Predictable Assembly of Substation Automation Systems: An Experiment Report, Second Edition

Scott Hissam
John Hudak
James Ivers
Mark Klein
Magnus Larsson
Gabriel Moreno
Linda Northrop
Daniel Plakosh
Judith Stafford
Kurt Wallnau
William Wood

Initial Publication September 2002

Revised September 2003

TECHNICAL REPORT
CMU/SEI-2002-TR-031
ESC-TR-2002-031

Pittsburgh, PA 15213-3890

Predictable Assembly of Substation Automation Systems: An Experiment Report, Second Edition

CMU/SEI-2002-TR-031
ESC-TR-2002-031

Scott Hissam
John Hudak
James Ivers
Mark Klein
Magnus Larsson
Gabriel Moreno
Linda Northrop
Daniel Plakosh
Judith Stafford
Kurt Wallnau
William Wood

Initial Publication September 2002

Revised September 2003

Predictable Assembly from Certifiable Components

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 Scondras
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.

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.

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

List of Figures

List of Tables

Acknowledgements

The authors would like to thank Otto Preiss and Holger Hoffman from ABB Ltd. for providing invaluable expertise on substation automation. We also wish to thank Ivica Crnkovic and Mälardalen University for their continuing support of our work in predictable assembly.

Preface to 2nd Edition

Our motivation for producing a second edition of this report is primarily to provide a proper conclusion to the experiment in developing a prediction-enabled component technology (PECT) for substation automation systems. The first edition ended the story prematurely—it reported a PECT that satisfied requirements on the reliability of predictions, but which nonetheless exhibited a sufficient level of random error to leave us unsatisfied with the result, both in terms of the PECT itself, and in the methods used to specify and document the PECT.

The major revisions found in this new edition are limited to Appendices A and B, which document the λ_{ABA} reasoning framework and its empirical validation, respectively. The authors have resisted, wherever possible, revising this report to reflect the most recent developments in our approach to PECT building. This is explained by our preference for an editorial process that has a good chance of terminating. Nonetheless, where practical, we indicate more recent thinking in footnotes.

Executive Summary

Background

The Predictable Assembly from Certifiable Components (PACC) Initiative at the Software Engineering Institute (SEI[SM])[1] is developing methods and technologies for predictable assembly. A software development activity that builds systems from components is *predictable* if the runtime behavior of an assembly of components can be predicted from known properties of components and their patterns of interactions (connections), and if these predictions can be *objectively validated*. A component is *certifiable* if these known properties can be obtained or validated by independent third parties.

The SEI's approach to predictable assembly[2] is through prediction-enabled component technology (PECT). At the highest level, PECT is a scheme for systematic and repeatable integration of software component technology, software architecture technology, and design analysis and verification technology. This scheme is not simply technological; it also includes the processes needed to design and validate the predictive powers of a PECT. A PECT, then, is a *packaging* of engineering methods and a supporting technical infrastructure that, together, enable predictable assembly from certifiable components.

PACC became an SEI initiative on October 1, 2002. Prior to that, from 1999-2001, PACC was considered an exploratory research project. The objective of such a project at the SEI is to develop an understanding of a problem area and proposed solutions for it. During the third year of exploratory research (2001-2002), the SEI—in partnership with an industrial sponsor, the ABB Ltd. Corporate Research Center (ABB/CRC)—undertook the prototyping effort described in this report.

Objective

The objective of this prototyping effort was not to develop a PECT solution for a particular design problem per se, but rather to explore as many aspects of PECT as was practical within a limited time frame. Our primary concerns were methodological and practical: could we define a process for developing, and validating PECTs? Would the process be practical? What are the risks to transitioning PECT to practice? What are the technical limits of PECT, and how might

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

2. In this report, the term *predictable assembly* means *predictable assembly from certifiable components*.

they be overcome? On more than one occasion, our prototyping effort was diverted to explore these and similar questions.

Approach

To focus and ground our research, we selected as a prototype problem the substation automation system (SAS), an application area in the domain of power generation, transmission, and management. This problem area was chosen because it is well bounded, well defined, and representative of the broader class of critical infrastructure systems. The SEI defined three model problems in the predictable assembly of SAS: the assembly of an operator station, a primary equipment controller, and an integrated operator/controller. The nominal objective of the prototype was to develop three PECTs, one for each model problem.

Results of This Work

This report describes the results of an experimental application of PECT to SAS, focusing on the primary equipment PECT, the SAS switch controller. A low-power, high-speed switch was developed by the SEI to simulate primary equipment. Controller assemblies were developed in accordance with the International Electrotechnical Commission (IEC) 61850 standard for substation controllers, and laboratory scenarios were developed for external switch control and over-current protection. A family of latency models and their interpretations (λ_*) were developed to predict point-to-point, intra-assembly (intra-controller) latency. One member of this family, average case latency with blocking and asynchrony (λ_{ABA}), was empirically validated.

Primary results. The primary results of this work were methodological. We developed an overall process model for the design, development, and validation of PECTs. We studied two key processes in depth: *co-refinement* and *empirical validation*. Technically, *co-refinement* is the process of defining an interpretation from component model to analysis model. Intuitively, it is a negotiation that results in a component model that is expressive enough to span an intended application area, but restrictive enough to ensure that analysis is tractable. Technically, *empirical validation* defines measurement procedures for obtaining component measures and assembly measures, and for designing the experiments to compare the predicted and observed assembly measures. Intuitively, it is a means of attaching meaningful statistical labels to components and the predictive powers of PECT analysis.

Secondary results. The secondary results of this work were technological. We developed a measurement and validation infrastructure and gained valuable experience in the construction of a laboratory environment for empirical validation. More significantly, we specified an *analytically extensible* component model. This model defines the basics of component interface and connection topology (assembly), and is extended by formal interpretations to one or more analysis models. An analysis model supports compositional reasoning about assembly properties P_A and defines the component properties P_C required to predict P_A. Each interpretation

may impose additional constraints on components and their topologies. In return, each analysis model guarantees that assemblies of components satisfying these constraints will be analyzable, and hence be *predictable by construction*, with respect to P_A.

Although the work emphasized empirical predictability, *formal* approaches to predictability were also explored. Specifically, safety conditions for the prototype (hardware) switch and IEC 61850 (software) controller dealing with the over-current protection components were expressed in branching time temporal logic. State machines for these components were developed, and the safety and liveness conditions were verified using the SMV model checker.[3]

Tertiary results. The tertiary results of this work were the PECTs themselves. A PECT was developed for the operator interface and controller. The operator PECT was developed on the Microsoft .NET environment. Latency predictions for operator interfaces were accurate (< 3% magnitude of relative error [MRE]) but not technically demanding. More interesting was the controller PECT, which was substantially more complex and demanding. Controller assemblies introduced threading, contention, priority-based scheduling, and periodicity not encountered in the operator PECT.

The original normative requirement for controller latency prediction was a 90% confidence interval that 80% of predictions would not exceed an upper bound 5% MRE. This norm was arbitrary and not particularly demanding. Nevertheless, the results were encouraging. λ_{ABA} predictions match observations (expressed as an MRE) to within 0.5% for 80% of all predictions, with greater than 99% confidence that 0.5% is indeed the upper bound.

Although the requirements imposed on the property theory (λ_{ABA}) were relatively lax, and although the property theory itself was significantly restricted, the PECT exhibited many interdependencies among component technology, components, property theories, property-theory-imposed well-formedness rules, and use scenarios. Several inconsistencies in the PECT itself were discovered during the empirical validation presented in the first edition of this report. This, in turn, prompted improvements in the way PECTs are specified and documented, and to the revalidation of λ_{ABA} reported in this edition.

Next Steps

The results of the exploratory prototype are encouraging. However, as expected, there are aspects of PECT that require further elaboration, and some that have yet to be explored. This report identifies two areas requiring attention.

3. SMV is a symbolic model-checking tool developed at Carnegie Mellon University. For more information about it, see <http://www.cs.cmu.edu/~modelcheck/smv/smvmanual.ps>.

First, a technology infrastructure for specifying and deploying components and their assemblies must be constructed; the substation automation prototypes relied on manual processes for much of this. We developed a substantial suite of tools to support empirical validation, but greater automation is still required to generate the massive data sets required to adequately validate even moderately complex analysis models.

Second, the methods and technologies needed to support independent (third-party) measurement and certification of components and PECTs must be established; the substation automation prototypes relied on *in situ* measurements. Means must be established for acquiring measurements in third-party environments, establishing and maintaining trust in assertions about those measurements, and extrapolating those assertions to different, possibly heterogeneous development and deployment environments.

Abstract

The Predictable Assembly from Certifiable Components (PACC) Initiative at the Software Engineering Institute (SEI[SM]) is developing methods and technologies for predictable assembly. A software development activity that builds systems from components is *predictable* if the runtime behavior of an assembly of components can be predicted from known properties of components and their patterns of interactions (connections), and if these predictions can be objectively validated. A component is *certifiable* if these known properties can be obtained or validated by independent third parties. The SEI's technical approach to PACC rests on prediction-enabled component technology (PECT). At the highest level, PECT is a scheme for systematic and repeatable integration of software component technology, software architecture technology, and design analysis and verification technology. This report describes the results of an *exploratory* PECT prototype for substation automation, an application area in the domain of power generation, transmission, and management. This report focuses primarily on the methodological aspects of PECT; the prototype itself was only a means to expose and illustrate the PECT method.

1 Introduction

The Predictable Assembly from Certifiable Components (PACC)[4] Initiative at the Software Engineering Institute (SEI[SM])[5] is developing methods and technologies for predictable assembly. A software development activity that builds systems from components is *predictable* if the runtime behavior of an assembly of components can be predicted from known properties of components and their patterns of interactions (connections), and if these predictions can be objectively validated. A component is *certifiable* if these known properties can be obtained or validated by independent third parties.

The SEI is not alone in pursuing the objectives of PACC, although academic and commercial terminology may obscure this fact for those not deeply familiar with the subject. The SEI technical approach to PACC rests on prediction-enabled component technology (PECT). At the highest level, PECT is a scheme for systematic and repeatable integration of software component technology, software architecture technology, and design analysis and verification technology. The results of this integration are engineering methods and a supporting technical infrastructure that, together, enable *predictable assembly from certifiable components.*[6]

This report describes an *exploratory* prototype of PECT. We emphasize the word exploratory, since the objective of the work was to touch on as many methodological and technological aspects of PECT as possible, using the broad constraints imposed by an industrially significant problem area as a guide. We selected the problem area of substation automation, an application area in the domain of power generation, transmission, and management, because it is well bounded, well defined, and representative of the broader class of critical infrastructure systems.

4. For more information on this initiative, see <http://www.sei.cmu.edu/pacc>.

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

6. We will henceforth use the term *predictable assembly* to mean *predictable assembly from certifiable components*.

1.1 Background

In 2000, the SEI initiated a feasibility study in predictable assembly. The study was undertaken to

- identify the fundamental science and engineering challenges posed by predictable assembly and software component certification
- determine if the need to address these challenges was widespread and of economic or strategic interest to the U.S. Department of Defense (DoD) and to the software industry as a whole
- ascertain whether technological and methodological elements to address these challenges existed, or could be created with reasonable probability and effort

In 2001, the SEI formed a collaboration with the ABB Ltd. Corporate Research Center (ABB/CRC). This collaboration was established to undertake a two-year feasibility study of predictable assembly in an industrial setting. The application area was in power generation and transmission systems, specifically, substation automation systems (SASs). Typically, an SAS is implemented as a distributed system composed of 20 to 100 computing elements (personal computers and/or other electronic computing devices) executing a mix of soft- and hard-real-time applications. An SAS is characterized by stringent performance and reliability requirements.

1.2 Approach

The SEI/ABB joint feasibility study was intended to address two broad feasibility questions:

1. Can a practical technology infrastructure be developed to automate significant aspects of predictable assembly?
2. Can an engineering method be developed as a basis for the systematic improvement of an industry-wide capability in predictable assembly?

To conduct this feasibility exploration, the SEI and ABB defined a series of model problems. Briefly, a model problem is a simplification of a more complex one such that a solution to it can be extrapolated to that of a real problem. In 2001 and 2002, three model SAS problems were specified; in 2002 and 2003, they were extended to include the domain of industrial robotics.

1.3 Objective of This Report

This report consolidates the results and discoveries of the SEI/ABB investigation in predictable assembly. It is, in effect, a retrospective on an extended laboratory experiment. Our intent

is that this report serve as a basis for continued research within the SEI's PACC Initiative. Our intent is also that this report, when combined with related SEI publications on predictable assembly and PECT, will provide a valuable resource for other applied research in this field.

1.4 Audience for This Report

The audience for this report is computer scientists and software engineering researchers with an interest in predictable assembly from certifiable components (by this or any other name). We assume that readers have technical backgrounds that are both diverse and deep, spanning areas of software architecture, software component technology, probability and measurement theory, real-time systems, and model checking, to name the major areas reflected in this report. Therefore, this report is not meant to serve as a general introduction to predictable assembly, component technology, or PECT.

1.5 Structure of This Report

An overview of substation automation and the details of the SAS model problems are described in Chapter 2. In Chapter 3, we provide an overview of the technical concepts of PECT. The key workflows of a process for developing and validating a PECT are summarized in Chapter 4. In Chapter 5, we summarize the chronology of a key development activity in the design of a PECT, called co-refinement, the result of which was a controller PECT for predicting point-to-point latency for any controller assembly. The techniques used to empirically validate the SAS-controller PECT are outlined in Chapter 6. In Chapter 7, the focus shifts from empirical to formal approaches to predictable assembly, where we describe the use of model checking to verify safety and liveness conditions of controller assemblies using temporal logic. Chapter 8 extracts the key results of the work to date, emphasizing the benefits and risks of the PECT approach to predictable assembly. The plan for the next phase of our investigation is outlined in Chapter 9.

More details on the work are provided in appendices of this report. Appendix A describes the λ_{ABA} analysis model for predicting controller latency and its interpretation; this is the model that emerged from co-refinement. A detailed account of the empirical validation of λ_{ABA} is provided in Appendix B. In Appendix C, we provide the temporal-logic specifications and state model used to reason about controller safety conditions. Appendix D provides a schematic for the switch hardware developed for the prototype. The Acronym List on page 145 defines the acronyms used in this report.

2 Model Problems for Substation Automation

The following description of an SAS is excerpted from a paper by Preiss and Wegmann [Preiss 01]:

> *Networks (power grids) of different topologies are responsible to transport energy over short or long distances and finally distribute it to end-consumers (such as households and companies). The nodes in such a network are called substations .... Substations may be manned or unmanned depending on the importance of the station and also on its degree of automation. Substations are controlled by Substation Automation Systems (SAS). Since unplanned network outages can be disastrous, an SAS is composed of all the electronic equipment that is needed to continuously control, monitor, and protect the network. This covers all the high voltage equipment outside the substation (overhead lines, cables, etc.) as well as those inside the substation (transformers, circuit breakers, etc.).*

This combination of challenging and concrete requirements means that an SAS provides a good basis for model problems in predictable assembly. We outline the basic structure of the SAS model problems in Section 2.1. Background information on the International Electrotechnical Commission (IEC) 61850 standard, the SAS domain model used to develop the controller PECT prototype, is provided in Section 2.2. An overview of the hardware and software used in the operation and controller prototypes is provided in Section 2.3.

2.1 Three Model Problems and Three PECTs

The scale of an SAS depends on the cost, size, and criticality of the primary equipment that it manages. For the model problems described in this report, we envisage a manned SAS that has an operator workstation and one or more controllers, each of which controls some primary equipment (e.g., breakers, switches, and transformers). The operator has a graphical display console for observing and interacting with primary equipment. For the purpose of this report, we are not concerned with details such as the allocation of controllers to bays or the interaction of multiple controllers (e.g., bay interlocking). Instead, we assume a single operator console, a single primary switch, and a single controller for that switch. From this general structure, we defined three model problems, each leading to a PECT as its model solution.

The overall scheme is depicted in Figure 1. The SEI developed a low-power, high-speed switch (labeled “SEI Switch” in Figure 1) to play the role of primary equipment. One model problem was to develop a PECT for predictably assembling SEI switch controllers, a second model problem was to develop a PECT for predictably assembling operator interfaces, and a third model problem was to develop a higher-order PECT for composing controller and operator assemblies into an overall SAS application. This PECT would allow operators to monitor and control the state of the switch.

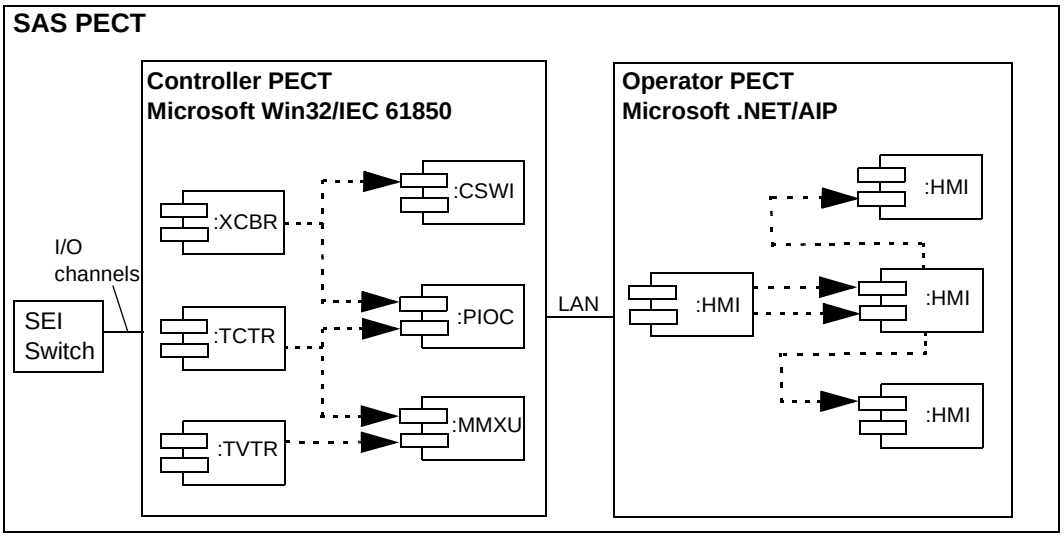

Figure 1: Three PECTs

The PECTs were intended to enable the prediction of the end-to-end latency of operations within an operator assembly, within a controller assembly, and over a combined operator/controller assembly. Nominal requirements were established for the accuracy and reliability of predictions and were expressed as statistical tolerance intervals:[7]

- controller PECT: 99% confidence that 80% of latency predictions will have a magnitude of relative error (MRE) < 5%
- operator PECT: 95% confidence that 80% of latency predictions will have MRE < 10%
- SAS PECT: 95% confidence that 80% of latency predictions will have MRE < 10%

It must be emphasized that these requirements were not selected with any consideration for the practicality of or resemblance to operational requirements. For example, the controller PECT was to be developed on Windows 2000, and we understood that it is problematic to achieve the stated tolerance interval on that platform (given that Windows 2000 is not a real-time operating system). Instead, the above norms were selected as a plot device for our study of empirical validation.

7. The statistical intervals selected for these model problems were derived from Preiss and Wegmann’s work [Preiss 01].

Although the model problems were purposely kept quite abstract, two technology decisions were made in the interest of reducing the gap between the model solutions and their ultimate deployment environments. The operator PECT was developed for the ABB Aspect Integration Platform (AIP). The AIP is a development and deployment environment for automation systems; it uses OPC[8] to enable operator interfaces to communicate with controllers. The controller PECT was developed using the IEC 61850 standard for substation automation.

