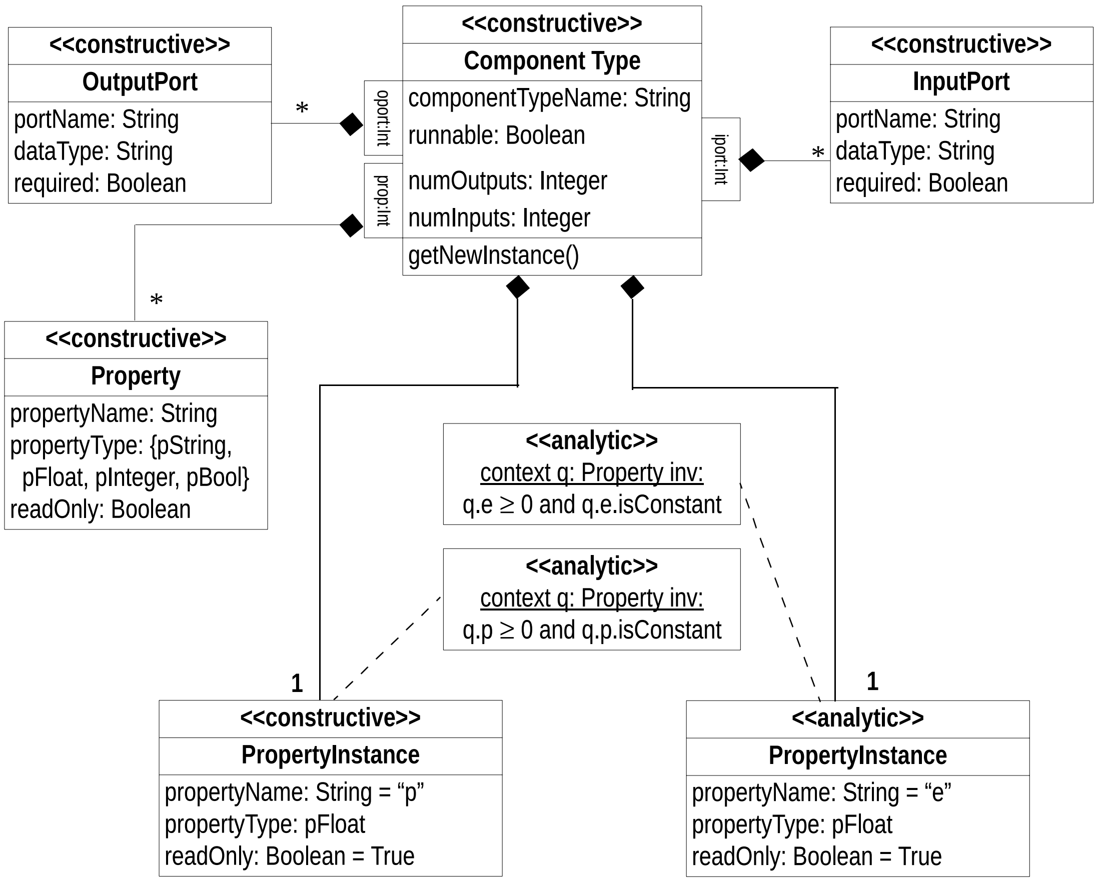


Figure 6: COMTEK-λ *Component Metatype*

More significant for our purposes is the differentiation of constructive and analytic interfaces. Input and output ports and a set of properties constitute the constructive interface. The analytic interface consists of two mandatory analytic properties, 'p' and 'e,' whose meanings are described later in this report. Note that UML does not permit co-habitation of classes and instances in models. Thus, although ‘p’ and ‘e’ are, logically, instances of ‘Property,’ they are depicted as classes in Figure 6. Also associated with ‘p’ and ‘e’ properties are constraints on their values, specified in the UML Object Constraint Language (OCL).

4.1.2 Interaction Rules

The interaction model describes rules for composing components into assemblies. Figure 7 shows some aspects of the COMTEK-λ assembly model. This model defines the way that components can be connected, and specifies invariants for the whole assembly. The ‘Component’ class in the diagram represents an instance of ‘ComponentType’ from Figure 6. The association class ‘Connection’ represents how components are linked together. In addition, OCL invariants represent the constraints that must be observed when composing components; for example, a component cannot connect to itself, and the data types of the output and input ports must be the same. In the case of COMTEK-λ applications, we are also interested in a property of the application that indicates whether an assembly is fully connected, because the COMTEK-λ runtime environment will not execute the application unless it is fully connected. The definition of ‘fullyConnected’ for component and application are defined in OCL.

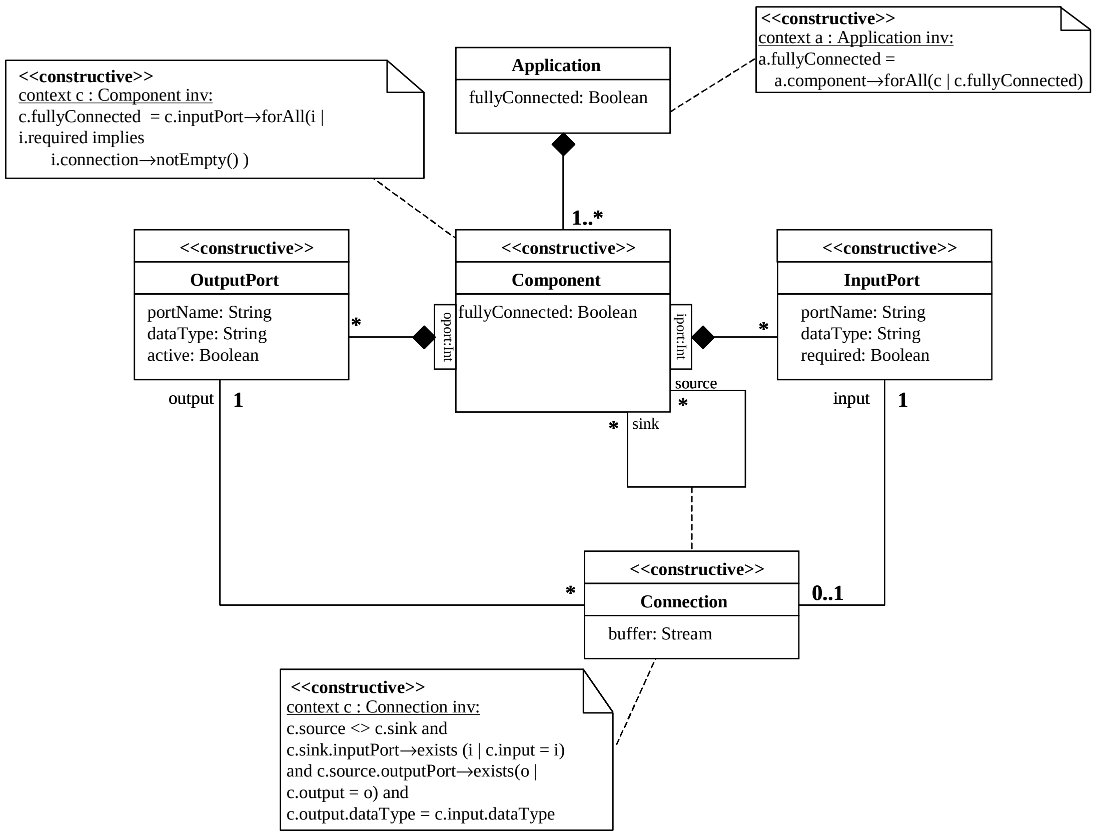


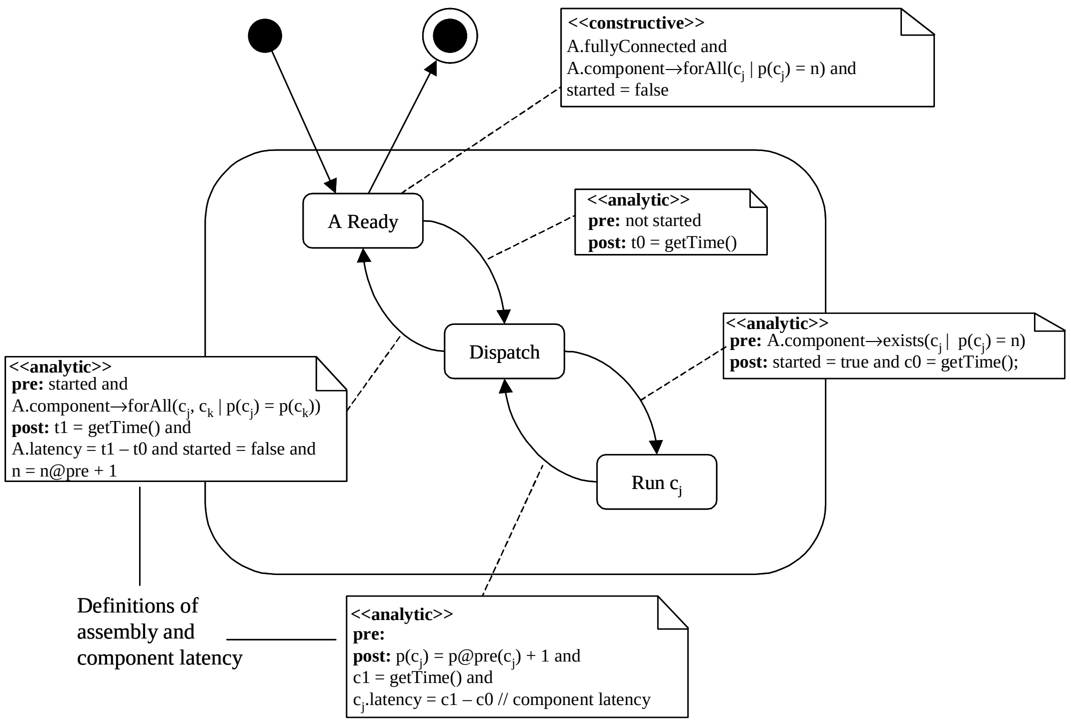
Figure 7: COMTEK-λ *Interaction Rules*

4.1.3 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 may be vital not only for implementing the component runtime environment, but also for constructing a PECT. For latency prediction, there are several questions we need to answer:

- 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.



