



Carnegie Mellon
Software Engineering Institute

Volume III: A Technology for Predictable Assembly from Certifiable Components

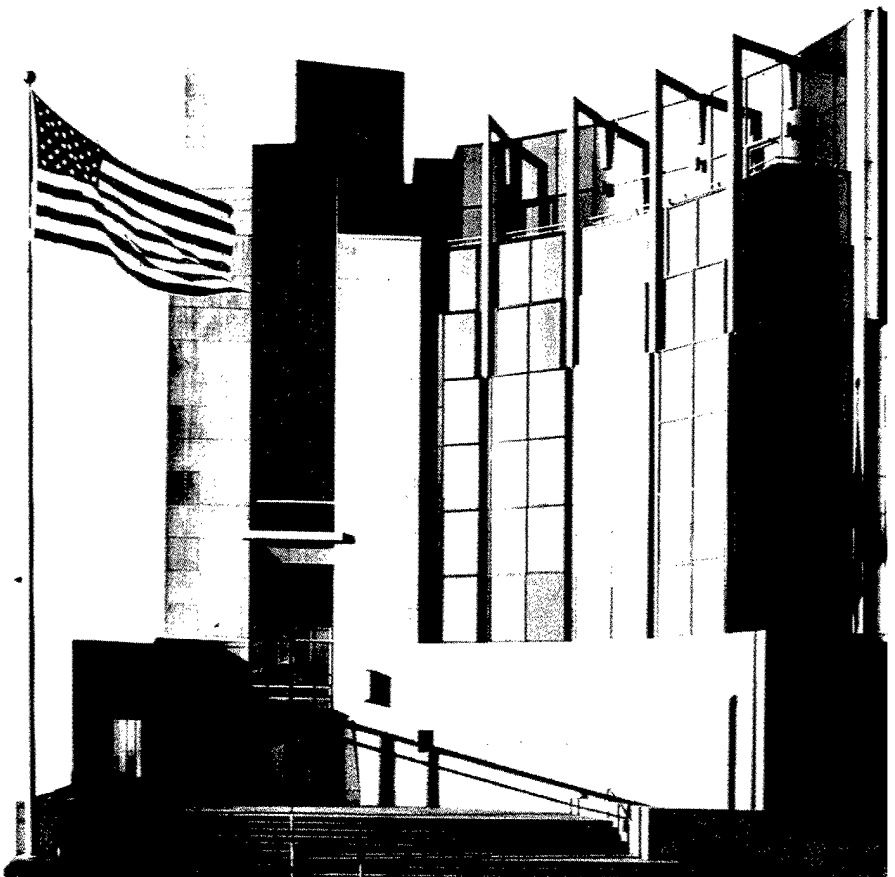
Kurt C. Wallnau

April 2003

20030519 013

TECHNICAL REPORT
CMU/SEI-2003-TR-009
ESC-TR-2003-009

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited





Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Volume III: A Technology for Predictable Assembly from Certifiable Components

CMU/SEI-2003-TR-009
ESC-TR-2003-009

Kurt C. Wallnau

April 2003

**Predictable Assembly from Certifiable Components
Initiative**

Unlimited distribution subject to the copyright.

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
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



Christos Scodras
Chief of Programs, XPK

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.

Copyright 2003 by Carnegie Mellon University.

Requests for permission to reproduce this document or to prepare derivative works of this document should be addressed to the SEI Licensing Agent.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (<http://www.sei.cmu.edu/publications/pubweb.html>).

Table of Contents

Acknowledgements	vii
Abstract	ix
1 Introduction	1
1.1 About This Series	1
1.2 About This Report	3
2 Motivation for PECT	7
3 Construction Framework	13
3.1 Components	13
3.2 Reactions	15
3.3 Interactions	16
3.4 Runtime Environment	17
3.5 Assemblies	19
3.6 Assembly Constraints	22
3.7 Properties	24
3.8 Construction Language	26
4 Reasoning Frameworks	29
4.1 Property Theory	30
4.2 Automated Reasoning Procedure	31
4.3 Validation Procedure	31
4.4 Illustration 1: Temporal Logic Model Checking	33
4.5 Illustration 2: Rate Monotonic Analysis	37
4.6 Illustration 3: n-Version Majority Voting Analysis	40
5 Concise Summary of PECT	43
6 Multiple Reasoning Frameworks	45
6.1 A Few Formal Definitions	46
6.2 Simplistic Versus Realistic Property Theories	47

6.3	Optimizing Qualities of Reasoning Frameworks	49
6.4	Incompatibility Among Reasoning Frameworks	52
6.5	Dealing with Incompatibility	55
7	Compositional Reasoning	59
7.1	Compositional and Modular Reasoning	59
7.2	Why Compositionality Is Too Strong	61
7.3	Why Compositionality Is Important	62
8	Status and Future Work	65
9	Conclusions	67
	Glossary.....	69
	Acronym List.....	75
	Bibliography	77

List of Figures

Figure 1: The Fundamental Premise of PACC	2
Figure 2: Constructive and Analytic Well-Formedness of Assemblies	8
Figure 3: Logical Structure of Prediction-Enabled Component Technology	10
Figure 4: Components, Labels, and Constructive Interface	14
Figure 5: Component Interface with Pins and Reactions	16
Figure 6: Enabled Interaction Between Two Components	17
Figure 7: Environment Services and Containment	18
Figure 8: Connectors Provided by Hypothetical Environment	19
Figure 9: A Simple Controller Assembly	20
Figure 10: Hierarchy via Gateways	21
Figure 11: Hierarchy via Partial Assembly	22
Figure 12: Extending the Well-Formedness Rules of an ACT	23
Figure 13: Property Annotations on Components	25
Figure 14: Annotations of Assembly and Component Properties	25
Figure 15: Component Specification in CCL	27
Figure 16: An Interpretation for Model Checking	35
Figure 17: An Interpretation for RMA Schedulability Analysis	39
Figure 18: An Interpretation for Reliable Voter Pattern Reliability Analysis	42
Figure 19: UML Class Diagram of PECT Concepts	43
Figure 20: Compatibility of Analytically Well-Formed Assemblies	45
Figure 21: Time Sequence of a Series of Co-Refinement Steps	51

Figure 22: Interference Among Property Theories 53

Figure 23: Interference Between Security and Reliability Property Theories 55

List of Tables

Table 1:	Selected (Analytic) Assumptions of Reliability Theory	47
Table 2:	Plausible Satisfiability of NVV Well-Formedness Constraints	48

Acknowledgements

This report reflects the contributions of many people. Judith Stafford played an early and instrumental part in defining the basic structure of prediction-enabled component technology (PECT). James Ivers provided continuous and instrumental feedback on the numerous drafts of this report. Daniel Plakosh's WaterBeans prototype provided an exemplar of pure composition, and he defined the earliest versions of the Pin component model. Len Bass, Mark Klein, and Paul Clements helped to solidify the connections between construction model and software architecture, and especially between construction model and analyzability. Scott Hissam, Mark Klein, Magnus Larsson, and Gabriel Moreno developed proofs of feasibility of PECT for substation automation, and Otto Preiss provided essential substation domain knowledge. John Hudak, James Ivers, and Bill Wood introduced model checking for safety and liveness to augment empirical predictions of time, while James Ivers and Nishant Sinha defined CL, a syntax and formal compositional semantics for Pin. Natasha Sharygina helped clarify the distinctions between formal and empirical theories. Linda Northrop provided nourishment and encouragement despite false starts and backtracking. This work would not have been possible without the encouragement of John Goodenough and Steve Cross.

Abstract

This report is the final volume in a three-volume series on component-based software engineering. Volumes I and II identified market conditions and technical concepts of component-based software technology, respectively. Volume III (this report) focuses on how component technology can be extended to achieve predictable assembly from certifiable components (PACC). An assembly of software components is predictable if its runtime behavior can be predicted from the properties of its components and their patterns of interactions. A component is certifiable if its (predictive) properties can be objectively measured or otherwise verified by independent third parties. This report identifies the key technical concepts of PACC, with an emphasis on the theory of prediction-enabled component technology (PECT).

1 Introduction

This report describes one means of achieving predictable assembly from certifiable components (PACC). An assembly of software components is predictable if its runtime behavior can be predicted from the properties of its components and their patterns of interactions. A component is certifiable if its (predictive) properties can be measured or verified by independent third parties. In our context, component and assembly properties are objective and susceptible to rigorous empirical and/or formal verification.

The goal of the Software Engineering Institute (SEISM)¹ PACC Initiative, which was launched in 2002, is to develop and transition the engineering methods and tools necessary to reliably predict the behavior of assemblies of components, and to certify the properties of components necessary to trust these predictions.

The SEI's approach to PACC is prediction-enabled component technology (PECT). A PECT is a development infrastructure that guarantees that critical runtime properties of assemblies of components are objectively analyzable and predictable. A PECT comprises a component technology and one or more analysis technologies. Composition tools ensure that component assemblies satisfy analytic assumptions, thus ensuring that assemblies are predictable by construction.

1.1 About This Series

This report is the third—and final—volume of a three-volume series that documents the results of an internal research and development (IR&D) activity of the SEI. This IR&D involved two phases of exploratory research, spanning three years of effort.

The first phase of the IR&D, carried out in 1999-2000, examined the broad outlines of software component technology. In particular, this phase

- surveyed, in Volume I, market conditions underlying industry adoption of software component technology [Bass 01]
- identified, in Volume II, technical concepts of software component technology, with an emphasis on application development infrastructure [Bachmann 00]

1. SEI is a service mark of Carnegie Mellon University.

These reports concluded that successful adoption of software component technology was inhibited by (1) the indeterminate (and often poor) quality of software components and (2) the lack of engineering techniques to predict the behavior of assemblies of components. Combined, these problems result in a heavy reliance on rapid prototyping in place of analysis and design, and on expensive integration testing and its concomitant late discovery of defects. Regardless of prototyping and test results, there was a reported expectation that component-based solutions involve decreases in system quality and increases in project risk.

The second phase of the IR&D, carried out in 2000-2002, was premised on the idea that the specification of components and the prediction of assembly behavior are codependent. In short, the properties of software components that must be trusted should be precisely those properties that support reasoning about assembly behavior. If supported by technology and appropriate business models, this codependency might form the basis of a virtuous cycle between component certifiability and assembly predictability, with advances in one area stimulating advances in the other. An emblematic representation of this premise, on which PACC is predicated, is depicted in Figure 1.

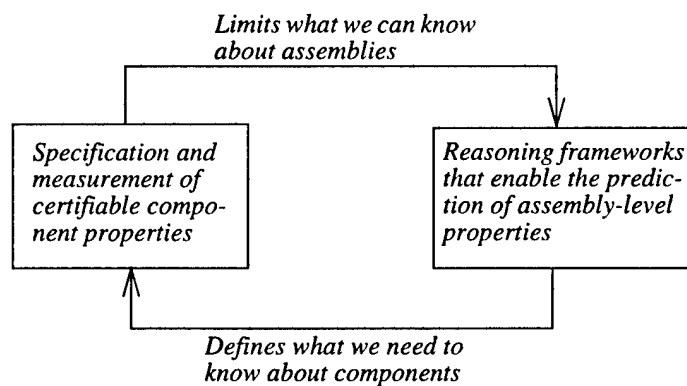


Figure 1: The Fundamental Premise of PACC

The objective of the SEI’s PACC research during 2000-2002 was to define a technological and methodological basis for PACC, specifically linking predictability and certifiability. Examined chronologically

- In 2000-2001, the major technological and methodological elements of PECT were defined and demonstrated in a simple laboratory experiment, as described in the paper titled “Packaging Predictable Assembly” [Hissam 02a].
- In 2001-2002, with an industry partner, the SEI applied PECT to a more realistic application area—power grid substation automation—as described in *Predictable Assembly of Substation Automation Systems: An Experiment Report* [Hissam 02b].

On October 1, 2002, PACC was established as an emerging initiative at the SEI. This report describes the key technical ideas underlying PACC and PECT, in effect defining an agenda for applied research and technology transition for this new initiative.

1.2 About This Report

In principle, this report could describe PACC in the abstract and then motivate PECT within this abstract context. However, for reasons of expediency, no attempt is made to find a bright line separating PACC from PECT. Instead, the key concepts of PACC are made material in the form of PECT. A positive aspect of this approach is that this report provides a basis for understanding the SEI's approach to PACC. A negative aspect is that the report is undeniably parochial.

Objectives

The primary objective of this report is to outline the key concepts of PECT. The intent is to describe these concepts in a rigorous way without sacrificing their intuitive appeal. This report provides a theoretical and technological basis to answer the following questions:

- What characteristics of an assembly make it predictable, and what kinds of assembly properties can be predicted?
- What characteristics of a component make it certifiable, and what kinds of component properties can be certified?
- How can we achieve objective and measurable confidence in certified component properties and predicted assembly properties?
- Can a technology infrastructure that provides answers to these questions be systematically developed and transitioned into practice?

Answers to these four questions are provided in Chapter 9.

It should be noted that this report describes work in progress. It is definitive of the SEI's approach in some cases, and speculative in others. Another objective of this report, then, is to generate constructive criticism of our ideas by exposing them to the scrutiny of the software engineering and computer science research communities, and to interested practitioners.

Caveats

The following caveats should be kept in mind when reading this report:

- This report does not dwell on fundamental philosophical issues of predictability, but rather stakes out positions that define our approach to PACC; however, pointers are provided to the literature of philosophy of science where appropriate.
- This report also does not provide a detailed survey of alternative approaches to PACC; however, pointers are provided to competing or augmenting ideas.
- This report is not comprehensive in its treatment of the theory and technology required for PACC. In particular, this report focuses on predictable assembly and leaves a detailed treatment of certifiable components to a later report.

- This report is not a primer on software component technology, language semantics, formal analysis, empirical analysis, measurement, or certification. Rather, this report addresses their integration to form a PECT.

Audience

This report is intended for a technical audience interested in predictable assembly and component certification in general, but more specifically in the SEI's approach to these topics. It is assumed that the reader has at least surface knowledge of software architecture, formal verification, and measurement theory. Some familiarity with real-time analysis techniques, fault tolerance, and software reliability would be helpful, but is not required.

This report makes occasional use of the Unified Modeling Language (UML) [Booch 99] and the UML Object Constraint Language (OCL) [Warmer 99]; for example, Figure 12 on page 23. Some familiarity with UML and OCL is useful, but because only their simple features are used in this report, no special expertise is required.

Relation to Other Reports

Familiarity with Volume I of this series is not essential [Bass 01], but familiarity with Volume II is assumed [Bachmann 00].² A straightforward application of the concepts presented in this report to a simple model problem can be found in the paper *Packaging Predictable Assembly* [Hissam 02a]; indeed, this may be considered a companion report to the present volume. A more comprehensive illustration of the ideas to a nontrivial application area can be found in *Predictable Assembly of Substation Automation Systems: An Experiment Report* [Hissam 02b], which introduces the notation and terminology of a composition language, CL, and its underlying construction model, Pin. A formal treatment of CL can be found in *A Basis for Composition Language CL* [Ivers 02], but is not required to appreciate the main points of this report.

All three volumes document, in varying degrees of rigor, the essentials of the SEI's view of PACC and our thinking about PECT. Various workshop papers are also available, some of which address topics not given adequate attention in this report [Stafford 01a], [Stafford 01b], [Wallnau 01], [Stafford 02], [Moreno 02], [Li 02]. Although those papers provide insight into PACC and PECT, they are by intent speculative. Naturally, some speculations have fared the test of time better than others. In short, *caveat emptor*.³

2. There are some inconsistencies in terminology between Volume II and this report. For example, "component framework" in Volume II is (less ambiguously) denoted as "component runtime environment" in this report. However, such inconsistencies are minor and should not pose difficulties to the reader.

Typographic Conventions

The first defining occurrence of terms that can be found in the glossary are underlined. Formal notations are *italicized*. Example specifications appear in `courier`. Important points are highlighted in **boldface**. Footnotes are used extensively and in two ways: (1) for additional explanation of ideas that may be obvious to most but not all readers, and (2) for ideas that will be of interest to some readers, but otherwise digress from the main flow of the report.

Structure of This Report

The report is organized in three main parts.

Part 1, comprising Chapters 2 through 5, focuses on the technological aspects of PECT. Chapter 2 motivates the main ideas. Chapter 3 describes construction frameworks, which constitute the “component technology” aspect of a PECT. Chapter 4 describes reasoning frameworks, which constitute the “prediction-enabled” aspect of a PECT. Chapter 5 provides a concise summary of how construction and reasoning frameworks are combined to form a PECT.

Part 2, comprising Chapters 6 and 7, focuses on theoretical and, in places, speculative aspects of PECT. Chapter 6 deals with different types of relationships that arise among reasoning frameworks during the construction and use of a PECT. Chapter 7 deals with the crucial topic of compositional reasoning, and, in particular, distinguishes compositional reasoning from reasoning about compositions.

Part 3, comprising Chapters 8 and 9, closes the report. Chapter 8 describes the current status of PECT and areas where further development is planned. Chapter 9 answers the four motivating questions posed earlier.

-
3. Let the buyer beware.

2 Motivation for PECT

Component technologies are not new, and neither are the technologies that are used to specify and reason about the behavior of software systems. What is new in PECT is the conscious design of component technology to enable automated and trustworthy analysis and prediction of system behavior.

All component technologies are designed to achieve specific goals. Microsoft's COM is designed to support independent deployment of components through a separation of interface and implementation, and to permit the use of different programming languages to develop components [Box 98]. Sun Microsystems' Enterprise JavaBeans (EJB) is designed to support quick development and deployment of distributed, secure, transactional business information systems. Other component technologies have their own design goals.

The key to a component technology is its component model. A component model, in effect, imposes design and implementation rules on component developers and application integrators (assemblers). To date, component models have focused mainly on the construction aspects of development—application programming interfaces (APIs), memory management conventions, concurrency management conventions, component and application deployment processes, and so forth. These aspects may all be thought of as imposing **constructive constraints** on developers; if the constraints are satisfied, an assembly can be constructed, that is, its components can be compiled and linked, integrated, deployed, and so forth.

Volume II of this series observed that component models share characteristics with architectural styles.⁴ Both define component types,⁵ patterns of interaction, and other design constraints. The significance of this observation is that a system's properties⁶ (e.g., reliability and performance) correlate strongly to its architectural structure; a style is in essence a structural

-
4. Shaw and Garlan provided one of the earliest and most extensive treatments of architectural style [Shaw 96b]. The sense of "style" adopted in this report and in Volume II (viz. component types and their interaction patterns) follows the book titled *Software Architecture in Practice* [Bass 98]. More recently, Clements et al. bring much-needed order to architecture-related terminology [Clements 02b].
 5. EJB, for example, defines different types of enterprise "beans" for handling stateless/stateful sessions and data entities. Analogously, there is a rich hierarchy of COM component types, specified as a hierarchy of interface specifications.

pattern [Bass 98]. In particular, Klein and associates showed that the assumptions of certain qualitative and quantitative theories for analyzing quality attributes can be expressed using design patterns such as styles [Kazman 99], [Bachmann 02], what we call “property theories.” In effect, these patterns define **analytic constraints** that, if satisfied, ensure that the design will be analyzable in the constraining property theory, and the behavior of the resulting system will therefore be predictable.

A good way to understand the complementary roles of constructive and analytic constraints is to think of a component model as a language that defines well-formedness rules for components and their assemblies. Figure 2 depicts this mode of thinking and introduces some terminology.

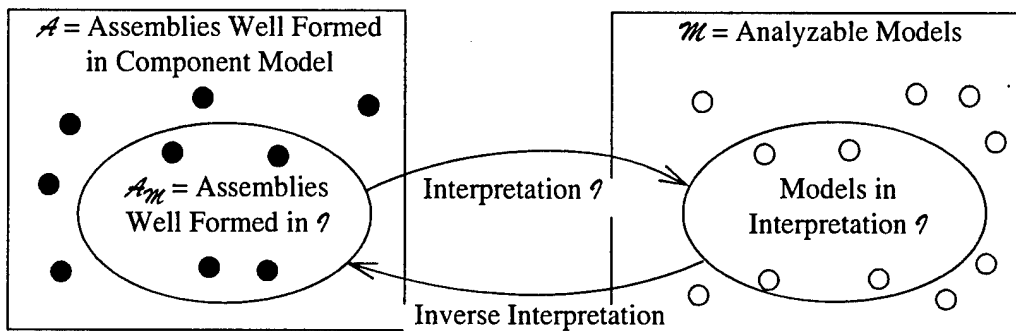


Figure 2: *Constructive and Analytic Well-Formedness of Assemblies*

The box on the left in Figure 2 represents the universe of all assemblies that are well formed with respect to the constraints of some component model. The box on the right represents the universe of all models \mathcal{M} in some property theory—a theory that can be used to predict some kind of assembly behavior. The validity of \mathcal{M} rests on some assumptions about the systems it models; for example, resource management policies. These assumptions are the analytic constraints imposed by \mathcal{M} 's property theory. $A_{\mathcal{M}}$ is the set of all assembly specifications that satisfy these analytic constraints **in addition to** satisfying the constraints of the component model. An interpretation \mathcal{I} maps assembly specifications to analyzable models in the property theory; to relate the results of analysis back to the original assembly specification, the inverse

-
6. The term “quality attribute” is used in most SEI literature on software architecture. The more generic term “property” is used in this report. The emphasis in PECT is on runtime quality attributes only (e.g., availability instead of modifiability), and attributes that have objective rather than subjective definition (e.g., latency instead of usability). Of course, these distinctions can be debated—a system that is unable to perform a runtime reconfiguration might become unavailable, and the usability of an interface can be studied using experimental subjects. Nonetheless, these distinctions have practical significance.

interpretation \mathcal{T}^{-1} is defined. Each assembly specification in \mathcal{A}_M is analyzable and predictable, with respect to the property theory underlying \mathcal{M} .