This report focuses on the controller PECT, since that model problem posed the most challenges. Although the latency prediction enabled by the controller PECT is quite flexible in defining the end points of an "end-to-end" latency prediction, two specific controller scenarios were defined:

1. switch control latency. In this scenario, we are interested in predicting the end-to-end latency of an operation to manually "throw" the SEI switch.
2. over-current protection latency. In this scenario, we are interested in predicting the time required to detect an over-current condition and automatically "throw" the switch.

These scenarios were selected in part because they correspond to actual SAS functionality, as defined in the IEC 61850 standard.

2.2 The IEC 61850 Domain Model

The IEC 61850 standard defines a domain model of the functionality of substation automation in a form that is readily adopted for use with software component technology [Ivers 02]. Figure 2 depicts, in the Unified Modeling Language (UML), the IEC 61850 concepts that are most important for SAS model problems. An SAS consists of a physical system and a logical system. The physical system comprises primary equipment, such as switches, breakers, and transformers, and secondary equipment, such as intelligent electronic devices (IEDs) or other controller hardware. The logical system comprises the data and software functions for Supervisory Control and Data Acquisition (SCADA), Energy Management Systems (EMSs), and other substation applications.

The IEC 61850 standard defines a variety of functions and the quality attributes required for each, including reliability, accuracy, and latency. Each function is defined in terms of one or more logical nodes (LNs). The standard defines many LNs; each one is a primitive building block from which many SAS functions may be composed. Conceptually, then, there is a close correspondence between LNs and software components. Although it is possible, and perhaps reasonable, to deploy several LNs as a single component, we chose to maintain a one-to-one correspondence between LNs and the software components that implement them.

8. OPC stands for OLE (Object Linking and Embedding) for Process Control.

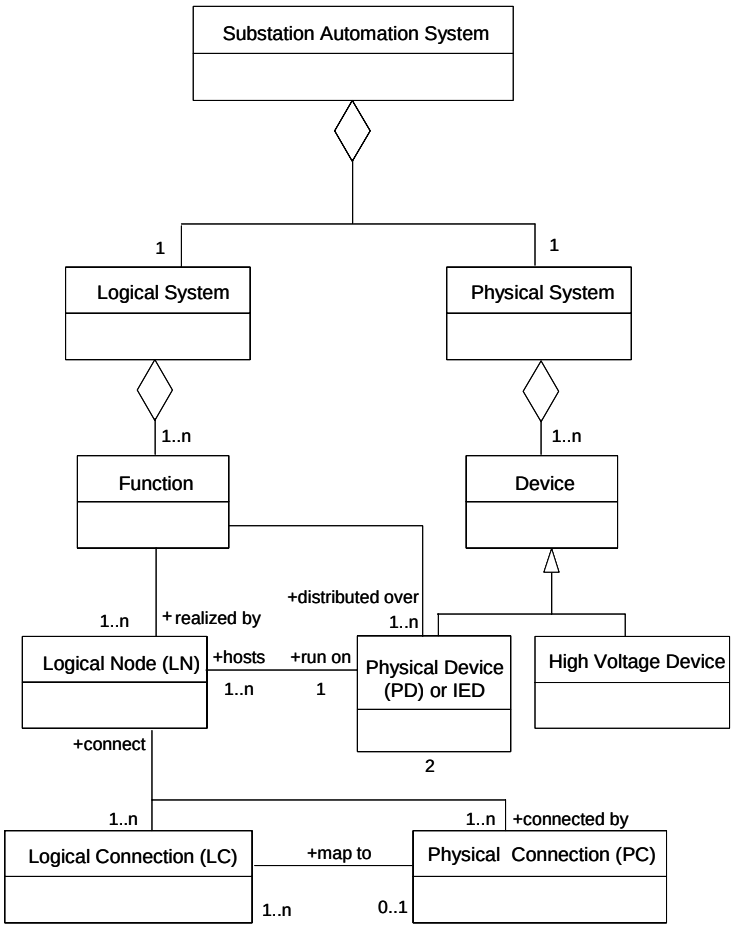

Figure 2: Key Concepts of the IEC 61850 Standard for a SAS Model Problem

The IEC 61850 components implemented for the controller PECT are summarized in Table 1. Only rudimentary functionality was implemented. (Other components that were developed but are not shown here include an OPC gateway for communicating with the operator PECT and a clock component for delivering periodic stimulus to controller components.)

Table 1: IEC 61850 LNs as Components of the Controller PECT

IEC 61850 LN	**Summary of Function**
TCTR	Calculates current
TVTR	Calculates voltage
MMXU	Calculates power
CSWI	Provides safety rules for the controller (e.g., select before use)
XCBR	Acts as the controller interface
PIOC	Provides over-current protection

2.3 Lab Environment and the SEI Switch

The hardware and software platforms used for the operator and controller PECTs were conventional, Intel-based personal computers running Microsoft's Windows 2000 operating system. The operator PECT was developed for Microsoft .NET in C#. The controller PECT used VenturCom's RTX[9] real-time extensions package to support priority-based scheduling. Other details of the hardware and software configuration for the controller PECT are described in Table 14 and Table 15 on page 120.

The SEI also developed a low-power, high-speed switch to facilitate the prototype development. An abstraction of this switch is depicted in Figure 3; the actual schematic is shown in Figure 61 on page 139.

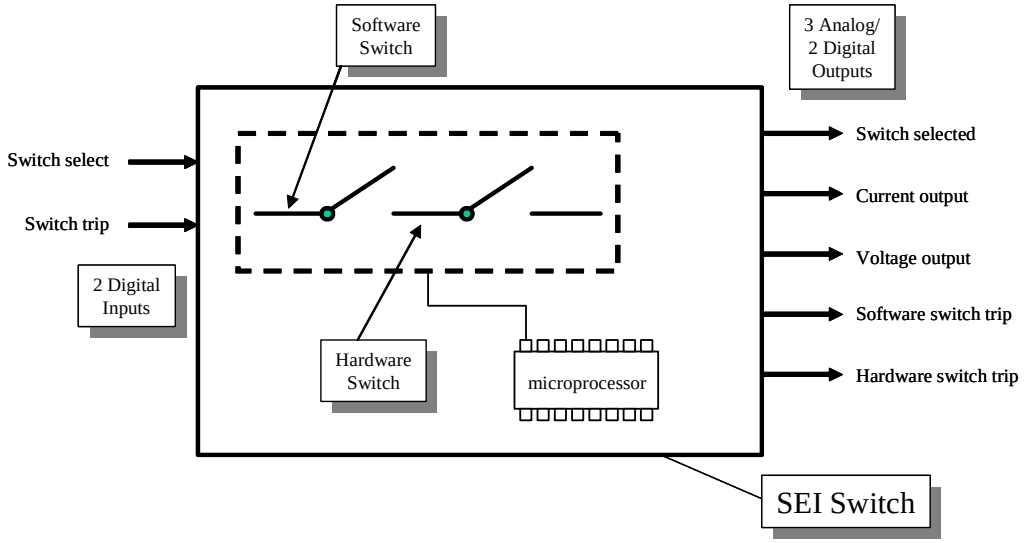

Figure 3: SEI Switch Abstraction

The SEI switch communicates with the software controller using two digital input channels for commands and five output channels (two digital and three analog) for status and values. Digital input is provided for the controller to select the switch and set its position to "open" or "close." Analog output provides positive feedback on switch selection status and reports on the switch's current, voltage, and position. The switch is implemented as a serial composition of two high-speed electronic switches, which are also known as TRIACs. One TRIAC represents the primary equipment being controlled (labeled "Software Switch" in Figure 3); the other is controlled internally on the SEI switch and used for over-current protection test scenarios (labeled "Hardware Switch" in Figure 3). The hardware switch is controlled by a microprocessor that monitors current and time. If the current exceeds a value for a specified duration (both are configurable), the microprocessor "throws" the hardware switch, representing a failure of the PIOC (see Table 1) component to protect the primary equipment.

9. See <http://www.vci.com> for details about this software package.

The software switch is operated by a software controller running on the Windows 2000 platform through a data acquisition card that is connected to the analog and digital lines. This controller is shown in detail in Figure 9 on page 23.

3 Prediction-Enabled Component Technology

Our approach to predictable assembly is to develop and use *prediction-enabled component technologies* (PECTs). We formulated the basic concepts of PECT prior to undertaking our SAS effort [Hissam 02], but refined and expanded them substantially as a result of the work reported here.

In this chapter, we first outline the theory of PECT in Section 3.1 and then describe the structure of a PECT implementation in Section 3.2. Next, we provide an overview of *Pin*, the core component model we use to build PECTs, in Section 3.3. Not all the Pin features described in this report were implemented in the model solutions; those that were are summarized in Section 3.4. Section 3.5 depicts the SAS software controller developed in Pin.

In the following discussion, we adopt the terminology of architecture documentation used by Clements and associates to describe PECT [Clements 02b]. You should consult their work if the meanings of these terms are not self-evident. Our only modification to this terminology is that, in some situations, we prefer the term *assembly* to **view**[10] for its connotations of compositionality.

3.1 The Theory of PECT

Component composition, in the literature, almost universally assumes an underlying component-and-connector **viewtype**.[10] Correspondingly, a PECT has a central **component-and-connector**[10] viewtype, called the *constructive* viewtype that models runtime components and their interaction topologies. In general, we denote a view in this viewtype as a *constructive assembly.* The constructive viewtype has one or more analysis viewtypes, called *analysis views*,[11] associated with it. Each analysis viewtype provides a basis for compositional reasoning about some runtime behavior; for example, latency, security, safety, liveness, or reliability.

10. As defined by Clements and associates [Clements 02b].

11. In previous documents, we referred to analysis views as analysis assemblies. *View* seems preferable to *assembly* here, because assembly has a stronger connotation of components and connectors than can be justified in the general case of views that support behavioral analysis technology. However, in this case study, the term *analysis assembly* turns out to be appropriate.

In PECT, analysis views are derived automatically from constructive assemblies by means of a syntactic transformation, called an *interpretation* (see Figure 4). An interpretation is a partial function from constructive assemblies to analysis views. Three interpretations are shown in Figure 4: I_{LATENCY}, I_{SAFETY}, and I_{AVAIL}.

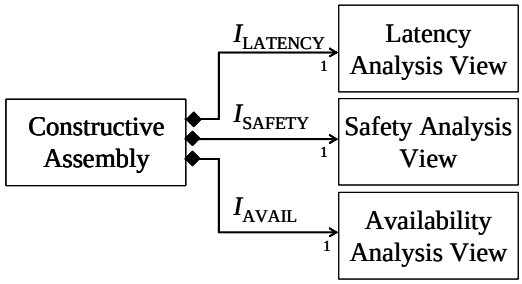

Figure 4: Interpretations on Constructive Assemblies and Analysis Views

Interpretations may impose constraints on constructive assemblies beyond those specified in the constructive viewtype, for example, restrictions on allowed topologies of component interactions or on component behavior. Interpretations may also require analysis-specific component information that is not defined in the constructive viewtype. For example, a latency analysis viewtype might require the priority assignment of component execution threads.

An interpretation is *consistent* if each constructive assembly is related to, at most, one analysis view under that interpretation. An interpretation is *valid* if there is empirical or formal justification for the claim that behaviors predicted in the analysis view will be manifested by the assembly of components specified in the constructive view. A PECT, then, consists (in part) of a set of consistent and valid interpretations.

3.2 The Structure of PECT

Figure 5 depicts the four different environments in PECT and their conceptual dependencies in terms of a UML class diagram. As implied by its name, a PECT consists of a *component technology* that has been extended with one or more *prediction-enabling technologies*. We do not claim to have discovered a universally acceptable definition of these technologies; instead, we define them relative to PECT. The assembly and runtime environments are provided by a *component technology* discussed in Section 3.2.1, while the analysis and validation environments are provided by a *prediction-enabling technology* discussed in Section 3.2.2.

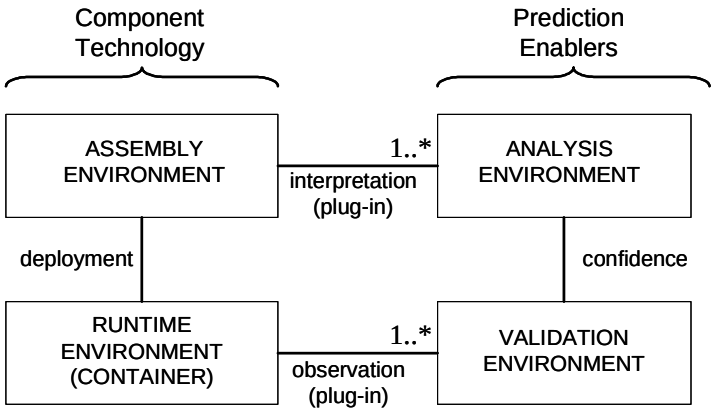

Figure 5: The Four Environments of a PECT

3.2.1 Component Technology

A component technology comprises an assembly environment and a runtime environment. In current literature and commercial products, a component runtime environment is often referred to as a *container.* We prefer our own terminology here.

- The assembly environment supports development-time activities such as selecting components, configuring component properties, and connecting components into assemblies. An assembly environment supports pure composition if those operations are the only ones required to develop applications. Pure composition involves developing applications entirely from components and connectors.
- The runtime environment manages the execution of components (e.g., their execution schedules and life cycles), manages resources shared by components, and provides services that allow components to interact with the external world.

Assemblies are deployed in their runtime environment, that is, by a mechanism either native or external to the component technology.

Not shown in Figure 5 is the fact that both of the above environments are specialized to support a particular *construction model*.[12] The construction model specifies the types of components in an assembly, their type-specific runtime behavior, and the constraints on their connection topology [Bachmann 00]. In our view, this concept of construction model is nearly (if not exactly) identical to that of *architectural style*, as that term has been used in current literature [Gardiner 00], [Bass 98], [Clements 02b]. Therefore, the *Pin construction model,* or,

12. Our use of the term *construction model* supersedes the earlier, and more ambiguous, term *component model*.

more ponderously, the *Pin constructive component model*, is also sometimes referred to as the *Pin style*. Pin is described in Section 3.3.

3.2.2 Prediction Enablers

Prediction enablers comprise an analysis environment and a validation environment:

- The *analysis environment* supports reasoning about the runtime behavior of a system. Simplistically, reasoning is *compositional* if it supports "divide and conquer" analysis.
- The *validation environment* is an infrastructure for controlled, experimental validation of predictions of assembly behavior made in the analysis environment.

The association between the analysis and validation environments shown in Figure 5 reflects a logical dependency between the theoretical model underlying a particular form of design analysis and the experimental apparatus needed to experimentally validate the theory.

3.2.3 PECT=Component Technology+Analysis Technology

We enable reasoning in a component technology by extending its assembly environment with one or more analysis environments. This extension is implemented via a plug-in interface. An analysis environment provides an implementation that satisfies the plug-in, which, in turn, allows users of an assembly environment to reason about as many assembly properties as there are plug-ins.

We enable the validation of analysis technology by extending the component runtime with one or more validation environments. This extension is also implemented by means of a plug-in interface, although in this case, we do not envisage multiple validation environments operating concurrently (but you never know). The observation mechanism is based on execution traces where observable trace events (e.g., procedure calls or message exchanges) have been annotated with property-theory-specific information.

3.3 The Pin Style[13]

Pin allows us to emphasize the compositional potential of software component technology. This may sound tautological at first, but, with a little study, it becomes apparent that components and composition do not always go hand in hand. Consider, for example, the amount of "glue" code that must often be developed, in conventional programming languages, to integrate components [Wallnau 02].[14] With Pin, we can explore the potential for pure composi-

13. What is referred to here as the Pin Style, or elsewhere as the Pin Component Model, or often just Pin, has become progressively more formal over time. Pin is now (at the time of the second edition of this report) the metamodel of the Construction and Composition Language (CCL). This language is described by Wallnau and Ivers [Wallnau 03b].

tion. It is our view that breakthroughs in programmer productivity and system quality require higher-level programming abstractions, and that pure composition has many of the qualities we might want such abstractions to possess, especially when it is "prediction enabled."

The Pin style (hereafter referred to as simply *Pin*) adopts the hardware metaphor that components interact with their environment through a set of *pins*. Components in Pin are specified as a set of pins and pin behaviors; assemblies are specified as a set of connections among pins. Pin defines a number of *connectors*, each specialized to a particular type of pin and pin-to-pin interaction scheme. In this report, we provide a high-level overview of the graphical notation for Pin and the connectors used in SAS PECTs. A more formal and detailed treatment is provided by Ivers and associates [Ivers 02].[15]

We describe the notation of components and pins in Section 3.3.1, assemblies and environments in Section 3.3.2, the behavioral specification of components and the composition of component behavior in Section 3.3.3, and assemblies of assemblies in Section 3.3.4.

3.3.1 Components and Pins

A component interacts with the external world exclusively through its *pins*; there are no other communication paths to or from a component. There are two types of pins: *sink* pins and *source* pins. A component receives communication (stimuli) *from* the environment via its sink pins and sends communication (responses) *to* the environment via its source pins. Figure 6 depicts the graphic notation we use. As suggested by the different shapes given to *pin heads* in Figure 6, there are different types of source and sink pins. These types and their notations are defined in the following sections.

14. By "glue" code, we mean something messy and ad hoc that integrators cobble together as needed. While connectors are also a form of "glue," we typically think of connectors as being well defined and provided by a compiler or infrastructure.

15. A more recent treatment of this topic since the first edition of this report is provided in a short paper that outlines an algorithm for the variant (near subset) of UML statecharts used by CCL [Ivers 03].

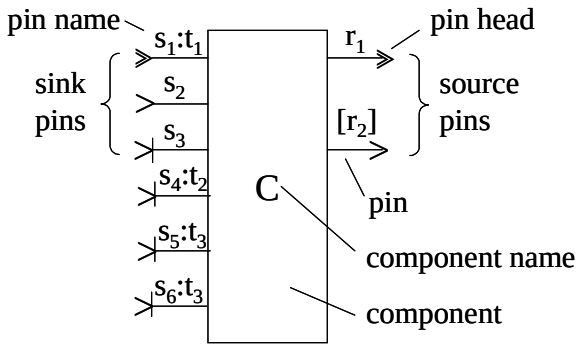

Figure 6: Pin Notation

Components are denoted by c_j, sink pins are denoted by s_j (for $stimulus_j$), and source pins are denoted by r_j (for $response_j$). Where the distinction between sink and source pin is irrelevant, we use p_j (for pin_j). In all cases, we omit subscripts where they are not required. Component names, which must be unique, are also used as pin names. The expression c.p denotes pin p of component c. We use capitalized names to denote the predicates defined on pins. For example, SomeCondition(c.p) is true if pin c.p satisfies SomeCondition, and is false otherwise. We use lowercase names to denote properties of pins. For example, someProperty(c.p) denotes the value of pin c.p's someProperty property.

Pin Data Interface

All pins have a data interface, denoted by interface(c.p). This corresponds to what is usually called the signature of a method and defined as a sequence of formal arguments <*name*, *type*, *mode*>, where *mode* is one of {In, Out, InOut}, and *type* is one of {Boolean, SByte, UByte, SWord, UWord, SDWord, UDWord, SDouble, Float, String}. For simplicity, we do not define the representation of these types in this report.

Source Pins

There are two types of source pins: *asynchronous* and *synchronous*. Informally, an asynchronous source pin represents the ability of a component to make a request, where the component does not wait (or block) for the request to be satisfied. A synchronous source pin also represents the ability to make a request by a component, but in this case, the component must wait for the request to be satisfied. Asynchronous source pins are graphically denoted with the ⩾ pin head, and synchronous ones use the > pin head. In Figure 6, $c.r_1$ is an asynchronous source pin, while $c.r_2$ is a synchronous source pin. It is reasonable to think of asynchronous source pins as *message sends* and synchronous source pins as *procedure calls*, but you shouldn't put too much faith in this gross interpretation; in fact, that is not how these pins were implemented

in the SAS model solutions. Nonetheless, arguments in an asynchronous pin data interface must be of mode *In*.

