Packaging Predictable Assembly

Scott A. Hissam, Gabriel A. Moreno,
Judith A. Stafford, Kurt C. Wallnau[1]

Software Engineering Institute
Carnegie Mellon University, Pittsburgh, PA, 15213

{shissam, gam, jas, kcw}@sei.cmu.edu
http://www.sei.cmu.edu/pacc

[1] Corresponding Author

Abstract. Significant economic and technical benefits accrue from the use of pre-existing and commercially available software components to develop new systems. However, challenges remain that, if not adequately addressed, will slow the adoption of software component technology. Chief among these are a lack of consumer trust in the quality of components, and a lack of trust in the quality of assemblies of components without extensive and expensive testing. This paper describes prediction-enabled component technology (PECT). A PECT results from integrating component technology with analysis models. An analysis model permits analysis and prediction of assembly-level properties prior to component composition, and, perhaps, prior to component acquisition. Analysis models also identify required component properties and their certifiable descriptions. Component technology supports and enforces the assumptions underlying analysis models; it also provides the medium for deploying PECT instances and PECT-compliant software components. This paper describes the structure of PECT. It discusses the means of establishing the predictive powers of a PECT so that consumers may obtain measurably bounded trust in both components and design-time predictions based on the use of these components. We demonstrate these ideas 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

Significant economic and technical benefits accrue from the use of pre-existing and commercially available software components to develop new systems. Variable component quality combined with their opacity require designers to rely upon extensive prototyping just to establish the feasibility of using a component in a particular assembly [1]. Many of the benefits of software component technology evaporate in the presence of high uncertainty and low consumer trust.

This paper 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 models. PECT is designed to enable predictable assembly from certifiable components. By this we mean:

- Assemblies of components are known, by construction, to be amenable to one or more analysis method for predicting their emergent (assembly level) properties.
- The component properties that are required to make these predictions are defined, certified, and trusted.

We therefore see component certification and predictable assembly as correlates.

Our underlying premise 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 [2][3], and it can also be applied to empirical analysis and prediction. It is a further premise of ours that software component technology is an effective way of packaging the design and implementation restrictions that yield analyzable designs, i.e., PECT.

Our research objective is to provide guidelines on how to construct PECT instances in different domains, and for different properties of interest. While research in component technology and software and system analysis continues, our focus is on how advances in these areas can be integrated and deployed. Our approach provides not only the architectural basis for this integration and deployment, but also criteria with which to empirically, statistically, and logically demonstrate the predictive effectiveness of PECT instances.

In the remainder of Sect. 1 we define the problem and our approach, and define key terminology used in this paper. In Sect. 2 we describe in some detail the PECT conceptual model. In Sect. 3 we give a brief illustration of these concepts using the COMTEK-λ prototype. Sect. 4 discusses related work, and Sect. 5 offers a few final thoughts.

1.1 Problem and Approach

Component technology, as it exists today, is more a marketplace phenomena than technology innovation. The major technical elements of component technology—for example, separate interfaces, multiple interfaces, encapsulation, and designed runtime environments—have been around for many years. What is significant about component technology is that IT producers and consumers have been rapidly adopting it in the form of Sun Microsystems' EJB™ and Microsoft's COM™ and many other "commercial off-the-shelf" component technologies [9].

Current generation component technologies address various syntactic and mechanistic aspects of component composition (*nee* integration), but more complex forms of behavioral composition have not been addressed. As software systems become more complex, and as their quality takes on greater social significance (safety, reliability, security, and so forth), the limitations of existing component technologies will (and already have) become manifest. Rely-guarantee reasoning is sufficient in a limited range of behavioral composition, but is not sufficiently expressive to be a general solution. And, of course, traditional testing has its own costs and limitations.

Our approach is to augment component technologies with sound analysis and prediction technologies. We refer to the resultant augmentation as a *prediction-enabled component technology* (PECT). Indeed, the marriage of component and analysis technology makes eminent sense:

- Analysis models are valid with respect to assumptions about the execution environment of an end application. For example, a performance model will likely depend upon assumptions pertaining to scheduling policy, process or thread priority, concurrency, resource management policies, and many other factors. These assumptions can be treated as design and implementation constraints, and made explicit, supported, and enforced by component technology. That is, assemblies of components can be rendered analyzable *by design and construction*.
- Analysis models refer to (are parameterized by) the properties of components being modeled. We refer to these as *analytic properties*, and the set of these as the component's *analytic interface*. An explicit, well-defined analytic interface provides an opportunity for certifying those component properties that support engineering analysis. This, in turn, provides a value proposition for certified, or *trusted*, component properties.
- Component technology is *par excellence* a means of packaging and deploying software technology. And, it is being adopted by industry. On a very practical level, we view component technology as a readily available distribution channel for packaging and deploying predictable assembly from certifiable components.

The last, and defining, element of our approach is that the packaging of a PECT is not complete until its predictive powers have been validated. Our objective is that each PECT be described by an objective, bounded confidence interval that is backed by mathematical and empirical evidence. We exclude from our purview any analysis technology that can not, in principle or practice, support such validation.