This discussion spotlights two fundamental theses underlying the theory of PECT:

1. A component technology imposes constructive and analytic constraints, and provides tools and environments that enforce these constraints. As a result, **component assemblies have predictable behavior by construction.**
2. Interpretations are defined to component and assembly specifications for analyzable models of assembly behavior. As a result, **component properties⁷ required for predictability are unambiguously defined, establishing a basis for trust and certification.**

In more practical terms, the above theses declare that rather than trying to predict the behavior of arbitrary assemblies of components (e.g., \mathcal{A}), we should construct only those assemblies whose behavior is predictable (e.g., \mathcal{A}_M). Further, rather than defining subjective notions of component quality, we should rigorously define those component properties that have meaning in some validated, predictive theory.

So far, the discussion has focused on the theory of PECTs rather than the technology itself. Before shifting to a more concrete focus on the latter, it is worth emphasizing that it is not our objective to develop PECT insofar as this implies a single component technology and a particular suite of property theories. That objective would be untenable for at least two substantial reasons. First, there will never be a “one size for all” component technology—as noted earlier, the key concepts of component technology can be applied to solve widely varying problems, ranging from embedded real-time to distributed enterprise systems. Second, and for an analogous reason, there will never be a “one size for all” property theory.

For these (and other) reasons, it makes little sense for the SEI to define PECT theory in terms of a particular component technology and suite of property theories. Moreover, many software development organizations will be similarly motivated to preserve some conceptual distance between the theory of PECT and the specific software component and analysis technologies

7. This is yet another use of the term “property.” Earlier, the term was used to denote the subset of quality attributes that are manifested at runtime and objectively observable; they are the properties that are the subject of prediction. Here, the term “property” denotes things that are known rather than predicted about a component; these properties are the subject of certification. It might be argued that it is better to have two terms to denote these two concepts. In Section 3.7, however, a single (formal) definition will be given to “property” that will accommodate both concepts.

used to realize the concept. This motivation provides the rationale for the generic structure of PECT that is described in this report and illustrated in Figure 3.

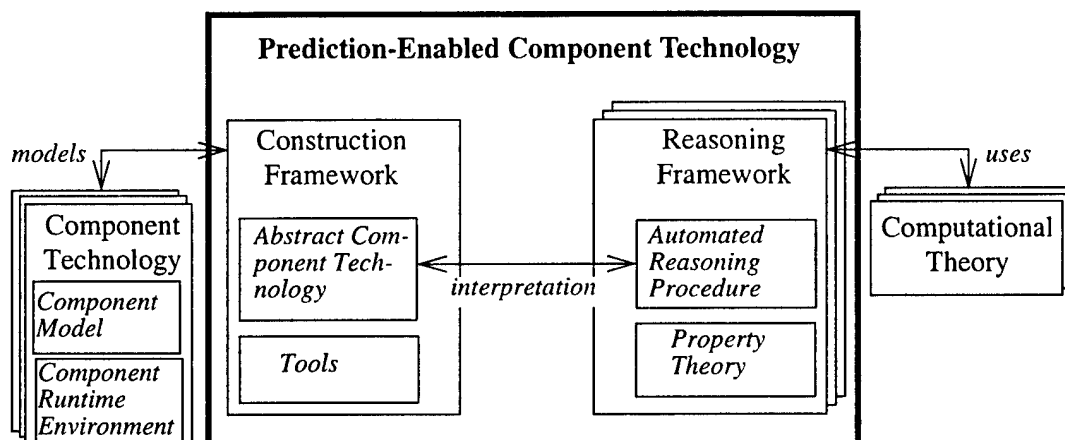


Figure 3: Logical Structure of Prediction-Enabled Component Technology

A PECT has two key ingredients, a **construction framework** and one or more **reasoning frameworks**. Each reasoning framework is linked to the construction framework by means of an **interpretation**.

The construction framework supports the construction activities of component-based software development. The **abstract component technology** is a proxy for one or more component technologies. It defines a conceptual vocabulary and notations for specifying components, assemblies, and their runtime environments⁸ in a component-technology-independent way; it also specifies the properties, imposed by reasoning frameworks, that must hold on these component technologies for predictions to be valid. Tools provide automation support for construction activities such as writing assembly specifications, checking well-formedness of assemblies, and generating code.

Reasoning frameworks support the prediction activities of component-based software development; they encapsulate the property theories mentioned earlier. A **property theory** is a proxy for, and is likely to be a specialization of, some computational theory. It defines a property-specific conceptual vocabulary and notation for reasoning about, and predicting, the behavior of assemblies of components. Each property theory has an associated **automated**

8. Recall that *Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition* defined a component technology to include, among other things, a component model and runtime environment, where the term "component framework" was used in place of "component runtime" [Bachmann 00]. The role of the runtime environment in an ACT is discussed in Chapter 3.

reasoning procedure. Automation is stressed not as its own end, but as a practical means of ensuring that a property theory is explicit and computationally tractable.

Last, interpretations map specifications in the “construction world” to and from specifications in the “analysis world.” Interpretations are formally defined; they are complete and consistent translations from the notations defined by construction and reasoning frameworks. Again, formal rigor and automation is stressed not as its own end, but as a practical means of ensuring that there is a sound correlation between construction and analysis.

3 Construction Framework

An abstract component technology (ACT) defines a vocabulary and notation for specifying components, assemblies, and their runtime environments in a component-technology-independent way, and for specifying the constraints, imposed by reasoning frameworks, that must be satisfied for predictions to be valid. A construction framework is an ACT and the tools that support its use (e.g., editors, constraint checkers, repositories). This chapter focuses on ACT.

The following discussion serves two purposes. The first is descriptive: it explains the essential concepts and defines their terminology and notation. The second is prescriptive: it strongly suggests what is required of a concrete component technology if it is to be a suitable foundation for predictable assembly.

3.1 Components

Components are the building blocks of predictable assembly, although, as you will see, a substantial amount of component substructure must be exposed in the interest of certifiability and predictability. Components

- are implementations in final form, modulo binding labels
- provide an interface for third-party composition
- are units of independent deployment

The term implementation distinguishes components from design abstractions in software architecture and architecture description languages [Bass 98], [Clements 96], which also use component as a primitive concept. The term final form means that the implementation is delivered in a form ready to be executed rather than as source code. This is more general than the term “binary form” [Bachmann 00], [Szyperski 97], although it conveys the same idea.⁹

The term binding label refers to linking mechanisms embedded in components to enable their interaction with other components.¹⁰ The interface of a component includes (among other

9. Language interpreters and “just-in-time” compilation blur but do not eliminate the usefulness of this distinction.

10. Similar but not necessarily equivalent terminology is used in the literature; for example, ports and provide/require interfaces.

things) a set of publicly defined binding labels; these labels are used by some composition mechanisms to “bind” the labels of one component to those of another, and therefore enable the components’ runtime interaction through those bindings. “Public” definition of these labels allows third parties to compose components on them.

The term “units of independent deployment” is quite subtle, more than is apparent at first glance,¹¹ with the result that it is quite difficult to propose a completely satisfactory definition. For the purpose of this report, it is sufficient to emphasize one particular aspect of independent deployment: a component is a unit of independent deployment if all its dependencies on external resources are clearly specified, and if it can conceivably be a substitute for, or substituted by, some other component.

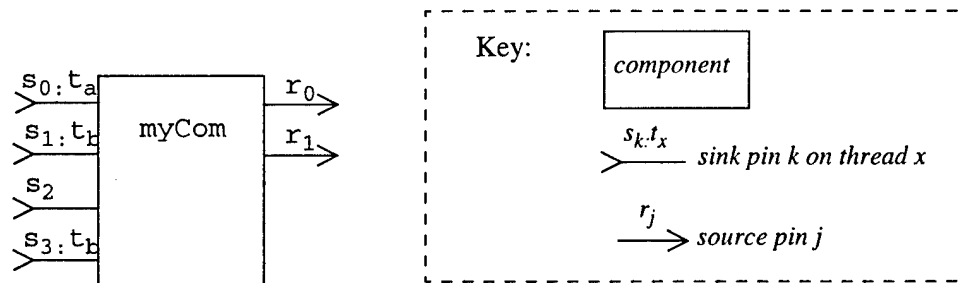


Figure 4: Components, Labels, and Constructive Interface

Figure 4 introduces terminology and graphical conventions used in this and other SEI reports on predictable assembly that generally adopt the “box and line” convention. A component is depicted as a box. A label within a box denotes the name of the component. Decorating the boundary of the component is one or more binding labels that are depicted as incoming and outgoing arrows, called “pins.”

Each pin has an associated label that denotes its name. Roughly speaking, incoming pins—called sink pins—denote incoming events to a component, and outgoing pins—called source pins—denote outgoing events, although they may represent other interaction mechanisms as well, such as procedure calls.¹² It is useful to make explicit concurrent behavior in components. Therefore, sink pins also indicate whether they execute on their own thread of control, or on their caller’s thread; threads may be shared by sink pins within a component but not across components.¹³ Here “thread” does not denote a particular implementation concept, but rather any unit of concurrent execution in the component technology.

11. For example, code (component) mobility is a special case of deployment. See Cardelli’s work for cross-fertilizing ideas on it [Cardelli 98].

12. The sink/source distinction mirrors the notions of provides/requires interfaces found elsewhere; for example, Van Ommering’s work [van Ommering 02].

A component defines a naming scope for (among other things) pins. We use dot notation to scope names. In Figure 4 for example, $c.r_0$ denotes pin r_0 of component c . However, fully scoped pin names are used only where ambiguity would otherwise result.

3.2 Reactions

Clearly, predicting the runtime behavior of assemblies of components requires that we know something about the runtime behavior of the components themselves. This requires additional specification mechanisms beyond pins, and certainly beyond what passes for interface specification in the vernacular of software developers—the API. The behavior of components is specified as reactions. A reaction specifies the behavioral dependencies between the stimulus of a component and its possible responses (i.e., between its sink and source pins). Roughly speaking, a component reacts to an event arriving on a sink pin by emitting one or more events on its source pins.

The simplest form of reaction is a simple dependency relation sRr on component pins such that $(s, r) \in R$ indicates that a component will react to stimulus on pin s by generating a response on pin r . This is the minimal characterization of component behavior required to predict any meaningful assembly property. Still, the consensus is that computational models that are richer than a simple dependency relation are needed in practice. We have, in the past, used the CSP process algebra [Hoare 85] to specify reactions.¹⁴ Although CSP is a complex specification language, only the simplest features of it will be used in the following illustrations.

Note that reactions can be either complete or abstracted descriptions of component behavior, and can be specified in a “formal” language (such as CSP) or in an “implementation” language (such as Java). The only requirement is that a reaction has a parsable syntax, as is discussed more fully in Chapter 4.

In Figure 5, the reaction Rs_0 shown inside component c (on the left) specifies that c can receive stimulus on s_0 (i.e., $Rs_0 = s_0\dots$), and its reaction will be to respond through source pins r_0 and r_1 (i.e., $\dots s_0 \rightarrow r_0 \rightarrow r_1 \dots$), after which it will be prepared again to receive stimulus on s_0 (i.e., $\dots r_1 \rightarrow Rs_0$). The other reactions are similar. We require that each sink pin appears in *exactly one* reaction (although several sink pins can appear in the same reaction), and that each source pin appears in *at least one* reaction rule. Component behavior is specified as an inter-

13. Some restrictions apply to thread allocation. Pins also specify a type signature to accommodate parameters. These and other minutiae are not discussed further in this report.

14. We are currently exploring less complex process algebras such as FSP [Magee 99] and alternatives to process algebra altogether, such as one of its many variants or statecharts [Harel 95].

leaved (indicated by the $|||$ symbol) and/or parallel (indicated by the $||$ symbol) composition (in the CSP sense) of these reactions. In the example, overall component behavior Rc is defined using interleaved composition, indicating that there is no synchronization among the component's reactions. The reaction Rs_0 , shown inside the simpler component c' , is specified directly as a labeled transition system; its meaning is the same as the CSP reaction in c .

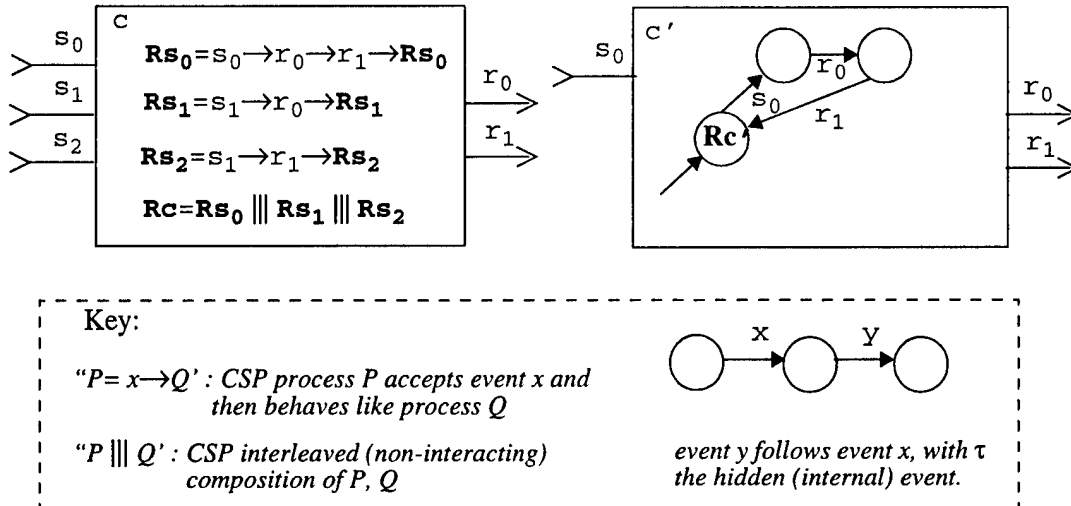


Figure 5: Component Interface with Pins and Reactions

As a notational convention, $R\{s_x\}$ is the reaction associated with sink pin s_x , and $R\{s_x, s_y\}$ is a reaction shared by sink pins s_x and s_y , and so forth. $R\{y\}$ is the overall behavior of component y . Braces ($\{ \}$) are omitted if confusion will not result, and dot notation is used to disambiguate reactions; for example, $c.Rs_0$ and $c'.Rs_0$.

3.3 Interactions

Where reactions specify the behavior of components, interactions specify the behavior of interacting components. To be a bit more precise, an interaction specifies the composite behavior of two or more reactions. An interaction may only occur between components that have been composed. Two components are composed when their labels (i.e., pins) have been bound.¹⁵ Note that such a binding only specifies that an interaction **may** occur between two components, not that it **must**. Note also that the definition of component composition is given in terms of a binding mechanism and not in terms of some abstract operator.

15. In this report, we assume that all interactions are binary and involve some form of "handshake" between the two interacting components. In general, n-ary interactions are possible (e.g., broadcast interaction), although it is possibly a matter of philosophy whether all such n-ary interactions comprise n-1 binary interactions.

Figure 6 illustrates the main ideas. Here components $c1$ and $c2$ are composed on $c1.r$ and $c2.p$. Graphically, this composition, or enabled interaction, is shown as a solid line connecting the composed pins. The semantics of composition is defined so that the behavior of composed reactions can be inferred from the reactions themselves. In the illustration, a very simple CSP semantics has been assumed: R_s and R_p synchronize on the shared event x , which is achieved by renaming both $c1.r$ and $c2.p$ to x ($[x|r]$ and $[x|p]$, respectively), and then using CSP parallel composition ($R_s \parallel R_p$).

Again, the details of CSP are less important than the requirement for a defined mechanism for inferring the behavior of composed components. Composition semantics is discussed in the next section.

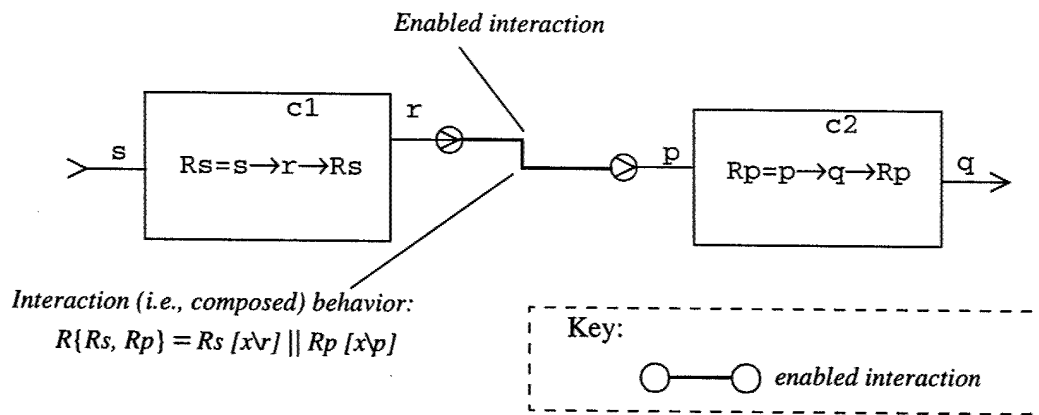


Figure 6: Enabled Interaction Between Two Components

3.4 Runtime Environment

As asserted in Volume II of this series, a component technology includes a runtime environment [Bachmann 00]. That runtime environment (called environment below) is a prominent feature of a component technology, although terminology varies; for example, framework, container, and platform are frequently used. In each case, the environment plays an analogous role: provider of services (e.g., transaction and security services), manager of resources (e.g., thread pools and database connections), and controller of component life cycles (e.g., initialization, preservation, and execution¹⁶). It is quite consistent to think of the component runtime environment as a kind of high-level, component-aware, possibly application-specific virtual machine.

16. Here, too, terminology varies substantially. Life-cycle terms analogous to the one cited include activation, passivation, and persistence, and they hardly exhaust the alternatives.

Environments, likewise, play a critical role in PECT. However, the PECT notion of environment is more general than the one discussed above. Rather than referring to a specific virtual machine, the PECT notion of environment refers to all relevant aspects of the execution environment that can influence the runtime behavior of components. Specifically, a component runtime environment

1. provides runtime services that may be used by components in an assembly. That is, environments can be thought of as a distinguished type of component with which (or in which) other components, and other environments, may interact.
2. provides an implementation for one or more interaction mechanisms, each supporting its own characteristic interaction protocol (e.g., blocking or non-blocking, buffered or unbuffered, and ordered or unordered interactions)
3. provides a closure for, and containment of, all assumptions made by a reasoning framework about the component runtime environment that can influence assembly behavior. (Reasoning frameworks are discussed in Chapter 4.)

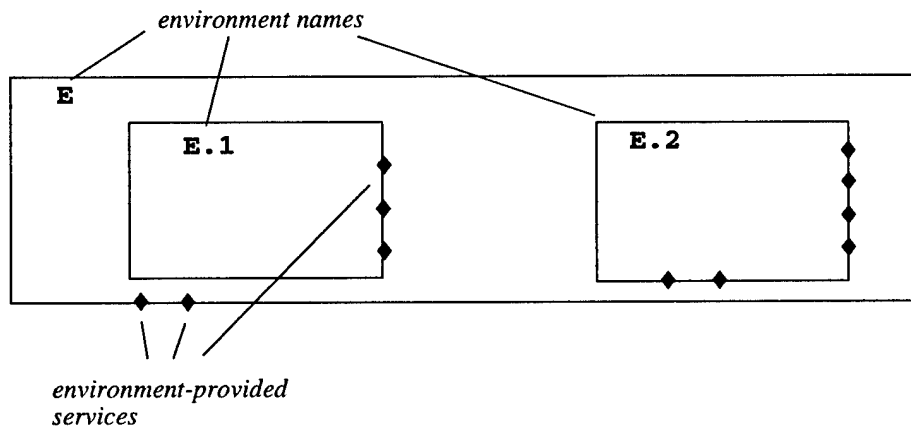
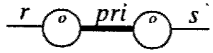
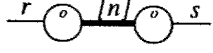

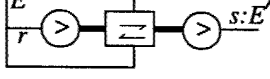


Figure 7: Environment Services and Containment

The graphic in Figure 7 introduces terminology and graphical conventions. First, environments, like components, are represented as boxes. Also, like components, environments have interfaces (in this illustration represented by the ♦ symbol—junctions on the environment boundaries), although later illustrations will extend this iconography to different types of environment services. Unlike components, though, environments have internal structure: they can contain components and other environments (a discussion of containment is deferred until Section 3.5).