If Mandatory(c.r), then c.r is a mandatory source pin; otherwise it is optional. Optional source pins need not be connected in an assembly, while mandatory source pins must be connected. This models the distinction between *uses* (mandatory) and *calls* (optional), respectively. A component c_0 uses sink pin c_1.s if the behavior of c_0 depends on the correct behavior of c_1.s; otherwise it only calls c_1.s. Graphically, we distinguish optional source pins from mandatory ones by enclosing the names of optional sources in square brackets ([ ]). So, c.[r_2] is an optional source pin in Figure 6.

Sink Pins

There are two types of sink pins: *asynchronous* and *synchronous*, referring to a component's ability to receive asynchronous or synchronous communication, respectively. As with source pins, the data interface of an asynchronous sink pin may include only arguments of mode *In*. Asynchronous and synchronous sink pins are graphically denoted with pin heads ⩾ and >, respectively. In Figure 6, c.s_1 is asynchronous, while c.s_k, $2 \leq k \leq 6$, are synchronous.

If Threaded(c.s), then c.s has its own thread of control, and threadId(c.s) denotes its identity. Threads represent units of concurrent execution and may be implemented by operating system threads, processes, tasks, and so forth. Graphically, t_j = threadId(c.s) is denoted as a suffix: t_j on the sink pin name. In Figure 6, sink pins c.s_2 and c.s_3 are unthreaded. Threads may be shared by sink pins. So, in Figure 6, c.s_5 and c.s_6 share thread t_3, but c.s_1 and c.s_4 have their own threads. Note that asynchronous sink pins must be threaded, that is, Asynchronous(c.s) ⇒Threaded(c.s).

If Mutex(c.s), then c.s is called a *mutex* sink, and only one caller may be active on c.s at any given time—in effect, the caller must obtain the semaphore for c.s. Conversely, if ¬Mutex(c.s), then c.s is called a *reentrant* sink and is *never* guarded by a semaphore. Note that even a reentrant c.s might force a caller to wait while c.s synchronizes on an internal (to c.s) resource. This characteristic pertains only to synchronous sink pins. Mutex sinks are represented by >|. In Figure 6, Mutex(c.s_k), where $3 \leq k \leq 6$. Note that Threaded(c.s) ∧ ¬Asynchronous(c.s) ⇒ Mutex(c.s).

3.3.2 Assemblies and Environments

An *assembly* is defined as a set of components and their connections. We denote an assembly as a_j, and a.c denotes the component *c* in the scope of assembly *a*. An assembly is graphically depicted as a box enclosing a set of connected components, which visually reinforces the notion that assemblies are composed of components. Figure 7 depicts a simple assembly to illustrate key points in the following discussion.

A connection, or connector, is established between two components when some source pin $c_i.r$ is connected to some sink pin $c_j.s$, $i \neq j$ (this inequality is assumed in further discussion). Components c_i and c_j must be part of the same assembly if they are to be composed. In text, we denote the connection as $a(c_i.r \rightsquigarrow c_j.s)$, although we usually omit the a() where doing so will not cause confusion. A connection $c_i.r \rightsquigarrow c_j.s$ requires that $c_i.r$ and $c_j.s$ be mutually conformant. The rule for mutual conformance is simple: both pins must be synchronous, or both must be asynchronous, and their data interfaces must have the same argument types, modes, and positions. Graphically, we denote a connection as a double-headed lollipop, with the lollipop heads circumscribing the connected pins. In Figure 7, $c_0.r \rightsquigarrow c_2.s$ is a connection.

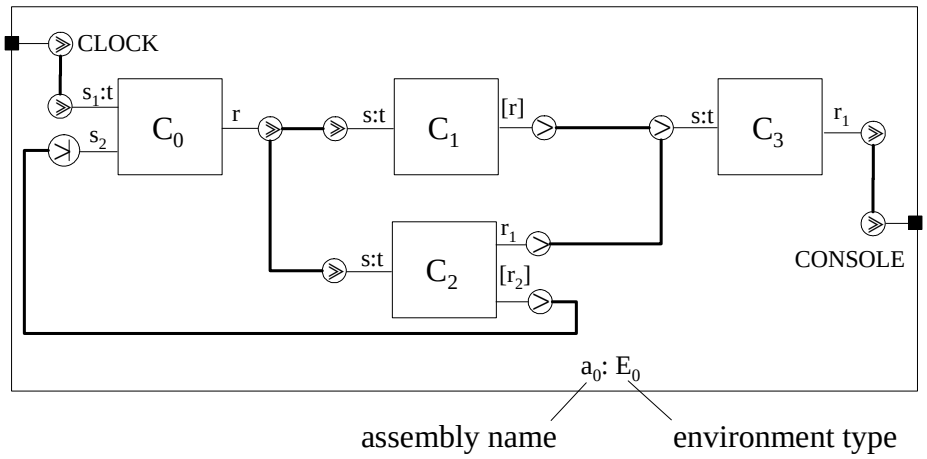

Figure 7: A Simple Assembly

Each assembly has an associated environment type, which is denoted by the $:E_j$ suffix of an assembly name. Environment types play two crucial roles in Pin. First, they define the services (specified as sink and source pins) that components may use. Graphically, these services are attached to the assembly by means of environment junctions, drawn as small black boxes. For example, in Figure 7, a runtime environment associated with assembly a_0 provides two services: the source pin CLOCK and the sink pin CONSOLE. The environment's services can be thought of as being implemented by an environment-provided component with a single sink pin (a_0.CONSOLE) and source pin (a_0.CLOCK). We denote a connection with an environment service in an analogous way to that defined earlier, that is, as $a_0.CLOCK \rightsquigarrow a_0.c_0.s_1$ and $a_0.c_3.r_1 \rightsquigarrow a_0.CONSOLE$. To make these expressions easier to read, we allow assembly names to distribute over $\rightsquigarrow$, so, instead of the above, we could write $a_0(CLOCK \rightsquigarrow c_0.s_1)$ and $a_0(c_3.r_1 \rightsquigarrow CONSOLE)$. Again, we might omit a_0 if doing so does not cause confusion.

Second, environment types supply the interaction models implemented by connectors. For example, consider the interaction topology in Figure 7 that includes

- $CLOCK \rightsquigarrow c_0.s_1$, $c_3.r_1 \rightsquigarrow CONSOLE$, $c_2.r_2 \rightsquigarrow c_0.s_2$

- $c_0.r \rightsquigarrow \{c_1.s, c_2.s\}$
- $\{c_1.r, c_2.r_1\} \rightsquigarrow c_3.s$

where we use sets $\{c.p_1, c.p_2,...\}$ to denote multiple sinks and sources on 1:N and N:1 compositions, respectively. The semantics of 1:1 composition (the first bulleted item, above) is likely to be clear intuitively, but what about the 1:N asynchronous composition (the second bulleted item)? How many connectors are there? Is a broadcast or a sequence of unicasts represented? If a sequence is, what is its order? What is the semantics of message buffering—first in, first out (FIFO) or last in, last out (LIFO)? What is the capacity of the message buffers? Analogously, what is the interaction order of the N:1 synchronous composition? We may want different answers to these questions in different component runtime environments. Ivers and associates give a detailed treatment of how the semantics is defined [Ivers 02].

3.3.3 Behavior: Reaction and Interaction

In PECT, the semantics of composition is defined by analysis views. That is, we define the semantics of an assembly as the observable behavior of an assembly. Informally, the Pin syntax describes *what an assembly looks like,* while the semantics describes *what it does* at runtime. We say that Pin is *semantically extensible* since different syntactic elements of Pin may be annotated[16] with information that is used to construct, via interpretations, the semantics of each assembly in the Pin style.

Nonetheless, one semantics is of such utility to predictable assembly that it is built into Pin—the behavior that is specifiable in a process algebra such as Hoare's CSP [Hoare 85], Magee and Kramer's FSP [Magee 99], or Milner's CCS [Milner 89] and π-Calculus [Milner 99]. We have chosen, at this time, to use CSP as the behavior specification language for Pin. In this report, we describe only the essence of our approach. For complete details on it, see *A Basis for Composition Language CL* [Ivers 02]. We first treat the specification of component behavior in the form of *reactions* and then treat the composition behavior in the form of *interactions*.

Component Behavior: Reactions

Components have intrinsic behavior, as defined by their implementations. We model the behavior of a component in terms of *reactions*. A reaction is a CSP process that relates one or more sink pins to one or more source pins, indicating how the component reacts to the stimulation of its sink pins. The general form is a process that looks something like $R = s \rightarrow r \rightarrow R$, where *s* is a sink pin that can be stimulated, *r* is a source pin that is stimulated in response to an interaction on *s*, and *R* is the CSP process that describes this pattern of behavior. Note that reactions are defined within the scope of a component, so the usual denotation of c.s, c.r, c.R and so on is superfluous.

16. The annotation mechanism or syntax is not described in this report.

Reactions also reflect the thread structure of a component. For example, all behavior implemented by a common thread is modeled as a single reaction. This allows analyses to take into account the actual degree of threading, and potential concurrency errors, of the component implementation. A component's behavior as a whole is specified as the CSP parallel (indicated by the || symbol) or interleaved (indicated by the ||| symbol) composition of its reactions, depending on which better models the actual interaction among the component's threads.

The gist of reaction rules is depicted in Figure 8.

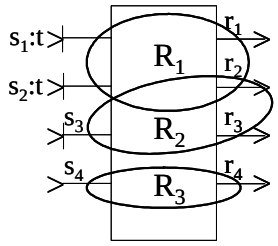

Figure 8: Reactions Specify Component Behavior

In this example, there are three reactions: R_1, R_2, and R_3. The ovals are used to illustrate which pins are related by each reaction; for example, R_1 is shown as relating sink pins s_1 and s_2 to source pins r_1 and r_2. R_1 represents the behavior of a single thread *t* that is used to process sinks s_1 and s_2. A definition of R_1 could be $R_1 = (s_1 \rightarrow r_1 \rightarrow R_1) \,\square\, (s_2 \rightarrow r_2 \rightarrow R_1)$ where $\square$ means *external choice*.

Reactions allow us to specify the causal dependencies among behaviors in an assembly of components. The most basic causal dependency is the *dependency chain*, as illustrated in the above reaction. In fact, this is the only causal dependency we required for the latency analysis in SAS model solutions. More complex behaviors, such as coordination among reactions or changes in behavior based on accumulated state information, can also be modeled.

Assembly Behavior: Interactions

Up to this point, we have been using the $\rightsquigarrow$ operator in an informal manner. Although we never made the claim, the reader might infer that $\rightsquigarrow$ has some sort of algebraic properties and associated composition semantics. A simple algebraic model for $\rightsquigarrow$ might be $\langle R, \{ \overset{a^n}{\rightsquigarrow} \bullet a^n \in E \} \rangle$, where *R* denotes reactions, and in place of the single operator $\rightsquigarrow$, we have a set of operators $\overset{a^n}{\rightsquigarrow}$, one for each ($n^{th}$) interaction scheme a^n defined for an environment *E*, where the operators of a^n may be of arbitrary arity.

So, for example, from Figure 7, the 1:N interaction $a_0(c_0.r \overset{\geqslant}{\rightsquigarrow} \{c_1.s, c_2.s\})$ denotes the syntactic composition of three components c_0, c_1, and c_2, in assembly a_0, on pins $c_0.r$, $c_1.s$, and $c_2.s$. The interaction scheme for this composition is $\geqslant$, as defined in environment type E_0. We can-

not tell whether $\overset{\geq}{\rightsquigarrow}$ is a binary (unicast) or n-ary (broadcast) operator. This and other aspects of the semantics of this interaction must be formally specified.

Such specifications are not trivial, but there are specification patterns that can simplify them. One objective of *A Basis for Composition Language CL* [Ivers 02] is to produce a general approach for specifying the composition semantics for arbitrary interaction schemes (connectors) for arbitrary environment type *E*. We hasten to add here that end users of PECTs never see these semantic complexities, any more than users of modern programming languages see the complexity of type checking or code generators.

3.3.4 Hierarchical Assembly

So far we have described a component model that is quite flat: components can interact only with components in the same assembly and only with a single runtime environment associated with that assembly. Pin will not scale to interesting problems without introducing some form of hierarchical composition. In fact, the algebraic model $\langle R, \{ \overset{a^n}{\rightsquigarrow} \bullet a^n \in E \} \rangle$ has a hierarchical meaning, because $R_3 = R_1 \rightsquigarrow R_2$ (for some arbitrary binary composition operator $\rightsquigarrow$ and reactions $R_{\{1,2,3\}}$) induces a hierarchy that is rooted at R_3 and composes R_1 and R_2.

Hierarchical assembly in Pin always involves treating an assembly as a component. That is, the assembly has an interface defined in terms of source and sink pins. The correspondence between component pins and assembly pins is established by means of assembly junctions. Two forms of junctions are defined: *null* junctions and *gateway* junctions. We note at the outset that hierarchical composition introduces many subtle complexities not generally addressed by component technology. We do not discuss these subtle issues in this report because our focus is on the controller assembly only.

3.4 Pin Subset in SAS Model Solutions

The preceding description of Pin is more general and complete than the version of Pin used in the SAS PECTs—many of the generalities were a result of our experience. The following summarizes those aspects of Pin that were actually used:

- The operator PECT used Microsoft .NET as its component runtime, while the controller PECT used Microsoft Windows 2000 augmented with support for real-time, priority-based scheduling. Differences between these runtimes were reflected in the latency theories used, but not in composition semantics.
- The assembly environments for the operator and controller PECTs were conventional text editors operating on component and assembly descriptions written in Extensible Markup Language (XML).

- The operator and controller PECTs had an analysis environment in the limited sense that latency prediction was automated, and both had validation environments based on traces of time-stamped pin activations.
- All the pin types described in Section 3.3.1 were implemented. However, components were restricted to one thread of control that was shared by all asynchronous sink pins. Further, synchronous sink pins were not threaded.
- Sink pins were annotated (in XML) with execution time. Components were annotated with a single execution priority. Callers on synchronous, mutexed sink pins would acquire the priority level of the called component.
- The notion of assemblies was implicit—although assemblies were units of deployment, they were not named entities. Environment types did not appear at all. Environment services, such as periodic timers, were implemented as environment-provided components.
- OPC gateway junctions were used to (hierarchically) compose an assembly of the operator PECT with an assembly of the controller PECT; analysis hierarchies using null junctions were not used.
- The CSP reactions were specified, on paper, for each controller component. These reactions were then used to manually construct the models used in the temporal-logic model checking, as described in Chapter 7 and Appendix C, and the ordering of interactions over 1:N connection topologies for controller latency prediction.
- There were no formal compositional semantics defined for the connectors.

3.5 SAS Software Controller in Pin

Figure 9 depicts the SAS software controller developed in Pin for this model problem.

The names and functional roles played by the components in Figure 9 (i.e., CSWI, TCTR, TVTR, MMXU, XCBR, and PIOC) are defined by the IEC 61850 standard. SwMonSource and SwMonSink manage the interface to the hardware switch, Clock is a component that is (logically) bundled with the λ_{ABA} latency model, and OPCGateway allows us to create an assembly of assemblies—one that allows the operator and controller assemblies to interact using the OPC protocol.

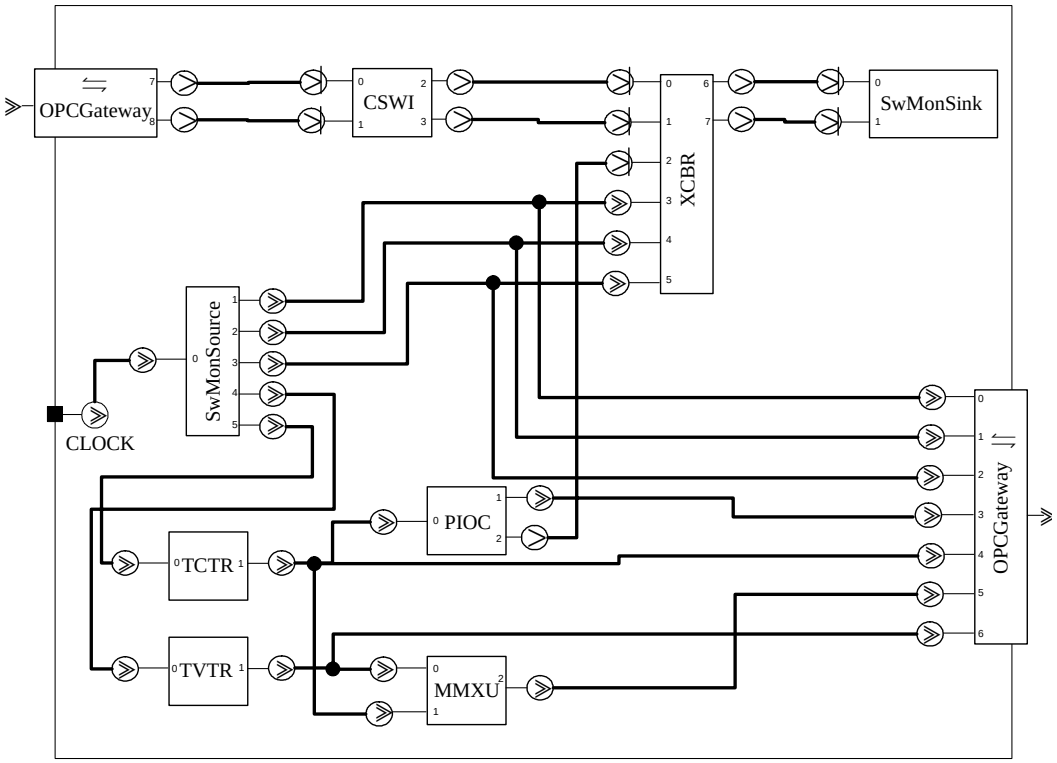

Figure 9: SAS Software Controller in Pin[17]

17. Each component in this figure had at most one thread of execution that was shared among all asynchronous sink pins. Those threads are not depicted in this figure for the sake of simplicity.

4 PECT Process and Workflows

4.1 The PECT Development Process

We develop and validate a PECT using a process that consists of four fundamental phases:

1. Definition
2. Co-Refinement
3. Validation
4. Packaging

During the Definition phase, we define the functional requirements, the assembly property to analyze, and the statistical goals for the PECT. Once those goals have been set, we create a component model, an analysis model, and a measurement framework through an iterative process known as co-refinement. A PECT instance, which integrates the component and analysis models, emerges from co-refinement. During the Validation phase, we validate the PECT instance against the defined goals. If it meets the goals sufficiently, the PECT is packaged and released. If the goals are not met, co-refinement is repeated until they are. Figure 10 shows a brief overview of the PECT process. As the arrows indicate, this process is highly iterative.