<<**constructive**>>
A.fullyConnected and
A.component→forAll(c_j | $p(c_j) = n$) and
started = false
A Ready
<<**analytic**>>
pre: not started
post: t0 = getTime()
Dispatch
<<**analytic**>>
pre: A.component→exists(c_j | $p(c_j) = n$)
post: started = true and c0 = getTime();
<<**analytic**>>
pre: started and
A.component→forAll(c_j, c_k | $p(c_j) = p(c_k)$)
post: t1 = getTime() and
A.latency = t1 – t0 and started = false and
n = n@pre + 1
Run c_j
Definitions of assembly and component latency
<<**analytic**>>
pre:
post: $p(c_j)$ = p@pre(c_j) + 1 and
c1 = getTime() and
c_j.latency = c1 – c0 // component latency

Figure 8: Assembly Behavior and Definition of Assembly Latency

Figure 8 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; ‘`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. Execution order is not required to predict end-to-end latency of a COMTEK-λ assembly in steady state (as will be explained in the appendix). However, this information might be needed in other circumstances.

Also defined in Figure 8 are the runtime interpretations of assembly latency (‘A.latency’) and component latency (c_j.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.

4.2 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.

4.2.1 The Property Theory

The COMTEK-λ latency theory, denoted as A_Λ, is summarized by the following equation:

$$A.\text{latency} = \max\left(\left(\sum_{\Phi_j \in A} \Phi_j.e + \sum_{\Theta_j \in A} \Theta_j.e\right), \max(\{\Phi_j \in A \mid \Phi_j.p\})\right) \qquad \text{Eq. 1}$$

We denote an assembly as the set of components A, and the k^th component of A is denoted as either or Θ_k or Φ_k. These correspond to one of two analytic component types: Θ refers to components that only have dependencies that are internal to A, while Φ refers to components that also exhibit dependencies on external periodic events (these symbols were chosen for their mnemonic value). A property of a component or assembly is denoted using ‘dot’ notation. A.latency is the end-to-end latency of an assembly. 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 (also, refer to Figure 6):

- *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. Note that this analytic model is not parameterized by invocation order or connections among components. Neither the summation nor *max* depend on the order of the components in A in Eq. 1.

A more precise specification of the COMTEK-λ component metatype in Figure 6 would have shown Θ and Φ as subtypes (UML subclasses) of ‘ComponentType.’ We have not done so only to avoid over-complicating the presentation. It is also worth noting that these component types only have meaning within an A_Λ interpretation, i.e., within an analytic assembly.

4.2.2 Adapting COMTEK for Latency Prediction

For reasons of expediency, we chose to derive A_Λ 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_Λ, and are therefore arguably part of the analytic interface of PECT-λ. 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.

4.3 Association Validity

PECT will be of little value unless its users (application designers and engineers) *trust* the predictions. Trust is a complex social phenomena 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. We provide the technical foundation for trust in a PECT prediction by validating the associations between a component technology and one or more prediction technologies. We referred to this in Figure 1 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 theoretical validity is provided in Appendix A, while Appendix B is concerned with the details of empirical validity.

4.3.1 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 true of the theories underlying software engineering practice.

There appears to be two key questions that must be asked prior to investing the time and effort required to empirically validate a PECT:[1]

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 any 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.

1. This may well be true in the physical sciences, too, although we make no strong claims on this matter.

- *Interpretation validity* establishes that each constructive assembly has an interpretation in the theory. This validation addresses the second two key questions by demonstrating the relationship between constructive assemblies and model theoretic assemblies (question two), and by providing a basis for theory falsification (question three).

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_Λ was derived from COMTEK. That is, the latency theory A_Λ 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_Λ depends upon, i.e., its assumptions.

To demonstrate COMTEK-λ association validity, we began by itemizing the assumptions that A_Λ would likely depend upon, and by assigning names to and carefully defining these assumptions. These assumptions included scheduling policy, concurrency policy, how components and assemblies are defined, what it means for an assembly to be in steady state, and so forth. From these definitions we proceeded in stepwise fashion to derive A_Λ. The entire derivation takes the form of a sequential argument, with the dependencies between later steps to earlier steps explicitly noted.

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. (This is true in natural science, too.)

Interpretation Validity

Recall that an analytic assembly is an interpretation, or mapping, of a constructive assembly under some property theory. For COMTEK-λ, 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_Λ 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_Λ (see Eq. 1 in Section 4.2.1 on page 19), and hence need not be considered. Demonstrating the consistency of A_Λ was likewise trivial. However, had A_Λ 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 slightly more involved.

4.3.2 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. We must use statistics to quantify both forms of error.

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 [Clarke+99].

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 manufacturing defects and wear that can only be detected by empirical means. Thus, all proof demonstrations ultimately rely upon empirical demonstration.

This reasoning carries a whiff (or a strong odor?) of sophistry. Yet 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 [Messersch+01]; 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 [Giannak+01]. 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 becomes 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-λ

The use of statistics in empirical validation allows us to infer characteristics of the property theory from samples of its application. Confidence intervals and significance levels provide more insight in the analytic model, so that an application assembler can interpret the results of the analytic model, or compare the results from competing models. The number of samples that are used for empirical validation depends on how difficult or expensive it is to get a data point for the sample. This, in turn, depends on the kind of property that is being predicted. Therefore, it is expected that there will be tradeoffs between the accuracy of empirical and statistical analysis, and the cost of producing this analysis. This tradeoff must be quantified.

We use the average of a set of measurements to estimate the population's mean. By using an estimator, we incur an error that we want to quantify. To do so, we define a confidence interval for the estimation, as follows:

$$\bar{x} - t_{\alpha/2}\frac{s}{\sqrt{n}} < \mu < \bar{x} + t_{\alpha/2}\frac{s}{\sqrt{n}} \qquad \text{Eq. 2}$$

where μ is the mean of the population; $\bar{x}$ and s are respectively the average and the standard deviation of the sample; n is the size of the sample; and $t_{\alpha/2}$ is the t-value from a t-distribution with $v = n - 1$ degrees of freedom leaving a tail with an area of $\alpha/2$ to the right [Walpole+89]. With this formula, we calculate a $(1 - \alpha)100\%$ confidence interval for the population's mean. In other words, if $\alpha = 0.05$, we can be 95% confident that μ falls within that interval. This calculation assumes that the sample comes from a normal population (i.e., a population that has a normal distribution). When the sample is large ($n \geq 30$), we can use the following formula even when normality cannot be assumed:

$$\bar{x} - z_{\alpha/2}\frac{s}{\sqrt{n}} < \mu < \bar{x} + z_{\alpha/2}\frac{s}{\sqrt{n}} \qquad \text{Eq. 3}$$

where $z_{\alpha/2}$ is the z-value from a normal distribution that leaves a tail with an area of $\alpha/2$ to the right. When the sample is not large and we cannot be sure about the normality of the population, there are some procedures that can be applied both to test for normality and to normalize the sample. An example of this can be found in Appendix B.

The core of establishing empirical validity is to demonstrate how accurate and repeatable the predictions are on average. Two methods we can use to do so are *magnitude of relative error* (MRE) and *correlation analysis*. These methods have been used in the empirical validation of other software related estimation models [Kemerer+87].

MRE is a measure of the percentage error of the predicted property, and it is defined as follows:

$$MRE = \left|\frac{A.p - A.p'}{A.p'}\right| \qquad \text{Eq. 4}$$

where A.p is the predicted property and A.p′ is the measured property. If we used absolute error, it would be difficult to compare the errors for different predictions. For example, an error of 0.3 would be relatively small for an actual value of 140 while it would be very large for an actual value of 1.44. Therefore, we want to use relative error instead of absolute error so that we can compare and analyze the predictions for different assemblies. As we are concerned with the goodness of the model on average, we can compute the predictions for a sample of assemblies and use the average MRE to estimate the mean of the MREs for all possible assemblies. Again, following the same rationale and procedure as before, we can calculate a confidence interval for this estimation.

The second method we can use for evaluating the goodness of a model is correlation analysis. This method measures the strength of the linear association between two variables [Walpole+89]. The *sample correlation coefficient R* is calculated from a sample of observed pairs of values for the two variables X and Y. The former is called the independent variable, and the latter is called the dependent variable. For our purposes, we can use the predicted property as the independent variable, and the actual property as the dependent variable.

In practice, the *sample coefficient of determination R^2* is used to evaluate the correlation. A value of $R^2 = 1$ means that there is a perfect correlation between X and Y, (i.e., between the predictions and the actual property values), whereas an $R^2 = 0$ indicates that there is no correlation at all. In general, we can think of R^2 as an indication of the amount of variability of Y that is accounted for by its linear relation with X. For instance, $R^2 = 0.9$ indicates that the model explains 90% of the actual value of the property.