1.2 Assumed Terminology and Notation

We assume as background context for our research the existence of two distinct technologies: component technology and analysis technology. Here we define, without further elaboration, a number of terms that denote aspects of these technologies. We define only those terms most useful for our exposition; we make no effort at completeness. Our terminology is, in the main, consistent with that found in the software component technology literature [4][5][6]:

- *Software component*: An independently deployable and executable software implementation. Hereafter referred to as *component*.
- *Component model*: A specification of component types, allowable patterns of interactions among instances of these types (*components*), and between components and a component runtime environment.
- *Component runtime environment:* An execution environment that enforces aspects of the component model. The runtime plays a role analogous to that of an operating system, only at a higher level of abstraction.
- *Component assembly environment*: A development environment that provides services for component development, composition, and component and application deployment. The assembly environment may also provide assembly-time enforcement of the component model.
- *Component technology*: An integrated component model, component runtime environment, and component assembly environment.
- *Component assembly*: Noun—A set of components and their enabled interactions. Verb—To integrate a set of components, thereby enabling their runtime interaction.
- *Component property*: Something that is known and detectable about a component, denoted with ‘dot’ notation, e.g., c.p for component ‘c’ with property ‘p’. Can be a measurable quantity or a behavioral model such as a state machine.
- *Assembly property*: Also known as *emergent property.* Something that is known about an assembly, also denoted using dot notation. Note that we reject the idea that *emergent property* is synonymous with *unpredictable property.*

Our terminology for analysis technology is of our own invention. It reflects the context in which we use analysis technology, i.e., component technology:

- *Analysis model*: A definition of terms and concepts pertinent to a particular emergent property. The language used by specialists in a particular property domain. An analysis model is analogous to a component model; it defines conceptual components and their relations to one another.
- *Property theory*: A theory, expressed in terms consistent with an analysis model, that can be used to predict the values of an emergent property. Can be, for example, a closed form formula or a formal theory such as used in model checking [7]. A property theory is analogous to a component assembly; it is a configuration of terms and concepts governed by an analysis model.
- *Analytic property*: A component property that is required by (is a parameter to) a property theory. The set of these properties for a property theory is the component’s *analytic interface* for that theory.
- *Analytic assembly*: A component assembly interpreted in terms of a property theory. This could be a closed form formula, or some alternative view of the component assembly such as a state transition model or component and connector model.
- *Analysis environment*: An environment for computer-aided analysis of, and predictions about, analytic assemblies.

- *Analysis technology*: An integrated analysis model, analysis environment, and property theory(ies).

We use UML as our graphical notation, and UML object constraint language (OCL) to specify invariants. We assume the reader has some familiarity with this notation. We occasionally stray from the standard UML, but we do so only when limitations of UML require it; we are careful to highlight each such apostasy. References in the text to specific elements of diagrams are (*parenthesized and italicized*).

2 PECT Structure and Method

To give the reader an idea of what PECT is all about, we first describe the use PECT instances from the perspective of an application assembler (Sect. 2.1). We then turn to the methodological aspects of producing PECT instances, first by concentrating on the structure of a PECT instance (Sect. 2.2), and then to the validation of PECT instances (Sect. 2.3).

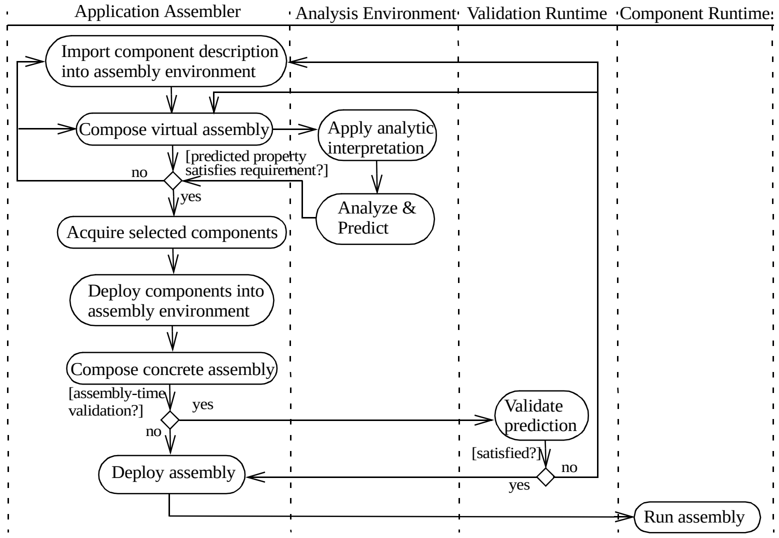

Application Assembler
Analysis Environment
Validation Runtime
Component Runtime
Import component description into assembly environment
Compose virtual assembly
Apply analytic interpretation
[predicted property satisfies requirement?]
no
yes
Analyze & Predict
Acquire selected components
Deploy components into assembly environment
Compose concrete assembly
[assembly-time validation?]
yes
no
Validate prediction
[satisfied?]
no
yes
Deploy assembly
Run assembly

Fig. 1: The user-level workflow for PECT. Note that the (*Validate prediction*) step is distinct from the validation of the PECT itself. In this workflow, validation refers to a spot check of the assembly against a prediction. The terms (*virtual assembly*), (*concrete assembly*), and (*analytic assembly*) are found in the conceptual schema, in Fig. 2. We have taken liberties with UML by permitting multiple ‘no’ paths on conditional branches.