The graphic in Figure 8 shows the main ideas of item (2) above and, likewise, introduces terminology and graphical conventions. The interaction mechanisms provided by an environment are encapsulated by connectors. A “connector” is best thought of as an environment-provided component whose behavior enforces an interaction protocol, or discipline, on the participants of an interaction or sequence of interactions.

Graphic	Connector Behavior (Informal)
	<p>Asynchronous (e.g., event-based) interaction.^a Unbounded priority queue. The priority of events is taken from the priorities of the reactions that emit them.</p>
	<p>Asynchronous (e.g., event-based) interaction. Bounded first-in, first-out (FIFO) queue of length n = maximum number of events. Oldest events are discarded on queue overflow.</p>
	<p>Synchronous (e.g., call-return) interaction. Semaphore acquired by calling reaction (i.e., the reaction on s is a [protected] critical section).</p>
	<p>Gateway from environment E to containing environment E'. Synchronous interaction is preserved. No semaphore is required (i.e., r is reentrant).</p>

a. Reaction on source r does not wait for reaction on sink s to complete.

Figure 8: Connectors Provided by Hypothetical Environment

The graphical notation of connectors has already been encountered, at least in part—the line connecting the component pins in Figure 6 on page 17 denotes a connector. Additional connector information can be encoded on the lines or in the symbols used to denote pins. Examples of the former are the *pri* for “priority queue” and *[n]* for “bounded queue” connectors in Figure 8 (the first two entries). Other examples are the use of the \triangleright symbol to denote connectors for event-based interaction (the first two entries) and the $\triangleright|$ symbol to denote connectors for protected critical sections (the third entry).

The behavior of each connector in Figure 8 is specified in such a way that the composite behavior of the participants in an interaction can be inferred from the behavior (reactions) of the participants themselves. The informal behaviors described in Figure 8 convey only a sense of what a complete connector specification must describe; in some circumstances, details such as threading and caching may also be required. Ivers and associates provide a detailed example of connector specifications (in CSP) for a mix of interaction protocols [Ivers 02].

3.5 Assemblies

The abstractions discussed in the preceding sections provide the necessary machinery to define what we mean by assembly. An assembly is a set of components and their enabled interactions. Since interactions are enabled by environments, each assembly is associated, through deployment, with exactly one environment.

Components are deployed to runtime environments; deployment defines where (ultimately, on which machine) behavior is executed. It is also sometimes useful to think of an assembly as

being deployed on an environment, although, strictly speaking, assemblies have no behavior other than that provided by their constituent components. $A:E$ denotes the assembly A deployed in environment E .

Components and assemblies are contained by other assemblies. Containment introduces hierarchy, with $A.c$ denoting component c contained in assembly A , $A1.A2$ denoting assembly $A2$ contained by assembly $A1$, and so forth. Containment defines a constructive closure—that is, the scope of all component interactions is restricted to (or closed within) the component’s immediately containing assembly.

The following illustrations are meant to appeal to the reader’s intuition; they are intentionally abstract and, in various ways, incomplete. For example, details such as allocation of environments to processors are ignored.¹⁷ Assume, for the present, that each environment denotes a distinct running instance of some (as yet anonymous) environment type; analogously, consider each component in Figure 6 to be a runtime instance of some anonymous component type.

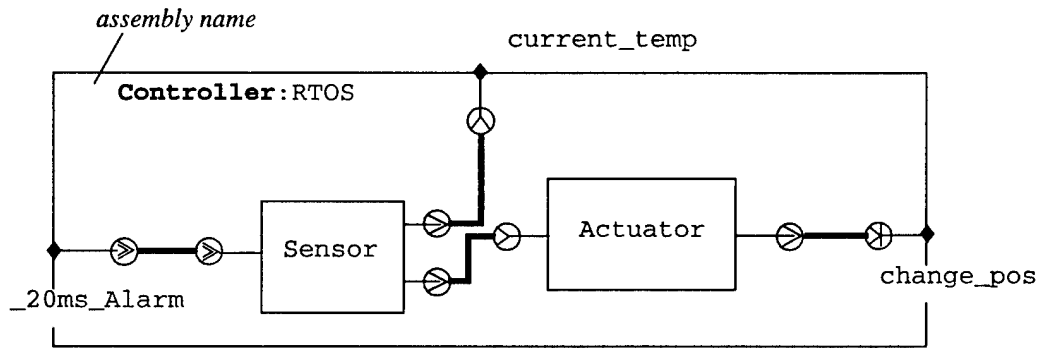


Figure 9: A Simple Controller Assembly

Figure 9 shows a simple two-component assembly, `Controller`, whose components are deployed in the runtime environment, `RTOS`. The environment provides three services: one service generates an alarm every 20 milliseconds (ms) (`_20ms_Alarm`), one provides system temperature (`current_temp`), and one changes the position of some device (`change_pos`). Although reactions are not shown, one scenario for the behavior of `Controller` is that the `Sensor` component reads the current temperature of some external system every 20ms and, based on what has been read, issues a command to the `Actuator` component.

17. This is not meant to imply that the abstracted details are trivial or unimportant, but merely that they are not critical to the following discussion.

Figure 10 illustrates another form of hierarchy. Controller¹⁸ from Figure 9 has been replicated, and each replicant is deployed into its own RTOS environment; these environments are, in turn, linked, via gateways, to the Voter component. Gateways are connectors that permit interaction among components across different environments. The outermost assembly, FTController, is deployed to the RTOS_4 environment. A reasonable scenario for the FTController assembly is that the Voter component implements some form of a fault-tolerance protocol. The important point, though, is to observe that all component interactions (i.e., their compositions) take place within a particular runtime environment.

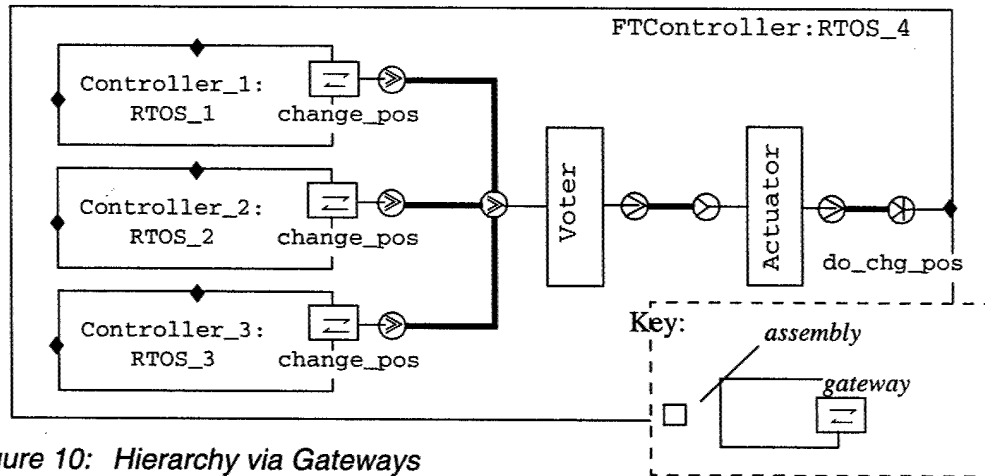


Figure 10: Hierarchy via Gateways

A third form of hierarchy is illustrated in Figure 11. In Figure 10, the three controllers were deployed assemblies. In Figure 11, the Controller_N assemblies and Voter component are partial assemblies. Like assembly, a partial assembly defines a scope of interaction. In this case, however, the scope is not associated with its own runtime environment; it “inherits” the runtime environment of its immediately containing assembly. Partial assemblies hide their contained components, but can expose selected pins through null junctions. A null junction has no behavior; it simply “unhides” selected pins. In Figure 11, Controller₂ is shown; the other controllers are, of course, not required to have identical internal structure.

18. The internal structure of the Controller assembly has been abstracted to make the graphic simpler.

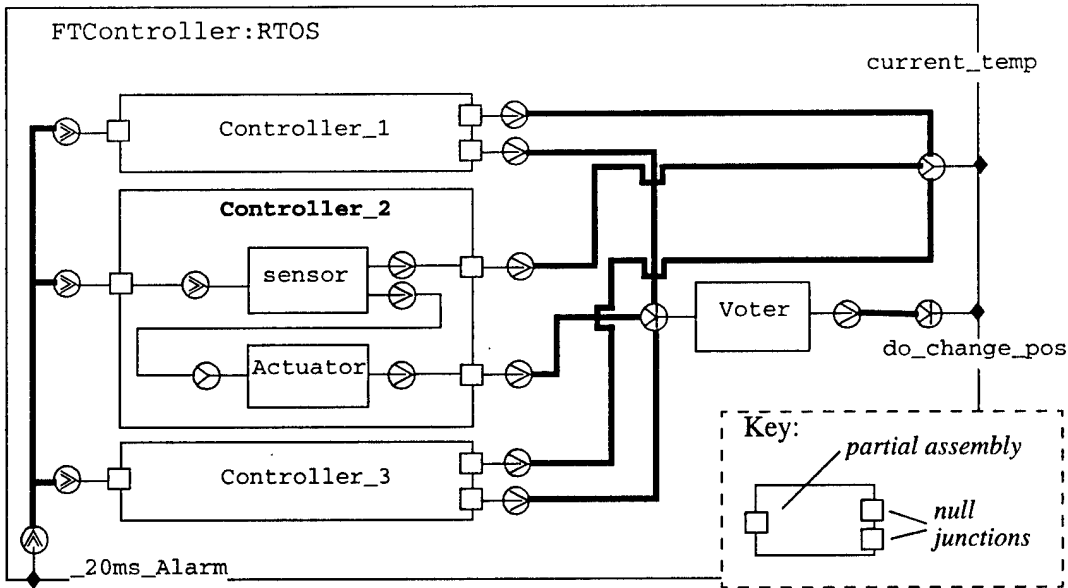


Figure 11: Hierarchy via Partial Assembly

The notion of hierarchy via partial assembly seems, on the surface, to be quite simple and natural. However, there are subtleties—in particular concerning analytic closures. An analytic closure defines a scope for assumptions that underlie predictions of assembly behavior. It is important to note that scopes defined by constructive and analytic closures do not always coincide; nor, in fact, do the scopes defined by different analytic closures (for different reasoning frameworks) always coincide.

3.6 Assembly Constraints

The previous sections described, in effect, a graphical language for an ACT consisting of components, reactions, interactions, environments, and assemblies. However, that description has not been particularly formal or complete. For example, the rules for well-formedness in the language are, for the most part, implicit. This informality is a necessary compromise, because a complete and formal description of a visual language is nontrivial, and its exposition would certainly distract from the objectives of this report.

In any event, to specify an ACT there must be at least one intended (target) component technology, and, insofar as ACT is specific to PECT, one intended reasoning framework. In particular, recall, from Chapter 2, that an ACT serves to make explicit constraints imposed by a component technology and one or more reasoning frameworks. In practice, then, well-formedness constraints will exist on components and their assemblies in addition to those discussed in Sections 3.1-3.5, which describe only the essential features of component technology.

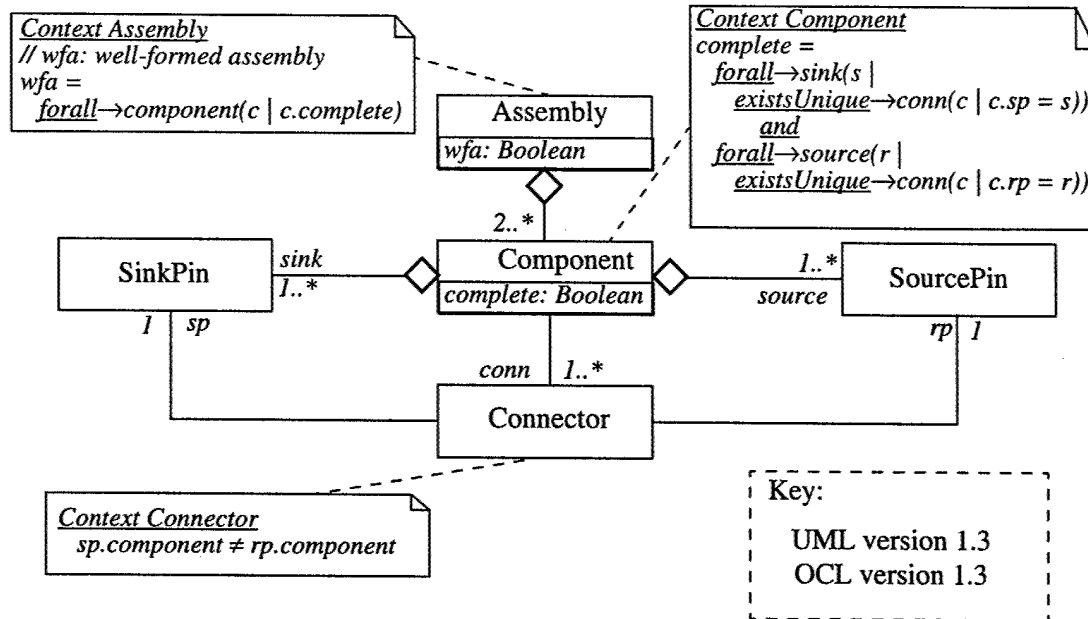


Figure 12: Extending the Well-Formedness Rules of an ACT

Figure 12 illustrates well-formedness constraints that might be imposed by a component technology or reasoning framework:¹⁹

- An assembly must have *at least two* components.
- Each component must have *at least one* sink pin and *at least one* source pin.
- Each sink and source pin must be connected *exactly once*.
- A component *cannot* be connected to itself (see the OCL annotation on Connector).
- An assembly is well formed if and only if all pins are connected (see the OCL annotation on Assembly).

This particular set of well-formedness constraints is easy to specify and understand, since it concerns syntactic (or topological) aspects of components and assemblies. Other constraints may go beyond well-formedness, specifying behavioral rather than syntactical constraints. For example, a reasoning framework might require that the component runtime environment enforces a specific scheduling discipline (e.g., the earliest deadline first) or that components conform to a particular start-up and shut-down sequence.

19. Note that these constraints are for illustration purposes only. They represent a component technology used in an early PECT prototype (see the paper titled "Packaging Predictable Assembly" [Hissam 02a]), but are too restrictive for general use.

Assembly constraints include the well-formedness and behavioral constraints imposed on an ACT by one or more component technologies and one or more reasoning frameworks. A construction framework comprises an ACT, tools to enforce the constraints imposed by an ACT, and other tools useful to automate the specification, development, and deployment of components and their assemblies.

3.7 Properties

Referring back to Figure 1, PACC is concerned with predicting what is unknown—assembly properties—from that what is known—component properties. A flexible but uniform treatment of both kinds of properties is desirable.

A property is an n -tuple $\langle name, value, \dots \rangle$, where *name* and *value* refer to the name of some property and the value it takes, respectively. The “...” portion of the definition refers to arbitrary and, perhaps, property-specific information. For example, it is often necessary to include a confidence interval with a property value.²⁰ However, at this level of generality, property types, hierarchies of property types, and the value sets of property types are not needed. Such details may ultimately be important, of course, but it is best to sidestep these complexities for as long as possible.

An annotation associates a property with a referent. An association of property P with referent R means that “ R has property P ” and is denoted as $R.P$. Although Figure 1 on page 2 implies only two kinds of referents—components and assemblies—that is far too restrictive. In fact, several kinds of referents have already been introduced: component, assembly, pin, reaction, environment, and environment service. Generally speaking though, properties of assemblies

20. Our notion of property annotation is an amalgam of ideas found in the Acme architecture description interchange language [Garlan 97]—which uses annotations to support extensible analyses—and credentials [Shaw 96a]—a proposal for treating architectural annotations as conjectures modified by the evidence to support the conjecture.

are predicted, while all other properties are asserted. That is, assembly properties are the purview of prediction, while all others are the purview of certification.

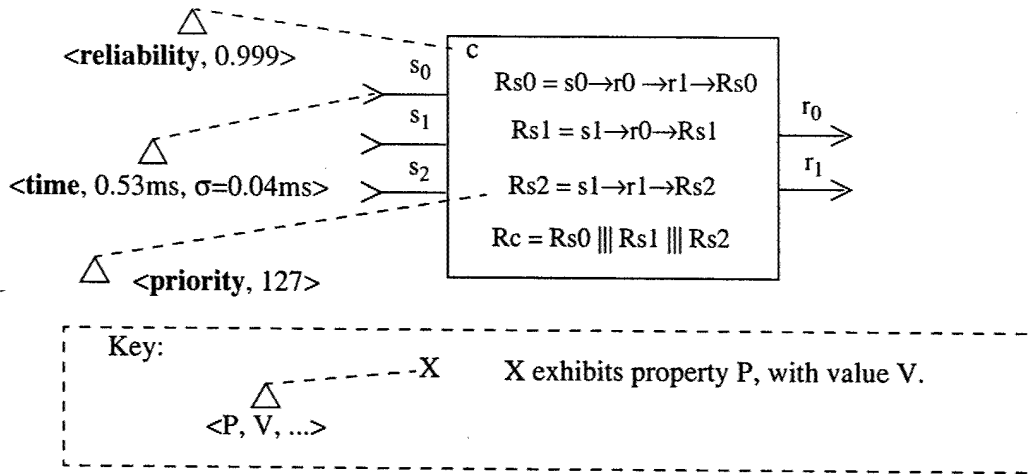


Figure 13: Property Annotations on Components

Figure 13 illustrates the main ideas and conventions with three annotations on the component discussed in Figure 5 on page 16. The component as a whole has been assigned a reliability of 0.999. Sink pin $c.s_0$ is annotated with an execution time of 0.53ms, with a standard deviation of 0.04ms. Lastly, the reaction R_{s2} has been annotated with its execution priority.

<latency, 139ms, (UB = 155ms, $\gamma=0.95ms$, $p=0.80$),
 (Controller.alarm) \sim (wake.Sensor.move) \sim
 (set.Actuator.move) \sim (change_pos.Controller)>

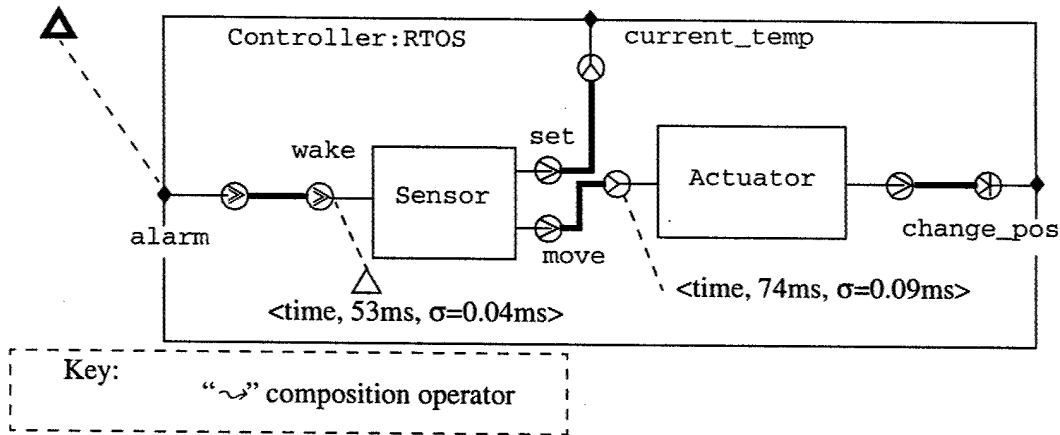


Figure 14: Annotations of Assembly and Component Properties

Figure 14 shows an annotation with a slightly more complex structure than that shown in Figure 13. The annotation appears to be attached to the alarm junction of the RTOS environment, but the actual referent is the series of interactions beginning with a response by the RTOS environment—alarm—and ending with a stimulus on the RTOS environment—change_pos. One interpretation of this annotation is that the predicted end-to-end latency of

the series of interactions is 139ms, a 95% tolerance interval, with an upper bound of 155ms and a confidence of 80%.

This is, however, just one possible interpretation of the latency annotation. **In fact, all of the annotations in Figures 13 and 14 are undefined, because they have not been assigned an interpretation in some reasoning framework.** Assigning meaning to component annotations is discussed in Chapter 4.

3.8 Construction Language

The graphical notation and concepts discussed above constitute an informal language for describing an ACT. A more formal definition is required to serve as a basis for automated interpretation from assembly specifications to analyzable models of assembly behavior. A construction language defines a concrete syntax for specifying ACTs, and for specifying components and assemblies of components that are well formed in ACTs. Each interpretation defined on a well-formed assembly can be thought of as assigning a semantics to the assembly.