When correlation analysis is used for empirical validation, it is also important to determine the significance of the correlation so that it can be shown that the linear relation in the sample did not happen by chance. This can be done by using the critical values for R, which can be found on tables or with computer software based on the size of the sample and the desired significance level. If R is greater than the critical value, then the correlation is considered statistically significant.

Statistical Validation of COMTEK-λ

A summary of the statistical validity of COMTEK-λ is shown in Table 1. The MRE leads to a confidence interval that suggests that latency predictions are, on average, quite accurate. That is, if we create several sets of random assemblies, the average MRE of each set will fall within the confidence interval for 95% of the sets. The high R^2 is also a strong indicator of the strength of correlation between observed data and predictions.

From the empirical (statistical) validation, it would seem that COMTEK-λ offers a promising solution to predicting latency of soft-realtime applications. But we are right to be skeptical and recall the old adage: “statistics don’t lie, but liars use statistics.” We offer the following caveats not in the mode of self confession, but rather to pose the question: how should a PECT be packaged to expose the limitations of an analysis model and its validation?

Caveats

The critical eye can, without too much effort, spot a variety of weaknesses of the COMTEK-λ latency theory and its empirical validation.

Statistical Analysis	
Latency Theory	
MRE	**0.554%**
95% Confidence Interval	Lower: 0.010% Upper: 0.701%
R^2	**0.9999**
Significance Level	0.01
MRE: measure of relative error R^2: coefficient of determination	

Table 1: Empirical Validation of COMTEK-λ

First, how do we know that the assemblies used in empirical evaluation (see Appendix C) represent the set of all possible COMTEK assemblies? An examination will show that they are quite similar in the number of components, component interactions, and component latencies. A thorough empirical validation would characterize COMTEK assemblies. For example, the assemblies might be characterized by the number of components, their average latency, the number or ratio of input/output connections, and so forth. Extreme values for these parameters, and parameter combinations, could be used to describe boundary cases with which to test empirical validity. The validity of the latency theory might well vary across these extremes.

Second, we did not parameterize the latency theory in Eq. 1 with a variety of environmental factors such as processor and memory speed, amount of free memory, resource contention, or other factors that may strongly influence latency. There is no doubt that we could extend the latency theory with such factors, and we may do so to incorporate sensitivity analysis and multi-regression analysis, and to further explore packaging PECTs and PECT-conformant components. While this serious deficiency in the empirical evaluation should be made explicit in the packaging of a PECT, the question remains: can any standard approach to PECT labeling reliably identify such limitations, whether they result from commission or merely omission?

Ultimately, the question devolves to developing standard labeling analogous to the nutritional labeling standards for foodstuffs. Backing up standard labeling will be the strictures of “truth in advertising” and the legal and economic consequences of violating these strictures.

5 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-λ prototype.

Not surprisingly, given the immature state of this research, this work and report have resulted in 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 will be addressed in upcoming research.

Appendix A Theoretical Validity

In the following, A.λ and C.λ are sometimes used as shorthand for A.*latency* and C.*latency*, respectively. Also, the terms 'assembly' and 'component' refer to *analytic* assemblies and components unless otherwise noted.

A.1 Association Validity—Derivation of A_Λ

Definitions

The following definitions are used throughout the derivation.

1. **Component** C: The scheduled[1] entity, C.
2. **Assembly** A: The set of components $A = \{C_1, C_2, ...C_k\}$.
3. **Schedule**: The execution order of components in A.
4. **Cycle** A^n, C^{nm}: The nth execution of the schedule. C^{nm} denotes that a component has executed *m* times; its last execution was in the *n*th cycle, denoted A^n.
5. **Component latency** C.λ: The duration of time that begins when the COMTEK-λ executive calls the `execute()` method on a component, and ends when the component returns from the `execute()` method.
6. **Assembly latency** A.λ: The latency of the assembly A, defined as the duration of time that begins when the COMTEK-λ executive calls C_0^n.`execute()` on an arbitrary first component C_0 (i.e., $\forall C_j \in A, n>0 \bullet C_j^{n-1} \wedge \neg C_j^n$), and the time C_k^n returns from its C_k^n.`execute()`, where C_k is last component in A^n (i.e., $n>0, C_k^{n-1} \wedge \forall C_j \in A \mid C_j \neq C_k \bullet C_j^n$).
7. **Steady state**: An assembly is said to be operating in steady state, when $A^j.\lambda = A^{j+1}.\lambda$ for all $j \geq k$, where A^k is the first steady state cycle. Prior to A^k, an assembly is in the **initializing state**. All assemblies eventually reach and remain in steady state. (This is not proven.)

1. The schedule-*able* entity is, in COMTEK-λ, a dynamic link library (DLL) that conforms to the COMTEK-λ API. The schedule-*ed* entity is an instance of the schedule-able entity in a particular assembly.

Observations

8. $C.\lambda$ has two constituents:[1] the duration of time, *e*, that C executes, and the duration of time, *b*, that C waits ("blocks") for resources that are required to complete its execution. That is, $\lambda = e + b$.

9. Different COMTEK-λ components can, and usually do, perform different functions, but each component must execute the same function in each cycle A^n. Further, the function must be independent of its data. Therefore, $C.\lambda$ can be represented by a single time duration rather than a vector of durations, and *e* is constant in $\lambda = e + b$. This is constraint 1) on page 20. This constraint greatly simplifies A_Λ.

10. COMTEK-λ uses round-robin scheduling, which ensures that each component in A is executed exactly once per cycle. The schedule is determined by data dependencies derived from an analysis of pipeline connections among components. The subassembly $C_j \bullet C_k$, where '•' denotes an allowed interaction, or pipe, is interpreted by the scheduler as asserting that C_k depends upon data produced by C_j. For $C_j \bullet C_k$, the scheduler guarantees that C_j^n`.execute()` executes prior to C_k^n`.execute()` in A^n.

11. From 10, the COMTEK-λ scheduler guarantees that $b = 0$ in $\lambda = e + b$, for all components that depend only upon other components.

12. Components may also exhibit dependencies on non-component resources, that is, resources external to A. These dependencies are opaque to COMTEK-λ: the scheduler does not take these external dependencies into consideration in scheduling, and COMTEK-λ makes no guarantees about *b* in $\lambda = e + b$ when blocking on external resources.

13. From (11) and (12), we adduce two classes of component: one class (Θ) consists of those components that depend only upon other components, the other (Φ) consists of those components that *also* exhibit external dependencies.[2] That is:

    a. $\Theta.\lambda = e$

    b. $\Phi.\lambda = e + b$, where *e* is a known constant, and *b* is a variable that depends upon the interaction protocol with the external resource, e.g., blocking or non-blocking.

14. *b* must be bounded in $\Phi.\lambda$. Since the COMTEK component technology is silent on external dependencies, **we must impose design constraints external to COMTEK** that bound *b* for all external dependencies. This is constraint 2) on page 20. Again, this constraint greatly simplifies A_Λ.

1. This definition is arguably true of latency in general.

2. As a mnemonic, Φ denotes pipes that extend beyond an assembly boundary, while Θ denotes pipes that are wholly contained by an assembly boundary.

a. We restrict Φ to dependencies on periodic events x, with a constant period, $x.p$. Both x and $x.p$ are independent of COMTEK-λ. In the following, we usually care about the period and not the event, and so we will usually denote $x.p$ as simply p.

b. In steady state, for an assembly consisting of exactly one component Φ_k, $\Phi_k.\lambda = e + b = e + (p - e)$. That is, the amount of time Φ_k blocks on x (i.e., $\Phi_k.b$), will be the period $x.p$ less the amount of time Φ_k spent executing (i.e., $\Phi_k.e$).

Derivation

A_Λ predicts A.λ in steady state. We derive latency prediction functions for three classes of assemblies:

- f^Θ for A consisting exclusively of Θ components.
- f^Φ for A consisting exclusively of Φ components.
- $f^{\Theta\Phi}$ for A consisting of arbitrary mixes of Θ and Φ.

Each are derived, in turn.

Class 1: Θ Assemblies (f^Θ)

15. Base case: if A = $\{\Theta_j\}$, then A.λ = $\Theta_j.e$, by (5) and (13.a).

16. If A = $\{\Theta_j, \Theta_k\}$, then A.λ = $\Theta_j.e + \Theta_k.e$, where '+' is arithmetic addition.

    a. Intuition: Assume COMTEK-λ is infinitely fast so that scheduling overhead can be ignored. Since Θ has by definition no blocking time, assembly latency will consist only of component execution time.

    b. Proof Scheme[1]: By (6), A.λ is the time elapsed between invocation of the execute method of the first component in an assembly, and the return from the execute method of the last component in that assembly. Figure 9 shows the execution timeline of an

1. The vertical axis represents components, while the horizontal axis represents time. Time is divided into ticks. The parenthetical subscript $\Theta_{(x)}$ represents the execution duration x clock ticks for component Θ . The subscript $\Phi_{(x, y)}$ represents the execution duration x and period of external resource y for component Φ.

assembly with two Θ components. As can be seen, the latency of the assembly is the sum of the latencies of the two components.