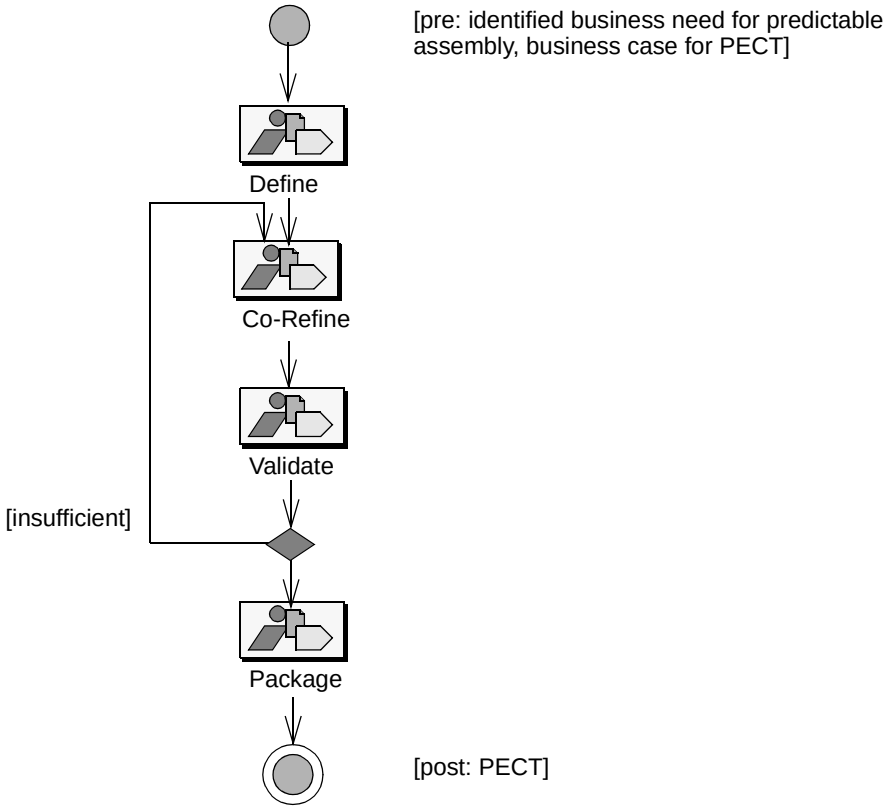

Figure 10: An Overview of the PECT Development Method

In the following sections, we translate the PECT process concretely into (using Kruchten's terms) deliverables, workers, activities, artifacts, and workflows [Kruchten 00]. Deliverables constitute the primary output of the PECT process. Workers are those involved in performing the PECT process. Artifacts describe what is produced during the PECT process. Activities describe how the PECT process is accomplished, and workflows provide insight into when in the process they occur.

4.1.1 Deliverables

The outcome of the process is a PECT that consists of

- a component technology
- one or more analysis views and their associated interpretations
- a validation environment or other means of establishing trust in analysis and predictions
- validation data and statistical labels

The delivered PECT has the following characteristics:

- zero programming assembly
  Components can be assembled into applications without additional programming. If additional code has to be provided, the code must be encapsulated in PECT-compliant components.
- automatic interpretation
  To perform predictions in an execution environment where the end assemblies are unknown, the PECT must be able to interpret a constructive assembly into an analysis view automatically. The analysis view is then used to perform the desired prediction.
- objective trust
  Objective trust is achieved by validating the prediction theory using a wide variety of assemblies. Statistical means can be used to calculate how many assemblies have to be executed to meet the statistical goals set for the prediction theory.

4.1.2 Workers, Activities, and Artifacts

In the Rational Unified Process (RUP), an activity is generally assigned to a specific worker and has a duration of a few hours to a few days [Kruchten 00]; we relax this stipulation somewhat and allow multiple workers to contribute to an activity, where that activity has an indeterminate duration. As with the RUP, each activity has a clear purpose, usually centered on creating or modifying artifacts—the tangible products of the PECT development method.

Table 2 depicts the major artifacts of the PECT development process that are the result of workers performing activities. The fact that each artifact is assigned to just one worker should be interpreted not as an exclusive assignment, but rather as an assignment of primary responsibility. In general, multiple workers are required for most artifacts. For example, defining the requirements for a PECT is primarily the responsibility of the customer, but setting the appropriate requirements requires the contributions of the PECT designer, attribute specialist, and component model specialist.

Table 2: PECT Development Activities

Activity	Workers	Artifacts
Define functional requirements	customer	PECT requirements
Define assembly property	customer	assembly property definition
Define statistical goals	customer	normative confidence labels
Define component model	component model specialist	constructive viewtype
Define analysis model	attribute specialist	analysis viewtype
Define property theory	attribute specialist	property theory and its validation concept

Table 2: PECT Development Activities (Continued)

Activity	Workers	Artifacts
Validate theory empirically	measurement specialist	experiment design, statistical labels
Develop analysis interpretations	PECT designer	component and assembly description schemas; interpretations
Develop component measurement apparatus	component developer measurement specialist	test components (synthetic, actual) testbench for component and assembly measurement
Develop assembly measurement apparatus	measurement specialist	assembly generator and analysis tools
Create validation sample	component developer application assembler	any additional assembly components sample assembly set
Obtain component measurements	measurement specialist	component measurements (labels)
Obtain assembly measurements	measurement specialist	assembly measurements and analysis (labels)
Deploy PECT	PECT designer	packaged PECT

Below, we describe workers whose roles are not implicitly clear by their names alone:

- component model specialist: defines the constructive viewtype used in the PECT. This person may or may not be the designer of a proprietary component technology. Either way, if the component technology has already been defined, this specialist will have in-depth knowledge about it.
- attribute specialist: defines and helps to validate both the property theory and the analysis model. This person must know the theories behind the chosen attribute (emergent property) that the PECT is engineered to predict. For example, if the attribute is latency, the attribute specialist would likely have in-depth knowledge about real-time performance and how to analyze for performance.
- measurement specialist: develops both the component and assembly measurement apparatus and obtains the component and assembly measurements. This person must know how to measure and empirically validate the property theories in the PECT and would typically have broad knowledge of statistical methods.
- PECT designer: unifies the constructive and analysis models through co-refinement (as manifested in the analysis interfaces) and then packages and deploys the PECT. This person holds the holistic view of the PECT and communicates with the other workers to keep the various parts of the PECT development synchronized.

In addition to those workers actually involved in the PECT process, two other roles are key to its successful execution:

- system specialist: has in-depth knowledge about the execution environment that usually affects the behavior of the components residing in the PECT. For example, the PECT may need to be designed to restrict the use of the operating system to meet the set goals.
- domain expert: knows how the PECT will be used from the end customer's perspective and provides input about existing standards, methods, and user profiles. This role is important, because it helps to ensure that the PECT's design will fit the system's intended usage.

4.2 Workflows

The workflows in this section are elaborations of the PECT overview workflow shown in Figure 10 on page 26.

4.2.1 Definition Phase

The workflow shown in Figure 11 is an elaboration of the Define process shown in Figure 10. The Definition phase has two parallel paths—one to define the functional requirements and another to gather the necessary requirements for the property of interest and set the PECT's prediction goals.

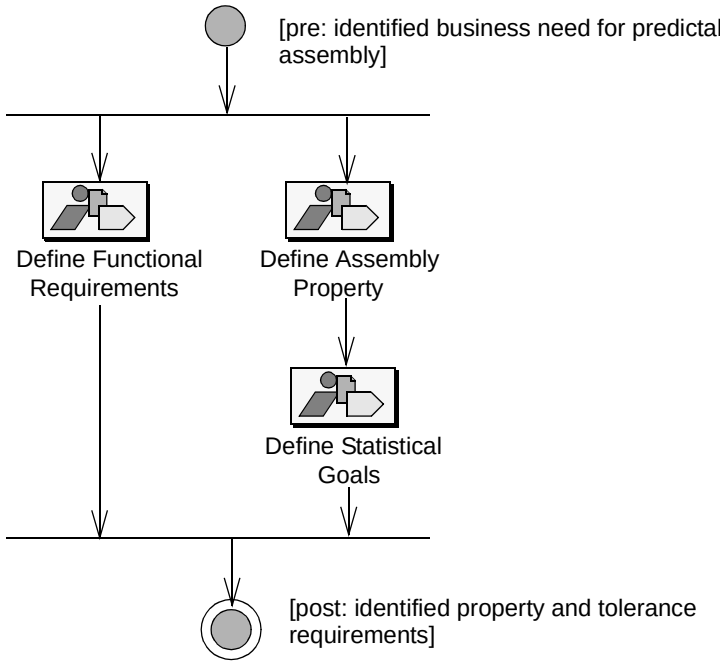

Figure 11: Definition Phase Workflow

Statistical goals should be based on the business need. Because producing it takes effort, a PECT should be only as general as necessary to fill that need.

4.2.2 Co-Refinement Phase

During the Co-Refinement phase shown in Figure 12, we define the constructive and analysis viewtypes, and the interpretation that integrates them. We also define a plausible and possibly high-level scheme for validating predictions based on the analysis viewtype.

The workflow depicted suggests a process that is somewhat more structured than what occurs in actual practice. The constructive and analysis viewtypes are depicted as being developed in parallel, but they are, in fact, mutually constraining, as explained and illustrated in Chapter 5. In our experience, considering validation places useful constraints on the development of the analysis viewtype, so we placed "Define Validation Concept" at the same level of activity in the workflow as "Define Constructive Viewtype" and "Define Analysis Viewtype." A similar argument can be made for placing "Define Interpretation" alongside "Define Validation Concept," although our experience suggests that defining the interpretation is usually the closing step in a co-refinement iteration.

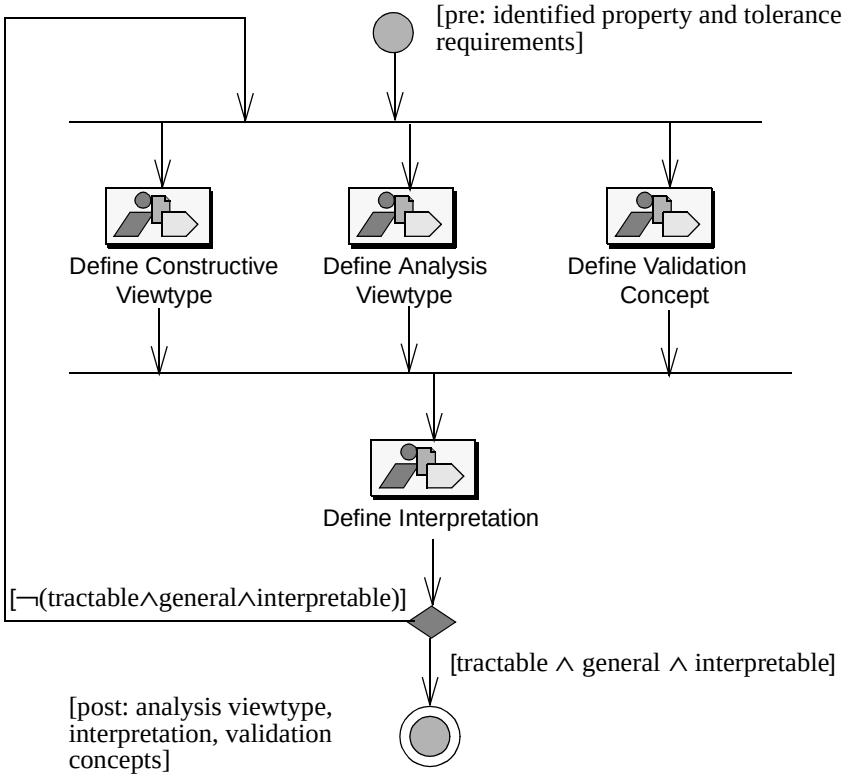

Figure 12: Co-Refinement Workflow

The iteration within the co-refinement workflow must not be confused with the iteration that results from the empirical validation shown in Figure 13. Within co-refinement, the condition for iteration is an assessment of whether the resulting constructive assembly is sufficiently general to handle an anticipated range of assemblies and whether the property theory underlying the analysis viewtype will, with reasonable computation and interpretation effort, yield predictions for all these assemblies.

4.2.3 Validation Phase

After co-refinement, the resulting PECT is validated against its requirements. The definition workflow shown in Figure 11 identifies functional requirements and tolerance requirements. Functional requirements define the range of assemblies that must be "spanned" by the construction model and its interpretations; the tolerance requirements refer to the desired quality of predictions based on the analysis viewtypes supported by the PECT. The workflow shown in Figure 13 is biased toward the validation of analysis viewtypes whose underlying property theories are empirical (i.e., based in measurement) rather than formal (i.e., based in logic). As this workflow is currently defined, it addresses the validation of both functional requirements and statistical goals.

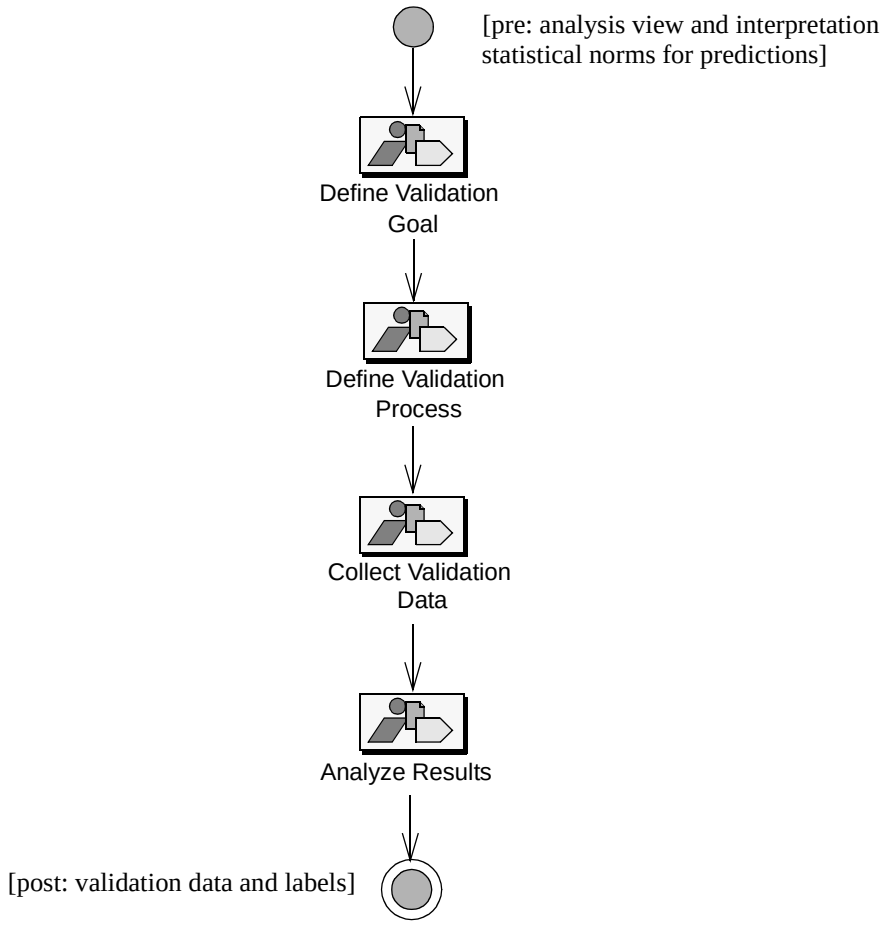

Figure 13: Empirical Validation Workflow

There are four general aspects to carrying out an empirical validation study:

1. Define the validation goal.
   The goal is a normative or informative statement about the predictive power of the PECT. Typically, a goal is stated in terms of the probability that a prediction will lie within some accuracy bounds, with some stated confidence level.

2. Define the validation process.
   Validating the accuracy of a prediction is analogous to validating a scientific theory. That is, behaviors are observed systematically under controlled circumstances, and this observed behavior is then compared to predicted behavior. The key word is systematic, since validation results should be, above all else, objective and hence repeatable.

3. Collect validation data.
   This step involves conducting the actual validation experiment. Important in this activity are good laboratory techniques—for example, keeping good notes and records so that anomalies can be studied and results can be repeated.

4. Analyze the results.
   The purpose of the analyses is twofold: first, to objectively and reliably describe the predictive powers of a PECT; second, to provide analysis data to support additional co-refinement, should normative goals fail to be satisfied.

This workflow is expanded considerably in Chapter 6.

4.2.4 Packaging Phase

The workflow for packaging is a lacuna at this time. The objectives of packaging are twofold. One objective is to ready the technology for deployment, including the development of installation support, documentation, and so forth. A second and more fundamental objective is to design automation support for the PECT to minimize the property-theory-specific expertise required by PECT users to make effective use of analysis viewtypes supported by the PECT. This area requires further work.

5 Co-Refinement of the Controller PECT

5.1 What Is Co-Refinement?

PECT creation is a process of iterative negotiation between the constructive and analysis points of view. We call this process *co-refinement*. Co-refinement does not necessarily follow a rigorously defined process, but rather it is channeled by a set of forces that guide co-refinement as the two points of view (models) converge into a PECT.

The constructive point of view pushes for assembly generality; the more assemblies that can be represented in the construction model, the better. The analysis point of view pushes for predictability, which implies constraining the construction model to adhere to the assumptions of the property theories that enable analysis and prediction. These forces tend to act in opposition to each other.

Practicability, tractability, and interpretability also weigh in as important forces and act as moderators. *Practicability* pushes for a level of generality that is necessary (but no *more* than necessary) to represent some class of realistic system; it acts as a bounds on generality. *Tractability* is concerned with the level of difficulty in solving the analysis model; it acts as a bounds on predictability. For example, it might not be viable to require a supercomputer to solve the analysis model. *Interpretability* ensures that automatic translation from the construction model to the analysis model is possible. This possibility implies the ability to formally specify both models and to translate the constructive representation into the analysis representation. *Validatability* is concerned with the level of difficulty in obtaining empirical validation of the analysis model.

5.2 How Co-Refinement Proceeds

The co-refinement process starts with an initial, not necessarily formal, description of the “languages” for the constructive and analysis models. The elements of the construction model language are influenced by the PECT’s target domains and how systems in those domains will be structured, developed, deployed, and sustained (evolved) over time. The elements of the analysis model language are simply a subset of some property theory that is suitable for analyzing the behavior of interest.

Initially, each model is independently subjected to its own starting criteria. The construction model[18] has to be expressive enough to describe the types of components and interactions expected in the domains that will use the PECT. However, initially, the construction model might not be interpretable vis-à-vis the analysis model, and the analysis model might not be general enough to account for all aspects of the construction model.

Given a starting point for the constructive and analysis models, the forces guide co-refinement as the initial models converge into final models that define the PECT. While there is no precise set of steps for determining the sequence of intermediaries from the initial models to the final models, there are some rules of thumb.

Generality increases monotonically. After the first iteration, the PECT designer should have reasonable confidence that the construction model is interpretable and that the analysis model is tractable. Therefore, the dominant force in co-refinement at this point is moving the construction model to the desired state of generality. The only reason to reduce generality would be if the designer's confidence in interpretability proved to be unwarranted.

Interpretability increases monotonically. While co-refinement might not start with interpretability, it must end with it. This leads us to believe that interpretability must monotonically increase as co-refinement proceeds. This means that, during co-refinement, the languages describing the constructive and analysis models, and the interpretation rules become more formal. In addition, a translator for the construction model is developed iteratively as co-refinement proceeds.

Validatability is an invariant. Validatability is a defining characteristic of a PECT. Unlike generality and interpretability, however, it is important that all analysis models be validatable, at least in principle, for each iteration of co-refinement. By *in principle*, we mean that there must be a plausible means for obtaining or validating component properties, and for falsifying predictions made in the analysis model. While plausibility is not an objective quality, it is one that has a reasonable intuitive meaning.

Using these rules, co-refinement drives for increasing generality in the construction model until all the following stopping conditions are met:

- The practicability boundary is reached—striving for more generality is not worth it.
- The interpretability boundary is reached—an automatic translation becomes impossible.
- The tractability boundary is reached—because of its complexity, solving the analysis model is too costly.

This iterative scheme was depicted in Figure 12 on page 30.

18. The terms *construction model* and *construction model language* are used interchangeably in this report.

5.3 Key Ideas for the Controller PECT

Performance is important in the intended domain for this PECT—power station control. Sensor inputs are gathered at various rates, inputs are coalesced, and actions are taken depending on the input values. Computations must adhere to both worst-case and average-case latency requirements. Typically systems are multiprocessing on a single processor or multiprocessors, connections can be both synchronous and asynchronous, and some data stores must be updated atomically.