The new jargon (“construction language”) adds to a field already crowded with composition language, architecture description language, coordination language, and module interconnection language. Admittedly, the boundaries among these different classes of notations are imprecise, and, in fact, particular notations are often classified in several ways.²¹ The introduction here is motivated by distinctions between what is needed to specify an ACT and what is provided by these existing classes of notation, such as the following:

- Components in architecture description languages tend to be design abstractions, rather than independently deployable component implementations in final form.
- Module interconnection languages tend to assume a fixed and limited repertoire of component interaction mechanisms, and do not represent “style” or “pattern” constraints.
- Coordination languages tend to ignore constructive issues such as module boundaries and focus instead on abstract behavioral specifications and their compositions.
- Composition languages tend to emphasize their roots in object-oriented (OO) programming languages, largely because their early formulations emerged from OO research.

Nonetheless, there is also significant overlap between these classes with one another and with the construction language concept discussed in this report. For example, the semantics for composing behaviors in the Piccola composition language [Achermann 01] is an earlier-speci-

21. For example, Papadopoulos and Arbab classify Polyolith [Purtillo 94] and Rapide [Luckham 95] as coordination languages even though the inventors of those languages described them as a module interconnection language and an architecture description language, respectively [Papadopoulos 98].

fied π -calculus variation of the CSP semantics defined by Ivers and associates [Ivers 02]; and the Acme architecture description interchange language [Garlan 97] defines syntactic elements for components, connectors, and annotations. In short, the new term “construction language” may not be justified in the long term; in the near term it is used to emphasize the distinctions.

A concrete syntax for one possible construction language has been specified. The construction and composition language (CCL) fragment in Figure 15 specifies the CSWI component used to implement a high-speed switch controller (see the substation automation case study for details [Hissam 02a]). Note that the sample specification assumes the use of CSP to specify reactions.²² (The meaning of the annotation is discussed later in Section 4.4 on page 33.)

```
#include "types.clh"
component CSWI: // From TR-031, pg. 54.

  sink async OpSel: Listener (Sel: in Select_t);
  sink async OpPos: Listener (Pos: in Position_t);
  source sync SwSel (Sel: out Select_t);
  source sync SwPos (Pos: out Position_t);
  react SwitchR [
    SwitchR =
      _OpSel.on-> _SwSel!on-> SwSel-> OpSel-> Selected
    [] _OpSel.off-> _SwSel!off-> SwSel-> OpSel-> SwitchR

    Selected =
      _OpSel.on-> _SwSel!on-> SwSel-> OpSel-> Selected
    [] _OpSel.off-> _SwSel!off-> SwSel-> OpSel-> SwitchR
    [] _OpPos?x-> _SwPos!x-> SwPos-> _SwSel!off-> SwSel-> OpPos-> R
  ];

  annotate react <SwitchR,
    claim: string =
      "!E[!(output = _sbosel_on) U (output = _sbopos_open)]">
end CSWI.
```

Figure 15: Component Specification in CCL

The CCL syntax also allows for the specification of environments (their services and connectors) and assemblies, and for the attachment of semi-structured annotations to an explicitly defined set of syntactic referents (component, assembly, pin, reaction, etc.).

22. A version of CCL under development uses executable statecharts in preference to CSP. See the SEI technical report by Ivers, J. & Wallnau, K. titled *CCL: A Parsable Syntax for a Construction and Composition Language*, currently in development. The CSP variant is used in this report for consistency with earlier reports.

4 Reasoning Frameworks

We want to reason about, and ultimately to predict, the behavioral properties (hereafter, simply properties) of assemblies. Our concern is with properties such as: the time it takes an operation to complete (latency); whether an operation exhibits erroneous behavior, and, if so, how frequently (reliability); whether an operation can always respond to a certain stimulus (liveness); and whether an operation invariably preserves certain conditions (safety). This is not a closed or precisely defined list, but it serves as a broad indicator of the different types of properties that are of interest to practicing software engineers.

Invariably, reasoning about complex systems requires the use of models. In general, a model is an abstraction that exposes some aspects of a system while simultaneously suppressing (or abstracting) others. A scientific theory provides models that objectively describe²³ the observable phenomena of natural or artificial systems,²⁴ and predict future observations in those systems. A scientific theory is always susceptible to falsification—that is, its **predictions can be subjected to tests designed specifically to refute (falsify) the theory.**²⁵

Property theories in PECT can be thought of as scientific theories about a particular runtime behavior, or property, of assemblies—they must be **objective, predictive, and testable.** Prop-

23. A scientific theory need not explain the cause of a phenomenon. For example, Newton's laws of motion described the effects of a hypothesized attractive force called gravity, but an explanation of these phenomena was not offered until Einstein formulated his general theory of relativity. Analogously, the ideal gas laws describe, but do not explain, the relation of pressure, temperature, and volume to each other.

24. This distinction is discussed at length by Herbert Simon in his classic *The Sciences of the Artificial* [Simon 96]. Natural systems are the purview of the traditional natural sciences such as physics, biology, and geology, while artificial systems are the result of human artifice. Simon argued that the scientific method developed for natural systems was applicable to, and required for, reasoning about modern artificial systems. A strong argument can be made that computing systems are artificial systems in the purest sense of that term, with complexity rivaling, if not exceeding, that of any other class of artificial system.

25. The falsifiability of scientific theories is the subject of Karl Popper's classic *The Logic of Scientific Discovery* [Popper 92]. Note that scientific models need only be falsifiable in principle. That is, an approach to falsification may be well defined but not technically feasible; for example, because of the limits of experimental apparatus. For example, there are models of quantum physics whose falsifiability, even in principle, is a matter of doubt.

erty theories must also be susceptible to automation so that their complexity can be, to the maximum practical extent, hidden from the end user—the practicing software engineer.

A PECT reasoning framework comprises a property theory, an automated reasoning procedure, and a validation procedure. These parts are discussed in Sections 4.1–4.3. The overall concept is illustrated with three reasoning frameworks, in Sections 4.4–4.6.

4.1 Property Theory

The design of an effective property theory can, and usually does, require significant theoretical knowledge and intellectual effort.²⁶ In short, its development requires a feat of ingenuity and skill no less than that implied by the development of any scientific theory. It is a research activity that, with luck and persistence, might eventually bear fruit in practice.

As discussed in Chapter 2, our objective is not to develop new property theories, but rather to specialize (restrict) existing ones to particular sets of systems, and, in this specialization, to satisfy required accuracy while also obtaining ease of use through automation. To understand how to achieve this, we must be more precise about the structure of a property theory. The basic elements of a PECT property theory are

- a calculus, which is “a system or arrangement of intricate or interrelated parts” [Merriam-Webster 93], where, in this context, the parts are symbolic
- a logic, which defines rules for transforming one sequence of symbols to another, establishing “principles and criteria of validity of inference and demonstration” [Merriam-Webster 93]
- an abstract interpretation, which is a map from the symbols of a calculus to elements of a (not necessarily component-based) computing system

The generality of this definition of property theory reflects the broad range of such theories that can be used in a PECT, while its formality reflects the emphasis placed by PECT on automated reasoning.²⁷

26. The following description of a property theory is based loosely on Hoare’s paper, “Algebra and Models” [Hoare 93]. In that paper, Hoare outlines an approach to developing algebraic models for reasoning about systems. The adaptation of Hoare’s ideas in this report generalizes from algebraic models to arbitrary calculi and logics; limiting PECT reasoning frameworks to algebraic models would be too restrictive. The generalization does no violence to Hoare’s ideas, however. It is also worth noting that Milner observed, in the introduction to his process algebra, that non-algebraic calculi, such as first-order logic, have their usefulness in conjunction with algebraic models [Milner 89].

4.2 Automated Reasoning Procedure

An automated reasoning procedure has three distinct elements, each of which must, in principle, be automatable: a decision procedure, a definite interpretation, and an definite inverse interpretation, where

- A decision procedure is a function that evaluates claims made on assemblies described in the property theory to the values “true” or “false.”
- A definite interpretation maps assemblies specified in a concrete syntax of a construction language to strings in the input language of the decision procedure.
- A definite inverse interpretation maps the results of the decision procedure back to the concrete syntax of the construction language.

Limiting decision procedures to evaluate claims to boolean values is not as restrictive as it might appear. A quantitative prediction of end-to-end latency of interactions can always, for example, be expressed as a boolean claim that latency does not exceed some value. The real effect of the definition, however, is to rule out decision procedures (and, indirectly, property theories) that cannot be used to express claims that are verifiable or, at a minimum, falsifiable.

Decision procedures must, of course, be computable, but there are no other restrictions on the type of algorithms used. In particular, a decision procedure could compute the value of a closed-form expression or compute its approximation using iterative means, do an exhaustive search, or perform a simulation. In general, complete decision procedures are preferred to partial procedures, but are not required. The only requirement is that if the procedure terminates, it yields a verifiable or falsifiable claim.

4.3 Validation Procedure

The most accurate predictions are of little use if they are not trusted. This trust is indispensable to achieving a fruitful separation of concerns among component developers, component certifiers, and application assemblers. Without trust in component properties, excess effort will be expended revalidating claims about components. Without trust in assembly predictions, excess effort will be expended in integration testing. Without trust in both, there will be little hope of

27. It may, in the future, be useful to classify property theories to expose their deeper structure. For example, Bachmann and associates, in *Illuminating the Fundamental Contributors to Software Architecture Quality* [Bachmann 02] and *Deriving Architectural Tactics: A Step Toward Methodical Architectural Design* (SEI technical report CMU/SEI-2003-TR-004, currently in development), describe a structure for property theories that distinguishes independent from dependent variables; in contrast, Hoare distinguishes between direct and indirect observations [Hoare 93]. The former seems a better fit for theories that emphasize reasoning about magnitudes, while the latter seems a better fit for reasoning about assertions in a formal logic.

establishing a value proposition for certified components or predictable assembly. A robust value proposition is needed, because the technical and social infrastructure for “trust” will require up-front investment that must be justified in terms of future efficiencies.

A validation procedure provides objective evidence for trusting the validity and soundness of a reasoning framework; it also defines its required component properties with sufficient rigor to provide an objective basis for trust in assertions of component behavior. A reasoning framework is valid if its predictions match observations and sound if its theorems, relations, and so on can be mathematically demonstrated.

As observed by Meyer and associates, trust is a social phenomenon likely to be achieved through a mix of technical and nontechnical means [Meyer 98]. These same authors also observed that trust is not absolute except in matters of religious faith. Looking beyond Meyer’s concern with trusting that a component implementation is correct, the means for establishing trust depends on qualities inherent to the

- property theory—for example, is it a stochastic or deterministic theory?
- to the problem domain—for example, is human safety of concern?
- and to the business context—for example, how much value accrues with additional evidence?

Minimally, the weight of objective evidence must justify a value proposition for component certification and other infrastructure development.

There are, however, two classes of objective evidence that are worth distinguishing: empirical evidence and formal evidence (in Polya’s terminology—plausible and demonstrable evidence²⁸), because different schools of thought place more trust in one form of evidence than the other. However, to the extent that these forms of evidence are truly distinct, both are needed. Empirical evidence is acquired through direct observation, preferably under controlled circumstances, with results reported in well-defined units of measure. Empirical evidence is therefore provisional, as any other observation might have been different; hence, conclusions based on empirical evidence are, to a lesser or greater extent, only plausible. Formal evidence is acquired through mathematical proof and is therefore irrefutable, as all such proofs are tautological; hence, conclusions based on formal evidence are inevitable or demonstrable.

In fact, both empirical and formal evidence have roles in providing objective confidence in property theories and in the asserted values of component properties. Different types of theo-

28. The distinction between plausible and demonstrable reasoning is discussed eloquently by Polya in *Mathematics and Plausible Reasoning: Volume I Induction and Analogy in Mathematics* [Polya 54].

ries and properties will result in different emphases on one or the other. For example, properties with established measures of magnitude, such as time and space, are more susceptible to empirical treatment than are properties with less obvious measures, such as reliability. The following statements illustrate the mix of reasoning used to establish trust:

- A reasoning procedure for predicting the absence of a behavior in an assembly might be defined in automata theory and temporal logic; the resulting claims will be demonstrable. However, establishing that a particular automata describes the behavior of a particular component will involve, ultimately, observation.
- A reasoning procedure for predicting the execution latency of an operation in an assembly might be defined as equations involving the measured time of components; the resulting claims will be plausible. However, the equations themselves might be derived from theorems whose validity must be formally demonstrated.

It is well worth repeating that the level and kind of objective evidence required to establish trust in predictions will vary from situation to situation.²⁹

Note that the problems of establishing trust in component properties and predictions are strongly interrelated, as premised in the introduction to this report (and implied by Figure 1 on page 2). This report is focused primarily, however, on PECT. A full treatment of how empirical and formal property theories are validated and how component properties are certified will be the topic of a future report. Only a passing treatment of the topic is provided through the following illustrations.

4.4 Illustration 1: Temporal Logic Model Checking

Temporal logic model checking (hereafter simply called model checking) is a technique for formally verifying system properties such as safety (informally, a certain condition never occurs) or liveness (informally, a certain condition eventually occurs).³⁰

29. In fact, it has been argued that the notion of a demonstration in a proof is also subject to different degrees of rigor, ranging from fully formal (mechanized) proof in the Hilbert style, to sketches of proof templates in which more can be assumed about the social context of a proof's audience [DeMillo 77].

30. Model checking has been used widely and successfully to verify hardware design; significant effort has been made recently to apply it to software design as well. There are many books and articles on the subject of temporal logic model checking. See Huth's work for a gentle but thorough introduction [Huth 00] or the work of Clarke and associates for a more in-depth and theoretical treatment [Clarke 99].

Property Theory

The property theory for model checking combines automata theory and temporal logic—in model checking literature, property theory is usually described as a combination of a computational model and a specification language for a particular class of temporal logic claims. There are many variations in the automata theory and temporal logics used, and each combination is equipped with its own rigorously defined (i.e., purely formal) calculus and logic. Although the distinctions among these property theories are theoretically important, they all bear strong family resemblances to one another. The illustration sidesteps distracting nuance.

Automated Reasoning Procedure

The decision procedure used in Illustration 1 is provided by the SMV model checker.³¹

The top half of Figure 16 shows a state machine corresponding to the CSWI . SwitchR reaction shown in Figure 15 on page 27. There, the temporal logic claim was expressed, in the annotation, directly in the notation of SMV; ideally, a neutral syntax would have been chosen. In natural language, the claim reads, “it is never possible for the switch to be opened (`_sbopos_open`) before it has been selected (`_sbosel_on`).”

31. See <<http://www.cs.cmu.edu/~modelcheck/smv>> for details on this model checker.

The key points are (1) a mechanical translation from CCL to SMV is possible, in principle, but its definition is complex, and (2) the resulting SMV is not easily comprehended by the end user. However, the complexity is apparent only once, when the definite interpretation is defined, and the resulting SMV input does not have to be made visible to the end user, any more than the internal results of any “code generation” process need to be exposed. All that is required, instead, is the inverse interpretation that maps the results of a decision procedure, SMV in this case, back to the original specification in CCL.

Validation Procedure

Model checking is a form of verification; the soundness of the model-checking algorithm is demonstrated by proofs involving; for example, the fixed-point semantics of the temporal logic operators.³²

Therefore, the basis for trusting the results of model checking depends on confidence

1. in the implementation of the model checker itself
2. that the finite models of component behavior used in the model checker are satisfied by the components (i.e., the implementations) they represent

Confidence in item (1) above is generally obtained by testing the model checker. Whether or not it's justified, most users trust the implementation of a model checker.³³ University-developed software might be regarded with some skepticism, but some model checkers are supported by commercial vendors, helping to obtain a level of trust.

Confidence in item (2) above is a more difficult matter. Some research prototypes extract state machines directly from source code [Corbett 00]. In that case, the formal translation process from source code to state machine (in fact, a definite interpretation in the PECT sense) reduces the question of trust to one of trusting the translator's implementation; an analogous situation arises in the reverse case, where components are derived from state machines. Some form of testing is required where state machines are not “formally” linked to components; for example, where state machines and components are implemented manually [Havelund 01].

32. An accessible treatment of the fixed-point semantics of computational tree logic (CTL)—a temporal logic—is provided by Huth [Huth 00].

33. Interesting is the experience of Brat and associates in evaluating verification and validation tools on Martian Rover software. In presenting the findings, one of the authors reported that the model-checking group encountered unexpected difficulties when it encountered a floating point error in the model checker. See <<http://sei.cmu.edu/pacc/smcw/>> for the proceedings of this workshop.

4.5 Illustration 2: Rate Monotonic Analysis

Rate monotonic analysis (RMA) is a technique for determining, among other things, whether a set of tasks can be scheduled for execution in such a way as to guarantee that hard-real-time deadlines are always satisfied.³⁴ Actually, RMA is a system for constructing property theories, although it does include results (such as the Liu Layland theorem discussed below) that can be thought of as property theories in their own right.

Property Theory

In addition to the usual symbols of classical algebra and finite mathematics, the RMA calculus defines symbols for tasks, parallel and sequential composition of tasks, task priority, period, task execution time, processor utilization, queues, and so forth. The underlying logic of RMA is based in classical algebra, with key principles and results expressed algebraically. This illustration makes use of the Liu and Layland theorem, which states that a set of n independent periodic tasks, when scheduled using a rate monotonic algorithm, will always meet its deadlines, for all task phasings, if

$$\text{Eq.1} \quad \frac{C_1}{T_1} + \dots + \frac{C_n}{T_n} \leq U(n) = n(2^{1/n} - 1)$$

where C_i = worst-case task execution time of $task_i$, T_i = period of $task_i$, and $U(n)$ = utilization bound for n tasks.

RMA was used as a basis for the λ_{ABA} property theory discussed extensively in the substation automation case study (see [Hissam 02b]). This property theory predicted the latency (λ) of the longest execution path in an assembly, for the Average case latency, where components could Block one another, and including Asynchronous interactions (ABA). A preliminary form of the property theory predicted worst case latency with blocking for synchronous interactions only (λ_{WB}); this simpler theory is more convenient for illustrative purposes.

$$\text{Eq.2} \quad L_{n+1} = \sum_{j=1}^{i-1} \left\lceil \frac{L_n}{T_j} \right\rceil C_j + C_i$$

34. *A Practitioner's Handbook for Real-Time Analysis* is the definitive source for applying RMA in practice [Klein 93]; it cites a variety of publications that establish the soundness of the underlying computational model.

The formula in Eq. 2 predicts the worst-case latency L_i for the i^{th} task. C_i is the execution time of the i^{th} task; T_i is the period of the i^{th} task. Tasks 1 through $i-1$ are assumed to be all the tasks whose priorities are higher than the priority of task i . The iterative calculation starts by using C_i as the first guess for the worst-case latency by setting L_1 to C_i , and it then computes L_{n+1} using the formula. It continues until $L_n = L_{n+1}$, at which time the fixed point has been reached, and L_{n+1} is the worst-case latency.

Automated Reasoning Procedure

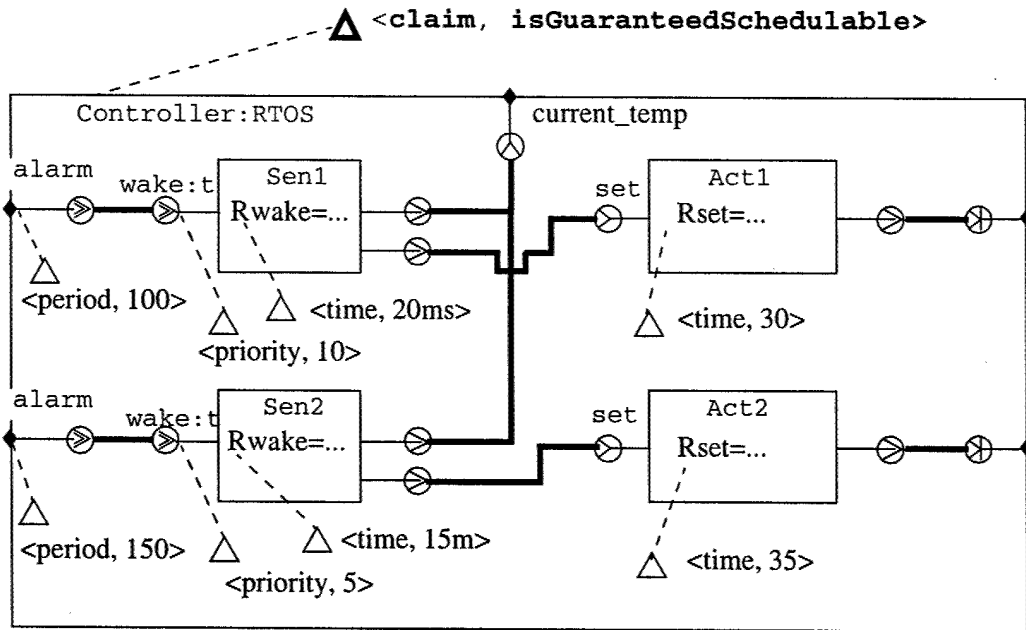
Figure 17 shows yet another variant of the controller assembly. Two sensors are in that assembly, each connected to its own actuator, but both connected to the same temperature gauge. In this figure, the reactions on asynchronous sink pins (\triangleright) are assumed to execute on their own threads of control, and are therefore schedulable tasks with fixed priorities, while the reactions on non-blocking synchronous sink pins ($>$) execute on the caller's thread of control.