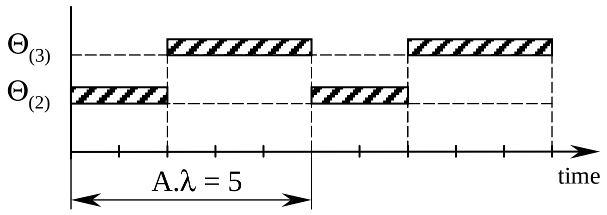


Figure 9: Execution Timeline for an Assembly with Two Θ Components

17. The previous case can be extended to an assembly with any number of components. If A = $\{\Theta_0, \Theta_1, ... \Theta_j\}$, then $A.\lambda = f^{\Theta}(A) = \Theta_0.e + \Theta_1.e + ... + \Theta_j.e = \sum_{\Theta_j \in A} \Theta_j.e$

Class 2: Φ Assemblies (f^{Φ})

18. Base case: if A = $\{\Phi_j\}$, then $A.\lambda = \max(\Phi_j.e, \Phi_j.p)$, where *max* is the larger value of $\Phi_j.e$ and $\Phi_j.p$.

    a. Intuition: We might expect $\Phi_j.e \le \Phi_j.p$, but this need not be so. It is true that if $\Phi_j.e > \Phi_j.p$, then Φ_j will miss at least one periodic event *x*. The significance of this, however, will depend upon the semantics of the application.[1]

    b. Proof Scheme: Figure 10 shows the timeline for the three possible cases: where $\Phi_j.e < \Phi_j.p$, where $\Phi_j.e = \Phi_j.p$, and where $\Phi_j.e > \Phi_j.p$.

1. In some applications, for example audio playback, all external events must be met. In other applications, for example radar position, only the “last” event must be met; in this situation, we may only need to put a bound on the number of periodic events missed per unit of time.

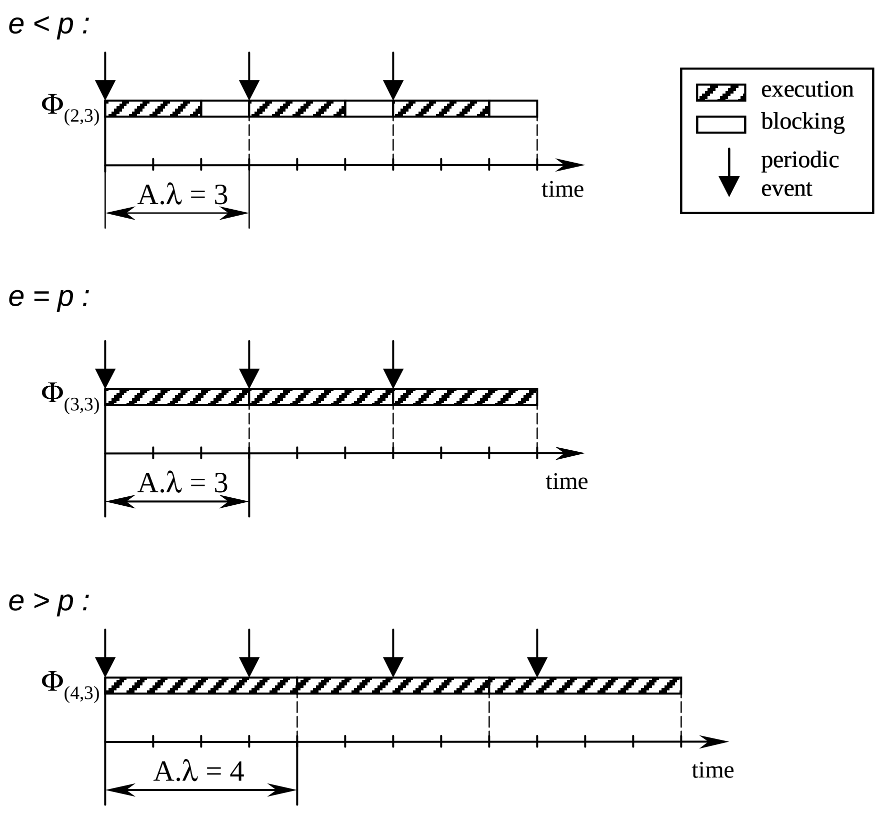


Figure 10: Timeline for Φ Components with e < p, e = p, and e < p

19. If A = $\{\Phi_j, \Phi_k\}$, then A.λ = max(($\Phi_j.e$ + $\Phi_k.e$), max($\Phi_j.p$, $\Phi_k.p$))

    a. Case 1, where $\Phi_j.p = \Phi_k.p$: A.λ = max(($\Phi_j.e$ + $\Phi_k.e$), p) . Figure 11 depicts this situation.

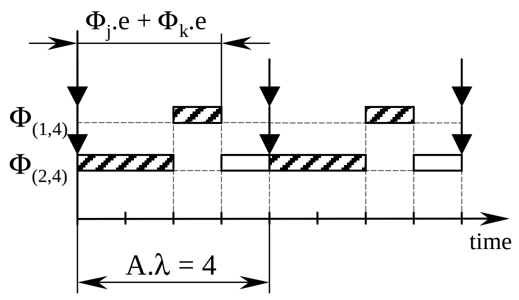


Figure 11: Timeline for Two Φ Components with the Same Period

b. Case 2, where $\Phi_j.p \neq \Phi_k.p$: This case can be considered as a special case of Case 1, with the term p in Case 1 replaced by $\max(\Phi_j.p, \Phi_k.p)$. Figure 12 shows how the largest period dominates the latency of the assembly.

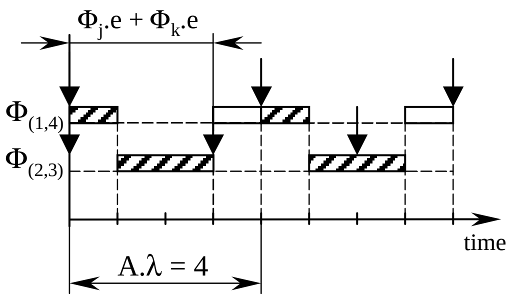


Figure 12: Timeline for Two Φ Components with Different Periods

20. If $A = \{\Phi_0, \Phi_1, \ldots \Phi_j\}$, then
$A.\lambda = f^{\Phi}(A)$
$= \max((\Phi_0.e + \Phi_1.e + \ldots + \Phi_j.e), \max (\Phi_0.p, \Phi_1.p, \ldots, \Phi_j.p))$
$= \max(\sum_{\Phi_j \in A} \Phi_j.e\, ,\ \max(\{\Phi_j \in A \mid \Phi_j.p\}))$.

Class 3: ΘΦ Assemblies ($f^{\Theta\Phi}$)

21. We observe that if $e \geq p$ in some $\Phi'.(e, p)$, then Φ' will never block on the periodic event $x.p$. A Φ component that never blocks is, for the purpose of latency computation, indistinguishable from a Θ component whose latency is $\Theta.e$. We can also view this from the other direction, and in a more specialized form. Specifically, a Θ component is indistinguishable from a Φ component with $\Phi.e = \Theta.e$ and $\Phi.p = 0$.

22. Using (21), we can apply (20) to the general case where A consists of a set of Θ and Φ components:

$$A.\text{latency} = f(A) = \max\left(\left(\sum_{\Phi_j \in A} \Phi_j.e + \sum_{\Theta_j \in A} \Theta_j.e\right), \max(\{\Phi_j \in A \mid \Phi_j.p\})\right) \qquad \text{Eq. 5}$$

where the term $\Sigma\Theta_j.e$ reflects the relation $\Theta_j.e = \Phi.(e, 0)$. Naturally, since $p = 0$ for the Θ components, we need not introduce an additional term into the set of component periods.

23. Although it may be useful for other purposes to distinguish between Θ and Φ components, this distinction can be represented implicitly using (21). That is, we might dispense with the distinction between Θ and Φ. If all components C possess two attributes, C.*e* execution time, and C.*p* the period of an external dependency, then Θ components would be represented implicitly by letting C.*p = 0*.

24. Under the convention in (23), Eq. 5 can be simplified as follows:

$$\text{A.latency} = f(A) = \max\left(\sum_{C_j \in A} C_j.e, \max(\{C_j \in A \mid C_j.p\})\right) \qquad \text{Eq. 6}$$

A.2 Interpretation Validity

We must demonstrate that every constructive assembly can be expressed as an analytic assembly (completeness), and that there is a unique latency prediction for each constructive assembly (consistency).

A.2.1 Completeness

Each component $C \in A$ is converted to an analytic component of one of two types, based on these rules:

- If C interacts with a periodic component C' external to A, then it is represented by an analytic component of type Φ, where the latency property is denoted as Φ.(e, p), where 'e' denotes the execution time of C and 'p' denotes the period of C'.
- If C interacts only with other components in A, then it is represented by an analytic component of type Θ, where the latency property is denoted as Θ.e, where 'e' denotes the execution time of C.

Since interactions in A (that is, pipes) are ignored, we need not consider ordering. We therefore treat A as the set of components $\{a_1, a_2, a_3, ..., a_j\}$. So constructed, it is clear that every possible COMTEK-λ constructive assembly can be represented as an analytic assembly in A_Λ.

The analytic assembly can also be ordered using the operator '•':$\Gamma \times \Gamma \to \Gamma$, where $\Gamma = \Phi \cup \Theta$. Informally, '•' is a left-associative concatenation operator that expresses the temporal relation, 'before'. That is, $a_1 \bullet a_2$ means 'a_1 executes before a_2'. In this case, the analytic assembly would be represented as the string $A_\lambda = a_1 \bullet a_2 \bullet a_3 \bullet ... \bullet a_j$, for $j \geq 1$, where $a_1, a_2, a_3 ... a_j \in \Gamma$. This temporal ordering would be appropriate and necessary should A_Λ be extended to address non-steady state latency.