The property theory for the controller PECT is based on rate monotonic analysis (RMA) [Klein 93]. RMA is well suited for predicting worst-case latency for a collection of processes on a single processor in a mostly (but not solely) deterministic situation; events occur mostly periodically, and process execution times are bounded and do not vary dramatically. RMA requires information such as process periods, and execution times and priorities; whether mutually exclusive access to shared resources is needed; whether there are intervals of nonpreemptability; and whether each process has a unique priority. In general, RMA strives to account for any factor that can contribute to latency—the time interval from when an event occurs until it has completely been processed.

5.4 Co-Refinement of the Controller PECT

This section describes the starting state (initial condition) for each model and then walks through the five iterations of the co-refinement process, leading to λ_{ABA}, the property theory whose validation is the subject of Appendix B. For each iteration, constraints on the construction model and model semantics (i.e., what is predicted) are described. The five iterations are

- predicting worst-case latency (λ_W)
- predicting average-case latency (λ_A)
- introducing mutual exclusion and worst-case blocking (λ_{WB})
- predicting average-case blocking (λ_{AB})
- introducing asynchrony (λ_{ABA})

It should be noted that this is an *a posteriori* examination of a co-refinement experience. *A priori*, you would not expect to know the specifics of each iteration or how many iterations are required.

5.4.1 Initial Conditions

The initial construction model was relatively simple. It consisted of a collection of components, where each component had a set of source and sink pins. Since concurrency was known to be important, each component could have either no threads or one thread associated with

it.[19] Since some components had to be activated periodically, there was a special source pin that was denoted as periodic. Naturally, these components were threaded. Sink pins could be either reentrant or non-reentrant. Both synchronous and asynchronous connections were allowed.

The analysis model is based on a collection of real-time analysis techniques known as RMA for predicting worst-case latency. Latency is the measure of how long it takes to service an event. For example, we consider reaching a voltage sensor's read time as the occurrence of an event. Servicing the event comprises all the computations that are necessary to respond to the event, such as acquiring input from the sensor, converting the input to a number with some physical meaning, filtering noise, combining the input with other data, testing the data against some important physical conditions, and then issuing some output. These computations take place on a single processor that is also servicing other events. Latency is how long it takes from the voltage sensor's read time (not necessarily when the sensor is actually read) to the completion of the response to the event.

The initial analysis model used a subset of the modeling capabilities of RMA including

- considering only periodic events, that is, those that occur at regular intervals
- each periodic event is allocated its own process (or thread)
- each process is assumed to have a constant, unique priority
- the variability of the execution time for any given component is bounded
- the worst-case latency of all events is less than or equal to the period
- only reentrant, synchronous connections are allowed
- no blocking is allowed (sources of blocking include critical sections and nonpreemptable sections)

5.4.2 First Iteration: Worst-Case Latency (λ_W)

Tractability and interpretability were the dominant forces during this iteration. They caused us to both extend the construction model with the annotations necessary to provide the analysis model with the needed information and to constrain the construction model to conform to the restrictions of the initial conditions of the analysis model. The annotations included

- specifying execution times as properties
- specifying unique priorities for each thread. This turned out to be a significant restriction, because Windows imposed severe limitations on the number of priority levels.

19. This has since been generalized. See Section 3.3.1.

Modifications to the construction model involved mainly the thread model. Since periodic invocations were important, we decided to extend the construction model with an explicit notion of time using an environment component, the clock component. This extension eliminated the need for a periodic source pin. Also, since the analysis model was not considering blocking during this iteration, we disallowed threaded components, because they would have resulted in non-reentrant components and hence blocking. The construction model was also constrained to allow only synchronous calls, conforming to this restriction of the analysis model.

With these restrictions to the analysis model, the worst-case latency for each process could be calculated using the following fixed-point calculation:

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

This formula can be used to compute the worst-case latency L_i for the *i*th process. C_i is the execution time of the *i*th process; T_i is the period of the *i*th process. Processes 1 through *i*-1 are assumed to be all the processes whose priorities are higher than that of process *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. Then it computes L_{n+1} using the formula above. It continues until $L_n = L_{n+1}$, at which point the fixed point has been reached, and L_{n+1} is the worst-case latency.

The interpretation from the construction model to this analysis model (the above formula) is straightforward. Execution time (associated with sink pins), periods (associated with clock components), and priority (associated with threads) all map directly to the formula. The only complication occurred for the case in which a clock component initiates a sequence of component calls, as shown in Figure 14.

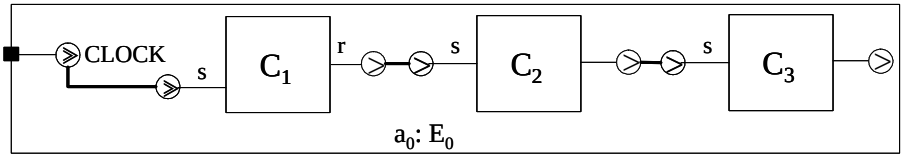

Figure 14: Clock Component Initiating a Sequence of Interactions

In this case, the period of the clock corresponds directly to the period of process *i* in the analysis model, the priority of component c_i corresponds to the priority of process *i* in the analysis model, and the sum of the execution times of components c_1, c_2, c_3, and so on, corresponds to the execution time of process *i*.

At this point, we did not yet have an automatic translation from the constructive to the analysis model, but we did have a high degree of confidence that such a translation could be created.[20]

5.4.3 Second Iteration: Average Latency (λ_A)

For the second iteration, practicability started to exert influence mainly in terms of the need of the construction model to satisfy average-case latency requirements in addition to worst-case latency requirements. Another facet of practicability influencing this iteration was the realization that the zero-phasing[21] assumption of the analysis model was not realizable in the construction model. Therefore during this iteration, the construction model did not change; the analysis model evolved to predict average-case latency and to handle non-zero-phasing.

Zero-phasing is significant because a key idea of RMA is that worst-case latency occurs for process *i* when it becomes ready to execute at exactly the same time that all higher-priority processes do. The fixed-point formula shown in the previous section computes worst-case latency assuming zero-phasing. If the construction model does not in fact adhere to zero-phasing, the prediction might produce a result that is too pessimistic. Accounting for non-zero-phasing requires only minor changes to the fixed-point formula shown in the previous section. These changes were made during this iteration.

A more substantive change to the analysis model was required to calculate average latency. The critical observation was that the execution profile of process *i* repeats itself. The interval of repetition (known as the hyper-period) is equal to the least common multiple (LCM) of T_1, T_2, ..., $T_{i.}$. Every NP period, process *i*'s performance behavior repeats, where $NP_i = LCM(T_1, T_2, ..., T_i)/T_i$. Therefore, the latency for the first instance of process *i* should be the same as the latency for the $1+NP_i^{th}$ instance. To calculate the average latency for process *i*, we simply compute the average of the NP_i instances of process *i* during a hyper-period. The fixed-point calculation above had to be generalized to work across the hyper-period instead of just stopping after the first instance of the job.

At this point, we performed a "spot validation" of the interpretation and analysis model. That is, we constructed a small experiment, consisting of a small sample (< five simple assemblies) to see if we were predicting average latency accurately. The results were encouraging.

---

20. Although rewrite rules were possible, we chose not to implement them, because the generality of this approach to interpretation was not clear.

21. We say two or more events are zero-phased in time when they happen at the same moment (i.e., there is no time delay between them). In the context of the controller PECT, zero-phasing means that process *i* will become ready to execute at the exact moment all higher-priority processes become ready to execute (the critical instant).

5.4.4 Third Iteration: Worst-Case Latency + Blocking (λ_{WB})

During this iteration, practicability continued to be a dominant influence. The controller domain required non-reentrant execution within components to ensure data consistency.

During the first two iterations, the only influences on process latency (in theory) were its own execution time and preemption from higher-priority processes. With non-reentrancy, a higher-priority process might have to wait while a lower-priority process is executing a non-reentrant component. This effect is known as blocking. It prolongs latency and must be accounted for in the analysis model.

To account for blocking in the analysis model, a blocking term is simply added to the expression on the right side of the worst-case latency equation shown on page 39. However, the real issue is how to control the magnitude of this term. To do so, we impose a restriction on the construction model that is based on another key result of RMA. One of the synchronization protocols discovered by RMA is the *priority ceiling protocol*. That protocol exhibits an important property known as the *blocked-at-most-once* property, which says that a process that executes in several critical sections can be blocked in, at most, one critical section.

The priority ceiling protocol requires that when more than one component interacts with a non-reentrant sink pin, the thread for that sink pin must execute at a priority level that is at least as high as the priority of the effective thread of any of the calling components. This priority level is known as the *ceiling priority.* Note that for this priority assignment to emulate the priority ceiling protocol, the thread that is assigned this priority cannot suspend for any reason, including input/output (I/O).

The prediction obtained using the priority ceiling is a worst-case prediction with respect to blocking, because a blocking term is used for every job in the hyper-period, regardless of whether the job is actually blocked during its execution. Assemblies must honor the priority ceiling to be well formed with respect to λ_{WB}. Since the construction model has no concept of priority, it is the interpretation rather than the construction model that enforces this topological constraint.

5.4.5 Fourth Iteration: Average Latency + Blocking (λ_{AB})

The goal of this iteration is to calculate average latency in the presence of blocking, taking into account the possibility that blocking will occur only for a portion of the jobs in a hyper-period. The initial challenge for this iteration was deriving a variation of the fixed-point calculation that could also determine when blocking did and did not occur. To make this determination, the notion of a subtask was introduced. A subtask is a portion of a component's computation distinguished by its priority; different subtasks have different priorities.

At this point, a purely analytic approach became unwieldy—in fact, nearly intractable. However, the analysis model yielded to a hybrid simulation-based approach [Schwetman 78]. In a hybrid simulation approach, an analysis model is used to compute values that define initial or test conditions in a simulator. Details of the hybrid simulation are provided in Appendix A.

At this stage, a formal interpretation was defined. Although we knew that the average-case latency theories were, in principle, validatable, it was by no means clear that the earlier spot validation would scale from λ_A to λ_{AB}. The PECT was not yet sufficiently general, since it did not accommodate assemblies with asynchronous pins. This led to the final iteration.

5.4.6 Fifth Iteration: Asynchronous Interactions (λ_{ABA})

The previous iterations had all involved adding new terms and features to the analysis model, and introducing or removing constraints on the construction model to accommodate those additions. At this point, it was clear that the final and necessary step of the co-refinement process would be to relax the prohibition against asynchronous pins. It was unclear at this point how this constraint could be relaxed, or how much its relaxation would affect the analysis model or simulation. Our great fear was that it would result in some kind of fundamental discontinuity, effectively demonstrating that the basic RMA approach was insufficient, or vastly complicating the simulator (loss of tractability).

Our fears proved to be unfounded, however. We made the fortuitous discovery that we could, through the application of rewrite rules, transform a constructive assembly with an arbitrary topology of asynchronous interactions into a behaviorally equivalent one (from the perspective of latency computation) with a more restrictive topology, as with synchronous pins only. These restrictions simplified the interpretation and preserved the hybrid simulation model. Thus, we were able to introduce asynchrony to the latency model strictly through rewriting and interpretation.

6 Empirical Validation

6.1 What Is Empirical Validation?

There are two classes of property theory: those based in logic and those based in measurement. Logical property theories, such as those found in model checking and other forms of program verification, involve demonstrative reasoning: the validity of a property is mathematically demonstrated, or proven, in a way that bars contradiction. Empirical property theories, such as λ_{ABA}, involve plausible reasoning: the validity of a property theory is established inductively, based on observations.

The fact that empirical theories are based on observations and measurement introduces uncertainty. The property theory itself may abstract aspects of an implementation that influence the assembly-level property. This, in turn, introduces variability and some degree of non-determinacy, for we cannot give a systematic accounting of an abstracted system aspect without undoing the effect of its abstraction.

Empirical validation is the means by which we validate empirical property theories. The validation does not yield a simple "yes" or "no." Instead, it produces statistical evidence of a property theory's effectiveness and of the component property measures on which the validation's predictions are based. Before describing the workflow for empirical validation in Section 6.3, we provide a brief overview of the statistical models used to express statistical evidence.

6.2 Introduction to Statistical Labels

Rigorous observation takes the form of measurement, and measurement invariably introduces error. Thus, measured component properties are described not as discrete values, but as probability density functions over a range of values. As was the case with component properties, measured assembly properties are described using probability density functions over a range of values. Likewise, determining the effectiveness of a property theory also involves measurement, this time in a more classical application of the scientific method. The property theory is a hypothesis; it is tested by comparing assembly properties predicted by the hypothesized property theory with observed assembly properties.

We refer to the statistical descriptors of component properties and property theory effectiveness as *statistical labels*. Our position is also that all empirical properties and property theories

can be described in a uniform way, using the same, or very similar, statistical models, resulting in standard labels.

6.2.1 Component Labels

The measurement and subsequent description of empirical component properties do not introduce novel methodological challenges. Objective measurement requires a measurement object (the component), a measurement scale for the component property of interest (e.g., time, in seconds), and a measurement apparatus (e.g., the Microsoft Windows' high-performance clock). Measurement is conducted using the apparatus within some controlled environment and with control exerted over all independent variables of the dependent (measured) property.

Component Property Estimators

In general, the measurand *Y,* or property of interest, is a function of *N* values: $Y = f(X_1, X_2, \ldots, X_N)$ [Taylor 94]. For latency, these values include execution time, blocking time, and period. We would like to know the *true* value of *Y,* component latency. Of course, the true value is not obtainable, as the following definition makes clear:

> True Value: the mean (μ) that would result from an infinite number of measurements of the same measurand carried out under repeatable conditions, assuming no systematic error.

Because we cannot, even in principle, know the true value of μ, we must use an estimator for it, one that is produced by statistical methods.

For example, we take sample observations of *X* and use their average $\bar{x}$ as the estimator of μ, a population parameter. The uncertainty associated with this estimator is expressed as the deviation *s* such that the true—and unknown—value of μ will fall within some interval $\bar{x} \pm ks$ with some specified confidence. The factor *k* is known as the *coverage factor.* When k=1, $\bar{x} \pm ks$ yields a 68% confidence interval (one standard deviation). That is, we have 0.68 confidence that this interval contains μ. Typically, we compute the 0.95 confidence interval (k=2), which yields higher confidence but a larger bound. This confidence interval expresses a fundamental component measure; it is fundamental because $\bar{x} \pm ks$ is how a component property is modeled in any property theory parameterized by that component property.

Component Property Intervals

It might be useful to have other descriptors, or labels, for component properties. One such label is the *tolerance* interval. For example, in the case of component latency, the consumer might want to know which latency interval $\pm x$ ms contains a specified proportion *p* of all executions of component *c*. For instance, a tolerance interval with *p*=0.95 might state that there is

a 0.95 probability that any given component's execution will have a latency of $50\text{ms} \pm 17\text{ms}$, where $\pm 17\text{ms}$ is the computed tolerance interval.

Another potentially useful label is the conceptual, but not mathematical,[22] inverse of the above tolerance interval. This label is the confidence interval on the probability of satisfying some property specification. Conceptually, this label is the inverse of the tolerance interval, since we specify the latency interval here and use it to compute the probability that any particular execution will fall within this interval. For example, we might specify the latency interval $\pm 5\text{ms}$ and use it to calculate that there is a 0.64 probability that any given component's execution will fall within the interval $50\text{ms} \pm 5\text{ms}$.

6.2.2 Property Theory Labels

Measuring and describing the effectiveness of property theories is methodologically more challenging than measuring component properties. As noted earlier, at the limit, this measurement process reduces to theory falsification as it is practiced in the empirical sciences; in that sense, at least, no new ground is broken. Our concern is with characterizing (labeling) the quality of a property theory. Such a characterization translates into labeling the accuracy of the theory's predictions and determining how often they are accurate.

One-Tail Inferential Intervals

We use inferential statistical models to characterize how effective a property theory is likely to be for future predictions. For this purpose, we use confidence and tolerance intervals. For property theories, we are usually interested in the MRE between predicted and observed values. For latency, this equation is used:

$$\text{MRE} = \frac{|a.\lambda - a.\lambda'|}{a.\lambda'}$$

where $a.\lambda'$ is the measured assembly latency and $a.\lambda$ is the predicted latency.[23] A normative confidence interval will describe the probability that the MRE for a particular prediction will lie within a specified MRE interval.

Notice that the lower bound for an MRE interval represents situations where predictions are better than the mean MRE of the statistical sample (i.e., in our experiment, for N=30 assemblies). While we might want to know how frequently our predictions are better than the mean, we might be concerned only when the predictions are worse than the mean. In this case, we

22. Different equations are used to compute conceptual and mathematical intervals.

23. We note in passing that $a.\lambda'$ is described using the fundamental label for components. That is, for statistical analysis, the assembly is treated as a component, and the estimator for assembly latency is described by an interval obtained using a coverage factor k=2.

use a one-tail interval, or bound, instead of a two-tail interval. The one-tail tolerance interval for λ_{ABA} is summarized in Table 3.

Table 3: Distribution-Free Tolerance Interval for λ_{ABA}

Part of Interval	Description
N = 156	sample size
$\gamma = 0.9929$	confidence level
p = 0.80	population
$\mu_{MRE} = 0.0051$	MRE
UB = .01	upper bound

The tolerance interval for λ_{ABA} can be interpreted as saying that 80% of latency predictions will not exceed an MRE of roughly 1%, and that the manner in which that interval is constructed yields greater than 99% confidence that this upper bound is correct. As with measures of component properties, one- and two-tail normative confidence intervals can be computed by specifying interval bounds and computing *p*, the probability that any particular prediction will satisfy these specified bounds.

Linear Correlation

Linear correlation is a descriptive statistic: it describes the strength of correlation between two data sets and is not directly useful for drawing inferences about future data sets. A consumer might be interested in linear correlation analysis as a descriptor of previous experimental validations of a property theory's accuracy.

We characterize that accuracy using linear correlation analysis, which allows us to assess the strength of the linear relation between two variables—in our case, predicted and observed assembly latency. The result of such analysis is the coefficient of determination, $0 \leq R^2 \leq 1$, where 0 represents no relation, and 1 represents a perfect linear relation. In a perfect prediction model, predicted and observed latency would be identical; therefore, the goal for the model builder is a linear relation.

For λ_{ABA}, a distribution-free linear correlation was used (Spearman rank correlation). The resulting $R^2 = 0.998$ means that there is < 0.001 probability that the reported correlation could have been achieved by chance.

6.3 Workflow for Empirical Validation

The workflow in Figure 15 expands the first two steps of the four-step validation process illustrated in Figure 13 on page 32. Figure 13's workflow is reproduced in the left portion of Figure

15; the first two steps of Figure 13 have been replaced by their expansions, resulting in a six-step workflow.