The interpretation is straightforward. First, observe that, in contrast to Figure 16, this interpretation requires annotations: the period of the alarm service (T_i in Eq. 1), the priority of threads `Sen1.wake:t` and `Sen2.wake:t`,³⁵ and the execution time of reactions (C_i in Eq. 1). Since `Act1.set` and `Act2.set` are not threaded, they are not schedulable in RMA, and they have no priority assignment. The execution time for the `Rset` reactions is simply added to the execution time of the calling reaction.

The decision procedure for λ_{WB} was more complex, but fully automated (again, unlike Figure 16, which was automated only "in principle"). The interpretation scheme translated assemblies specified in CCL³⁶ to an input string of a simulator developed especially for the property theory. The simulator used a combination of analytic and iterative means to compute the fixed point of a set of tasks sharing the same processor, and from that fixed point, the latency was derived.

35. Recall from Section 3.1 that where $x \neq y$, $cx.s:t$ and $cy.s:t$ denote different threads of execution.

36. The translation rules are specified completely in Appendix A of *Predictable Assembly of Substation Automation Systems: An Experiment Report* [Hissam 02b]. Note, however, that the syntax of CCL has changed since the time of that initial case study.



which can be interpreted in Liu and Layland (Eq. 1) as...

$$\left(\frac{20 + 30}{100} + \frac{15 + 35}{150} \right) = .75 \leq 2(2^{1/2} - 1) \cong .83$$

Figure 17: An Interpretation for RMA Schedulability Analysis

Validation Procedure

Unlike model checking, RMA theories depend explicitly on measured (empirical) phenomena: execution time—an explicit parameter of the property theory—and blocking time—the property that is predicted (i.e., latency = execution time + blocking time). While many of the basic results of RMA are demonstrable (e.g., Lui-Layland), trust in the overall effectiveness of the property theory rests on empirical, experimental evidence.

Trust in the effectiveness of λ_{WB} depends on statistical trust in the

1. quality of λ_{WB} predictions
2. measures of component execution time

Confidence in the quality of λ_{WB} predictions required the development of a measurement infrastructure for collecting timed traces of component and assembly execution in a controlled environment. Also required was a statistically valid (representative) sample of assemblies, so that a confidence interval for the property theory could be obtained. The technique used to generate this sample combined elements of random graph generation and topology constraints

to restrict the random assemblies to those that were, in some predefined way, “stereotypical” of the intended application.

Confidence in the measures of component execution time also required the development of a measurement infrastructure—this time for collecting traces of component execution time in a controlled environment, where the total blocking time of a component under all execution traces could be eliminated. As with the property theory itself, a statistical measure of confidence was obtained that reflected an inherent quantum of nondeterminism (random measurement error) introduced by the runtime environment and additional measurement apparatus.

In both cases, trust was established using statistical means for λ_{WB} statistical confidence in the representativeness of the random sample and for component measures’ statistical confidence that the component execution time established in a measurement environment would be equivalent to the execution time in the deployed runtime environment.

4.6 Illustration 3: n-Version Majority Voting Analysis

Many design patterns have been developed to make systems more resilient to hardware and software failure.³⁷ Most (if not all) of these patterns are based on some underlying notion of redundancy. The form that this redundancy takes in Illustration 3 is the n-version majority voting (NVV) pattern. In the NVV pattern, a reliable voting component chooses the majority response from a set of n unreliable components, where each component computes the same function but has a different implementation (i.e., n versions).

Note that, unlike Illustrations 1 and 2, the NVV reasoning framework is more a thought experiment than reality. It is introduced here as a foil for exposing different aspects of PECT theory. Nonetheless, while NVV is a thought experiment, doubts about its realism can be challenged, as will be discussed in Section 6.2 on page 47.

Property Theory

A simple calculus is defined for this example. In addition to the usual symbols of classical algebra and discrete probability theory, the symbol π is defined to denote the probability of component failure. As with most theories of reliability and fault tolerance, the underlying

37. Musa’s *Software Reliability Engineering* is widely cited as being authoritative [Musa 98]. Daniels and associates merge several design patterns for fault tolerance [Daniels 97]. McAllister and associates give an in-depth analysis of one particular voting pattern that is representative of the computational model shared by various models for reasoning about system reliability [McAllister 90].

logic is based in probability theory. In this contrived property theory, the reliability of an assembly of components using the NVV pattern is given as

$$\text{Eq3} \quad p_{\text{Fail}} = \sum_{k=m}^n \binom{n}{k} \pi^k (1-\pi)^{n-k}$$

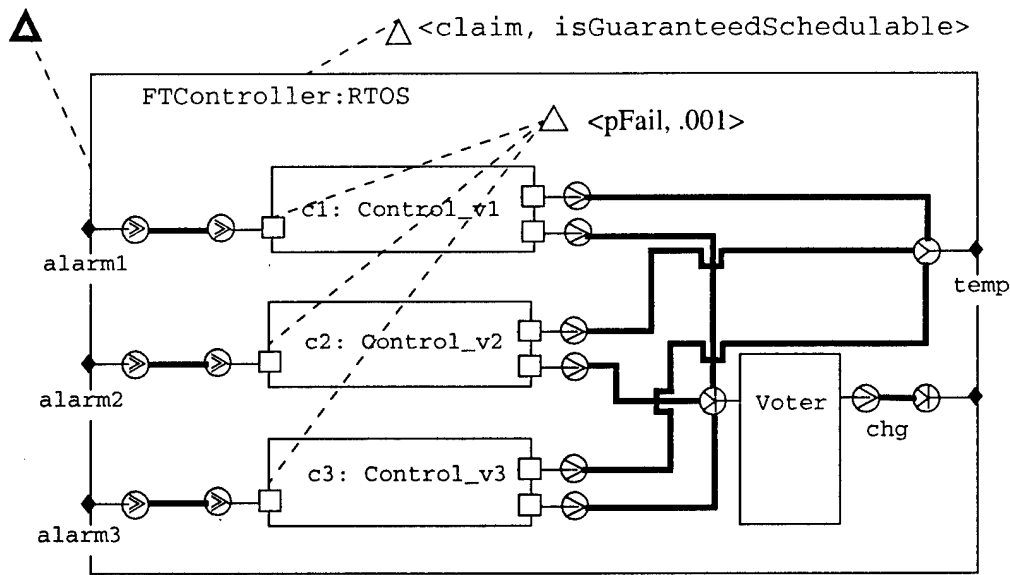
where p_{Fail} = the probability that an assembly will fail, n = the number of unreliable components, $m = (n + 1) / 2$ for odd n , and π = the probability of component failure. Of course, this is nothing more than the sum of simple binomial distributions of the probability that a majority k out of n components will experience a failure on some event, where $k \in \{m, m+1, \dots, n\}$. Although the theory is undeniably simple, it captures the essence of the voting pattern.

Automated Reasoning Procedure

The illustration in Figure 18 reprises Figure 11 with minor modifications. In this example, the failure rate of unreliable components is specified as an annotation on the partial assemblies, $c1$, $c2$, and $c3$. The reliability claim is the topmost, boldfaced annotation in the figure.³⁸ It asserts that the probability that the end-to-end interaction starting with one of the three RTOS alarms and ending on RTOS.temp will fail (p_{Fail}) is less than or equal to 1×10^{-5} .

38. The notation for the interaction omits pin names for brevity. As with CSP, the symbol \parallel denotes parallel composition. The intent is that the reactions composed by \rightsquigarrow in $(a \rightsquigarrow b) \parallel (c \rightsquigarrow d)$ execute concurrently.

$\langle \text{claim}, p_{\text{Fail}} \leq 1 \times 10^{-5}, \{\text{alarm1} \rightsquigarrow \text{c1} \parallel \text{alarm2} \rightsquigarrow \text{c2} \parallel \text{alarm3} \rightsquigarrow \text{c3}\} \rightsquigarrow \{\text{Voter}, \text{temp}\} \rangle$



which can be interpreted in Eq. 3 as...

$$p_{\text{Fail}} = 3\pi^2(1 - \pi) + \pi^3 = 3\pi^2 - 2\pi^3 < 3 \times 10^{-8}$$

Figure 18: An Interpretation for Reliable Voter Pattern Reliability Analysis

The interpretation uses the sink annotations to fill in the value of π in Eq. 3 and the assembly topology to fill in the values of n and m . The decision procedure for the property theory, as formulated, is a trivial mathematical function and interpretation. More complex decision procedures (e.g., using Monte Carlo simulations of failure rates with probability distribution functions) can be envisioned.

Validation Procedure

Similarly to RMA-based property theories, the NVV property theory is parameterized by, and predicts, an empirical (observable) phenomenon: rate of failure. Accordingly, confidence in the NVV property theory will be based in part on demonstration (e.g., the applicability of basic probability theory) and experimental evidence.

However, while time is, for all practical purposes, a universally understood phenomenon, the failure rate of software components is not. Still, it is possible to fix on a particular definition of failure rate (e.g., the number of “hard crashes” per invocation for each reaction) and obtain a statistical measure of a property defined in this way through established testing procedures.

5 Concise Summary of PECT

A PECT extends the notion of a component technology with one or more reasoning frameworks, such that assemblies of components are predictable with respect to those reasoning frameworks. In Figure 19, a UML class diagram illustrates how a component technology relates to reasoning frameworks in a PECT.

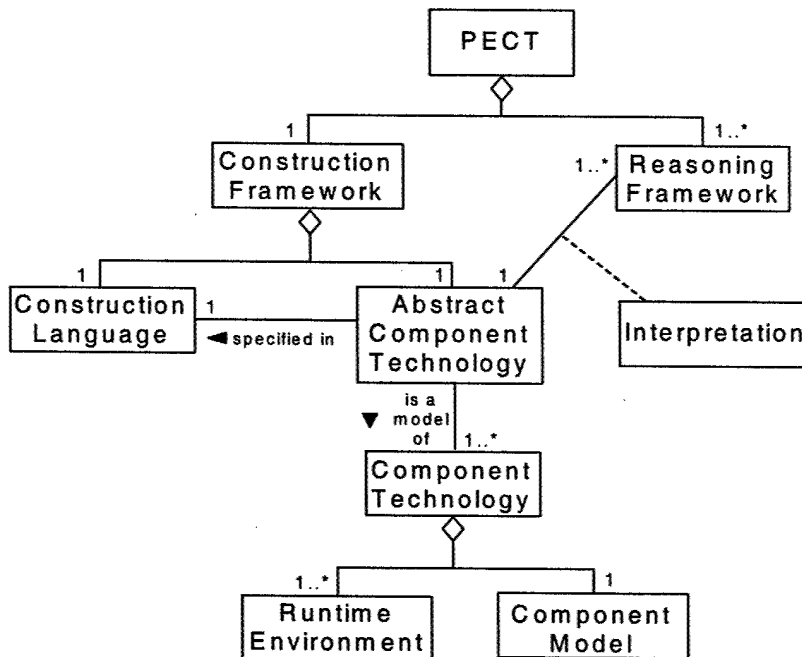


Figure 19: UML Class Diagram of PECT Concepts

A component technology consists of a component model and one or more runtime environments. The component model specifies allowable component types, interaction mechanisms, services provided by the runtime environment, and constraints among them. A runtime environment is an execution environment that enforces aspects of the component model. A runtime environment plays a role analogous to that of an operating system, serving as the context in which components execute. Different runtime environments for the same component technology enforce the same component model, but may differ in terms of quality attributes, such as performance or reliability.

A component technology is incorporated into a PECT by means of a construction framework. Such a framework consists of an ACT and a construction language. An ACT is a description of a particular component technology in a construction language. ACTs are described using a

common language—a construction language—to allow the same tools to be used with PECTs containing different component technologies.

The construction language is also used to describe assemblies constructed in accordance with the ACT and associated reasoning frameworks. A construction language includes the syntactic elements needed to capture three kinds of information:

1. the topology of an assembly (the composition of components that defines the structure of the assembly)
2. the behavior of each component in the assembly, the interaction mechanisms defined by the component model, and the services provided by the component technology's runtime environments
3. arbitrary property descriptions that are required by specific reasoning frameworks and attached to various syntactic elements, such as components or interactions

Each reasoning framework included in a PECT embodies the concepts and theories needed to analyze, and hence predict, properties of an assembly of components. An interpretation is defined for each reasoning framework of a PECT that relates the concepts of the ACT to the concepts of the reasoning framework. An interpretation is used during development to translate an assembly specification, as documented using the construction language, to a specification that can be used with the interpretation's reasoning framework.

6 Multiple Reasoning Frameworks

The value of PECT will be enhanced as reasoning frameworks are developed, validated, and, possibly, certified. However, additional complexity also attends the introduction of multiple reasoning frameworks. Questions will arise, for example, about the consequences of using one reasoning framework in place of another, or about whether reasoning frameworks that have contradictory assumptions (assembly constraints) can be safely used together.

The situation is shown informally as a Venn diagram in Figure 20, which refines the earlier depiction in Figure 2 on page 8. Figure 20 shows several reasoning frameworks, and the space of possible assemblies has been subdivided into a number of distinct sets, each containing the assemblies that are well formed with respect to a reasoning framework theory whose models (input to its decision procedure) are in the set \mathcal{M}_2 .

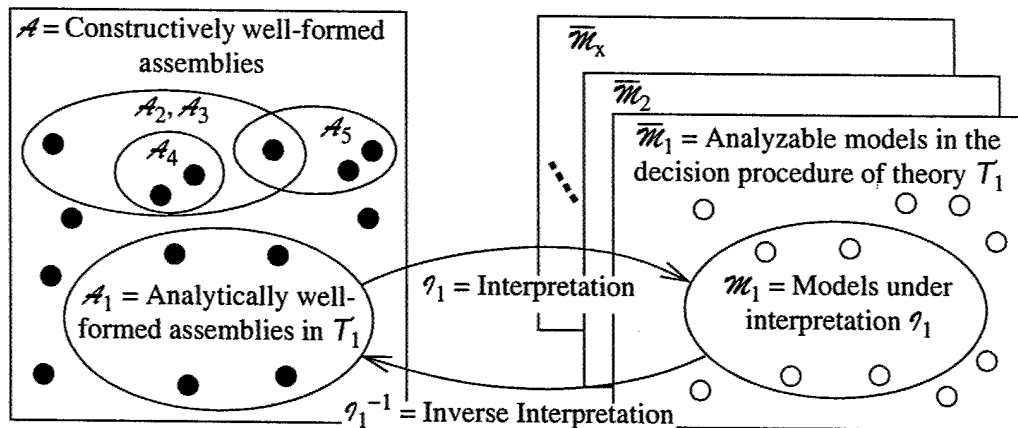


Figure 20: *Compatibility of Analytically Well-Formed Assemblies*

To illustrate, perhaps \mathcal{M}_2 and \mathcal{M}_3 model different properties; for example, security and performance (what will later be defined as heterogeneous properties). Since the well-formed assemblies for these reasoning frameworks are equivalent in extent ($\mathcal{A}_2 = \mathcal{A}_3$), all is well. However, what if we now add \mathcal{M}_5 that models, say, reliability? How will it be ensured that only the intersection $\cap\{\mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_5\}$ is constructed? Alternatively, given an assembly, is it possible to infer which reasoning frameworks are applicable?

We do not yet have robust answers to these questions. However, the concepts elaborated in the earlier chapters appear to provide a good foundation for providing answers—and indeed some progress has been made already. This chapter outlines, in Sections 6.2 through 6.4, what we

already know about several key issues introduced when considering multiple reasoning frameworks, and how these issues can be addressed. Section 6.2 addresses the question “when does a simple theory become too simple?” This discussion is a precursor to Section 6.3, which discusses co-refinement—the process by which an optimal balance is achieved among various qualities of property theories, including simplicity. Section 6.4 discusses the issue of incompatibilities among reasoning frameworks, while Section 6.5 discusses some still speculative approaches to dealing with these incompatibilities. This last is an area for needed research.

Before embarking, though, formal definition of some key terms (in Section 6.1) will prove useful. Readers not formally inclined may skip Section 6.1 and refer to it only when encountering unfamiliar terms or notation.

6.1 A Few Formal Definitions

Standard mathematical notation is used. The symbols \Leftrightarrow , \wedge , \neg , $\exists x.P$, and $\forall x.P$ denote boolean “if and only if,” “and,” “not,” and the existential and universal quantifiers, respectively. The letters p , q , r , denote (boolean) predicates, and $p(x)$ denotes the application of p to some x . The symbols \in , \cap , \supseteq , \supset , $=$, and \emptyset denote set “element,” “intersection,” “subset,” “proper subset,” “equality,” and “the empty set,” respectively.

General Definitions

- Let $\mathcal{A} = \{a_1, a_2, \dots, a_k\}$ be the set of assembly specifications, in CCL, that are well formed with respect to the constructive constraints of some ACT.
- Let $\mathcal{T} = \{T_1, T_2, \dots, T_x\}$ be the set of property theories, embedded in reasoning frameworks, that are integrated in the same ACT.
- Let $\mathcal{M}_x = \{m_{1_x}, m_{2_x}, \dots, m_{k_x}\}$ be the set of models in (or input strings to) the decision procedures of property theory T_x .
- Let $\mathcal{P}_x = \{p_{1_x}, p_{2_x}, \dots, p_{k_x}\}$ be the set of predicates that encodes the analytic constraints of property theory T_x .
- Let $\models : \mathcal{A} \times \mathcal{T}$ be the “satisfies” relation such that $a \models T_x \Leftrightarrow \forall p_{k_x} \in \mathcal{P}_x \cdot p_{k_x}(a)$ —that is, an assembly a satisfies property theory X if and only if a satisfies X ’s analytic constraints.
- Let $\mathcal{I}_x : \mathcal{A} \times \mathcal{M}_x$ be the “interpretation” relation such that $(a, m) \in \mathcal{I}_x \Leftrightarrow a \models T_x$, and a derives m through some sequence of syntactic transformations.
- Let $\mathcal{A}_x = \{a_x \mid \exists m \cdot (a_x, m) \in \mathcal{I}_x\}$ be the set of assembly specifications that satisfy and have interpretations in T_x . We say “assembly a is well formed with respect to T_x ” if and only if $a \in \mathcal{A}_x$.

Definition of Compatibility Relations

- $T_1 \equiv T_2$ (pronounced “ T_1 and T_2 are compatible”) $\Leftrightarrow A_1 = A_2$
- $T_1 \sim T_2$ (pronounced “ T_1 and T_2 are incompatible”) $\Leftrightarrow A_1 \cap A_2 = \emptyset$
- $T_1 \cong T_2$ (pronounced “ T_1 and T_2 are partly compatible”) $\Leftrightarrow \neg(T_1 \equiv T_2) \wedge \neg(T_1 \sim T_2)$
- $T_1 \geq T_2$ (pronounced “ T_1 subsumes T_2 ”) $\Leftrightarrow A_1 \supseteq A_2$
- $T_1 > T_2$ (pronounced “ T_1 strictly subsumes T_2 ”) $\Leftrightarrow A_1 \supset A_2$

So, in Figure 20, $T_2 \equiv T_3$, $T_1 \sim T_2$, $T_2 \equiv T_5$, and $T_2 > T_4$.

6.2 Simplistic Versus Realistic Property Theories

Practitioner and theorist alike will be skeptical of the NVV pattern and its accompanying equation in Eq.3, referred to as T_{NVV} , or “the [reliability] theory” in the following discussion. In fact, the theory might be dismissed out of hand as being simplistic, perhaps justifiably so. Still, being explicit about the basis for this dismissal highlights useful aspects of the theory of PECT.

The reliability theory can be mooted in a number of ways; for example, the vagueness of the definition of failure rate. However, our concern is not with reliability theory itself; assume that “failure rate” has a precise meaning and is measurable on components. Instead, our concern is with whether T_{NVV} is “simplistic” in the sense that it is unrealistically simple. To address this concern, it helps to look at analytic assumptions that are implicit in the theory.³⁹ A small selection of assumptions is presented in Table 1 (there are many assumptions in this and any other property theory):

Table 1: Selected (Analytic) Assumptions of Reliability Theory

1. A well-formed assembly satisfies the voter topology.
2. Failure rate (π) is defined for each sensor.
3. All sensors have the same failure rate.
4. Voter and connectors are 100% reliable.
5. Sensor output space is finite and small. ^a
6. Sensor failure rate is uniform over all input.
7. Sensor responses remain synchronized.
8. Failures are independent events.

a. This assumption simplifies the voting protocol.

39. The issue is not that the assumptions are implicit—that is a by-product of the necessarily attenuated description of the theory itself. As already noted, a complete description of any reasoning framework, while necessary in some contexts, would overwhelm the main points of discussion in this report.