A.2.2 Consistency

Since Eq. 5 and Eq. 6 involve operations (Σ and max) that do not depend on the order of their arguments, all analytic assemblies produced using the interpretation in Section A.2.1 will be mathematically equivalent, regardless of the order in which components are considered.

QED

Appendix B Empirical Validity

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. Measure components.
2. Design assemblies with the components and predict the property of interest.
3. Build the assemblies and measure the property of interest.
4. Statistically analyze the results.

To empirically validate COMTEK-λ, we selected the Audio Playback (APB) application. This simple application was conducive to studying end-to-end latency, the property that we wanted to predict. APB is comprised of the following components:

- Audio Input—reads digital audio data from devices within the host operating system and makes that data available to other components. It has two outputs, left and right audio channels.
- Generator—generates a digital audio signal (frequency and amplitude). It has one output.
- Adder—produces a digital audio signal from two input signals (additive). It has two inputs and one output.
- Subtractor—produces a digital audio signal from two input signals (subtractive). It has two inputs and one output.
- Audio Out—outputs digital audio data to devices within the host operating system to produce sound. It has two inputs, left and right audio channels.
- Audio View—outputs a graphical representation of input digital audio data. It has one input.

The following sections describe how the experiments to empirically validate COMTEK-λ were conducted.

B.1 Measuring the Latency of Individual Components

Latency for an individual component is defined as the duration of time starting from the invocation of its `execute()` method to its return from that method (see Figure 13). The execute method is required by COMTEK-λ; it is the method in which the component performs its work.

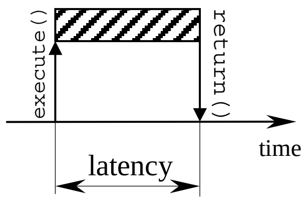


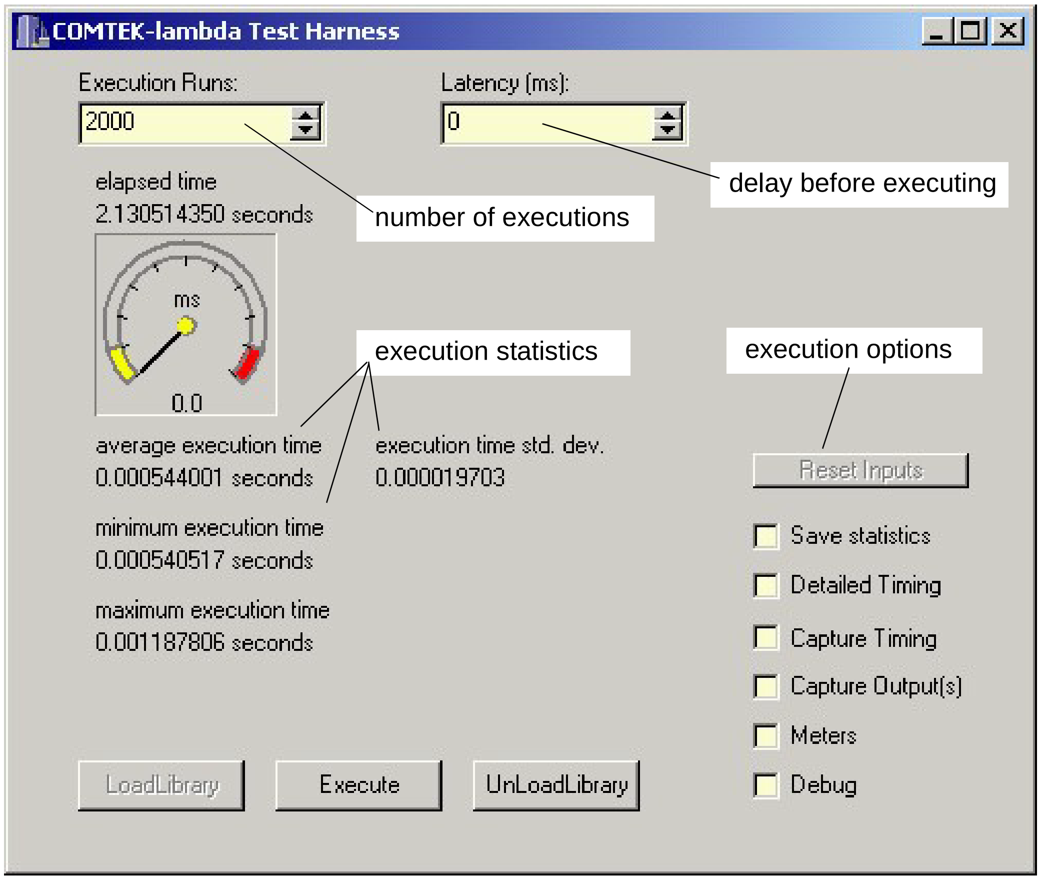
Figure 13: Latency of a COMTEK-λ *Component*

Measuring the latency of an individual component was not straightforward. First, a COMTEK-λ assembly cannot execute until it is "fully connected" (see Figure 7 on page 17). Which means, in short, that components in a COMTEK-λ application having required inputs ('InputPort') must be connected to a source ('OutputPort') before the COMTEK-λ executive will run the application. A COMTEK-λ component can have zero or more InputPorts and zero or more OutputPorts, but must have at least one port; it may not have zero of both. Therefore, we would have to have at least two components in a COMTEK-λ application to calculate the latency of the component we wanted to measure. Lastly, we wanted to measure the latency of a component in an external environment to better simulate the role of a component property certifier.

For these reasons, we created a test harness to measure the latency of single components. The test harness supplied the services of COMTEK-λ sufficient to execute the component, and no more. A useful analogy is an engine placed in a dynamometer to measure its performance. To run the test, we put oil in the engine, hook up a battery and fuel line (input), attach it to a wiring harness (for control), hook up a transmission and exhaust (output), and conduct the performance test. This is essentially what the COMTEK-λ test harness does for COMTEK-λ components.

The test harness, shown in Figure 14, supported loading and executing one COMTEK-λ component at a time. A component in the test harness could be executed any number of times while the test harness measured and recorded the time the component spent in its `execute()`

method. The test harness computed several parameters for a component depending on its settings, but the most useful were the average execution time and the standard deviation of the execution times.



Execution Runs: 2000

Latency (ms): 0

elapsed time
2.130514350 seconds

average execution time
0.000544001 seconds

execution time std. dev.
0.000019703

minimum execution time
0.000540517 seconds

maximum execution time
0.001187806 seconds

- [ ] Save statistics
- [ ] Detailed Timing
- [ ] Capture Timing
- [ ] Capture Output(s)
- [ ] Meters
- [ ] Debug

Figure 14: COMTEK-λ *Component Test Harness*

Input and output (I/O) from a component in the test harness were handled in precisely the same manner as I/O within the COMTEK-λ run-time environment, as memory streams. Because the test harness mimicked the precise behavior of the memory input and output streams, we not only established a suitable environment for testing components (with actual test data), we also achieved a very close approximation of the individual component latencies we would expect in the actual COMTEK-λ runtime.

Measuring the time spent in the `execute()` method gives the latency of a Θ component, but for Φ components we also need the period of the event that Φ components depend on. To measure the actual execution time of a Φ component, exclusive of any blocking, we inserted a timed delay that was longer than the external period before dispatching the `execute()`. In this way, only the time spent performing work was measured, not time waiting for the period to expire. The length of this timed delay can be set in the test harness, as can be seen in Figure 14.

In addition to the components that were used in the original APB application, we created variations of two of them to have a broader palette. One variation was a delayed adder that performed the same operation as the original adder but also had a forced delay that would make its execution take roughly 60ms. The other variation was a Φ version of the subtractor component. This new subtractor would depend on a periodic event of 80ms in addition to its 20ms required to perform its computation.

The COMTEK-λ framework is implemented on the Microsoft Windows™ platforms. We ran the empirical validation experiments in one workstation running Windows 2000 Professional operating on a DELL Dimension 4100 comprised of a Pentium™ III 1GHz processor, 512Mb of system RAM, ATI Rage Pro graphics adapter with 16Mb RAM, and a Creative Sound Blaster AudioPCI audio device. To minimize the effects of other processes on the measurements, the test harness sets itself to high priority while running the benchmarks.

Table 2 shows the results of the measurements for all the components. What we consider to be the certified latency of a component is the average of a large sample of measurements. In our empirical validation, we took 20,000 measurements for each component. We have included in Table 2 the maximum latency estimate error at 95% confidence. For example, for the component *Generator*, we can be 95% confident that the average of the sample (i.e., what we use as latency) is off by less than 9.61655E-07 from the true mean of the latency.

Table 2: Latency of Individual Components[a]

Component	**Execution (e) (in seconds)**			**Period (p) (in seconds)**		
	Average	**Std. Dev.**	**Max. error**	**Average**	**Std. Dev.**	**Max. error**
Audio Input	0.000161557	0.000371361	5.1468E-06	0.046409895	0.002502847	3.46877E-05
Generator	0.000250372	0.000069387	9.61655E-07			
Adder	0.000030612	0.000037586	5.20915E-07			
Subtractor	0.000030488	0.000037678	5.2219E-07			
Audio Out	0.000185987	0.000050028	6.93353E-07	0.046400214	0.009928901	0.000137608
Audio View	0.001159573	0.000206997	2.86883E-06			
Delayed Adder	0.060077160	0.000370087	5.12914E-06			
Dependent Subtractor	0.020026375	0.000174137	2.41342E-06	0.080112472	0.000530736	7.35563E-06

a. The latencies shown in the table are the average of 20,000 measurements taken for each component.