2.1 PECT User-Level Overview

A PECT is an infrastructure for predictable assembly and trusted components. But how is this infrastructure used in practice? The answer to this depends upon which user role is considered. Fig. 1 specifies a simple view from the perspective of the application assembler. The workflow is certainly optimistic, since all paths lead to a running assembly. A more realistic workflow would include exit paths on failure, and paths to modify the PECT, perhaps by using an alternative property theory.

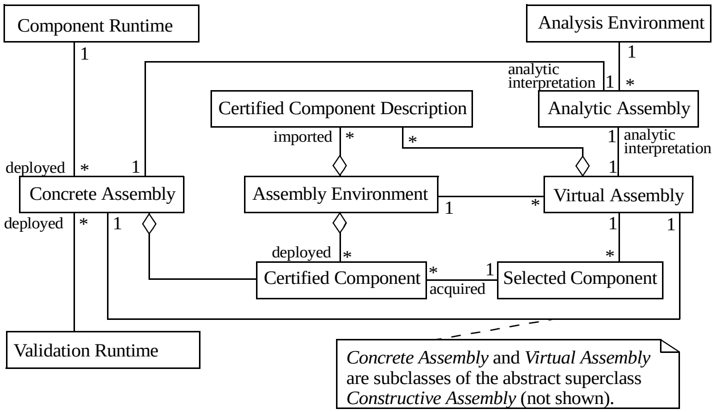


Fig. 2: The PECT conceptual schema identifies key terms and their relationships.

A conceptual schema in Fig. 2 identifies and relates terms found in the above workflow and used in the following discussion. The key new terminology is:

- *Certified Component Description*: A component can be *described* by its interface. Analytic interfaces are descriptive not normative: they describe the properties of a component; they do not specify the values that components must achieve. An interface description of components that includes analytic interfaces can be imported into an assembly environment and used for the purpose of assembly-time analysis and prediction.
- *Virtual Assembly* and *Concrete Assembly*: An assembly of component descriptions is sufficient for analysis and prediction, but provides no runtime behavior. A concrete assembly, on the other hand, consists entirely of components, and therefore has runtime behavior; naturally, it is also sufficient for analysis and prediction. Virtual and concrete assemblies are mapped to analytic assemblies via analytic interpretations. Analytic interpretations are discussed in Sect. 2.3 under *Model Validity.*
- *Validation Runtime*: The predictive power of a PECT is qualified by a statistical confidence interval or its formal equivalent. Nonetheless, an application engineer may wish to validate a PECT prediction. Doing so may require a specialized component infrastructure (i.e., a validation runtime) that is, for example, instrumented or based on an alternative runtime scheme. Assembly-time validation might be performed on a subset of assemblies, or as a ‘sanity check’ on a particular prediction.

2.2 Conceptual Structure of PECT Instances

A PECT arises from the association of an analysis technology with a component technology. Therefore, our description of the overall structure of PECT instances in Fig. 3 centers on the UML association class (*Predictable Assembly Model*).

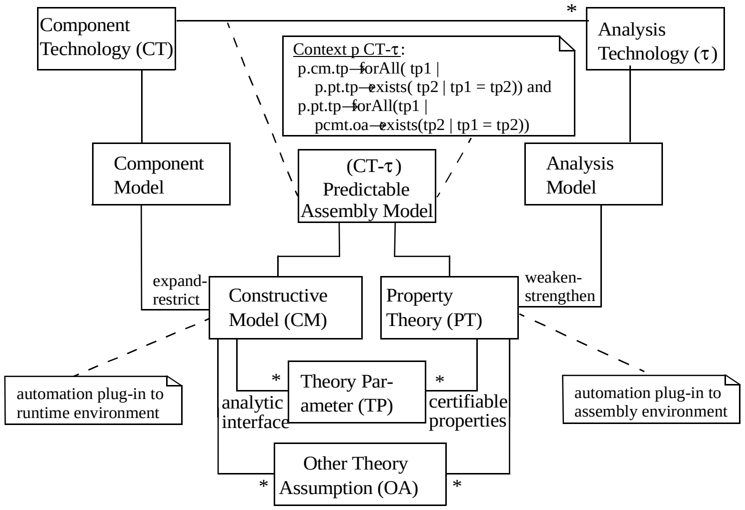


Fig. 3: The naming convention we use for PECT instances is to use the name of the component technology, e.g., (*CT*), and one or more symbols to denote analysis models, e.g., (τ). In this figure, role cardinalities are 1 unless otherwise specified.