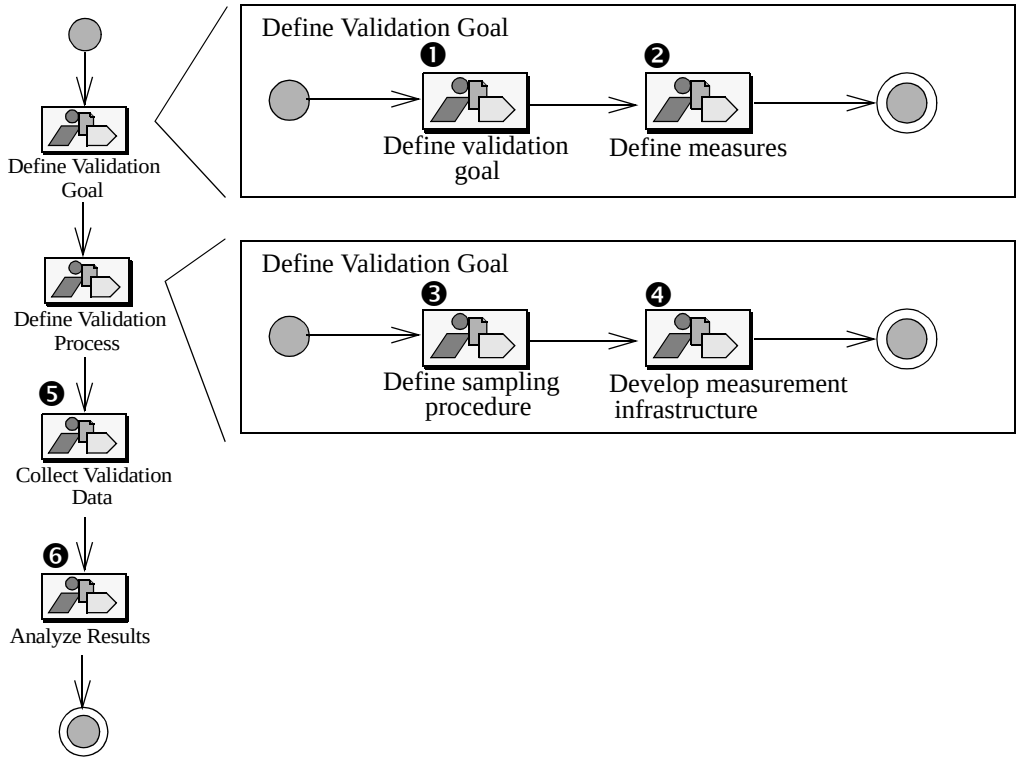

Figure 15: Expanded Workflow for Empirical Validation

In the remainder of this section, we describe the six steps shown in Figure 15.

6.3.1 Define Validation Goal

The overall objective of empirical validation is to assign a statistical label to a property theory that describes the effectiveness of that theory and to design the means of producing trusted component labels in support of that theory.

Two types of goals yielding different types of statistical labels are possible: *normative* and *informative*. Normative goals express a prediction requirement that has to be met. For example, predictions might be required to be, on average, accurate to within 0.5%. An informative goal is free of norms—it's up to the validator to describe the effectiveness of predictions. For example, the validator might compute the upper bound of a confidence interval for 60, 70, 80, and 90% of predictions, assuming varying levels of confidence.

Part of defining the validation outcome involves selecting the statistical models used for labels. As discussed in Section 6.2.2, we have used the MRE as a basis for the empirical vali-

dation of property theories and two forms of intervals for descriptive and inferential labels. Those choices are not a necessary consequence of the initial "customer" objective—rather they are choices made by the measurement specialist of the best labels for a particular validation effort. As we learn more about empirical validation, we anticipate a broadened repertoire of statistical models.

For λ_{ABA}, the design problem was to develop a controller PECT that would predict latency with an upper bound of a tolerance interval for MRE ≤ 0.05, for a population $p = 0.80$, with a confidence level $\gamma = 0.99$. Thus, the empirical validation for λ_{ABA} was dominated by normative goals.

6.3.2 Define Measures

It is not always obvious what exactly must be measured to validate a property theory, or which measurement units are best to express those measures. Indeed, frequently there are several possible measurement units, depending on whether a direct or indirect approach is desired, or on the degree of accuracy or stability required. A good discussion of measures and measurements can be found in the seminal work of Fenton and Pleeger [Fenton 97].

From λ_{ABA}, we have identified the property of interest—latency—and defined it as the total time a task requires to perform its work. The constituents of this total time—execution time and blocking time—also had to be expressed in terms of the Pin component model and measurement units. We also had to know whether there was a technical basis for measurement at all, which, surprisingly perhaps, was not trivial.

Translating the notion of time to Pin was conceptually straightforward. Sink and source pins provided convenient "hooks" for hanging instrumentation. Each pin can be instrumented to record the moment of its activation (e.g., when a sink pin receives a request or when a source pin sends a message or makes a request), and the moment of its deactivation (e.g., when the sink pin satisfies its request or when a source pin completes its message send or has its request satisfied). These four measurement points permit a variety of time durations to be recorded on any assembly.

Defining the appropriate units of time depends on the available instruments. For λ_{ABA}, we chose to use the Microsoft high-performance clock, which records times in nanoseconds. Before making this selection, though, we had to establish that that clock could be used to measure component latency. To do so, we had to first demonstrate that the latency of a process, as recorded by the clock, was equal, within measurement tolerance, to the sum of system time and wait time for a process, as recorded by the central processing unit (CPU) clock. Once we had established this correlation, which itself required a validation activity, we were satisfied with our defined measures.

6.3.3 Define Sampling Procedure

Defining the validation experiment eventually requires that we select a set of components and assemblies for measurement. This can be viewed as selecting components and assemblies from a population, with one important proviso: if we expect a PECT to work for all future assemblies, we must assume that not all of its components and assemblies exist. So a question arises as to how to select from a population of nonexistent entities. The three main approaches to selecting samples from a population are random, convenience, and judgment sampling.

Random sampling is useful when samples can truly be selected randomly. *Convenience sampling* occurs when the sample is taken while considering some *a priori* knowledge. *Judgment sampling* occurs when the sample is selected explicitly and sufficient domain knowledge exists to judge a sample set that represents the whole population. Of the three methods, random sampling provides the best statistical data, but it might be difficult to obtain a real random set of samples. On the other hand, convenience and judgment sampling are more practical and easier to perform, but the resulting statistical data might be biased and therefore not fully representative.

For λ_{ABA}, random sampling was used to collect latency measures from a set of synthetic components—components whose functionality consists of consuming CPU resources. We used a combination of judgment and random sampling to collect measures for assembly latency measures, and we used judgment sampling to define variation points for assemblies. Doing so created an assembly design space for constructing viable assemblies. Then, from that design space, we selected assemblies randomly. For example, in the controller PECT, no realistic assemblies have more than 50 components. Thus, that was an upper bound for one variation point in the assembly design space that we defined.

In an analytic study, we must define the design space and select a random sample from it. Our approach to validating λ_{ABA} was to exploit the "pure composition" aspects of Pin to define various dimensions of variation—the number of input and output connectors, the number of instances of each type of component, and so forth—and thereby sketch the outlines of the design space. Using these variations, we generated the pseudorandom assemblies.

We have made only tentative steps in understanding how to analytically characterize the design space and draw a random sample from it. At this point, we have only two conjectures. The first is that a component model that provides well-defined and restricted rules for allowable patterns of component interaction is likely to be better suited for statistical analysis than a wholly unconstrained component model. The second conjecture is that product line settings may augment a component model's structure-oriented rules with rules governing semantic variation (i.e., product feature variation [Clements 02a]).

Section B.2.3 on page 107 describes the details of the sampling procedure for both components and assemblies used in the empirical validation of the controller PECT.

6.3.4 Develop Measurement Infrastructure

As with any experimental process, a measurement infrastructure must be developed to collect data. Where objective measurements are taken, as with λ_{ABA}, this infrastructure must be sufficiently well constructed to produce reliable and repeatable measurements. For λ_{ABA}, we developed a shared-memory mechanism for capturing traces of pin activation and deactivation. We also developed a distributed, trace mechanism based on the Universal Datagram Protocol (UDP) for measuring higher-order SAS assemblies (i.e., an assembly of operator and controller subassemblies).

The measurement infrastructure must also be sufficiently well documented to allow the results to be repeated, possibly by disinterested third parties not associated with the original validation experiment. Just what should be documented has much to do with the property of interest. Often, it is clear which environmental factors have a direct correlation to a property, for instance, latency. Latency for a component will vary from environment to environment, based on processor speed. Other environmental characteristics could have an indirect correlation to a property—which may not be immediately obvious—for example, the compiler (or compiler flags) used to produce the component.

Describing the test environment becomes increasingly important when trying to establish causal relationships between runtime and observed behavior and is especially critical when a discrepancy exists between the predictions and observations. Trying to find a correlation between such environmental characteristics, and behaviors of components and assemblies can be difficult. Altering the environment to limit or prevent unexpected behaviors is one viable approach. However, having a description about what characteristics were present during the validation may hold clues as to what was and was not considered to be possible sources of error.

For λ_{ABA}, the environmental characteristics that were recorded for the validation included platform hardware characteristics (e.g., processor, bus, and memory speed), software characteristics (e.g., operating system and compiler versions), and residual processes (e.g., those background tasks running at the time of the validation). These characteristics are discussed in Appendix B. A summary of the overall measurement and analysis infrastructure developed for λ_{ABA} is discussed in Appendix B.

6.3.5 Collect Validation Data

This step more or less speaks for itself. The only issue that requires attention is that a realistic validation experiment will generate a large volume of data and may require significant com-

puting resources and time to do so. All generated data should be archived so that it can be subjected to historical analysis.

6.3.6 Analyze Results

As noted earlier, the primary objective of empirical validation is to assign statistical labels to components and property theories. The computation of these labels should be straightforward, given an adequate definition of goals, measures, and sampling procedures. However, this does not mean that the only utility of empirical validation is the assignment of labels. In particular, the data generated by the validation experiment can be an invaluable resource for improving the quality of the property theory.

This proved to be a source of particularly valuable lessons in the case of λ_{ABA}. The data produced by validating the property theory uncovered several subtle and not so subtle inconsistencies in the PECT implementation. It is the repair of these inconsistencies that led to a second edition of this report.

7 Safety and Liveness Analysis in the Controller PECT

This report is focused on an empirical property theory (λ_{ABA}) and its validation. Equally vital to PECT are formal property theories. In this chapter, we illustrate the main ideas of model checking, one approach to formally verifying safety and liveness properties of components and assemblies. In that approach, a system model is extracted from component reactions and their compositions in an assembly. Safety and liveness conditions are expressed in a modal (temporal) logic and tested by an exhaustive search of the state space described by the system model.

We describe only the modeling and verification of temporal claims against a single component in the controller assembly. We defer a description of composed controller models and verifications to a follow-on report. A formal description of composition in Pin is provided by Ivers and associates [Ivers 02].

7.1 Model Checking

Model checking is a formal method for verifying finite-state concurrent systems. It is based on algorithms for system verification that operate on a system model usually expressed in some form of state-transition representation.

This system model is then checked against a set of desired properties expressed in a modal (temporal) logic. Model checking is a desirable verification approach, because it is automated and exhaustive. State-transition representations, in essence finite state machines, are intuitive to the average engineer. Temporal logics, such as computational tree logic (CTL) and linear temporal logic (LTL), are not as intuitive, but can be mastered with practice [Prior 62], [Pnueli 85]. Verification of the system is accomplished by “checking” the model—exhaustively searching the state space described by the system model and, for each state, checking whether the desired properties have been satisfied.

Characteristics of system models that favor model checking over other behavioral verification techniques include

- ongoing I/O behavior (in so-called “reactive” systems)
- concurrency (not a single flow of control)
- control intensive (minimal data manipulation)

The controller assembly has each of the above characteristics, indicating that model checking is likely to be a good fit.

The next section addresses how we apply model checking in the context of a PECT.

7.2 Process

The goal of our analysis is to determine whether the assembly of components used to implement the controller satisfies various safety and liveness properties. Therefore, we begin by producing a model of the software components used to implement the controller. In this example, we focus on the CSWI component of the SEI switch controller (see Figure 9 on page 23).

First, we build a Pin model of the CSWI component, basing the model on the implemented component, rather than its requirements. This is an important distinction; we care about what the implemented component *actually* does, not what it *should* do. Essentially, we want to ensure a strong correspondence between the model and the component implementation. Extraction of the model from the implementation is one way to accomplish this, even when performed manually, as in this example. (Alternatively, we could have built the model and generated the implementation; however, since we already had the latter, that approach was not useful.)

Next, we use a model checker to analyze the CSWI model. However, the Pin model uses a formal language—CSP [Hoare 85]—that is not understood by the model checker used in this exercise—NuSMV [Cimatti 00]. Consequently, we must define an interpretation of Pin to NuSMV. In this exercise, the interpretation is performed manually, although there is no obvious reason why it cannot be automated.

Once the CSWI model is expressed in a form that is understood by the model checker, we formalize the safety and liveness properties we want to analyze and use the model checker to determine whether the model satisfies them. Any failure that is uncovered by the tool must then be analyzed. Most failures include a counterexample—an execution trace in which the desired property is not satisfied.

A failure could mean one of three things:

1. There is an error in the CSWI component model. If the counterexample is not a possible trace of the implemented component, the formal component model must be corrected.
2. There is an error in the formalization of the safety or liveness property. If the counterexample does not violate the desired property we meant to check, the property has not been formalized correctly.
3. The model does not satisfy the property. If the counterexample is a possible trace of the implemented component and fails to satisfy the desired property, we have a real error, and the component implementation must be repaired.

Any type of failure can mean that we iterate back through the steps of this process. Modeling and analysis are often iterative processes that conclude only when a statement can be made regarding whether the component satisfies its required safety and liveness properties.

The steps of this process are elaborated in the following sections.

7.3 Building the Model

We assume that a Pin model of an assembly is available, such as the one shown in Figure 9 on page 23. As noted, the model checker we used, NuSMV, does not accept input in the formal notation used by Pin. Consequently, we translate our Pin model into a formal notation that the model checker will accept. The steps to translating the model include (corresponding to Sections 7.3.1-7.3.4, respectively)

1. Determine the scope of problem analysis.
2. Produce a Pin model of the problem.
3. Translate the Pin model into a state machine.
4. Translate the state machine into NuSMV.

These steps are elaborated in the following sections.

7.3.1 Determining the Scope of Problem Analysis

The context diagram shown in Figure 16 shows the controller assembly in Pin and highlights the region of that assembly used for our verification. The CSWI component contains the logic for interfacing to the operator and carrying out the selection and operation protocol with the XCBR component. The XCBR component controls the signals sent to the physical switch, and contains protection logic (via PIOC) and the logic needed to accept commands (i.e., select, open, and close) from the operator (via CSWI).

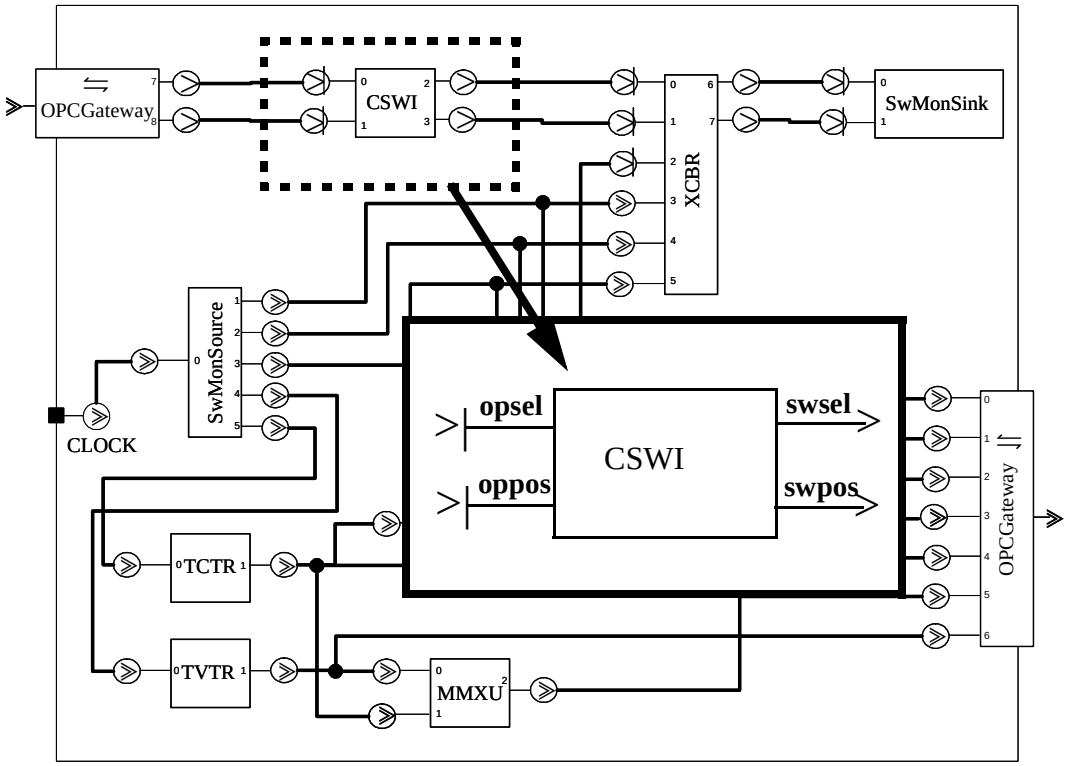

Figure 16: Controller System Diagram

As shown in Figure 16, the operator has two inputs to the controller—opsel and oppos—that indicate requests to select or position the switch. The CSWI component can send two outputs to the XCBR component—sbosel and sbopos—the effect of which is to select or position the switch. Before continuing, CSWI always waits until it receives a reply from XCBR indicating that the task has been performed. XCBR has two outputs to the physical switch where a selection or position change actually occurs—swsel and swpos. Two inputs from the physical switch—sosel and sopos—provide acknowledgement feedback to XCBR, through the respective sbo channels, and ultimately to the operator (perhaps in the form of indicator lights, etc.).

In this discussion, we focus on modeling the behavior of the controller assembly's CSWI component.

7.3.2 Producing a Pin Model of the Problem

CSWI has two sink pins—opsel and oppos—each of which is a synchronous mutex pin. Additionally, each sink pin is threaded, and the two sink pins are handled by the same thread. CSWI also has two synchronous source pins—sbosel and sbopos.

Because CSWI only has one thread of control, it is formally modeled in CSP as a single reaction. This specification is shown in Figure 17. Note that the leading underscore character (_) in event names indicates the initiation of an interaction on a pin of that name; the absence of a leading underscore indicates the completion of an interaction.

```
CSWI = _opsel.on -> _sbosel!on -> sbosel -> opsel -> Selected
    [] _opsel.off -> _sbosel!off -> sbosel -> opsel -> CWSI
Selected = _opsel.on -> _sbosel!on -> sbosel -> opsel ->
              Selected
        [] _opsel.off -> _sbosel!off -> sbosel -> opsel -> CSWI
        [] _oppos?x -> _sbopos!x -> sbopos -> _sbosel!off ->
              sbosel -> oppos -> CSWI
```

Figure 17: CSWI Pin Specification

7.3.3 Translating the Pin Model into a State Machine

While the Pin model of CSWI is based on CSP, NuSMV (our target model checker) requires input in the form of a finite state machine. As shown in Figure 18, we translated the Pin model to a simple state machine as an intermediate representation.

The transitions in that diagram correspond to CSP events in the Pin model. The names associated with these transitions, and subsequently in the NuSMV model, are based on the names of CSP events in the Pin model. For example, the Pin model defines the operator selection event as `_opsel`, with a data parameter of either `on` or `off`, written as `_opsel.on` or `_opsel.off`, respectively. To preserve traceability to event names in the Pin model, the state machine modifies this convention slightly by changing `_opsel.on` to `_opsel_on` or `_opsel.off` to `_opsel_off`. This transformation is necessitated by the syntax of NuSMV, which uses the period (.) operator for a different purpose. Consequently, periods (as well as exclamation points [!] and question marks [?]) were changed to underscores.