B.2 Predicting the Latency of an Assembly

Having measured all the components we were going to use, we were ready to predict the latency of the different assemblies that would constitute our empirical validation. In addition to the APB application shown in Figure 15, we created eight more assemblies. Figure 15 shows one APB assembly as it is displayed in the COMTEK-λ assembly environment.

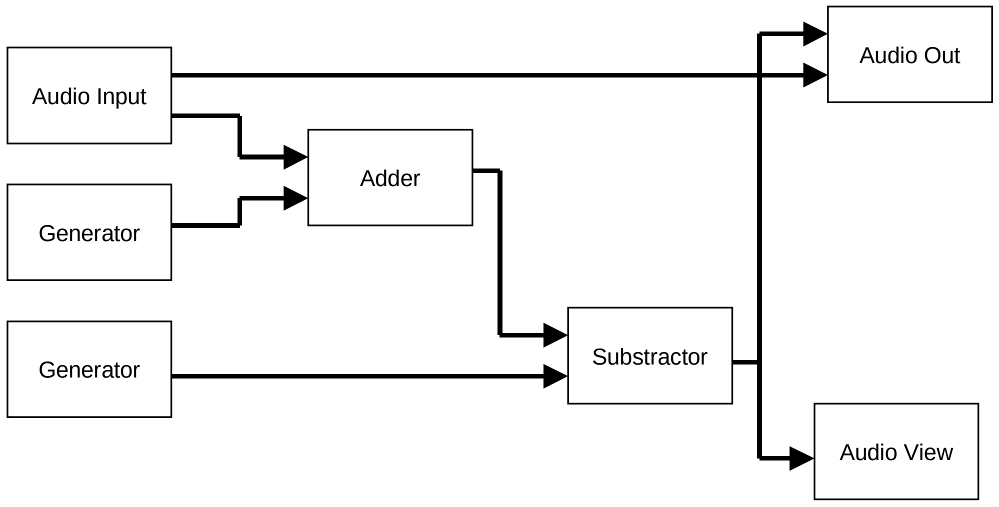


Figure 15: Audio Playback Application Assembly

For each assembly, we calculated the predicted latency using the analytic model (Eq. 6 on page 37):

$$A.\text{latency} = f(A) = \max\left(\sum_{C_j \in A} C_j.e, \max(\{C_j \in A \mid C_j.p\})\right) \qquad \text{Eq. 6}$$

Table 3 shows the components used in the APB assembly with their analytic interfaces. In the table, we have highlighted the sum of the execution times, and the largest period.

Table 3: Components Used in the APB Assembly

Component	Execution (e) (in seconds)	Period (p) (in seconds)
Audio Input	0.000161557	**0.046409895**
Generator 1	0.000250372	
Generator 2	0.000250372	
Adder	0.000030612	
Subtractor	0.000030488	
Audio Out	0.000185987	0.046400214
Audio View	0.001159573	
Sum	**0.002068961**	

Filling in the values in the previous formula, we obtained the predicted assembly latency:

$$A.\text{latency} = \max(0.00206896, 0.0464099) = 0.0464099$$

Following the same procedure for all the assemblies in our experiment, we obtained the predicted latencies shown in Table 4. The rest of the assemblies can be found in Appendix C.

B.3 Measuring the Latency of an Assembly

As described earlier, the COMTEK-λ framework pre-computes the schedule (order of execution) of components for a COMTEK-λ assembly at assembly time. This schedule is simply a fixed list; component X goes first, then Y, and so on until the last component is dispatched—at which time the scheduler reverts back to the first component in the ordered list and continues in a cyclic fashion. We defined assembly latency as the elapsed time between two consecutive invocations of the `execute()` method of some component $C_k \in A$.

There is a slight difference between this definition of measured assembly latency and the assembly latency as defined in the analytic model. Figure 16 depicts this difference. Since the COMTEK-λ assembly environment pre-computes the schedule, there is little or no scheduling overhead at run time. Therefore, we assume Δ to be negligible.

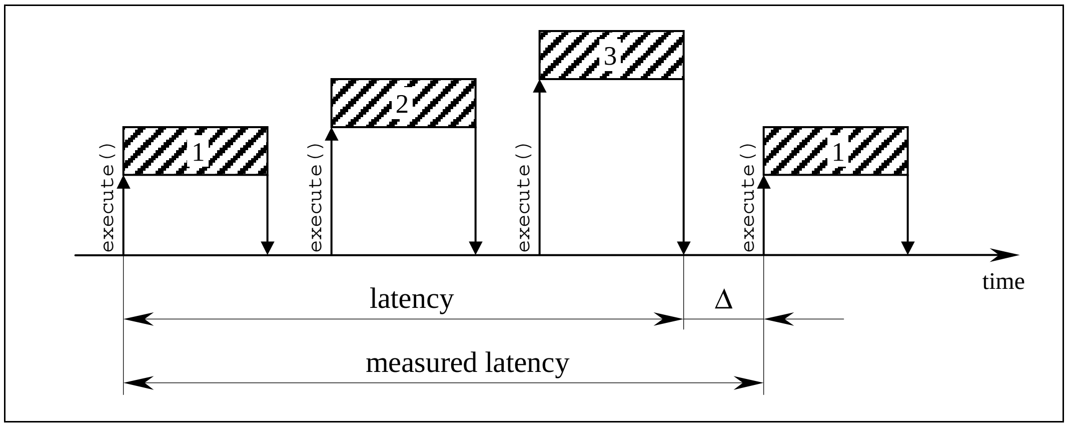


Figure 16: Measured vs. Theoretical End-to-End Latency for an Assembly

In addition to instrumenting the COMTEK-λ run-time environment to measure the cycle time of an assembly, we also instrumented it to compute and record the total number of cycles actually executed (i.e., number of samples taken) along with average and standard deviation of the cycle times. For both the test harness and the COMTEK-λ executive, the overhead of instrumenting and calculating component latency was taken into consideration when measuring A.λ and C.λ. Table 4 shows the measured latency for all the assemblies in the experiment.

Table 4: Predicted and Measured Latency for the Assemblies in the Experiment

Assembly	**Predicted Latency**	**Measured Latency**		
		Average	**Samples**	**Std. Dev.**
1	0.046409895	0.046432115	15000	0.005848142
2	0.080451079	0.080130336	15000	0.000491377
3	0.141687812	0.140204650	15000	0.000603960
4	0.060675076	0.060083619	15000	0.000278201
5	0.046400214	0.046401125	15000	0.009964861
6	0.046400214	0.046404150	15000	0.035717654
7	0.003131486	0.003206257	15000	0.000310752
8	0.046400214	0.046401195	15000	0.009883773
9	0.001440557	0.001442853	15000	0.000242908

B.4 Statistical Analysis of the Results

Although just by looking at the predicted and measured latencies in Table 4 we could see that the analytic model produced useful predictions, we needed to establish credibility in the model by backing it with statistical evidence. We used the magnitude of relative error (MRE) and correlation, two methods that have been used in the empirical validation of other software related estimation models [Kemerer+87].

MRE is a measure of the percentage error of the predicted latency, and it is defined as follows:

$$\text{MRE} = \left|\frac{A.\lambda - A.\lambda'}{A.\lambda'}\right| \qquad \text{Eq. 7}$$

where $A.\lambda$ is the predicted latency and $A.\lambda'$ is the measured latency. If we used absolute error, it would be difficult to compare the errors for different predictions. For example, an error of 0.3 ms would be relatively small for a latency of 140 ms while it would be very large for a latency of 1.44 ms. Therefore, we want to use relative error instead of absolute error so that we can compare and analyze the predictions for different assemblies together. In addition, using the absolute value of the relative error prevents errors with opposite signs to cancel out when the average of them is taken. Table 5 shows the MRE for all sample predictions.

Table 5: MRE for the Predicted Latency

Assembly	MRE	log MRE
1	0.000479	-3.320074412
2	0.004003	-2.397639777
3	0.010579	-1.975573828
4	0.009844	-2.006832907
5	0.000020	-4.707010133
6	0.000085	-4.071501732
7	0.023320	-1.632265142
8	0.000021	-4.674860158
9	0.001591	-2.798250203
Average	**0.005549**	**-3.064890**

For the nine assemblies in our experiment, the average magnitude of relative error was about 0.5%. To infer how the model would behave when used in other situations, we wanted to define a confidence interval for the expected value of MRE. In other words, we wanted to calculate a range within which the mean of all the MREs is most likely to fall. The statistical methods to do this required either that data be normally distributed or that the sample size be greater than or equal to 30. We only had a sample size of 9, so we checked whether the data was normally distributed or not.

We used the Shapiro-Wilk test for normality. The results were $W = 0.7621$ and *p-value* = 0.007482, where W is the statistic used in the test. The *p-value* is used to decide whether to reject or accept the null hypothesis that the data is normally distributed. When the *p-value* is less than or equal to the level of significance used, then the null hypothesis of normality is rejected. Using a level of significance[1] $\alpha = 0.05$, we concluded that the sample data was not normally distributed.

We applied a log transformation to the sample data intending to make it normally distributed. The values of the transformation are also shown in Table 5. We again used the Shapiro-Wilk test on the transformed values, with the result that $W = 0.9$ and *p-value* = 0.2522. Since the *p-value* was greater than the level of significance, we accepted the hypothesis of the transformed data being normally distributed. With this, we defined a 95% confidence interval for the expected value of the transformed MRE:

$$-3.976083 < \mathrm{E}(\log \mathrm{MRE}) < -2.1537$$

Applying anti-logarithm to revert the transformation, we obtained the confidence interval for the MRE:

$$0.000106 < \mathrm{E}(\mathrm{MRE}) < 0.007019$$

This means that there is a probability of 95% that the average MRE for all the predictions (i.e., the population of the predictions) will fall within this interval. To be clear, this does not mean that every single MRE value will fall within this range; we are estimating the average.