The (*Constructive Model*) is the original component model plus any modifications that are required to specify the assumptions of a property theory. Component assemblies that are conformant to a constructive model are called *constructive assemblies*. The (*Concrete Assembly*) and (*Virtual Assembly*) in Fig. 2 are constructive assemblies. The OCL constraint (*Context p CT-t*) stipulates that each certifiable component property (*Theory Parameter*) is found in the analytic interface of the relevant constructive component types. An analogous constraint, not shown, is required for (*Other Theory Assumption*). These associations are confirmed during model validation (see Sect. 2.3).

2.3 Validation of PECT Instances

A technology that purports to enable predictable assembly would be meaningless if its predictions could not be validated. To paraphrase (and perhaps debauch) the wisdom of Wittgenstein: *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. This is provided by two distinct but related forms of validity: *model* and *empirical* validity.

Model Validity. Our concern is to establish the validity of the (*Predictable Assembly Model)* depicted in Fig. 3. This consists in establishing that two conditions hold:

1. The mapping of assumptions from a (*Property Theory)* to elements of the (*Constructive Model)*, again, in Fig. 3, is consistent.
2. The analytic interpretation from (*Constructive Assembly)* to (*Analytic Assembly)* in Fig. 2 is both consistent *and* complete.

The first condition, which we refer to as *logical validity*, is analogous to establishing the validity of a theorem in formal logic. That is, *a theorem is valid if the conclusions follow from the premises.* In PECT, the theorem is a property theory; its assumptions are the theorem's premises; and its predictions are the theorem's conclusions. Establishing logical validity involves demonstrative (mathematical) reasoning.

The second condition, which we refer to as *interpretation validity*, demonstrates that each constructive assembly can be interpreted in terms of the property theory (completeness), and each constructive assembly has at most one interpretation (consistency). This last is a bit subtle. It is possible for a single constructive assembly to correspond to several analytic assemblies, but these analytic assemblies must form an equivalence class with respect to predictions made under the property theory.

Empirical Validity. Returning to logical validity, we said that a property theory is valid if its predictions follow from its assumptions (conclusions follow from premises). Continuing with the analogy to formal logic, a theorem is *sound* if, in addition to being valid, *the premises hold true*. In PECT, establishing that the assumptions hold true amounts to demonstrating that each theory assumption is enforced by the PECT runtime or assembly environments, *or by engineering practices*.

Unfortunately, the soundness of a property theory can almost never be formally established. Modern computing environments are complex, and one can never be absolutely certain that a property theory has adequately enumerated all its assumptions. In practice, then, empirical evidence is required. That is, we must treat predictions as falsifiable hypotheses, and each failure to falsify the prediction, within explicitly stated measurement tolerance, incrementally adds evidence that the theory is sound. That is, we do not *demonstrate* that the assumptions of the property theory hold true, we *infer* their truth inductively from experimental evidence.

Of course, this is no more than asserting the need for using traditional scientific method to demonstrate the soundness of property theories. It seems that Simon's thoughts on the application of scientific methods to artificial systems, and to engineering design of complex systems in particular, remain ahead of their time [10].

2.4 Integration by 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 Where mismatches arise, either or both must be adjusted, as illustrated in Fig. 4.

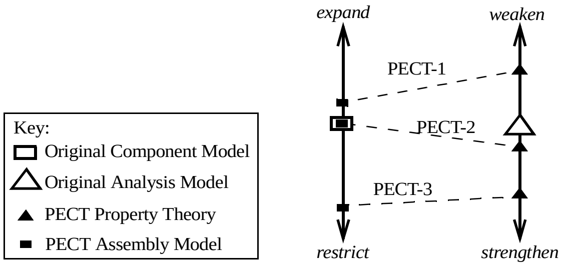


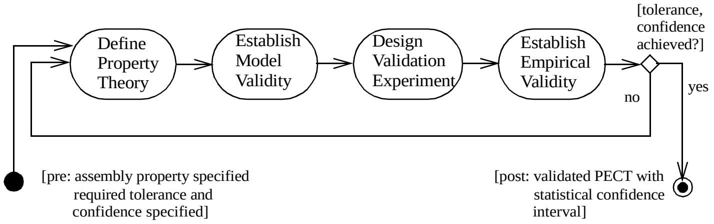
Fig. 4: Given a component model and analysis model, a PECT instance is produced through a process of co-refinement. Co-refinement (including a null refinement) transforms a component model to an assembly model, and an analysis model to a property theory.

Our idea of co-refinement is informal. Intuitively, expanding a component model removes or weakens design and implementation restrictions, and thus increases the set of allowable assemblies; restricting the component model has the opposite effect. Similarly, weakening an analysis model removes or weakens the assumptions (which can be thought of as preconditions) of property theories, again increasing the scope of the property theory to a larger set of assemblies, but perhaps at the cost of precision or reliability of predictions; strengthening an analysis model has the opposite effect. Three alternative co-refinements are depicted in Fig. 4, (*PECT-1*), (*PECT-2*) and (*PECT-3*). Each of these alternatives will exhibit different degrees of design freedom and expressiveness, and different degrees of predictive accuracy.