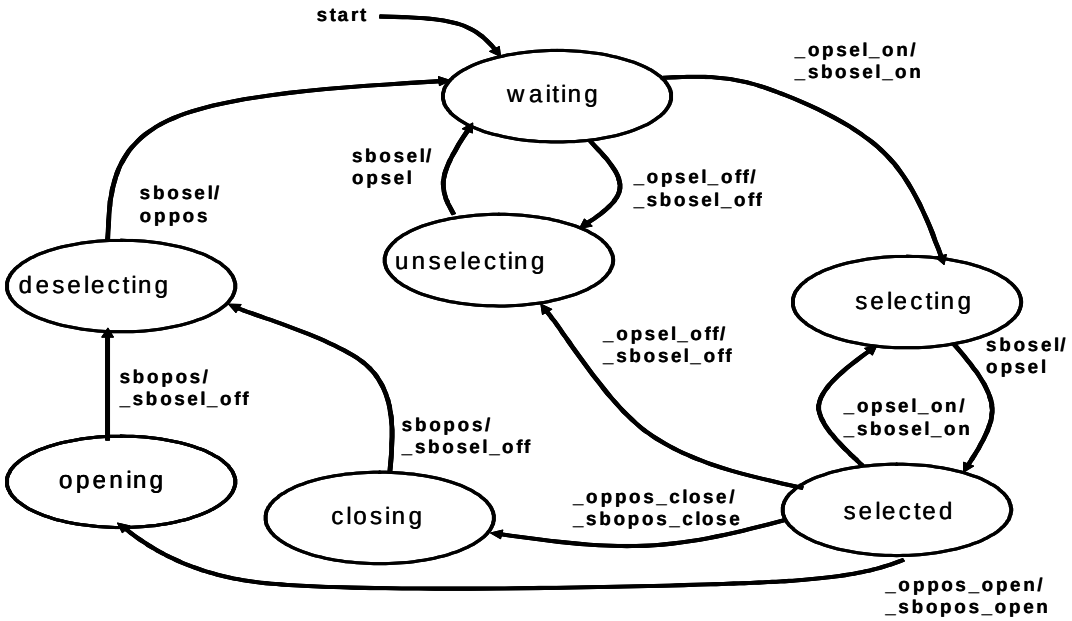

Figure 18: CSWI State Machine

Seven states compose CSWI: waiting, selecting, selected, opening, closing, deselecting, and unselecting.

The states and major transitions are described below.

- waiting state: CSWI starts in the waiting state and transitions to another state only when an `_opsel_on` or `_opsel_off` event occurs. An `_opsel_on` event, for example, indicates that the operator is selecting a switch and results in a transition to the selecting state. As part of this transition, CSWI initiates an `_sbosel_on` event. The "event1/event2" convention indicates that an occurrence of event1 not only triggers a transition, but also results in the generation of event2.[24] This convention is how we reflect the behavior found in the Pin reaction specification.
- selecting state: CSWI remains in the selecting state until it receives an acknowledgement that the selection has been accomplished, an sbosel output is received, and an opsel input is generated. CSWI then transitions to the selected state.
- selected state: CSWI waits in the selected state until one of the following events occurs:
  - An `_opsel_off` event is received, indicating the operator action of deselecting the switch and causing a transition to the unselecting state.
  - An `_oppos_open` event is received, indicating the operator action of requesting to open the switch and causing a transition to the opening state.

24. This follows the convention of Harel's statecharts, which are reflected in UML [Rumbaugh 00], where "event2" is called "action."

- An _oppos_close event is received, indicating the operator action of requesting to close the switch and causing a transition to the closing state.
- An _opsel_on event is received, indicating the operator action of selecting the switch and causing a transition back to the selecting state.

- opening state: If CSWI is in the opening state and an sbopos output is received, CSWI transitions to the deselecting state and generates an _sbosel_off event to deselect the switch, since the positioning is complete. If the acknowledge is not received, the system continues to wait in the opening state.
- closing state: If CSWI is in the closing state and an sbopos output is received, CSWI transitions to the deselecting state and generates an _sbosel_off event to deselect the switch, since the positioning is complete. If the acknowledge is not received, the system continues to wait in the closing state.
- deselecting state: If CSWI is in the deselecting state and an sbosel output is received, CSWI transitions to the waiting state and generates an oppos input.
- unselecting state: If CSWI is in the unselecting state and an sbosel output is received, CSWI transitions to the waiting state and generates an opsel input.

Although they are similar, the deselecting and unselecting states are both needed; the select and operate actions require different acknowledgements to be generated, even though the transition to each is caused by the same thing—an sbosel output.

7.3.4 Translating the State Machine into NuSMV

This section discusses the translation of the model from the state machine represented in Figure 18 to the NuSMV textual model shown in Appendix C. It was done in a straightforward manner following the steps listed below. Note that since the Pin model indicates that CSWI has a single thread of control, there is no need to use multiple processes in the NuSMV model.

1. The transitions in the state machine (e.g., _opsel_on and _oppos_open) appear in the NuSMV model as symbolic names associated with the input variable, which can be assigned any of the input values shown in the state diagram. The input variable can also have the value “none,” indicating that no inputs currently exist. The input variable is declared as an IVAR,[25] directing NuSMV to consider all combinations of input sequences when analyzing claims.

25. IVAR is a keyword from the NuSMV tool. It’s short for “input variable.”

2. The outputs in the state machine (e.g., `_sbosel_on` and `_sbopos_open`) appear in the NuSMV model as symbolic names associated with the `output` variable, which is declared as a VAR.[26] As with `input`, `output` can have the value "none," indicating that no outputs currently exist. The `output` variable is changed only within the NuSMV model. The other VAR declaration is the `state` variable, which takes on the state names from the state machine (e.g., waiting and selecting).

3. There is a case expression relating the behavior of the `output` variable at the next state to the current values of the `state` and `input` variables. For example, the first rule states that if the current `state` is waiting and the current `input` is `_opsel_on`, the next value of the `output` is `_sbosel_on`. Hence, each output is delayed by a single tick from the condition causing it. The last rule in the case expression (`1: none`) indicates that if none of the other rules in the case expression apply (e.g., if the `state` is waiting and the `input` is `_oppos_open`), the `output` variable will default to "none."

4. The other case expression relates the changes in the `state` variable to the current `state` and the `input` variable. For example, if the `state` is waiting and the `input` is `_opsel_on`, the next `state` will become selecting. The last rule (`1: state`) indicates that if none of the other rules apply, the next `state` remains equal to the current `state`.

7.4 Analyzing the Model

Once we have a model of CSWI in a form that NuSMV understands, we can apply model checking to determine whether the model (and, therefore, the component's implementation) satisfies various system properties. For example, this type of analysis can determine whether deadlock is possible.

In this section, we begin by discussing different types of system properties that we can analyze using model checking. We then introduce temporal logic, a formal language commonly used to express system properties. Finally, we show examples of properties expressed in temporal logic that we used in analyzing the CSWI model.

7.4.1 Types of System Properties

The types of system properties evaluated by model checkers are typically classified as safety or liveness properties. Although other taxonomies have been proposed by Naumovich and Clarke [Naumovich 00], we continue to use the safety and liveness classification. Intuitively, a safety property specifies that "bad things" cannot happen, and a liveness property specifies that "good things" eventually happen [Lamport 77].

---

26. VAR is a keyword from the NuSMV tool. It's short for "variable."

To illustrate these points, consider a software application in which two or more concurrent processes must access the same critical section of code. A condition that must hold is mutual exclusion, which means that two (or more) concurrent processes cannot enter and execute a common critical section of code simultaneously. This condition can be specified as the safety property “two processes cannot be in the same critical section at the same time.” Another condition that is usually desirable in concurrent processes is starvation freedom, which means that eventually each process will enter the critical section. This can be specified as the liveness property “whenever process P1 wants to enter the critical section, it will eventually do so.”

7.4.2 Temporal Logic

Temporal logic is frequently used to formally define safety and liveness properties. Two well-known logics used in verification are LTL and CTL.

In LTL, time is viewed as a linear sequence of time instants, with each time instant having exactly one successor. The semantic intuition of LTL expressions (where **a** and **b** are primitive prepositions) is as follows:

- □ **a**
  This is read as “henceforth (or always) **a**” and means that statement **a** is true now and in all future time instants.
- ◊ **a**
  This is read as “eventually **a**” and means that **a** is true either now or at some time instant in the future.
- ○**a**
  This is read as “in the next state or at the next time instant, **a** will be true.”
- **a** $\mathcal{U}$ **b**
  This is read as “**a** is true until **b** becomes true.”

In CTL, time is viewed as a branching tree of time instants, each with one or more possible successors. A path is a particular linear sequence of time instants (a path in the tree) in which each successive time instant is one of the possible successors of the current time instant. The semantic intuition of CTL expressions (where **p** and **q** are primitive prepositions) is as follows:

- AG **p**
  Along all paths, **p** holds globally (henceforth).
- EG **p**
  At least one path exists where **p** holds globally.

- AF **p**
  Along all paths, **p** holds at some time instant in the future (eventually).
- EF **p**
  A path exists where **p** holds at some time instant in the future.
- AX **p**
  Along all paths, **p** holds in the next time instant.
- A[**p** U **q**]
  Along all paths, **p** holds until **q** holds.
- E[**p** U **q**]
  A path exists where **p** holds until **q** holds.

Both LTL and CTL include the usual $\mathbf{a} \wedge \mathbf{b}$, $\mathbf{a} \vee \mathbf{b}$, $\neg\mathbf{a}$, $\mathbf{a} \rightarrow \mathbf{b}$ logic expressions. While LTL expressions appear to be a subset of CTL, their expressive powers differ. It is possible to express properties in LTL that cannot be expressed in CTL, and vice versa. The NuSMV model checker allows claims to be expressed in either CTL or LTL; however, CTL was used as the primary specification language in this experiment.

7.4.3 CSWI Claims

The desired system properties analyzed with respect to the CSWI model include safety and liveness properties. The specific behavioral characteristics of each property are often first expressed in a natural English language format and then translated into CTL (or LTL). These temporal logic expressions are often called *claims*. Examples of claims for CSWI are discussed in the following sections.

Example Liveness Claim

A number of liveness claims were constructed for CSWI, the most obvious of which investigate its operational correctness. Those claims follow the pattern of “if an operator action occurs, a signal is conveyed to the appropriate entity.” In this example, the entity would be XCBR. A specific desired liveness property is “if the operator selects the switch, the ‘select before operate’ signal is sent to the XCBR.” Translating this into the appropriate CTL expression results in

$$\text{AG } ((\text{state} = \text{waiting}) \wedge (\text{input} = \text{\_opsel\_on}) \text{ -> AX}(\text{output} = \text{\_sbosel\_on}))$$

The above claim is read as “for all paths globally, whenever CSWI is waiting and the operator select comes on, the sbo select signal will come on.”

Example Safety Claim

Safety properties are logical expressions of conditions that should never occur, presumably because the resultant action could endanger a person or harm equipment. One of the operational characteristics of the controller assembly is the enforcement of the "select before operate" protocol. This can be rephrased as the desired safety property "under human control, the switch cannot be operated unless it is first selected." Translating this into the appropriate CTL expression results in

$$\neg E\,[\neg(\text{output} = \text{\_sbosel\_on})\ U\ (\text{output} = \text{\_sbopos\_open})]$$

The above statement is read as "it is impossible for the switch to be opened (`_sbopos_open`) before it has been selected (`_sbosel_on`)."

Example Failed Claim

The CSWI model does not satisfy the following safety claim:

$$AG(\text{input} = \text{\_opsel\_on} \rightarrow AG\ \text{output} = \text{none})$$

This is actually a good thing, because we don't want the safety claim to be true. The claim asserts that whenever the operator selects the switch, nothing will ever happen (i.e., no output will be generated). When we run this claim in NuSMV, it confirms that the model does not satisfy the claim, and it also presents a counterexample showing a particular trace in which the claim is not satisfied. The counterexample is shown in Figure 19.

```
-- specification AG (input = _opsel_on -> AG output = none)
-- is false
-- as demonstrated by the following execution sequence
State 1.1:
input = none
output = none
state = waiting

State 1.2:
input = _opsel_on

State 1.3:
input = _oppos_close
output = _sbosel_on
state = selecting
```

Figure 19: NuSMV Counterexample

The counterexample represents three time instants (1.1, 1.2, and 1.3), which have the following meanings:

- 1.1: CSWI begins in the initial state, waiting, and there is no input or output.
- 1.2: An input of `_opsel_on` is received, indicating that the operator selects the switch.
- 1.3: An output of `_sbosel_on` is generated, and CSWI transitions to the selecting state.

In the final time instant, 1.3, CSWI generates an output, violating the claim that nothing will happen after an operator selects the switch.

The NuSMV model and the claims verified against it are contained in Appendix C.

8 Discussion

The basic structure and concepts of a PECT had been prototyped before the SAS model problems were defined [Hissam 02]. Nonetheless, the SAS problems, although simple by the standards of production systems, were complex enough to reveal previously unappreciated, or underappreciated, subtleties of PECT. We discuss some of those subtleties, with an emphasis on those aspects of PECT that were shown to require further study. In Section 8.1, we discuss methodological results, while, in Section 8.2, we briefly touch on the model solutions.

8.1 Results on Method

8.1.1 Interfaces Between Specialized Skills

One striking observation is that designing, implementing, and validating a PECT requires a combination of specialized expertise. Some indication of this can be gleaned from the different workers identified in Chapter 4. Table 4 focuses not on workers, but on their skills, and in particular those skills needed to build the SAS PECT. To some extent, these skills are coherent with the workers identified earlier. However, it is significant that we often found it necessary for a single worker to straddle several skills, suggesting, as is anticipated in the RUP, that a single person will fill many roles, perhaps simultaneously. Note that some divergence from the general set of workers and their implied skills outlined in Chapter 4 is expected, given our focus on research rather than production.

Table 4: Areas of Expertise Needed to Build SAS Operator and Controller PECTs

Area of Expertise	Use of Expertise in SAS Model Solutions
component technology	Design the Pin component model, application program interface (API), runtime environment, and deployment model.
component-based development	Implement the component technology, and define and implement conformant SAS components (including their behavioral models) and their SAS assemblies.
systems programming	Implement real-time extensions to Windows 2000, observation mechanisms for validation environments, and low-level features of the component runtime environment.
measurement, validation, experimental design, statistical analysis	Define measurement scheme, and statistical analysis and labeling approach for component execution time and latency property theories. Conduct component measurement and empirical validation.

Table 4: Areas of Expertise Needed to Build SAS Operator and Controller PECTs (Continued)

Area of Expertise	Use of Expertise in SAS Model Solutions
performance modeling	Define the latency property theory based on the theory underlying RMA.
model checking	Develop model-checker-specific state models and define, develop, and check temporal claims of controller safety and liveness conditions.
language design and formal semantics	Design the specification language(s) for Pin assemblies and the formal (CSP) compositional semantics of connectors between component (CSP) behaviors.
electronics	Design, develop, and calibrate test hardware to simulate switch control. This custom hardware was used to empirically validate the controller latency theory.
substation automation	Define the model problems and evaluation criteria such that the PECT reflects a representative class of design and engineering problems.

As discussed in Chapter 5, co-refinement is a means by which actors possessing expertise in two domains, each with specialized conceptual schemes and vocabularies, can nonetheless negotiate mutual satisfaction of a shared concern (practicability), while ensuring satisfaction of their unique concerns (generality and tractability). Co-refinement is a negotiation process that establishes a shared conceptual interface between those actors. Note that although the interface is conceptual, it serves an analogous role to the more familiar notion of software interface: it helps to make explicit, and therefore manage, the dependencies in a system.

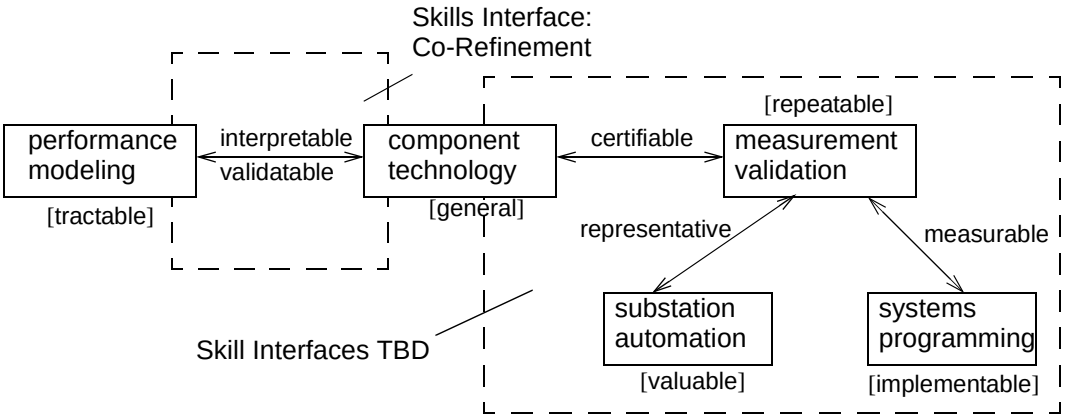

Figure 20: Interfaces Among PECT Development Skills

This metaphor of negotiation is apt and can be applied elsewhere in the PECT development process. Consider, for example, the interface between the skills depicted in Figure 20. Above or beneath each skill, we placed, in brackets ([ ]), a one-word descriptor for the PECT design concern that motivates the application of that skill. The associations are labeled with a one- or two-word descriptor of a *shared* concern. In Figure 20, we show a shared concern between component technology and measurement validation—"certifiable." This concern covers issues such as whether the measured property can be validated by an independent party, whether the

measures fall within required confidence or tolerance intervals, or whether the level of systematic and random measurement error can be quantified. Measurement and validation also share a concern with the domain specialist—whether the assemblies used for empirical validation are representative of real design problems.

It is unreasonable to expect a single person to possess the range of expertise reflected in Figure 20. A transitionable PECT development method must provide a clear distribution of these forms of expertise across different workers and processes that define a separation of the skills across activities.

8.1.2 Infrastructure Complexity

Carl Sagan introduced his television series, *Cosmos*, with the statement, "*To bake an apple pie, we must first create the universe*."

Before we can assemble components and predict their aggregate properties, we must create a PECT infrastructure that includes the collection of programs *exclusive of* application-level (e.g., controller-level) components and their assemblies. The view of PECT depicted in Figure 5 on page 13, *The Four Environments of a PECT*, is a good guide to what constitutes infrastructure. Nonetheless, that structure is guilty of oversimplification. The practicality of PECT rests, to some extent, on the cost of developing and maintaining its infrastructure. We must therefore strive to understand which part of a PECT infrastructure's complexity is inherently related to PECT, and which part arises from concerns that have already been addressed by available technology or is inherent to a problem domain such as SAS.

One approach to obtaining this understanding is to examine Figure 5 from the perspective of a *software development environment* (SDE). Clearly, more is required of an SDE than the assembly and analysis environments shown in Figure 5. An SDE would also include facilities for configuration and version management, and a repository of components and assemblies, for example. The components themselves will likely be implemented in a conventional programming language such as C#, C++, or Java. This suggests that code browsing, compiling, linking, and debugging facilities are also required in a PECT. Still, none of this is part of a PECT-specific infrastructure, and it is all readily available—for example, Microsoft's VisualStudio and Borland C++'s Developer. What remains is to ensure that a PECT infrastructure is integrated with an existing SDE. While not trivial, this is not a substantial risk to PECT.

Another way to look at Figure 5, however, is from the perspective of a language-specific SDE (LS-SDE). A language comprises its syntax and semantics. In PECT, the syntax of the language is defined by the construction model [Ivers 02]. However, the language used to specify (constructive) assemblies for any PECT is not necessarily simple in structure; its syntax and semantics are dependent on an open-ended set of environment types and analysis views, only some of which may be applicable in any given application. This added complexity must be

laid directly on the PECT doorstep—it cannot be waived off as a simple matter of SDE tooling.