The second method we used for evaluating the empirical validity of this PECT was the correlation. We used the predicted latency as the independent variable, and the actual latency as the dependent variable. We obtained a *sample coefficient of determination* $R^2 = 0.999965721$, meaning that there is an almost perfect correlation between the predictions and the actual latencies of the assemblies. In other words, we can say that the analytic model accounts for 99.99% of the variations in the actual latency. Since the sample size we were using was not too large, we wanted to be sure that this correlation could not have happened by chance. The critical value of R for 7 degrees of freedom (N-2) at a 1% significance level ($\alpha = 0.01$) is 0.798. The correlation coefficient for our sample is $R = 0.99998286$, which is greater than the critical value. Therefore, we conclude that the correlation is statistically significant.

1. The significance level α is the probability of rejecting the null hypothesis when it is true.

In light of these results, we believe that the association between the constructive and analytic models in our exemplar PECT is valid. In addition, the fact that the analytic model explains or predicts 99.99% of the latency and that its predictions have a relative error less than 1% on average shows strong evidence that the predictions made by the analytic model are accurate.

Appendix C Assemblies Used for Empirical Validation

The following diagrams show the nine assemblies that were used for the empirical validation of this PECT.

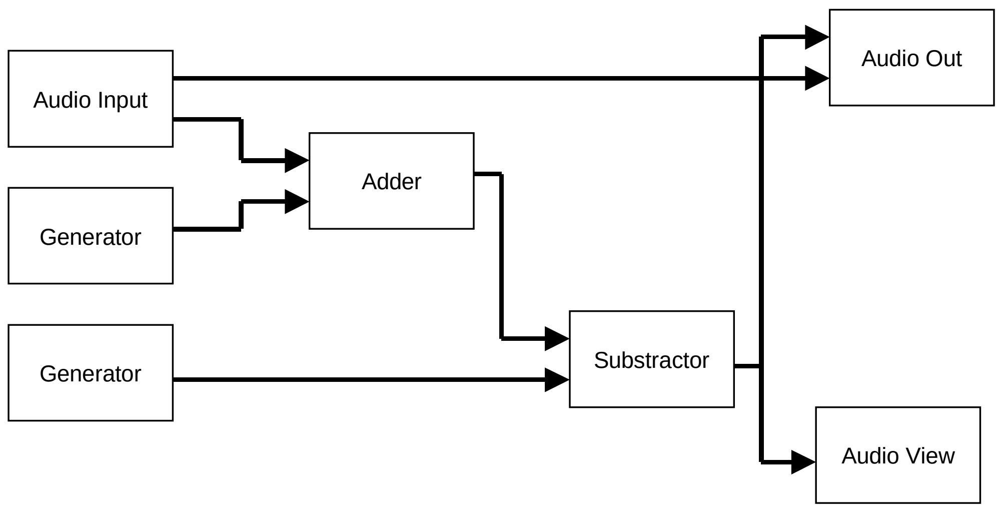


Figure 17: Assembly 1

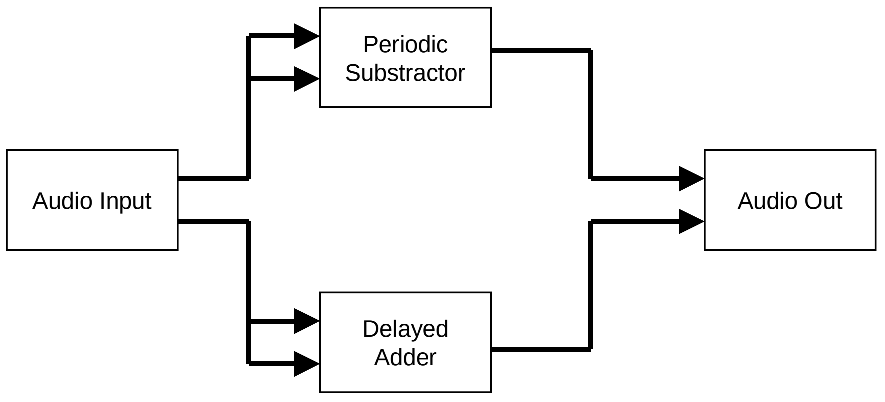


Figure 18: Assembly 2

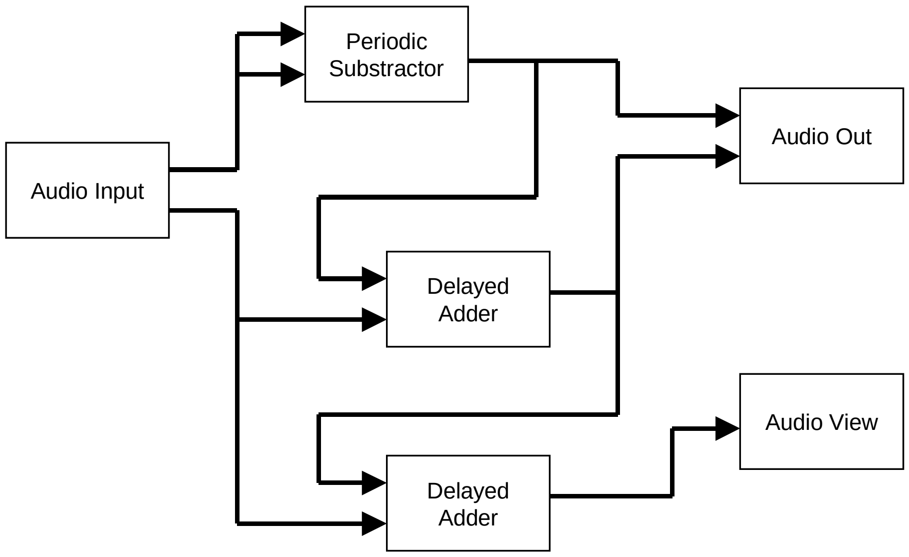


Figure 19: Assembly 3

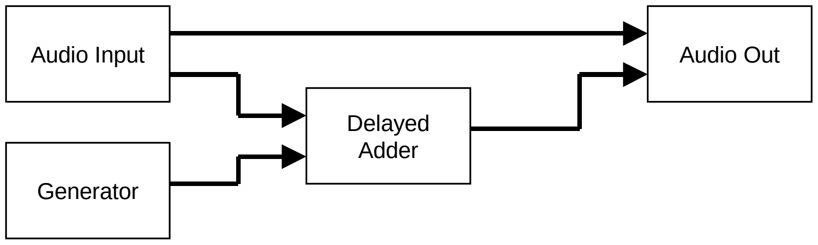


Figure 20: Assembly 4

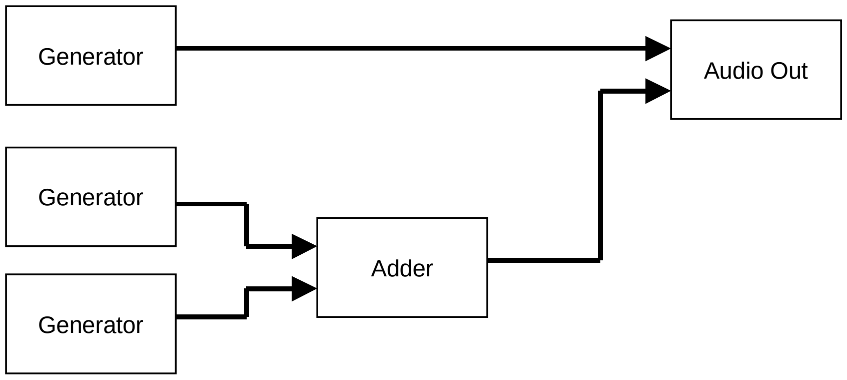


Figure 21: Assembly 5

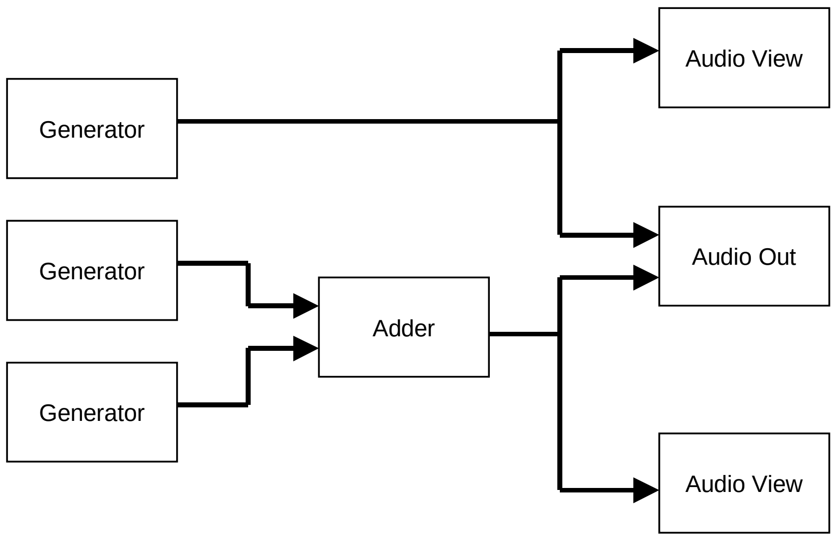


Figure 22: Assembly 6

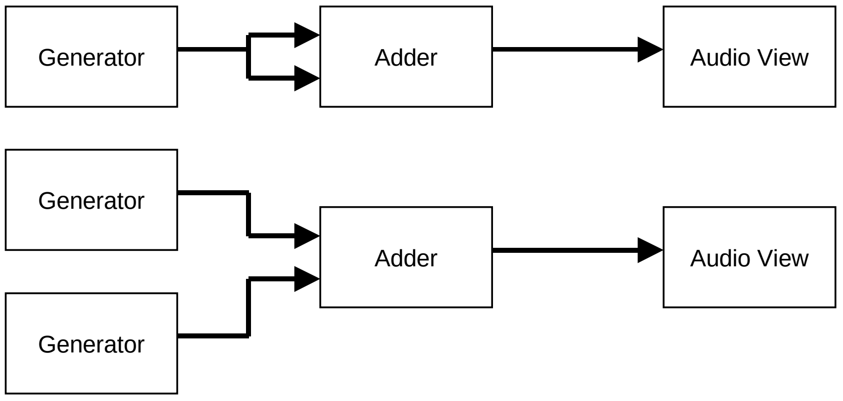


Figure 23: Assembly 7

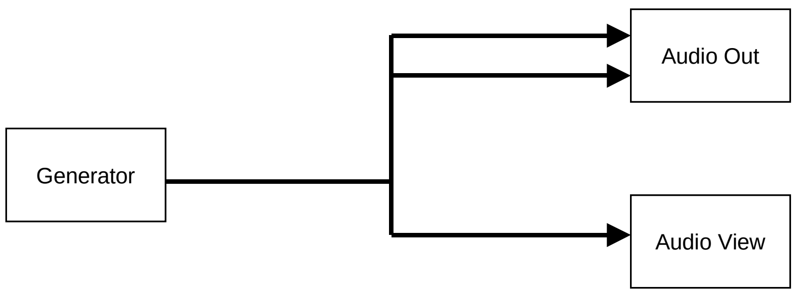


Figure 24: Assembly 8

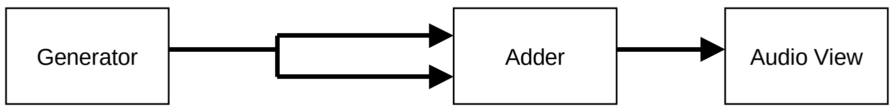


Figure 25: Assembly 9

References

[Bachmann+00] 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* (CMU/SEI-2000-TR-008), Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. URL: <http://www.sei.cmu.edu/publications/documents/00.reports/00tr008.html> (2000).

[Bass+00] Bass, L.; Klein, M.; & Bachmann, F. *Quality Attribute Design Primitives* (CMU/SEI-2000-TN-017, ADA392284), Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. URL: <http://www.sei.cmu.edu/publications/documents/00.reports/00tn017.html> (2000).

[Bass+01] Bass, L.; Buhman, C.; Comella-Dorda, S.; Long, F.; Robert, J.; Seacord, R.; & Wallnau, K. *Volume I: Market Assessment of Component-Based Software Engineering* (CMU/SEI-2001-TN-007), Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. URL: <http://www.sei.cmu.edu/publications/documents/01.reports/01tn007.html> (2001).

[Clarke+99] Clarke, E.; Grumberg, O.; & Peled, D. *Model Checking*, Cambridge, MA: MIT Press, 1999.

[Finkbeiner+01] Finkbeiner, B. & Kruger, I. “Using Message Sequence Charts for Component-Based Formal Verification,” in *Proceedings of the OOPSLA workshop on Specification and Verification of Component-Based Systems, Tampa, Florida, 2001*.

[Giannak+01] Giannakopoulou, D. & Havelund, K. “Runtime Analysis of Linear Temporal Logic Specifications,” in *Proceedings of the 16th IEEE International Conference on Automated Software Engineering, San Diego, California, 2001*.

[Heineman+01]	Heineman, G. & Council, W. *Component-Based Software Engineering Putting the Pieces Together*, Reading, MA: Addison-Wesley, 2001.
[Kemerer+87]	Kemerer, Chris F. "An Empirical Validation of Software Cost Estimation Models." *Communications of the ACM 30*, 5 (May 1987): 416-429.
[Klein+93]	Klein, M.; et al. *A Practitioner's Handbook for Real-Time Analysis*, Boston, MA: Kluwer Academic Publishers, 1993.
[Klein+99]	Klein, M. & Kazman, R. *Attribute-Based Architectural Styles*, (CMU/SEI-99-TR-022, ADA371802), Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. URL: <http://www.sei.cmu.edu/publications/documents/99.reports/99tr022/99tr022abstract.html> (1999).
[Lyu+96]	Lyu. M., ed. *Software Reliability Engineering*, Los Alamitos, CA, New York: IEEE Computer Society Press; McGraw-Hill, ISBN 0-07-039400-8, 1996.