Of course, there are refinements of component and analysis models that do not fit neatly into the expand/restrict and weaken/strengthen dichotomies. For example, does modifying a component model to use asynchronous rather than synchronous communication expand or restrict set of assemblies that are conformant to the component model? At present, co-refinement appears to be the essential design problem of constructing a PECT instance—that is, it requires the use of judgment, experience, and taste on the part of the PECT designer to make the appropriate trade-off’s that arise when associating an analysis model with a component model.

2.5 Refinement and Validation Taken Together

Co-refinement and validation are major activities in the development of PECT instances. Although we do not yet have a complete workflow to describe PECT development, one thing is clear: these two activities are mutually reinforcing, and the final workflow will almost certainly include an approximation of the fragment in Fig. 5.



Define Property Theory
Establish Model Validity
Design Validation Experiment
Establish Empirical Validity
[tolerance, confidence achieved?]
no
yes
[pre: assembly property specified required tolerance and confidence specified]
[post: validated PECT with statistical confidence interval]

Fig. 5: A PECT development process will involve iteration among model validation, co-refinement, and empirical validation.

Although there are various intermediate steps missing in the workflow (such as implementing any required changes to the component runtime or assembly environments), the crucial point is that empirical validation is required to satisfy the guard condition for exiting the workflow. Our own experience suggests that empirical validation is effective at uncovering hidden, and therefore unsatisfied, assumptions of the property theory—that is, where these hidden assumptions are exposed by a violation of required measurement tolerance.

3 Illustration: COMTEK-λ

We now describe a prototype PECT, constructed from the COMTEK[1] component technology [11], and a widely known, if largely implicit, analysis model for predicting end-to-end latency. The objective of the prototype was to test the conceptual model of PECT and the PECT development process sketched in the previous section in a realistic but not overly complex problem setting.

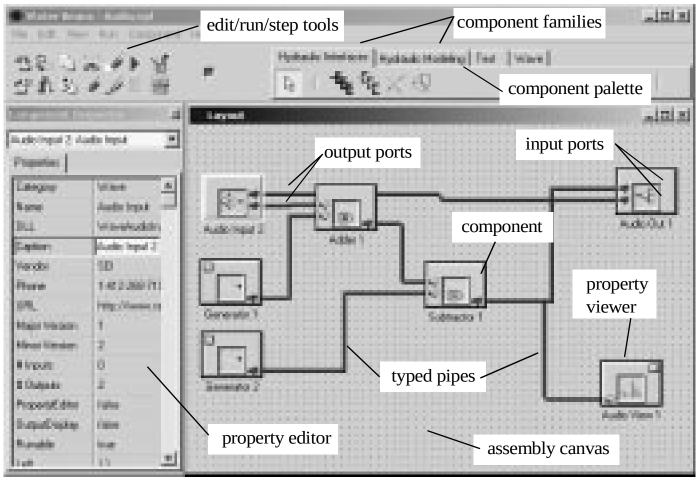
3.1 Component Technology: COMTEK

COMTEK was developed by the Software Engineering Institute (SEI) for the U.S. Environmental Protection Agency (EPA) Department of Water Quality to support end-user composition of water quality simulation from third-party simulation components. COMTEK runs on the Microsoft Windows-NT family of operating systems, and its

[1] Formerly known as *WaterBeans*.

components are deployed as Microsoft dynamic link libraries (DLL). Fig. 6 presents a screenshot of the COMTEK assembly environment. Despite its simplicity, the generality of COMTEK was demonstrated in several application domains.

The menu tabs above the assembly canvas in Fig. 6 display four families of components: Hydraulic Interfaces, Hydraulic Models, Wave Interfaces, and Test Interfaces. Fig. 6 shows an assembly built from components of the Wave Interface family. This and similar assemblies are the subject of the COMTEK-λ illustration. These assemblies implement audio signal sampling, manipulation, and playback functionality. We chose to develop a PECT for assembling audio playback applications since we could develop a simple performance analysis model to accommodate the relative simplicity of COMTEK.



Audio Input 2: Audio Input

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

Fig. 6: COMTEK has the following high-level characteristics: a) it enforces a typed pipe-and-filter composition style; b) components may have multiple input and output ports, each of which may be connected by a pipe to other component ports; c) a round-robin schedule is calculated from component input/output dependencies; d) the execution of an assembly is sequential, and components are single-threaded and non-preemptive.

3.2 Analysis Technology: Latency Prediction

The problem we posed was to predict the end-to-end latency of an assembly, 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. We required that predicted assembly latency is, on average, within +/ -10% of observed assembly latency. We permitted ourselves such liberality because WindowsNT provides few performance guarantees. As will be seen, however, we did much better than a 10% margin of error, although this was never our goal.