The basic question is, “which assumption, or set of assumptions, brands the reliability theory as simplistic?” More to the point, what criteria can be used to make this assessment? One reasonable test is whether an assumption is overly constraining: in short, is it realistic to assume that systems of interest—that is, systems likely to be built in practice—will satisfy the assumption? If the answer is “no,” any theory based on that assumption is likely to be simplistic. If, conversely, the answer is “yes,” then the theory can be considered as “realistic,” at least for the satisfying system.

The assumptions in Table 1 can be characterized in PECT terms by defining each i^{th} assumption as a predicate P_i on assemblies that evaluates to “True” if the assumption i is satisfied by the assembly, and “False” otherwise; in this way, assumptions are converted to well-formedness constraints. Then, the reliability of $FTController$ (from Figure 18) can be established if and only if $P1(FTController) \wedge P2(FTController) \wedge \dots \wedge P8(FTController)$.

It is possible, even in the absence of specific application requirements, to conduct a thought experiment on whether a sufficiently large set of assemblies can be constructed that is well formed with respect to these predicates. After all, if no such experiment is possible, the original critique of the reliability theory as simplistic must be discounted. The results of the experiment are presented in Table 2.

Table 2: Plausible Satisfiability of NVV Well-Formedness Constraints

P1 is satisfiable.	By virtue of the ACT, it is a simple topology.
P2 is satisfiable.	By assumption (see earlier assumption about the validity of failure rate)
P3 is satisfiable.	Even though the design pattern “assumes” that each component has a different implementation, the failure rate $\pi = .001$ might be a minimal norm for components. Eq.3 can also be generalized, without altering the theory, to handle unique failure rates.
P4 is satisfiable.	The binomial distribution can be generalized easily to handle additional sources of failure events. It is also plausible that the correctness of a voter component can be formally verified. Failure-free connectors can be assumed in some (but not all) situations.
P5 is satisfiable, given plausible assumptions.	The temperature sensor in the example might report <i>temperature</i> $t \in \{safe, caution, critical\}$; a switch sensor might report <i>position</i> $p \in \{closed, closing, opening, open\}$.
P6 is satisfiable, with loss of precision.	If components exhibit a range of failure rates over their input ranges, the worst rate can be chosen, yielding a worst-case (pessimistic) estimate for the assembly. Assigning a distribution to failure rates would require substantial changes to the theory (a new theory), especially if the distributions were nonstandard.
P7 is satisfiable, assuming RMA schedulability as a precondition.	The NVV pattern requires synchronization to ensure that voting refers to the same event. This could be done by tagging sensor output or through some other interaction protocol. In this illustration, the approach is to guarantee that sensor deadlines are never missed. Adding the RMA schedulability precondition, ^a in effect, defines the ordering $T_{RMA} > T_{NVV}$.

Table 2: *Plausible Satisfiability of NVV Well-Formedness Constraints (cont'd.)*

<p>Satisfiability of P8 is problematic, but plausible.</p>	<p>This is a key assumption of most reliability theories;^b its satisfaction is the primary motivation for n-version programming. However, even the assumption that the sensors are independent versions is not sufficient—erroneous assumptions may be found in the shared specification for sensors. The FTController assembly, as depicted, likely executes on one processor—another single source of failure. A distributed assembly could be constructed, but distributed inter-process communication will not be 100% reliable (see P4).</p>
---	--

a. The schedulability claim for FTController is an explicit annotation on Figure 18 on page 42.

b. The term “common mode failure” is often used in reliability literature to denote the condition in which the assumption of independent failure events is not satisfied.

To summarize the results of Table 2: assumptions 1, 3, and 4 are readily satisfiable; assumptions 6 and 7 are satisfiable, assuming that a pessimistic estimate of reliability is acceptable, and assuming that T_{RMA} is a valid theory. The satisfiability of assumptions 2 and 8 is no more (or less) questionable here than in any other reliability theory. This leaves only assumption 5, which is satisfiable in any assembly in which sensors have a limited output range, a common scenario.

It appears that peremptory dismissal of T_{NVV} may not be justifiable. As simple as the theory is, its viability appears to be dependent only on the application setting (i.e., assumption 5) and not due to anything intrinsically weak about the theory itself. **In more general terms, a property theory, however simple it appears, can be dismissed as simplistic only if the set of assemblies well formed in that theory is a small subset of the assemblies likely to arise in practice.**

6.3 Optimizing Qualities of Reasoning Frameworks

The discussion in the previous section concerned the quality of property theories called “simplistic,” or put another way, with the question of judging when a particular property theory T_X exhibits (the quality called) “realism.” The criterion for making this judgement was reduced to relating the set A_X of analytically well-formed assemblies in T_X to the set of stereotypical assemblies in some potential solution set, call it A_S . In this view of the matter, a necessary condition for judging a theory to be “simplistic” is $A_S \supset A_X$.

There are other qualities of property theories, several of which reflect different aspects of the the “complexity” quality. Some examples of these other qualities include

- computational complexity: growth rate of the decision procedure’s time/space costs
- confidence bounds: objective statements about theory reliability, stability, and accuracy
- certification effort: the cost of acquiring required component properties
- verification/validation complexity: the cost of showing theory soundness and validity
- end-user complexity: the level of difficulty perceived by users of a property theory

The subsumes relation (indicated by the $>$ symbol) was defined in the introduction to Section 6. Assuming that T_X and T_Y address the same property (e.g., latency), the subsumes relation can be thought of as a preference relation—that is, “**all else being equal**, T_X is preferable to T_Y if and only if $T_X > T_Y$.” Similar ordering relations can, in principle, be defined for each of the above itemized qualities, each with an analogous meaning in “preference” to that of subsumes.

The key phrase for each preference relation, though, is “all else being equal.” Not surprisingly, the above qualities of theories interact with one another. For example, a decrease in the end-user complexity of a property theory (a desirable result) might be achieved by abstracting some phenomenon from the theory, which leads to nondeterminism—that is, behavior not predicted by the theory—and this, in turn, reduces the accuracy and stability of the theory (an undesirable result).⁴⁰

In this characterization, a classical multi-objective⁴¹ optimization solution is required to integrate a new property theory into a PECT. In general, this is a hard problem; however, the emphasis on formal (or at least explicit) representation of ACTs and reasoning frameworks and on automated interpretation and reasoning imposes a helpful constraining influence on the integration process. These constraining influences are harnessed through “co-refinement.” The co-refinement process derives its name from the coincident and iterative corrections (or “refinements”) made to an ACT and property theory. The main concepts are shown, in Figure 21, as a trace over four iterations (steps 0 through 3) of the co-refinement process.

40. Assuming of course that the abstracted phenomenon is not extraneous to begin with.

41. There is an enormous body of literature on multi-objective decision theory, also known as “multi-attribute decision theory,” “multi-objective optimization,” “multi-objective design,” and other synonyms and their permutations. Roy provides a concise but accessible formalization of preference relation, preference structure, and families of decision procedures [Roy 91].

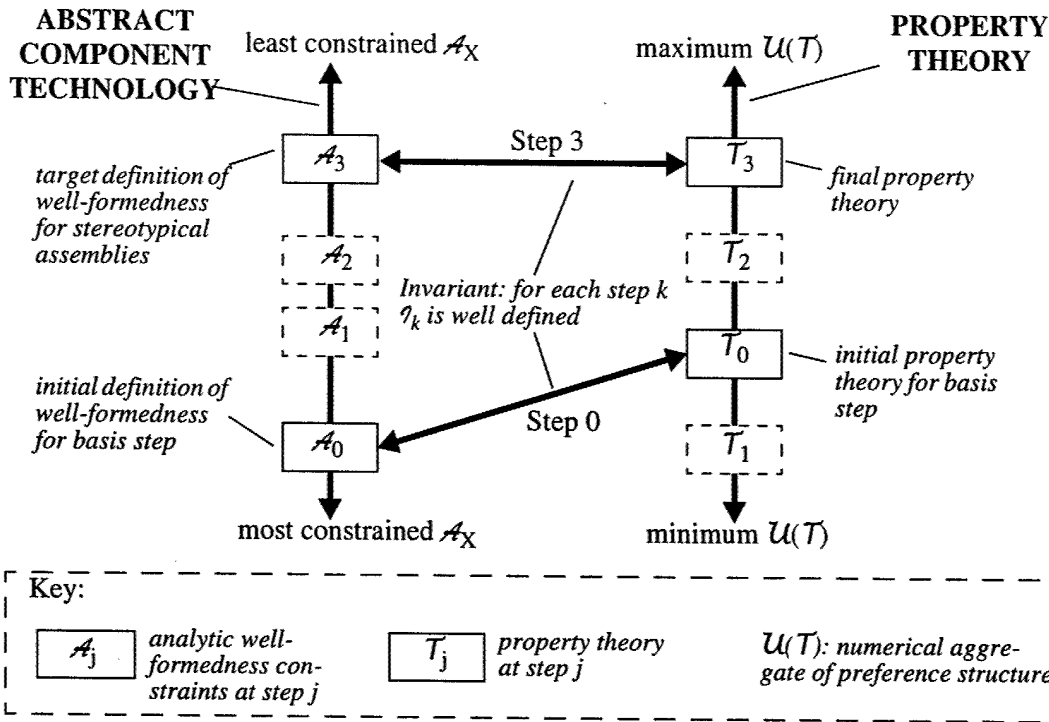


Figure 21: Time Sequence of a Series of Co-Refinement Steps

The vertical arrow on the left side of Figure 21 represents the level of restrictiveness of assembly constraints, perhaps quantified (intuitively) as the cardinality of the set of well-formed assemblies. The vertical arrow on the right side represents the state of the multi-objective optimization problem; the hypothetical function $U: T \rightarrow \mathbb{N}$ assigns an integer score to a property theory to quantify its level of optimality or utility.⁴² All else being equal, fewer constrained well-formedness rules are preferred to more constrained rules, since fewer constraints translate to a greater number of possible assemblies, and therefore a more general property theory. Likewise, greater values of U are preferred to lesser values, since the former translate to better tradeoffs of theory qualities.

There are different strategies for converging on the right balance between restrictiveness and utility. The one illustrated in Figure 21 begins with highly restrictive assembly constraints and proceeds by systematically relaxing these constraints while adding utility to the property the-

42. There is nothing special about this function, which can take many forms. It is a standard component of the vast majority of multi-objective decision aids. Strictly speaking, the utility function should be defined on a vector of valuations assigned to each objective, rather than on the theory as a whole.

ory. An alternative is to start with target assembly constraints and a property theory with little or no utility, and then add assembly restrictions as needed, while improving theory utility.

In Figure 21, the co-refinement process begins with the basis step, Step 0, that involves defining highly restrictive well-formedness rules—which can, in fact, restrict well-formedness to a single model assembly—and specifying a property theory that “solves” the model problem posed by the model assembly(ies). The crucial element of the basis step is that it establishes an interpretation from the model assembly(ies) to the property theory. **The only invariant between steps in any co-refinement process is that an interpretation is defined as the post-condition of each step.**

Each j^{th} step after the basis step is guided by a primary objective and a secondary objective. The **primary** objective is that \mathcal{T}_j subsumes \mathcal{T}_{j-1} , that is, $\mathcal{T}_j \geq \mathcal{T}_{j-1}$, as defined earlier. This subsumption is valid only because of the invariant on the defined-ness of interpretations. That is, if $\mathcal{A}_j \supseteq \mathcal{A}_{j-1}$, the interpretation \mathcal{I}_j ensures $\mathcal{T}_j \geq \mathcal{T}_{j-1}$, by definition. The **secondary** objective is that $\mathcal{U}(\mathcal{T}_j) \geq \mathcal{U}(\mathcal{T}_{j-1})$. The intent is that the primary objective takes precedence over the secondary objective.

However, neither objective is sacred; in some circumstances, the secondary objective may have priority; in others, regression in one or both objectives may be warranted in the interest of “one step backward, two steps forward.” The possibility of regression on utility is shown at Step 1 in Figure 21, as $\mathcal{T}_1 \geq \mathcal{T}_0$, but $\mathcal{U}(\mathcal{T}_0) \geq \mathcal{U}(\mathcal{T}_1)$. Such pragmatic decisions have been encountered in practice (see *Predictable Assembly of Substation Automation Systems: An Experiment Report* for a case study of co-refinement [Hissam 02b]).

An interesting question is how to define the termination condition (or exit criteria) for co-refinement. Figure 21 suggests that some target set of stereotypical assemblies was defined, perhaps prior to Step 0. This might be the case if a PECT were developed for an established product line. Even so, there will likely be normative requirements imposed on the property theory—perhaps a required accuracy or confidence interval that must be achieved by a property theory (again, see *Predictable Assembly of Substation Automation Systems: An Experiment Report* for a case study of normative requirements on property theories [Hissam 02b]).

6.4 Incompatibility Among Reasoning Frameworks

Even assuming that $\mathcal{T}_j \geq \mathcal{T}_{j-1}$ is a primary objective for successive steps in co-refinement, it is possible for incompatibilities to arise, that is, $\mathcal{T}_j \sim \mathcal{T}_{j-1}$ or $\mathcal{T}_j \equiv \mathcal{T}_{j-1}$. To illustrate, a performance theory at Step j might be modified to rely on a task-scheduling policy different from the policy assumed at Step $j-1$; for example, a change from “fixed-priority scheduling” to “earliest-dead-line-first scheduling.” Since tasks in an assembly cannot be scheduled simultaneously with two different policies, the two property theories (whatever they are) cannot be valid simulta-

neously, hence $T_{j-1} \sim T_j$. However, this incompatibility is not problematic since, by definition, T_{j-1} and T_j would never be applied simultaneously to the same assembly.

For the purpose of this discussion, let reasoning frameworks that are defined as a consequence of a particular co-refinement activity be called homogeneous; assume that, at most, one of these reasoning frameworks will be used to reason about the same assembly.⁴³ Also, let reasoning frameworks that span different co-refinement activities be called heterogeneous; assume that one or more of these frameworks can be used simultaneously to reason about the same assembly. For the following discussion, Greek subscripts are used to distinguish heterogeneous from homogeneous reasoning frameworks—that is, T_α and T_β are heterogeneous, while T_j and T_k are homogeneous.

In the case of co-refinement, the incompatibilities $T_j \sim T_{j-1}$ and $T_j \cong T_{j-1}$ are not, in general, problematic to the end user of a PECT, since T_j and T_{j-1} are homogeneous. However, complications do arise with incompatibilities among heterogeneous theories, and in particular where $T_\alpha \cong T_\beta$. In this case, the two properties may be susceptible to interference. Interference arises when, for example, changes to an assembly specification are made to optimize predicted properties in T_α , but where those changes violate the well-formedness constraints imposed by some other property theory T_β . Figure 22 depicts a canonical form of interference, using analogous Venn diagram and naming conventions used in Figure 20.

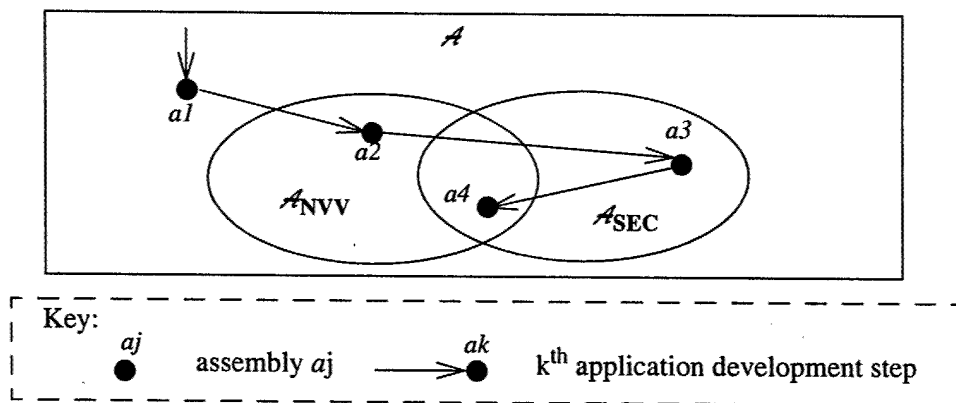


Figure 22: Interference Among Property Theories

To illustrate interference, imagine a PECT that supports some security theory— T_{SEC} . Say that T_{SEC} can be used to predict the probability that the confidentiality of message traffic can be compromised within a specified time interval. Further, to be well formed in T_{SEC} , an assembly

43. It is possible, and may be desirable, for more than one homogeneous theory to be embedded in its own reasoning framework and integrated into the same PECT.

must encrypt messages explicitly with the encryption component `Crypto`, or implicitly with the (security environment-specific) connector \sim_{crypto} . In this illustration, \mathcal{T}_{SEC} will be used in conjunction with \mathcal{T}_{NVV} , and we assume $\mathcal{T}_{\text{SEC}} \cong \mathcal{T}_{\text{NVV}}$.

Now, let assembly $a1$ be `Controller`, as depicted in Figure 14 on page 25. Assume that our objective is to ensure that $a1$ satisfies specific reliability and security requirements that are predictable in \mathcal{T}_{SEC} and \mathcal{T}_{NVV} . Unfortunately, $a1$ is not well formed to either \mathcal{T}_{SEC} or \mathcal{T}_{NVV} . With luck, tools will provide diagnostics that provide clues about how to repair the specification. As a practical matter, $a1$ might first be modified to be well formed in \mathcal{T}_{NVV} ; assume that the resulting $a2$ assembly is `FTController`, as depicted in Figure 18 on page 42. At this point, $a2$ is modified yet again to be well formed to \mathcal{T}_{SEC} ; assume that the resulting assembly $a3$ is `FTSecController`, as depicted in Figure 23.

Unfortunately, there are various ways in which `FTSecController` is no longer well formed with respect to \mathcal{T}_{NVV} . For example, it probably violates any naive definition of the voter topology. Only a bit more subtle is the violation of the \mathcal{T}_{NVV} assumption about independent failure modes—`Crypto` is a single source of failure (see P8 in Table 2 on page 48). To complete the scenario, assembly $a3$ is modified to replace `Crypto` with an encrypted connector, resulting in $a4$ assembly in Figure 22, which is well formed to both \mathcal{T}_{NVV} and \mathcal{T}_{SEC} .

This is, admittedly, a rather simple illustration—it is just as easy to imagine that, with only two property theories to accommodate, the initial sequence of assemblies $\{a1, a2, a3\}$ might never have been specified, or $a1$ could be transformed directly into $a4$. On the other hand, the scenario would look more likely if the hypothetical PECT had supported a dozen or more heterogeneous property theories, with several homogeneous alternatives for each such theory. Although this more complicated scenario has not yet arisen, it is inevitable should the PECT concept achieve widespread industrial use.

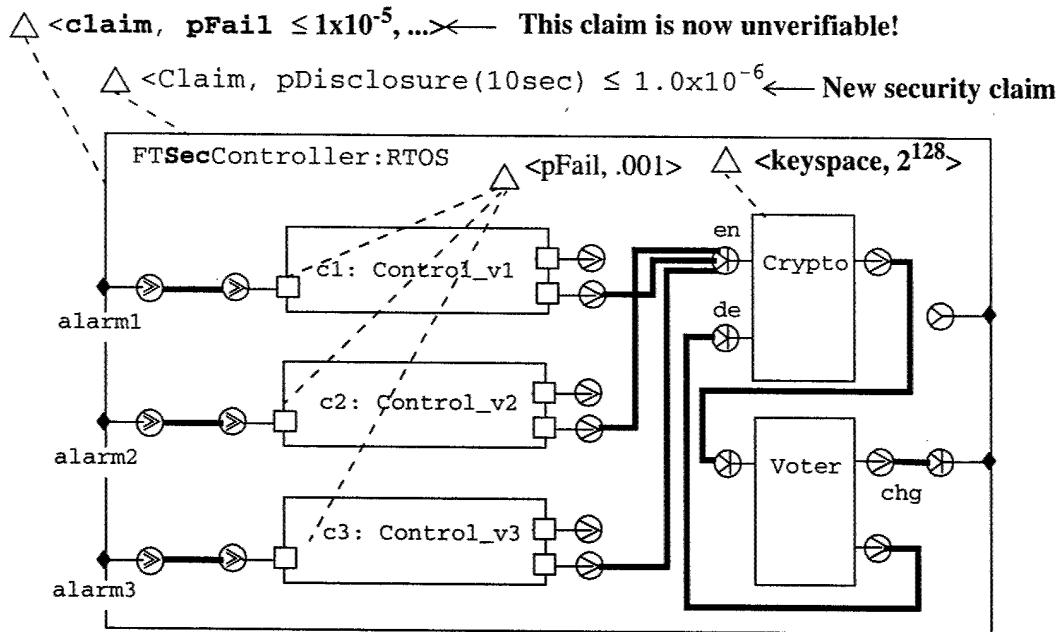


Figure 23: Interference Between Security and Reliability Property Theories

6.5 Dealing with Incompatibility

Incompatibilities among heterogeneous reasoning frameworks are almost certainly inevitable, but they must be managed and, where possible, eliminated. PECT is as yet an immature technology, and as such, a sufficient number of reasoning frameworks don't exist to make this a pressing near-term issue. However, as the issue directly affects end users and issues of scale, it is vital that sound and practical approaches to dealing with incompatibility be developed. Three approaches are described below, from the least to the most speculative.