[Messersch+01]	Messerschmitt, D. G. & Szyperski, C. "Industrial and Economic Properties of Software: Technology, Processes, and Value." University of California at Berkeley, Computer Science Division Technical Report UCB//CSD-01-1130, Jan. 18, 2001, and Microsoft Corporation Technical Report MSR-TR-2001-11, Jan. 18, 2001. URL: <http://divine.eecs.berkeley.edu/~messer/PAPERS/01/Software-econ> (2001).
[Plakosh+99]	Plakosh, D.; Smith, D.; & Wallnau, K. *Builder's Guide for WaterBeans Components* (CMU/SEI-99-TR-024, ADA373154). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. URL: <http://www.sei.cmu.edu/publications/documents/99.reports/99tr024/99tr024abstract.html> (1999).

[Rivera+96] Rivera, J.; Danylyszyn, A.; Weinstock, C.; Sha, L.; & Gagliardi, M. *An Architectural Description of the Simplex Architecture*, (CMU/SEI-96-TR-006, ADA307890), Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. URL: <http://www.sei.cmu.edu/publications/documents/96.reports/96.tr.006.html> (March 1996).

[Sha+95] Sha, L.; Rajkumar, R.; & Gagliardi, M. *A Software Architecture for Dependable and Evolvable Computing Systems* (CMU/SEI-95-TR-005, ADA301169), Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. URL: <http://www.sei.cmu.edu/publications/documents/95.reports/95.tr.005.html> (1995).

[Sharygina+01] Sharygina, N.; Browne, J.; & Kurshan, R. "A Formal Object-Oriented Analysis for Software Reliability: Design for Verification," in *Proceedings of the ACM European Conferences on Theory and Practice in Software, Fundamental Approaches to Software Engineering (FACE), Genova, Italy, April 2-6, 2001. URL:* <http://st72095.inf.tu-dresden.de:8080/fase2001> (2001).

[Shaw+97] Shaw, M. & Clements, P. "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems," in *Proceedings of COMPSAC, Washington, D.C., August 1997*.

[Simon+96] Simon, H. *The Sciences of the Artificial, 3rd Edition*, Cambridge, MA: MIT Press, 1996.

[Szyperski+97] Szyperski, C. *Component Software Beyond Object-Oriented Programming*, New York, Reading, MA: ACM Press, Addison-Wesley, 1997.

[Wallnau+01] Wallnau, K.; Stafford, J.; Hissam, S.; & Klein, M. "On the Relationship of Software Architecture to Software Component Technology." *Proceedings of the 6th International Workshop on Component-Oriented Programming (WCOP6)*. Budapest, Hungary, June, 2001.

[Walpole+89] Walpole, R.E. & Myers, R. H. *Probability and Statistics for Engineers and Scientists*. New York: MacMillan Publishing Company, 1989.

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. **AGENCY USE ONLY** (leave blank)	2. **REPORT DATE** November 2001	3. **REPORT TYPE AND DATES COVERED** Final
4. **TITLE AND SUBTITLE** Packaging Predictable Assembly with Prediction-Enabled Component Technology		5. **FUNDING NUMBERS** C — F19628-00-C-0003
6. **AUTHOR(S)** Scott A. Hissam, Gabriel A. Moreno, Judith Stafford, Kurt C. Wallnau		
7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-2001-TR-024
9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		10. **SPONSORING/MONITORING AGENCY REPORT NUMBER** ESC-TR-2001-024
11. **SUPPLEMENTARY NOTES**		
12.a **DISTRIBUTION/AVAILABILITY STATEMENT** Unclassified/Unlimited, DTIC, NTIS		12.b **DISTRIBUTION CODE**

13. **ABSTRACT** (maximum 200 words)

This report describes the use of prediction-enabled component technology (PECT) as a means of packaging predictable assembly as a deployable product. A PECT results from integrating a component technology with one or more analysis technologies. Analysis technologies allow analysis and prediction of assembly-level properties prior to component assembly, and, presumably, prior to component acquisition. Analysis technologies also identify required component properties and their certifiable descriptions. This report describes the major structures of a PECT. It then discusses the means of validating the predictive powers of a PECT so that consumers may obtain measurably bounded trust in design-time predictions. Last, it demonstrates the above concepts in a simple but illustrative model problem: predicting average end-to-end latency of a ‘soft’ real-time application built from off-the-shelf software components.

14. **SUBJECT TERMS** PECT, prediction-enabled component technology, analysis, analysis technology, prediction, component technology			15. **NUMBER OF PAGES** 68
			16. **PRICE CODE**
17. **SECURITY CLASSIFICATION OF REPORT** UNCLASSIFIED	18. **SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	19. **SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	20. **LIMITATION OF ABSTRACT** UL