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 event is tolerable. In the audio playback application, audio signals received from an internal CD player must be sampled every 46 milliseconds for each 1,024 bytes of audio data. A failure to sample the input buffer, or to feed the output buffer 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.

3.3 COMTEK-λ Assembly Model

We begin with the more complex and extensive constructive model, and then turn to the property theory.

COMTEK-λ Constructive Model. The question of what information to include in a constructive model, and how to organize this information, is the same question we might ask of component models. We are not yet prepared to offer normative guidelines on either, and so the documentation we now describe should be interpreted as suggestive. With time and experience, we hope to propose a standard approach. For this relatively simple prototype, only a few views were needed to describe the essence of the constructive model. We use UML stereotypes (<<*component*>>) and (<<*analytic*>>) to distinguish those aspects of the specification that derive from the component model, and those that derive from the property theory, respectively.

Fig. 7 depicts the UML model of COMTEK-λ component metatypes. Instances of (*ComponentType*) are deployable units that are themselves instantiators—they provide runtime instances of themselves via (*getNewInstance()*). The property theory introduces two new component types, Φ and Θ, whose meaning is described later[1]. These types of components appear only in analytic assemblies, and have associated analytic constraints: they must possess specific properties that themselves are constrained, as specified in the pertinent OCL expressions.

[1] These symbols were chosen for their mnemonic value, as will be seen.

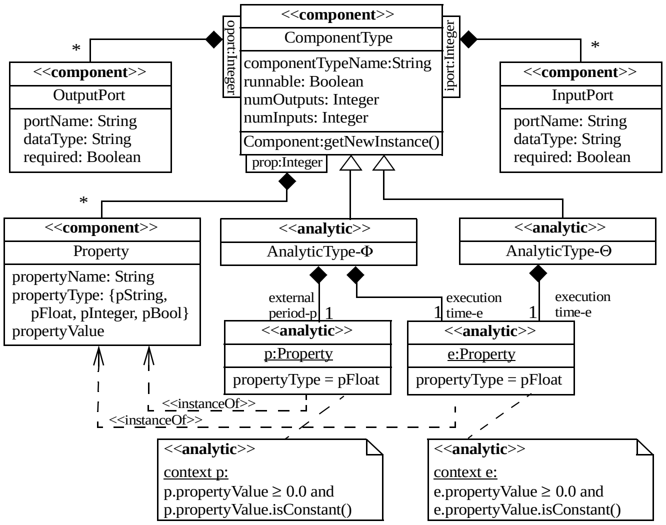


Fig. 7: COMTEK-λ Component Metatype Specification. The property theory introduces two analytic component types, (Φ) and (Θ). These subclasses of (*ComponentType*) have the additional constraint that their instances possess specific analytic component properties (*p*) and (*e*). This constraint is represented using non-standard UML.

Fig. 8 associates the definition of component latency used by the latency property theory with the COMTEK runtime[1]. Latency is defined as the time interval between two readings of a hypothetical, infinitely accurate clock. This definition was used for empirical validation. Fig. 8 also makes explicit the property theory assumptions relating to the execution schedule of components. Other assumptions, such as single threaded components and non-preemptive scheduling have been omitted for simplicity.

Property Theory. The COMTEK-λ latency theory is summarized by the following open formula:

$$A.\lambda \;=\; f(A) \;=\; \max\left(\left(\sum_{\Phi_j \in A} \Phi_j.e + \sum_{\Theta_j \in A} \Theta_j.e\right),\; \max(\{\Phi_j \in A \mid \Phi_j.p\})\right) \qquad (1)$$

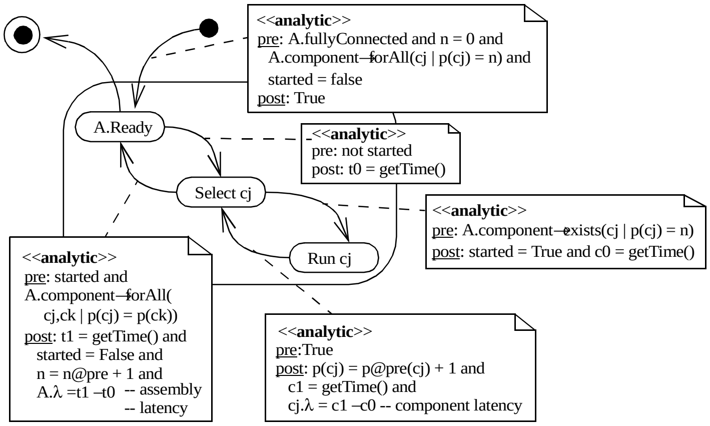
Assembly A is an enumerated set of components, and the kth component of A is denoted as either or Θ_k or Φ_k. These correspond to one of two analytic component types: Θrefers to components that only have dependencies that are internal to A, while Φrefers to components that also exhibit dependencies on external periodic events. A.λ