Multi-Refinement

One way to minimize the potential for heterogeneous incompatibility is by introducing a "multi-refinement" process that has as its primary objective maximizing the extent of $\cap \mathcal{A}_*$, where * ranges over the set of heterogeneous theories being integrated.

Multi-refinement might yield one or more intersection theories. For example, the new reliability theory $T_{\text{NVV}'}$ might be defined, $T_{\text{NVV}} \geq T_{\text{NVV}'}$, that requires a reliable filter to be interposed between unreliable components and the Voter component. Similarly, a new $T_{\text{SEC}'}$ might be defined, $T_{\text{SEC}} \geq T_{\text{SEC}'}$, that requires the use of Crypto and disallows the use of \sim_{crypto} . Then, $T_{\text{SEC}'} \equiv T_{\text{NVV}'}$, where both can be thought of as intersection theories designed "in the context" of one another. Here, though, $T_{\text{SEC}'}$ seems a bit too restrictive. An alternative is to leave T_{SEC} as it is and define $T_{\text{NVV}'}$ to allow, but not require, cryptographic

topologies such as in `FTSecController`. Then, although $\mathcal{T}_{\text{SEC}} \cong \mathcal{T}_{\text{NVV}}$, at least matters have improved somewhat, since $\mathcal{T}_{\text{SEC}} \cap \mathcal{T}_{\text{NVV}} \supseteq \mathcal{T}_{\text{SEC}} \cap \mathcal{T}_{\text{NVV}}$.

In brief, as with co-refinement, selecting a mix of intersection theories for a set of heterogeneous property theories will almost certainly involve multi-objective optimization. Although this is certainly achievable for a reasonably small number of reasoning frameworks (say, fewer than five), the process will be messy and ad hoc if many theories, with overlapping constraints, are introduced. In the long run, a more rigorous approach will be required.

Expert Tool Assistance

The interference between the reliability and security theories also strongly implied the importance of automated detection of theory incompatibility, as well as detection of the introduction of incompatibilities where previously there were no incompatibilities. However, achieving automation will be challenging.

For example, defining well-formedness constraints for a single property theory, with sufficient formality to support automated checking, is nontrivial. However, identifying contradictory constraints over heterogeneous reasoning frameworks, where each will likely have its own “domain of discourse,” will be doubly challenging. A large number and variety of automated constraint management tools have been developed to develop artificial intelligence applications. Such tools, in concert with some form of type system for property theories and other construction-language-sensitive tools, may well form the core of automated tool support for PECT users.

The notion of tactics for quality attribute design [Bachmann 02], currently being investigated by Bachmann, Bass, and Klein, is also intriguing and appears to have bearing on interference. The authors sketch a decision procedure that operates on analyzable design fragments known as tactics and produces refinements of those fragments.⁴⁴ In terms of the above scenario, the procedure manages design dialogues leading to the sequence of refinements $\{a2_1, a2_2, \dots, a2_j\}$ to improve reliability, and refinements $\{a3_1, a3_2, \dots, a3_k\}$ to improve security.⁴⁵ As currently defined, the procedure regards these dialogues as independent; it does not yet address refinement over multiple interacting theories. An extension to manage an agenda of design tactics

44. See *Deriving Architectural Tactics: A Step Toward Methodical Architectural Design* (SEI technical report CMU/SEI-2003-TR-004, currently in development) for a description of the following decision procedure.

45. This description takes liberties with the concept of a design tactic that operates at a more primitive level of abstraction than property theories. One motivation for design tactics is for them to serve as building blocks for attribute-based architecture styles (ABASs) [Kazman 99]. The NVV pattern and its property theory \mathcal{T}_{NVV} can be defined as an ABAS.

spanning several property theories and the associated techniques to compose tactic-based design fragments would support the scenario envisaged in Figure 22 (e.g., $\{a2_1, a3_1, a2_2, \dots\}$).

A Type System for Reasoning Frameworks

A speculative alternative to multi-refinement is to define a type system for reasoning frameworks that would permit them to be selected on the basis of a well-defined subtype relation. That is, rather than adapting an assembly to meet the well-formedness requirements of a property theory, it might be possible to select, or even infer, the “best fit” reasoning framework for a given requirement or assembly.

For example, the decision procedure for T_{NVV} might be defined as $eval: \mathcal{M}_{\text{NVV}} \times V_{\text{NVV}}$, where V_{NVV} is the probability of failure—that is, $\forall v \in V_{\text{NVV}} \cdot 0 \leq v \leq 1$. Since \mathcal{M}_{NVV} is derivable, we can instead use $eval: \mathcal{A}_{\text{NVV}} \times V_{\text{NVV}}$. With this notation, a conventional contravariant/covariant subtype relation $X <: Y$ (pronounced “ X is a subtype of Y ”) on homogeneous reasoning frameworks can be defined in a straightforward way: $T_{*'} <: T_* \Leftrightarrow \mathcal{A}_{*'} \subseteq \mathcal{A}_* \wedge V_{*'} \subseteq V_*$. That is, $T_{*'}$ is a subtype of T_* if and only if it applies to a superset of assemblies and yields a subset of behaviors. This can be further generalized; for example, by also encoding in V_{NVV} the notion of confidence. In this case, $T_{*'}$ is a subtype of T_* if and only if it applies to a superset of assemblies and yields a subset of predicted behaviors with at least as much confidence.

This discussion, of course, risks trivializing the task of defining such a type theory. For one thing, conventional type theory assigns one domain of values to each syntactic phrase (i.e., its semantic domain); in a construction language, each phrase would have several semantic domains, one for each reasoning framework. Further, a typing scheme along the lines described above would require that reasoning frameworks be compositional; as discussed in Chapter 7, this is not always possible. Nonetheless, the potential benefits of treating “nonfunctional” properties on a par with functional properties (as in the type theory of modern programming languages) is certainly intriguing.

7 Compositional Reasoning

It is almost (but not quite) a tautology that components are valuable insofar as they are composable with other components. In the parlance of software component technology, “compositionality” is simply assumed. As observed in Chapter 2, however, current parlance tends to focus almost exclusively on the constructive aspects of composition—namely, on the binding of component labels to enable their runtime interaction.

Compositionality is also desirable for the analytic aspects of component-based development; however, unlike constructive compositionality, it cannot be assumed. It is also not an easy or straightforward topic—a substantial body of research literature under the general heading of “compositional reasoning” has been developed since the early 1980s. To the extent that there is a general understanding of compositional reasoning and the closely related notion of modular reasoning, this understanding appears to be limited to particular research communities, such as those concerned with formal (demonstrable) verification and, more particularly, with verification of concurrent systems. In fact, even within these research niches, there is continued interest in developing first principles of compositionality.

To state the conclusion first: **Support for compositional reasoning is a desirable property of reasoning frameworks. However, PECT does not always support compositional reasoning. It does, however, always support reasoning about compositions.** The following discussion will clarify and substantiate this conclusion.

The following discussion is based on a survey paper by de Roever that also served as an introduction to the proceedings of a conference devoted to the first principles of compositional reasoning [de Roever 98]. The terminology and definitions used by de Roever have been adapted to better fit the terminology and definitions provided in this report. There is, therefore, some risk that this adaptation does violence to de Roever’s intent.

7.1 Compositional and Modular Reasoning

According to de Roever:

“The purpose of a compositional verification approach is to shift the burden of verification from the global level to the local component level, so that global properties are established by composing together independently (specified and) verified component properties.” [de Roever 98]

This is entirely consistent with the objectives of a PECT, modulo its restriction to demonstrable reasoning (i.e., verification)—an important proviso and one that must be generalized to accommodate plausible reasoning, if the purpose of compositionality is to be achieved by PECT.

Setting aside for the moment the distinction between component and assembly, the following outlines a compositional proof scheme. To compositionally demonstrate that a component C satisfies⁴⁶ some property P , written $C \vDash P$

1. If C cannot be further decomposed, then demonstrate $C \vDash P$ directly on C .
2. If C can be decomposed into components $\{C_1, C_2, \dots, C_k\}$, then
 - a. Find properties $\{P_1, P_2, \dots, P_k\}$ and demonstrate that $P_1 \wedge P_2 \wedge \dots \wedge P_k \Rightarrow P$.
 - b. Recursively demonstrate for each j^{th} component in $\{C_1, C_2, \dots, C_k\}$ that $C_j \vDash P_j$.

Interestingly, the above (widely accepted) compositional proof scheme works “top-down”—that is, it is a decompositional proof scheme. Modular proof schemes, in contrast, work “bottom up”—that is, they begin with non-decomposable components and their demonstrated properties. The task is to show that a composite property on a composite component is entailed by these previously established component properties and so on, in a “bottom-up” fashion.

It can be observed, however, that while the distinction between compositional and modular reasoning may be significant to the theory of PECT in the long run (with modular reasoning apparently closer to what is needed where systems are composed from preexisting components), the description of reasoning frameworks in this chapter does not expose this distinction. In particular, Step 1 above is delegated to component certification, and Step 2a is delegated to the design and validation of a reasoning framework; Step 2b, however, has not been assumed to be a valid reasoning step for all reasoning frameworks.

In fact, “component” was not given an inductive definition in Section 3.1, and the different forms of hierarchy defined for assembly in Section 3.5 flow from the fact that Step 2b is **not** always a valid reasoning step for **every** property theory (although it **might** be valid for **some** property theories). This is one reason why compositionality and modularity—as currently formulated—impose conditions that are too strong in practice, even if they are ideal in theory. A justification of this important assertion is discussed next.

46. Recall that even plausible (i.e., empirical) properties can be expressed as factual assertions. Thus, casting the following discussion into the terms of formal logic does not restrict the applicability of the discussion to demonstrable reasoning. Note also that in this discussion, the notion of “ C satisfies p ” (or “ $C \vDash p$ ”) refers to the usual definition of this term in formal logic, and not to that given in the general definitions of Section 6.

7.2 Why Compositionality Is Too Strong

As observed by de Roever, the proof Steps 2a and 2b lead to the following technical requirements, recast in PECT terms. For each n -ary connector \rightsquigarrow in (a CCL specification of) an ACT, an n -ary operator $\rightsquigarrow_{\mathcal{T}}$ exists in (a calculus and logic of) a property theory \mathcal{T} such that

Requirement 1: If $C_j \vDash P_j$, $1 \leq j \leq n$, then $\rightsquigarrow(C_1, C_2, \dots, C_n) \vDash \rightsquigarrow_{\mathcal{T}}(P_1, P_2, \dots, P_n)$.

Requirement 2: If $\rightsquigarrow(C_1, C_2, \dots, C_n) \vDash P$, then some P_j exists for each C_j , $1 \leq j \leq n$, such that $C_j \vDash P_j$ and $\rightsquigarrow_{\mathcal{T}}(P_1, P_2, \dots, P_n) \Rightarrow P$.

These requirements flow quite naturally from the compositional proof scheme. The first states that each connector has some corresponding property theory that describes the semantics of the connector. The second states that properties can be decomposed into wholly independent subproperties.

These conditions are too strong, however, and exclude property theories that are nonetheless quite useful. In fact, none of the three property theories illustrated in Sections 4.4 - 4.6 satisfy these requirements:

- Neither $\mathcal{T}_{\lambda ABA}$ (discussed Section 4.5) nor \mathcal{T}_{NVV} (Section 4.6) are defined on individual connectors— $\mathcal{T}_{\lambda ABA}$ is defined on sequences of interactions beginning with periodic stimuli, while \mathcal{T}_{NVV} is defined on its own topological arrangement;⁴⁷ this fact violates requirement 1.
- The timing properties of an assembly of components (discussed in Section 4.5) cannot be decomposed into independent subproperties—the timing behavior of one component may (and will likely) influence the timing behavior of others; this fact violates requirement 2.
- It is, in general, impossible to reason compositionally about liveness properties in temporal logic (discussed in Section 4.4)—the “rely/guarantee” reasoning procedure is compositional, but, in general, works only for safety properties; this fact violates requirement 2.

This is not to say that no compositional (or modular) theory for reasoning about liveness, timing, or reliability is possible. Of course, such theories must exist, since the assembly-level behaviors exist and since computation is deterministic. But the effort to develop or use such theories might outweigh the benefits of compositionality. For example, there is an interaction between components and a shared central processing unit (CPU) that is abstracted by $\mathcal{T}_{\lambda ABA}$.

47. It is interesting to speculate about the objection to the compositionality of \mathcal{T}_{NVV} . The NVV pattern of interaction could, in principle, be encoded as an n -ary connector for m unreliable components, 1 voter and 1 client. A corresponding polyadic operator, \rightsquigarrow_{NVV} , would be defined to take as its argument one or more failure probabilities and return a failure probability. This would satisfy requirement 1.

The CPU is a global resource; changes in the timing behavior of one component are transmitted through this global resource, leading to a loss of compositionality. However, this loss is offset by a simplification in the reasoning procedure, both in terms of human and computational complexity.

Before deciding if the loss of compositionality is worth the gain in simplicity, it is worth reflecting on the value of compositionality.

7.3 Why Compositionality Is Important

Compositionality implies all the benefits inherent to the “divide and conquer” problem-solving strategy, namely, significant reductions in complexity. Where automated reasoning is used (e.g., in model checking), a lack of compositionality leads to exponential growth complexity, either in the number of states in a model, or in the number of propositions in a temporal logic claim. At this time an objective for research in model checking is to discover an adequate resolution to exponential complexity, and no wholly satisfactory compositional approach has yet been defined to address this complexity.

Also, without compositional reasoning to link components to assembly properties, the long sought-after goal of independently substitutable component parts will remain chimerical. Until design theories permit reasoning about the behavior of components (and assemblies) compositionally—that is, in a way that is independent of other components (and assemblies)—unexpected interactional behaviors⁴⁸ among components will remain an open-ended source of undocumented intercomponent dependencies. As long as these dependencies exist, one component can never be substituted for another without the expectation of unpredictable results.

Consider the analogy between module types in higher order functional languages such as ML [Harper 94] and a theory of component types based upon compositional component properties. In ML, a subtype relation on modules⁴⁹ $M_1 <: M_2$ is defined such that M_1 can always be

48. Unexpected and unexplained interactional behaviors are usually also undesirable, as is any source of unpredictability in an engineering discipline. Such behavior is sometimes denoted as emergent behavior. Any effort to understand or control emergence is an effort to achieve predictable assembly. Earlier reports on PECT went as far as to assert that the terms “assembly property” and “emergent property” were synonymous. However, because the term “emergent property” has been widely associated with “unpredictable property,” it is probably better to concede the term and move on.

49. Essentially, a module in ML is a set of functions, each of which has its own type signature and exists within a well-defined type lattice. The analogy being drawn here is between modules in ML and components in CCL. Of course this does not do justice to the theory of higher order module types, but it is sufficient to situate the analogy.

“safely” substituted for any module M_2 —that is, no function in M_1 will be “surprised” by a value it receives from the environment in which it is deployed, and no function in M_1 will compute a value that “surprises” its environment. Achieving a similar feat for software components will require that, in addition to the ML sort of functional-type theory, an extra-functional-type theory be defined **for each** of the assembly-level properties of interest to application assemblers. That is, by analogy with type theory $C_1 < C_2$ if and only if $\forall P \cdot C_1.P < C_2.P$, where P includes the functional interface and all extra-functional properties of a component.

Compositional property theories might be abandoned in the near term for reasons of expediency, but, ultimately, such theories are required to achieve the degree of scalability and flexibility that is so often promised to be the inevitable accompaniment to software component technology.

8 Status and Future Work

The major elements of PECT have been demonstrated in progressively more demanding proofs of feasibility [Hissam 02a], [Hissam 02b]. The overall design structure of a PECT has proven sufficient for these initial trials, and several parts of a generic PECT development infrastructure are currently being developed—for example, a trace-based execution monitor, and tools for specifying and processing CCL specifications. Currently, a more ambitious proof of feasibility in industrial robotics is underway that incorporates automated model checking and performance theories based in both RMA and real-time queuing theory.

Other areas of near-term effort are described below:

- The PECT concept, as described in this report, focuses mainly on a theory and infrastructure for predictable assembly. Of equal importance is a theory and infrastructure for certifiable components. The technology for functional certification, perhaps along the lines of contracts [Meyer 97], extended to software components [Meyer 98], is a precondition to PECT. Future work will be more explicit about this assumption, and generalize it to address certification of properties defined and required by reasoning frameworks.
- The emphasis of PACC research, to date, has been on the science and technology of predictable assembly. Although PECT is relatively immature and although the theory of PECT will continue to evolve, a body of experience with the main ideas is accumulating, and the basic elements of a development method are emerging—for example, co-refinement and statistical validation of property theories. Future work will capture those elements, initially as method fragments, and later as an integrated method.
- The emphasis of PACC research, to date, has been on systems with highly deterministic, periodic, and reactive behavior, and property theories that are either verifiable (e.g., model checking) or present a clear falsification strategy (e.g., latency). Future work will extend the focus to systems exhibiting increasingly stochastic behavior (e.g., behavior sensitive to the distribution profiles of stimuli) and property theories whose falsification strategies are not inherently clear (e.g., reliability theories based on statistical testing of component reliability).

Areas for longer range research include developing

- sound value propositions for certifying components for predictable assembly
- automation for simultaneously optimizing assemblies for one or more properties
- type systems for component specifications extended to nonfunctional properties

9 Conclusions

*Expressiveness arises from strictures: restrictions entail stronger invariants.
Flexibility arises from controlled relaxation of strictures, not from their
absence.*

—Robert Harper⁵⁰

This report has described the structure and underlying theory of PECT. Answers to the questions that were posed in Section 1 are now provided, in the terminology of PECT.

Which characteristics of an assembly make it predictable, and which kinds of assembly properties can be predicted? An assembly is predictable if it is well formed with respect to the assembly constraints imposed by one or more property theories. An assembly is then said to be predictable with respect to the properties addressed by these theories. Property theories can be developed for any property that can be shown, in an objective way, to adhere to (well-formed) assemblies. Objective evidence can be demonstrable—in the form of verifiable proof—or plausible—in the form of empirical observation.

What characteristics of a component make it certifiable, and what kinds of component properties can be certified? A component is certifiable if it has properties that can be demonstrated, in an objective way, to adhere to the component. As before, objective evidence can be demonstrable or plausible. Any such property can be the subject of certification, but only those properties that are parameters to a property theory are of direct interest to a PECT. In this case, the property theory always provides a definition of component properties in sufficient detail, and with sufficient rigor, to enable their certification.

How can we achieve objective and measurable confidence in certified component properties and predicted assembly properties? All property theories provide an explicit and objective basis for confidence, both in the predictions themselves, and in the

50. This quotation is from the invited presentation titled “The Practice of Type Theory in Programming Languages,” Dagstuhl 10th Anniversary Symposium. Saarbruecken, August, 2000. Available online at <<http://www-2.cs.cmu.edu/~rwh/talks/Dagstuhl%202000.ppt>>.

component properties on which they depend (as described in the previous two answers). **Measurable** confidence, in the form of statistical labels, attends any component or assembly property that depends directly on empirical evidence. Even formally demonstrated properties are amenable to statistical treatment, where proof results depend on assumptions that can be validated through observation.

Can a technology infrastructure that provides answers to these questions be systematically developed and transitioned into practice? A PECT provides for two parallel frameworks, one for specifying and deploying well-formed assemblies of components (the construction framework), and one for imposing well-formedness constraints and providing automated analysis tools that exploit these constraints. A discipline for systematically developing and integrating these frameworks, and for validating the resulting integration, has been demonstrated in increasingly realistic industrial proofs of feasibility.