[1] Some of the terms used in Fig. 8 are introduced in views of the constructive model not discussed in this paper.

is the end-to-end latency of an assembly. Each Φcomponent has two required properties that describe its latency information: ‘Φ.e’ and ‘Φ.p,’ while each Θcomponent has only the required property Θ.e (refer to Fig. 7); e and p are defined as:

- *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 components of (*A*) in Eq. (1).



```
<<analytic>>
pre: A.fullyConnected and n = 0 and
  A.component→forAll(cj | p(cj) = n) and
  started = false
post: True
```

```
<<analytic>>
pre: not started
post: t0 = getTime()
```

```
<<analytic>>
pre: A.component→exists(cj | p(cj) = n)
post: started = True and c0 = getTime()
```

```
<<analytic>>
pre: started and
A.component→forAll(
    cj,ck | p(cj) = p(ck))
post: t1 = getTime() and
  started = False and
  n = n@pre + 1 and
  A.λ =t1 –t0  -- assembly
               -- latency
```

```
<<analytic>>
pre:True
post: p(cj) = p@pre(cj) + 1 and
  c1 = getTime() and
  cj.λ = c1 –c0 -- component latency
```

Fig. 8: Specification of Assembly Latency and Component Latency. The function (*p(ck)*) returns the number of times component ck has been run. The functions (*getTime()*) and (*setTime()*) read a hypothetical, infinitely accurate clock.

3.4 PECT Validity

The details of COMTEK-λ model and empirical validation are available in [12]. Here, we outline only the most important aspects of these validation exercises.

Model Validity. Our approach to constructing COMTEK-λ was driven by expediency. Accordingly, our strategy was to adopt as far as possible the design scheme represented by (*PECT-2*) in Fig. 4. To do so, we used the COMTEK component model as given assumptions for a custom fit latency theory. Following this approach was not especially demanding and worked well. Logical validity was demonstrated by a semi-rigorous derivation of the latency property theory from our knowledge of the internals of COMTEK. Interpretation validity was trivial to establish, mostly because the property theory was strengthened to deal just with steady state latency, and this, in turn, meant that order of component execution was irrelevant to latency prediction.

Nevertheless, our first "valid" property theory was falsified when we attempted to establish empirical validity. As detailed as our understanding of COMTEK was, we still manage to err by failing to appreciate the distinction between the $\Phi$and Θcomponents of Eq. (1). In fact, this distinction never mattered until the latency property theory was introduced, which accounts for our error. We were pleased to discover the effectiveness of a combined model and empirical validation process, and at the same time confirm that property theories might impose constraints beyond those of an existing component technology, and that these constraints would be relevant *only* to that property theory.

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. Obtain analytic properties of components (for example, through measurement).
2. Design validation assemblies and predict the assembly property of interest.
3. Construct the assemblies and observe their properties.
4. Statistically analyze the difference between predicted and observed properties.

We constructed a component benchmarking environment for (1) and instrumented component runtime for (3). The first turned out to be non-trivial since it was required to simulate but not re-implement the COMTEK runtime. We used statistical methods for two different purposes: latency measurement (1) (3), and quantification of accuracy and repeatability of the predictions (4). The sources of our statistical approach are [13][14][15]. We consider a large sample of measured latencies and use their mean as the value to be used.

For statistical analysis of the predicted latency (4) we used both descriptive and inferential statistics, namely correlation analysis, and confidence and tolerance intervals of the magnitude of relative error (MRE). We used nine sample assemblies as the basis for statistical analysis of our latency theory (2). Thus, in the summaries in Tables 2 and 3, (N) refers to the number of distinct assemblies that we tested, i.e., ($N = 9$). As we show, this small sample size does not cause difficulties with the confidence interval, but its flaws are revealed in the tolerance interval.

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

Table 1: Correlation Analysis Results

Statistic	Meaning
R2 = 0.9999657	Coefficient of determination
α = 0.01	Significance level

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

We use tolerance intervals for statistical inference. Three types of questions are addressed by tolerance intervals [14]:

1. What interval will contain (*p*) percent of the population?
2. What interval guarantees that (*p*) percent of the population will not fall below a lower limit?
3. What interval guarantees that (*p*) percent of the population will not exceed an upper limit?

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

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

Table 2: Second MRE Tolerance Interval

N = 30	sample size
γ = 0.95	confidence level
p = 0.90	proportion
μ_{MRE} = 1.99%	over 30 assemblies
UB = 6.33 %	upper bound

4 Related Work

Related Work. Compositional reasoning techniques are a natural foundation upon which to build PECT property theories. Fisler and Sharygina exploit the idea of design restrictions to ameliorate the state space explosion associated with compositional model checking [17][3]. Neither of these approaches, however, address components as independently deployable implementations. Analysis algorithms that have been developed for architecture description languages (ADLs) are also relevant, not surprisingly given their use of "component and connector" abstractions. ADL based analysis models address liveness and safety [18][19], and performance [20][21]. However, ADLs treat components as abstractions, not implementations. The difference between interface specification and description causes difficulty in applying ADL-based results to component technology. Also related is work in component certification and trust. Representative is the use of pre/post conditions on component interfaces [22]. This approach does support compositional reasoning, but only about a restricted range of properties. Quality attributes, such as security, performance, availability, and so forth, are beyond the reach of these assertion languages. Commercial ventures in component certification, such as specified by Underwriter's Laboratory (UL), lack empirical validation or compositionality; but these may nonetheless prove influential [23]. Voas has defined rigorous mathematical models of component reliability derived from testing [24], but he does not provide an assembly model nor any means of demonstrating the empirical validity of the resultant reliability properties. Hamlet attacks the problem of empirical and compositional theories of reliability, but his approach is far too restricted and microscopic to be of practical utility [25].

5 Conclusion

We have described and illustrated the key ideas of prediction-enabled component technology. Our approach asserts that component certification and assembly-level prediction are correlates. PECT emphasizes the technical affinities between component technology and analysis technology to enforce design rules imposed by analysis models. This enforcement will lead to systems that are, by design and construction, analyzable and predictable, and to trusted components, where this trust is bound to specific engineering predictions. While we have only made initial and tentative steps to develop PECT ideas, the results are encouraging.

References

1. Wallnau, K., Hissam, S., Seacord, R., *Building Systems from Commercial Components*, Addison-Wesley, July, 2001.
2. 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.

3. 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).
4. 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).
5. Heineman, G. & Council, W. Component-Based Software Engineering Putting the Pieces Together, Reading, MA: Addison-Wesley, 2001.
6. Szyperski, C. Component Software Beyond Object-Oriented Programming, New York, Reading, MA: ACM Press, Addison-Wesley, 1997.
7. E. M. Clarke, E.A. Emerson, and A.P. Sistla, “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications,” ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, April 1986, pp. 244-263.
8. Klein, M., Ralya, T., Pollak, W., Obenza, R., A Practitioner’s Handbook for Real-Time Analysis, Boston, MA: Kluwer Academic Publishers, 1993.
9. Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau, Volume I: Market Assessment of Component-Based Software Engineering Assessments (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).
10. H. Simon. The Sciences of the Artificial, 3rd Edition, Cambridge, MA: MIT Press, 1996.
11. D. Plakosh, D. Smith and K. Wallnau. 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).
12. S. Hissam, G. Moreno, J. Stafford, K.Wallnau,.Packaging Predictable Assembly with Prediction-Enabled Component Technology (CMU/SEI-2001-TR-024). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. URL: <http://www.sei.cmu.edu/publications/documents/01.reports/01tr024.html> (2001)
13. C.F. Kemerer. “An Empirical Validation of Software Cost Estimation Models,” in Communications of the ACM 30, 5, May 1987: 416-429.
14. NIST, NIST/SEMATECH Engineering Statistics Internet Handbook, National Institute of Standards and Technology (NIST), online at <http://www.itl.nist.gov/div898/handbook/>.
15. R.E. Walpole and R. H. Myers. Probability and Statistics for Engineers and Scientists. New York: MacMillan Publishing Company, 1989

16. Astley, M., Sturman, D., and Agha, G., “Customizable Middleware for Modular Distributed Software,” in Communications of the ACM, Vol. 44, No. 5, May 2001, pp. 99-107.

17. K. Fisler, S. Krishnamurthi and D. Batory. “Verifying Component-Based Collaboration Designs,” in Proceedings of the Fifth ICSE Workshop on Component-Based Software Engineering, Toronto, Canada, May 2001: 84-87

18. R. Allen and D. Garlan. “A Formal Basis for Architectural Connection,” ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 3, Jul. 1997, pp. 213-249.

19. J. Magee, J. Kramer, and D. Giannakopoulou, “Analysing the Behaviour of Distributed Software Architectures: A Case Study,” Proceedings, Fifth IEEE Workshop on Future Trends of Distributed Computing Systems, Oct. 1997, 240-247.

20. S. Balsamo, P. Inverardi and C. Mangano, "An Approach to Performance Evaluation of Software Architectures", Proceedings of the 1998 Workshop on Software and Performance, Oct. 1998, 77—84.

21. B. Spitznagel, D. Garlan, “Architecture-Based Performance Analysis,” Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering, San Francisco, California, 1998.

22. B. Meyer, Object-Oriented Software Construction, Second Edition, Prentice Hall, London, 1997.

23. Underwriter Laboratories, UL-1998, UL Standard for Safety for Software in Programmable Components, Northbrook, IL, 1998.

24. Jeffrey Voas, Jeffery Payne, “Dependability certification of software components,” in the Journal of Systems and Software, no. 52, 165-172, 2000

25. D. Hamlet, D. Mason and D. Woit. “Theory of Software Reliability Based on Components,” in Proceedings of the 23rd International Conference on Software Engineering, Toronto, Canada, 2001: 361-370.