Glossary

abstract component technology	a vocabulary and notation for specifying components, assemblies, and their runtime environments in a component-technology-independent way, and for specifying the constraints, imposed by reasoning frameworks, that must be satisfied for predictions to be valid
abstract interpretation	a map from the symbols of a calculus (in a property theory) to elements of a (not necessarily component-based) computing system. See also <i>definite interpretation</i> .
analytic closure	the minimum scope of component interactions for which the assumptions of a particular reasoning framework can be satisfied
analytic constraints	constraints imposed by one or more reasoning frameworks on an abstract component technology
annotation	a property P associated with a referent R , meaning that “ R has property P ,” denoted as $R.P$
assembly	a set of components and their enabled interactions
assembly constraints	behavioral and topological rules of well-formedness imposed on components and assemblies by one or more (real) component technologies, and one or more reasoning frameworks
automated reasoning procedure	a decision procedure, a definite interpretation, and a definite inverse interpretation, each susceptible to full automation. See also <i>property theory</i> .
binding label	a linking mechanism embedded in components to enable their interaction with other components. See also <i>pin</i> .
component	an implementation in final form, modulo bound binding labels, that provides an interface for third-party composition and is a unit of independent deployment

connector	a mechanism provided by the runtime environment that enforces an interaction protocol, or discipline, on the components that are participants in an interaction
construction framework	an abstract component technology, tools to enforce assembly constraints, and other tools used to automate the specification, development, and deployment of components and their assemblies
construction language	a language for specifying abstract component technologies (ACTs) and their well-formed components and assemblies
constructive closure	See <i>containment</i> .
compose, composed, composition	components that have interactions enabled through some connector are composed; composition (n): is a set of such enabled interactions; compose (vb): to enable component interaction. See also <i>assembly</i> .
contained, containment	All interactions among components are restricted to the scope of the most immediately enclosing (“containing”) assemblies and partial assemblies.
co-refinement	a process for developing reasoning frameworks, and in particular, for finding an acceptable tradeoff among various qualities of a reasoning framework, such as generality, complexity, and stability
decision procedure	a function that evaluates claims made on assemblies, described in the property theory, to the values “true” or “false”
definite interpretation	a map from assemblies specified in a concrete syntax of a construction language to strings in the input language of the decision procedure. See also <i>property theory</i> .
definite inverse interpretation	a map of the results of a decision procedure back to the concrete syntax of the construction language. See also <i>property theory</i> .
deploy	defines where (in which instance of a runtime environment, and, ultimately, on which physical computing device) component behavior is executed

empirical evidence	evidence acquired through direct observation, preferably under controlled circumstances, with results reported in well-defined units of measure. Empirical evidence is therefore provisional, as any other observation might have been different. See also <i>formal evidence</i> .
final form	a software specification that is ready for execution on a physical or virtual machine. See also <i>component</i> .
formal evidence	evidence acquired through mathematical proof. Formal evidence is therefore irrefutable, as all such proofs are tautological. See also <i>empirical evidence</i> .
gateway	a connector that enables interactions among components deployed in different environments
heterogeneous reasoning framework	property theories developed in different co-refinement activities. One or more heterogeneous theories can be used simultaneously to reason about the same assembly. See also <i>homogeneous reasoning framework</i> .
homogeneous reasoning framework	property theories developed in a particular co-refinement activity. In general, only one homogeneous theory will be used to reason about the same assembly. See also <i>heterogeneous reasoning framework</i> .
interaction	a composition of two or more reactions, from distinct components, using a runtime-environment-provided connector
interpretation	See <i>abstract interpretation</i> , <i>definite interpretation</i> .
multi-refinement	a generalization of co-refinement for finding an acceptable tradeoff among various qualities of heterogeneous reasoning frameworks
null junction	a graphical notation on partial assemblies that indicates which pins of components contained in the partial assembly are visible outside the assembly

partial assembly	a (recursively defined) abstraction that aggregates a set of components and their enabled interactions, and exposes selected component pins through null junctions. Logically, a partial assembly is a component implemented entirely in terms of other components. See also <i>assembly</i> .
prediction theory	synonym for property theory
property	an n -tuple $\langle name, value, \dots \rangle$, where <i>name</i> and <i>value</i> refer to the name of some property and the value it takes, respectively. See also <i>annotation</i> .
property theory	a calculus, logic, and abstract interpretation that provides an objective, rigorous, and verifiable or falsifiable basis for predicting the properties of assemblies. See also <i>prediction theory</i> .
reaction	specifies the behavior of units of concurrency within a component (e.g., a thread) and the behavioral dependencies between the sink pins and source pins of a component
reasoning framework	comprises a property theory, an automated reasoning procedure, and a validation procedure
pin	binding labels in the construction and composition language (CCL). See also <i>source pin</i> , <i>sink pin</i> , <i>connector</i> .
runtime environment	provides runtime services that may be used by components in an assembly, provides an implementation for one or more connectors, and enforces assembly constraints
sink pin	a pin that accepts interactions with the environment of a component (i.e., from other components or the runtime environment). See also <i>pin</i> , <i>source pin</i> .
source pin	a pin that initiates interactions with the environment of a component (i.e., from other components or the runtime environment). See also <i>pin</i> , <i>sink pin</i> .
unit of independent deployment	A component is independently deployable if all its dependencies on external resources are clearly specified (e.g., as pins), and if it can be substituted for, or substituted by, some other component. See also <i>deployment</i> .

validation procedure provides an objective basis for trusting the validity and soundness of a reasoning framework, and defines its required component properties with sufficient rigor to provide an objective basis for trust in assertions of component behavior

Acronym List

ABAS	attribute-based architectural style
ACT	abstract component technology
API	application program interface
CCL	construction and composition language
CPU	central processing unit
CTL	computational tree logic
DoD	Department of Defense
EJB	Enterprise JavaBean
FIFO	first in, first out
IR&D	internal research and development
NVV	n-version majority voting
OCL	object constraint language
OO	object oriented
PACC	predictable assembly of certifiable components
PECT	prediction-enabled component technology
RMA	rate monotonic analysis
SEI	Software Engineering Institute
UML	Unified Modeling Language

Bibliography

URLs are valid as of the publication date of this document.

- [Achermann 01]** Achermann, F.; Lumpe, M.; Scheider, J.; & Nierstrasz, O. "Piccola—a Small Composition Language," 403-426. *Formal Methods for Distributed Processing: A Survey of Object-Oriented Approaches*. New York, NY: Cambridge University Press, 2001.
- [Allen 97]** Allen, R. *A Formal Approach to Software Architecture*. PhD diss., CMU-CS-97-144, Carnegie Mellon University, May 1997.
- [Ayer 46]** Ayer, A. *Language, Truth, and Logic*, 2nd ed. London, England: V. Gollancz, 1946.
- [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*, 2nd ed. (CMU/SEI-2000-TR-008, ADA379930). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. <<http://www.sei.cmu.edu/publications/documents/00.reports/00tr008.html>>.
- [Bachmann 02]** Bachmann, F.; Bass, L.; & Klein, M. *Illuminating the Fundamental Contributors to Software Architecture Quality* (CMU/SEI-2002-TR-025, ADA407778). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. <<http://www.sei.cmu.edu/publications/documents/02.reports/02tr025.html>>.
- [Bass 98]** Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*. Reading, MA: Addison-Wesley, 1998.

- [Bass 01]** Bass, L.; Buhman, C.; Comella-Dorda, S.; Long, F.; Robert, J.; Seacord, R.; & Wallnau, K. *Volume 1: Market Assessment of Component-Based Software Engineering Assessments* (CMU/SEI-2001-TN-007, ADA395250). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. <<http://www.sei.cmu.edu/publications/documents/01.reports/01tn007.html>>.
- [Booch 99]** Booch, G.; Rumbaugh, J.; & Jacobson, I. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.
- [Box 98]** Box, D. *Essential COM*. Reading, MA: Addison-Wesley, 1998. (See especially Chapter 1 for a concise summary of design rationale for the design of COM.)
- [Buschmann 96]** Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; & Stal, M. *Pattern-Oriented Software Architecture*. New York, NY: John Wiley and Sons, 1996.
- [Buskens 02]** Buskens, V. *Social Networks and Trust*. Boston, MA: Kluwer Academic Publishers, 2002.
- [Cardelli 98]** Cardelli, C. & Gordon, A. D. "Mobile Ambients," 140-155. *Proceedings of Foundations of Software Science and Computation Structures FoSSaCS'98* (Lecture Notes in Computer Science LNCS 1378). Lisbon, Portugal, March 30-31, 1998. New York, NY: Springer-Verlag, 1998.
- [Clarke 99]** Clarke, E.; Grumberg, O.; & Pelad, D. *Model Checking*. Cambridge, MA: MIT Press, 1999.
- [Clements 96]** Clements, P. *A Survey of Architecture Description Languages*. <<http://www.sei.cmu.edu/publications/articles/survey-adl.html>> (1996).
- [Clements 02a]** Clements, P. & Northrop, L. *Software Product Lines: Practices and Patterns*. Boston, MA: Addison-Wesley, 2002.
- [Clements 02b]** Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Nord, R.; & Stafford, J. *Documenting Software Architectures: Views and Beyond*. Boston, MA: Addison-Wesley, 2002.

- [Corbett 00]** Corbett, J.; Dwyer, M.; Hatcliff, J.; Laubach, S.; Pasareanu, C.; & Hongjun Zheng, R. "Banadera: Extracting Finite-State Models from Java Source Code," 439-448. *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*. Limerick, Ireland, June 4-11, 2000. New York, NY: Association for Computing Machinery, 2000.
- [Cross 02]** Cross S.; Forrester, E.; Hissam, S.; Kazman, R.; Levine, L.; Linger, R.; Longstaff, T.; Monarch, I.; Smith, D.; & Wallnau, K. *SEI Independent Research and Development Projects (CMU/SEI-2002-TR-023, ADA407792)*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002.
<<http://www.sei.cmu.edu/publications/documents/02.reports/02tr023.html>>.
- [Daniels 97]** Daniels, F.; Kim, K.; & Vouk, M. *The Reliable Hybrid Pattern: A Generalized Software Fault Tolerant Design Pattern*.
<<http://st-www.cs.uiuc.edu/users/hanmer/PLoP-97/Proceedings/daniels.pdf>> (1997).
- [DeMillo 77]** DeMillo, R.; Lipton, R.; & Perlis, A. "Social Processes and Proofs of Theorems and Programs," 206-214. *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*. Los Angeles, California, January 17-19, 1977. New York, NY: Association for Computing Machinery, 1977.
- [de Roever 98]** de Roever, W.-P. "The Need for Compositional Proof Systems: A Survey," 1-22. *Proceedings of the International Symposium COMPOS'97 (Lecture Notes in Computer Science LNCS 1536)*. Bad Malente, Germany, September 8-12, 1997. New York, NY: Springer-Verlag, 1998.
- [Fenton 97]** Fenton, N. & Pleeger, S. *Software Metrics: A Rigorous and Practical Approach* (ISBN 1-85032-275-9). Boston, MA: PWS Publishing Company, 1997.
- [Garlan 97]** Garlan, D.; Monroe, R.; & Wile, D. "Acme: An Architecture Description Interchange Language," 169-183. *Proceedings of CASCON '97*. Toronto, Canada, November 10-13, 1997. Toronto, Ontario, Canada: IBM Canada Ltd. Laboratory, Centre for Advanced Studies, 1997.

- [Harel 95]** Harel, D. & Naamad, A. *The STATEMATE Semantics of State-charts* (Technical Report CS95-31). Rehovot, Israel: The Weizmann Institute of Science, Department of Applied Mathematics and Computer Science, 1995.
- [Harper 94]** Harper, R. & Lillibridge, M. "A Type Theoretic Approach to Higher-Order Modules with Sharing," 123-137. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. Portland, Oregon, January 17-21, 1994. New York, NY: Association for Computing Machinery, 1994.
- [Havelund 01]** Havelund, K. & Rosu, G. *Testing Linear Temporal Logic Formulae on Finite Execution Traces* (RIACS Technical Report 01-08). Moffett Field, CA: Research Institute for Advanced Computer Science, 2001.
- [Hissam 02a]** Hissam, S.; Moreno, G.; Stafford, J.; & Wallnau, K. "Packaging Predictable Assembly," 108-125. *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment* (Lecture Notes in Computer Science Volume 2370). Berlin, Germany, June 20-21, 2002. Berlin, Germany: Springer Verlag, 2002.
- [Hissam 02b]** Hissam, S.; Hudak, J.; Ivers, J.; Klein, M.; Larrison, M.; Moreno, G.; Northrop, L.; Plakosh, D.; Stafford, J.; Wallnau, K.; & Wood, W. *Predictable Assembly of Substation Automation Systems: An Experiment Report* (CMU/SEI-2002-TR-031). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. <<http://www.sei.cmu.edu/publications/documents/02.reports/02tr031.html>>.
- [Hoare 85]** Hoare, C. A. R. *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice Hall, 1985.
- [Hoare 93]** Hoare, C. A. R. "Algebra and Models." *SIGSOFT Software Engineering Notes* 18, 5 (December 1993): 1-8.
- [Huth 00]** Huth, M. *Logic in Computer Science: Modelling and Reasoning About Systems*. New York, NY: Cambridge University Press, 2000.

- [Ivers 02] Ivers, J.; Sinha, N.; & Wallnau, K. *A Basis for Composition Language CL* (CMU/SEI-2002-TN-026, ADA407797). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. <<http://www.sei.cmu.edu/publications/documents/02.reports/02tn026.html>>.
- [Kazman 99] Kazman, R. & Klein, M. *Attribute-Based Architectural Styles* (CMU/SEI-99-TR-022, ADA371802). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. <<http://www.sei.cmu.edu/publications/documents/99.reports/99tr022/99tr022abstract.html>>.
- [Klein 93] Klein, M.; Ralya, T.; Pollak, B.; Obenza, R.; & Harbour, M. G. *A Practitioner's Handbook for Real-Time Analysis*. Boston, MA: Kluwer Academic Publishers, 1993.
- [Kuhn 96] Kuhn, T. *The Structure of Scientific Revolution*, 3rd ed. Chicago, IL: University of Chicago Press, 1996.
- [Li 02] Li, P. L.; Shaw, M.; Stolarick, K.; & Wallnau, K. "The Potential for Synergy Between Certification and Insurance." <<http://www.sei.cmu.edu/staff/kcw/icsr02.pdf>> (2002).
- [Luckham 95] Luckham, D.; Augustin, L.; Kenney, J.; Veera, J.; Bryan, D.; & Mann, W. "Specification and Analysis of System Architecture Using Rapide." *IEEE Transactions on Software Engineering* 21, 6 (April 1995): 336-355.
- [Magee 99] Magee, J. & Kramer, J. *Concurrency State Models & Java Programs*. New York, NY: John Wiley and Sons, 1999.
- [McAllister 90] McAllister, D.; En Sun, C.; & Mladen, V. "Reliability of Voting in Fault-Tolerant Software Systems for Small Output-Spaces." *IEEE Transactions on Reliability* 39, 5 (December 1990): 524-534.
- [Merriam-Webster 93] Merriam Webster, Inc. *Merriam Webster's Collegiate Dictionary*, 10th ed. Springfield, MA: Merriam-Webster Incorporated, 1993.
- [Meyer 97] Meyer, B. *Object-Oriented Software Construction*, 2nd ed. London, UK: Prentice-Hall International, 1997.

- [Meyer 98] Meyer, B.; Mingins, C.; & Schmidt, H. "Providing Trusted Components to the Industry." *IEEE Computer* 31, 5 (May 1998): 104-105.
- [Milner 89] Milner, R. *Communication and Concurrency*. New York, NY: Prentice Hall, 1989.
- [Milner 99] Milner, R. *Communicating and Mobile Systems: The π -Calculus*. New York, NY: Cambridge University Press, May 1999.
- [Moreno 02] Moreno, G.; Hissam, S.; & Wallnau, K. "Statistical Models for Empirical Component Properties and Assembly-Level Property Predictions: Toward Standard Labeling." *Proceedings of the 5th International Workshop on Component-Based Software Engineering*, in conjunction with the 24th International Conference on Software Engineering (ICSE2002). Orlando, Florida, May 19-20, 2002. <<http://www.sei.cmu.edu/pacc/CBSE5/Moreno-cbse5-final.pdf>>
- [Musa 98] Musa, J. *Software Reliability Engineering*. New York, NY: McGraw-Hill, 1998.
- [Necula 97] Necula, G. "Proof Carrying Code," 106-119. *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-97)*. Paris, France, January 15-17, 1997. New York, NY: Association for Computing Machinery, 1997.
- [Nierstrasz 02] Nierstrasz, O.; Arevalo, G.; Ducasse, S.; Wuyts, R.; Black, A.; Muller, P.; Zeidler, C.; Genssler, T.; & van den Born, R. "A Component Model for Field Devices," 200-209. *Proceedings of the First IFIP/ACM Working Conference on Component Deployment (Lecture Notes in Computer Science Volume 2370)*. Berlin, Germany, June 20-21, 2002. Berlin, Germany: Springer Verlag, 2002.
- [Pierce 02] Pierce, B. *Types and Programming Languages*. Cambridge, MA: MIT Press, 2002.
- [Polya 54] Polya, G. *Mathematics and Plausible Reasoning: Volume I Induction and Analogy in Mathematics*. Princeton, NJ: Princeton University Press, 1954.

- [Papadopoulos 98]** Papadopoulos, G. & Arbab, F. *Coordination Models and Languages* (Technical Report SEN-R9834, ISSN 1386-396X). Amsterdam, The Netherlands: Centrum voor Wiskunde en Informatica, 1998.
- [Popper 92]** Popper, K. *The Logic of Scientific Discovery*. New York, NY: Routledge, 1992.
- [Purtillo 94]** Purtillo, J. "The Polyolith Software Bus." *ACM Transactions on Programming Languages and Systems* 16, 1 (January 1994): 151-174.
- [Rabinovich 00]** Rabinovich, S. *Measurement Errors and Uncertainties*, 2nd ed. (ISBN 0-387-98853-1). New York, NY: Springer Verlag, 2000.
- [Roy 91]** Roy, B. "The Outranking Approach and the Foundations of the ELECTRE Methods." *Theory and Decisions* 31, 1 (July 1991): 49-73.
- [Ryan 01]** Ryan, B. & Schneider, S. *Modeling and Analysis of Security Protocols*. New York, NY: Addison-Wesley, 2001.
- [Sharygina 02]** Sharygina, N. "Model Checking of Software Control Systems." PhD diss., University of Texas at Austin, 2002.
- [Shaw 96a]** Shaw, M. "Truth vs. Knowledge: The Difference Between What a Component Does and What We Know It Does," 181-185. *Proceedings of the Eighth International Workshop on Software Specification and Design*. Schloss, Germany, March 22-23, 1996. Los Alamitos, CA: IEEE Computer Society Press, 1996.
- [Shaw 96b]** Shaw, M. & Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall, 1996.
- [Simon 96]** Simon, H. *The Sciences of the Artificial*, 3rd ed. Cambridge, MA: MIT Press, 1996.

- [Stafford 01a]** Stafford, J. & Wallnau, K. "Predicting Feature Interactions in Component-Based Systems," 35-42. *Proceedings of the Workshop on Feature Interaction of Composed Systems*, in conjunction with the 15th European Conference on Object-Oriented Programming (ECOOP 2001). Budapest, Hungary, June 18-22, 2001. New York, NY: Springer, 2001.
- [Stafford 01b]** Stafford, J. & Wallnau, K. "Is Third-Party Certification Necessary?" <http://www.sei.cmu.edu/pacc/CBSE4_papers/StaffordWallnau-CBSE4-22.pdf> (2001).
- [Stafford 02]** Stafford, J. & McGregor, J. "Issues in Predicting the Reliability of Composed Components." <<http://www.sei.cmu.edu/pacc/CBSE5/StaffordMcGregor-cbse5.pdf>> (2002).
- [Szyperski 97]** Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Reading, MA: Addison-Wesley, 1997.
- [TCSEC 85]** Department of Defense. *Trusted Computer Security Evaluation Criteria (TCSEC)*, DoD Standard DOD 5200.28-STD. <<http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html>> (1985).
- [UL 98]** Underwriter's Laboratory. *UL Standard for Safety for Software in Programmable Components*. Northbrook, IL: Underwriter's Laboratory, 1998.
- [van Ommering 02]** van Ommering, R. Part 5, Ch. 12, "The Koala Component Model," 223-236. *Building Reliable Component-Based Software Systems*. Boston, MA: Artech House, 2002.
- [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)*, in conjunction with the European Conference on Object-Oriented Programming (ECOOP 2001). Budapest, Hungary, June 19, 2001. New York, NY: Springer, 2001. <<http://research.microsoft.com/users/cszypers/events/WCOP2001/>>.
- [Warmer 99]** Warmer, J. & Kleppe, A. *The Object Constraint Language: Precise Modeling With UML*. Reading, MA: Addison-Wesley, 1999.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (leave blank)		2. REPORT DATE April 2003	3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Volume III: A Technology for Predictable Assembly from Certifiable Components			5. FUNDING NUMBERS F19628-00-C-0003
6. AUTHOR(S) 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-2003-TR-009
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-2003-009
11. SUPPLEMENTARY NOTES			
12.a DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS			12.b DISTRIBUTION CODE
13. ABSTRACT (maximum 200 words) <p>This report is the final volume in a three-volume series on component-based software engineering. Volumes I and II identified market conditions and technical concepts of component-based software technology, respectively. Volume III (this report) focuses on how component technology can be extended to achieve predictable assembly from certifiable components (PACC). An assembly of software components is predictable if its runtime behavior can be predicted from the properties of its components and their patterns of interactions. A component is certifiable if its (predictive) properties can be objectively measured or otherwise verified by independent third parties. This report identifies the key technical concepts of PACC, with an emphasis on the theory of prediction-enabled component technology (PECT).</p>			
14. SUBJECT TERMS software components, predictable assembly, component-based development			15. NUMBER OF PAGES 98
			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

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102