Some of the infrastructure needed to support such a semantically extensible LS-SDE can be regarded as a one-time investment—once developed, a construction model and its infrastructure can be used repeatedly for a family of applications [van Ommering 02]. This suggests that a PECT may be more easily justified in the context of a product line. Although product lines require a substantial organizational commitment, they are justified by business considerations [Clements 02a], and PECT may in fact make product lines more effective. On the other hand, there is not likely to be an infinite supply of useful construction models. For example, Pin bears a strong resemblance to other component languages, notably Darwin [Magee 93], Wright [Allen 97], and the emerging UML 2.0 standard.[27] There is hope, then, for convergence on an industry-standard component language.

8.1.3 Design Space: Rules for Inclusion *and* Exclusion

We have adopted the idea of a PECT design space as a basis for validating empirical property theories—in this report, latency. The term *design space* has a metaphorical connotation, but it is given concrete meaning in empirical validation through its definition as a set of discrete *variation points*. That is, we define design space as an N-dimensional cartesian coordinate system,[28] where coordinates on each dimension take values from a (small) finite set representing one possible variation at that variation point, or on that dimension. From this design space, we select the sample of assemblies to use for the statistical analysis of predicted versus observed assembly behaviors.

The coordinate system for this design space defines the set of assemblies that are included within the scope of a property theory. That is, the coordinate system defines inclusion rules for assemblies. That definition reflects a positive statement of where a property theory must be valid, in the sense that any sample taken from this design space will be representative of the class of systems to which this PECT will be applied. We want the predictions to be valid for all members of this class. There is evidently a close connection between this concept of design space and that of variation points in product line architecture (e.g., Theil and Hein's work [Theil 02]), and in fact we use the term *variation point* to emphasize this connection.

In practice, however, we must also define rules for exclusion from the design space. That is, we will almost certainly want to characterize not just where we think the property theory

27. UML 2.0 is currently a work in progress. See <http://www.omg.org/uml/> for more information about it.

28. To be more precise, the coordinate system under discussion defines sufficient conditions for the assemblies of a class of products; the necessary conditions are defined by the construction model and its interpretation under the property theory being validated. For example, the interpretation of the controller latency theory λ_{ABA} restricts assembly topologies to only those that satisfy the priority ceiling constraint. All assemblies must satisfy these purely syntactic, and therefore necessary, conditions.

should be valid, but also where we know it is not valid. One way to approach this is to discover the analysis limits for each variation point. The analysis limits of a variation point define the greatest lower bounds and/or the least upper bound for the set of values defined for each variation point.[29] More simply, each variation point can be tested to destruction to find its analysis limits. This is analogous to the approach taken by Gorton and Liu [Gorton 02]; however, their measurement object is a large-scale commercial off-the-shelf (COTS) component (an Enterprise JavaBeans server) and not the property theory used to predict performance.

In general, there has been insufficient work in software engineering research regarding the empirical validation of design theories. PECT provides a conceptual vocabulary and technical means for making much needed progress in that area.

8.1.4 Confidence Intervals for Formal Theories

If there has been insufficient work in the empirical validation of design theories based on observation (and, therefore, measurement), there has been almost no work in the empirical validation of theories based on formal logic. (Barnett and Schulte's work is an exception [Barnett 01].) Indeed, the very notions seem to be mutually exclusive, which may account for the lack of attention to this subject. Nonetheless, assertions about safety and liveness properties of assemblies (as discussed in Chapter 7) are made with respect to a model of the components' behaviors and their interactions. We can have 100% confidence that a claim is satisfied or falsified by a model. However, there are two more substantial questions:

1. Can we have objective confidence that the model corresponds to reality?
2. Can we have confidence that the model is sufficient to justify the claims?

The answer to question 1 depends, to some extent, on how the correspondence (the hypothesized "satisfies" relation) between the model and reality is established in the first place. It might be established through automated (implying formal) translation. Two forms of automated translation are possible: generating components from models (e.g., Sharygina's work [Sharygina 02]) and generating models from components (e.g., Havelund and Pressburger's work [Havelund 00]). The first form is common in practice, while the second is more of a research topic. In both cases, our confidence in the satisfaction claim is proportional, if not equal, to our confidence in the correctness of the translator implementation. This does not lead to infinite regress, however, since we have an empirical means of establishing this confidence through the testing of the translator. If, however, the correspondence between the model and component is manual, there is no basis for objective confidence in the hypothesized "satisfies"

29. We assume that the set of values for each variation point is partially ordered. This assumption seems reasonable if there is a correspondence between the variation point and some property theory parameter. Only those variation points will be of interest when testing that theory's failure.

relation. The implication is that we must use automation to generate models and/or components in our approach to PECT.

The answer to question 2 is more subtle and rests on equivalence relations over models. Asking whether a model satisfies an implementation (we now use this term in place of *reality*) is just a more specific form of the question of whether two models are equivalent for some purpose. Sometimes, but by no means always, the more concrete model is said to *refine*, or be *a refinement of,* the more abstract model; the equivalence relation is then called *refinement.* A property proven to hold for a model will also hold for its refinements.[30] However, there are many different ways of defining refinement. Roscoe defines a hierarchy of three forms of refinement for the CSP process algebra—trace, trace failures, and failures-divergence refinement [Roscoe 98]; Davies (citing work by Reed [Reed 88]) describes a lattice of nine refinement equivalences for CSP [Davies 93]. Not to be outdone, van Glabbeek [van Glabbeek 01] identifies 13 equivalence relations, not all based on CSP traces. In each case, different notions of equivalence can be used to justify different claims.

We can draw two inferences from the discussion of question 2. First, we should leave the definition of equivalence relations over formal models—and the theorem provers based in them—to the experts; we should apply the fruits of their efforts in PECT. Second, a PECT infrastructure must be flexible enough to accommodate a variety of formal theories, each of which provides a notion of semantic equivalence sufficient to make the kinds of specification claims required.

8.2 Results on Model Solutions

Three model problems were posed, one leading to an operator-level PECT, one to a controller-level PECT, and one to a higher-order PECT for assemblies of assemblies. In the final analysis, only the operator and controller PECTs were developed. While a higher-order operator/controller assembly was demonstrated, no property theory was developed for higher-order assemblies.

The operator-level PECT was developed without incident. However, it became apparent during its development that the operator interface was quite simple—in fact, only reentrant sink pins were required. Moreover, operator display latency is completely dominated in any higher-order assembly—the time required to interact with the display and perform data validation on operator input is minor (often less than a simple scheduling question), compared with network and controller latency.

30. This is actually too strong a statement; not all properties are preserved under refinement—deadlock freedom, for example.

Nonetheless, we developed a simple latency property theory for the operator PECT that proved to be, under the highly constrained construction model used, highly accurate. A summary of the property theory's performance is shown in Figure 21. We do not summarize the results of this performance as a confidence or tolerance interval, since the simplicity of the PECT led us to focus our validation efforts elsewhere.

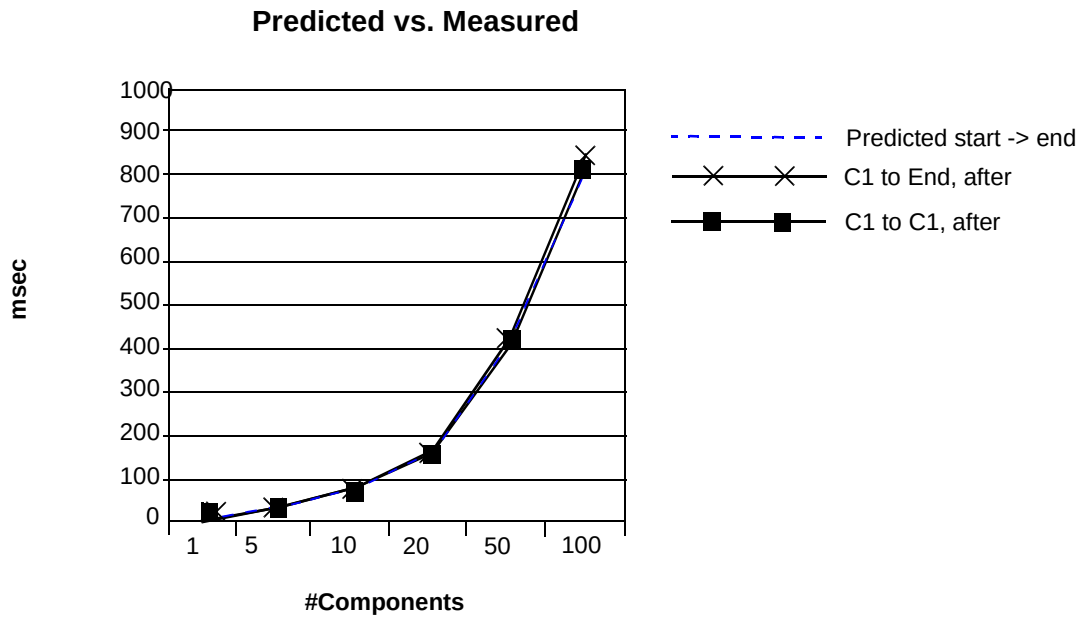

Figure 21: Results of Spot Validation of Operator PECT Latency Theory

The controller-level PECT presented considerably more methodological and technological challenges, most of which have been discussed already. To reiterate the main results, the λ_{ABA} produces stable and accurate predictions—a 99% confidence interval for an upper bound of 1% MRE for 8 out of 10 predictions. A full report on the empirical validation of λ_{ABA} is provided in Appendix B.

The λ_{ABA} validation exercise was more interesting for its methodological implications than for its validation of a specific latency model. The initial validation data revealed a significant quotient of systematic and random error. Several experiments were required to stabilize the model, each of which exposed different sources of experimental error. The first three served to remove errors in the measurement infrastructure itself—the instrumentation was developed in tandem with the validation work and was therefore itself a work in progress. The last two runs served to remove inconsistencies between the model and the controller runtime implementation. In other words, the empirical validation exposed an inconsistency between model assumption and runtime implementation; moreover, the raw data produced by the validation allowed us to quickly identify the sources of error.

9 Next Steps

As noted in the introduction to this report, the objective for this work was exploration—the technical concepts of PECT and the methods for developing and validating a PECT. As with any exploration, a body of knowledge was generated that must now be consolidated. Improved understanding leads to new, and more challenging, questions that must be explored.

In this chapter, we summarize the most important areas of consolidation and exploration. Motivating these priorities is our plan to extend the work reported here during the next year to accommodate more rigorous feasibility requirements and, working with our industry partner, ABB, to incorporate feasibility requirements drawn from the domains of substation automation *and* industrial robotics.

9.1 PECT Infrastructure

The infrastructure of a PECT is only a means to an end—predictable assembly from certifiable components. However, it is a necessary means, and our ability to "scale up" the PECT approach requires some up-front investment in that infrastructure. Some of this investment has already occurred as a by-product of producing SAS model solutions and their PECTs; more needs to be done.

9.1.1 Measurement and Validation Environment

A substantial suite of tools was developed to measure component execution time, automatically generate assembly samples, interpret those assemblies to instantiate and operate on analysis views, capture traces of assembly execution, and perform statistical analysis of the timed execution traces vice their predicted traces. Most of the tools in this tool suite were used for both the operator and controller PECTs, suggesting that those tools may well form the core of a common measurement and validation environment for future PECT development.

The future work required is consolidative in nature; it involves integrating, refining, and ruggedizing the existing tool set. Integration will achieve the end-to-end automation that is required to generate the large data sets necessary to empirically validate a substantial design space. This automation will require more attention to the interfaces defined between tools and to data interchange syntax and semantics. A good foundation for data interchange has been established by using XML to externalize component assemblies, but this externalization must be made consistent with an evolving construction model and its specification language.

9.1.2 Core Pin Language and Assembly Environment

One of the tenants of the PECT approach is that complexity can be "packaged" and hidden from end consumers. Still, there will be some minimum, irreducible complexity exposed by each analysis view. If this complexity is to be manageable in the aggregate, we must strive to eliminate as many forms of unnecessary complexity as possible. To (mis)appropriate a now famous political aphorism, "*it's the packaging, stupid!"*

The process used in the SAS model solutions for creating and deploying assemblies of components was tedious and error prone. Component properties and assemblies of components were encoded directly in XML, and although the connector "wiring" was generated automatically from XML, there is no disguising the awkwardness of representation and process. A visual composition environment (e.g., see the work of Plakosh and associates—specifically Chapter 5 [Plakosh 99]) is a form of packaging that can dramatically simplify, and make more pleasurable, the process of attributing component behavior, and developing, analyzing, and deploying their assemblies.

As discussed in Section 8.1.2 on page 67, a (visual) PECT assembly environment is a form of LS-SDE—in this case, one that emphasizes pure composition. The future work needed in this area is a mix of exploration and consolidation, and requires a definition of a core language for the Pin component model (see the work of Ivers and associates for a start at this [Ivers 02]) and the development of a visual environment to support it (a small matter of programming). By *core*, we mean a graphical and textual syntax (including a syntax for reaction rules) and syntax-directed externalization.

9.1.3 Real-Time Component Runtime Environment

One pleasant surprise in developing the SAS model solutions was that the component runtime environments for the operator and controller were rather simple to develop. Only the controller environment posed a challenge due to the artificial circumstance of our building a priority-based real-time PECT on an operating system (Windows 2000) with very limited support for priority-based scheduling. Even so, the operator and controller component runtime environments emerged largely from a restriction of component freedom to only a limited set of environment (runtime) interfaces.

Still, for the next stage of PECT development, more care must be taken when selecting an underlying operating system and specifying the environment types for hierarchical assemblies (see Section 3.3.2 on page 17)—the latter supports distributed assemblies. A definition of these environment types will also be essential to the compositional semantics of environment-specific connectors added to the core Pin component model.

9.1.4 Model Checking (Analysis) Environments

The SAS model problems were focused primarily on theories of latency. For PECT to be successful, it must be applicable to other behavioral theories. Compositional theories of reliability for components and assemblies are also of interest to the software industry. Empirical theories for compositional reliability are possible, but generalized, falsifiable theories have proven to be elusive. See the work of Mason [Mason 02], and Stafford and McGregor [Stafford 02] for an entree to this topic. Formal proofs of correctness have traditionally been thought of only as a means of obtaining reliability, not for predicting it. However, as discussed in Section 8.1.4 on page 69, there may be an empirical dimension to formal theories of correctness after all.

Complete proofs of correctness remain problematic, but within recent years, substantial progress has been made in improving the practicality of technologies for so-called *partial proofs of correctness*. In particular, proofs based on the exhaustive model checking of specific safety and liveness properties specified in a temporal logic are becoming almost mainstream. For recent developments in model checking of control systems, see Sharygina's work [Sharygina 02], for details on process algebra and model checking of concurrent Java programs, see the work of Magee and Kramer [Magee 99], and for a more theoretical perspective on the model checking, see the work of Clarke and associates [Clarke 99]. These results and those related to them are finding their way into practice. For example, Microsoft is using compositional verification to certify third-party device drivers [Ball 02].[31] Our principle motivation for using the CSP process algebra to specify models of component behavior (i.e., reactions) is that models of assembly behavior can be composed for analysis automatically.

Our concrete next step is to develop back ends to generate lower-level model representations for use in formal model checkers such as SMV[32] [Cimatti 00], Failures-Divergence Refinement [Gardiner 00], and SPIN [Holzman 97], probably in that order. Although this is not trivial, it is fairly simple. More complexity is involved in the question of compositional reduction and abstraction, and other techniques to minimize the state space of composed assembly behaviors. Our approach is to defer as many of these details to the back-end analysis tool as is practical and to introduce such techniques in the Pin infrastructure only as needed.

9.2 PECT Method

This report has outlined in a broad way the processes involved in developing and validating a PECT. The general area of certification is ripe for consolidation into a PECT validation guide and further exploration into the certification locale (i.e., the environment where the certification will be done). The next steps in these areas are outlined below.

---

31. For more information, see <http://research.microsoft.com/slam/>.

32. SMV is a symbolic model-checking tool developed at Carnegie Mellon University. For more information about it, see <http://www.cs.cmu.edu/~modelcheck/smv/smvmanual.ps>.

9.2.1 PECT Empirical Validation Guide

How is a validation experiment constructed? How are measurements collected, and how is systematic and random error quantified? How is systematic error removed? And how is the remaining error propagated? How many samples are required to achieve the desired statistical confidence? Which form of confidence interval is most appropriate? How is the assembly population defined, and how is the sample (or frame) of that population selected? What inferences can be drawn given this sampling procedure? What statistical tools can be used to analyze unexpected results?

Most of these questions are reflected in the workflows discussed for empirical validation in Chapter 6 and illustrated in detail in Appendix B. However, our workflows are descriptive rather than prescriptive—they describe *what* steps should be performed, but do not, in most cases, specify *how.* Any prescriptions provided are quite general. As a result, the skills needed to validate a PECT are not well contained within the current method; a more prescriptive guide for laboratory validation work is a necessary consolidation effort. The objective of a prescriptive guide would be to describe the validation process and identify (if not describe) the experimental and statistical foundations necessary for conducting a sound validation. Elements of such a guide are outlined by Moreno and associates [Moreno 02], but many more elements are needed.

9.2.2 Certification Locale and Foundations for Trust

Our work so far has focused on establishing component measures in a laboratory setting. We have not yet addressed issues of how these measures can be established independently. Of equal importance to the ability of third parties to validate the predictive power of a PECT is their ability to assign trusted and objective properties to components. One aspect is the potential of insurance underwriting (e.g., see the work of Li and associates [Li 02]) as a basis for a weaker notion of trust, which we may call *confidence*. This is, however, just one aspect of a set of aspects required to establish what amounts to a social network of trust. We plan to explore them in the context of our next round of industry model problems.

Appendix A λ_{ABA} Property Theory

An integral part of a PECT is its reasoning framework, which comprises three elements: a property theory that allows reasoning about a specific quality attribute using models of the system; an automated reasoning procedure that maps assemblies to elements of the property theory (interpretation) and produces predictions about the assemblies; and a validation procedure that is used to assess the validity of the predictions. This appendix describes the first two elements of the λ_{ABA} reasoning framework; Appendix B shows how empirical evidence is used to validate the predictions.

λ_{ABA} can be used to predict the average latency of arbitrary execution paths through an assembly comprised of concurrent periodic tasks with bounded execution times running on a single processor. The essential concepts of the λ_{ABA} property theory are presented in Section A.1, and the algorithms used for prediction are discussed in Section A.2. Section A.3 describes the Pin-to-λ_{ABA} interpretation, and Section A.4 gives an (abstract) syntax-directed translation scheme for this interpretation.

A.1 Essential Concepts of the λ_{ABA} Property Theory

A *task* is the unit of concurrency in the system; it executes periodically and has a fixed period. Initially, let's assume that a task has two other properties: priority and CPU time (sometimes referred to as execution time). We refer to each complete execution of a task as a *job*. The job *latency* is the amount of time it takes from the moment the task is ready to run to the moment it finishes executing. In Figure 22, there are three tasks that are ready to run at time zero. The timeline shows six jobs for task T1, each one with a latency of one unit of time. It shows four jobs for task T2, of which the first, second and fourth have a latency of three units of time, and the third has a latency of two units of time. Finally, the timeline shows one job for task T3 that has a latency of 14 units of time.

The latency of different jobs of the same task may be different due to the phasing of tasks and their priorities. In Figure 22, for example, as a result of delays, T2 latency varies from two to three units of time. The start of a task is sometimes delayed because of a higher-priority task (e.g., the start of the first job of T2 is delayed by T1). A delay can also occur when a task is already running and is *preempted* by a higher-priority task (e.g., in its second job, T2 is preempted by T1).

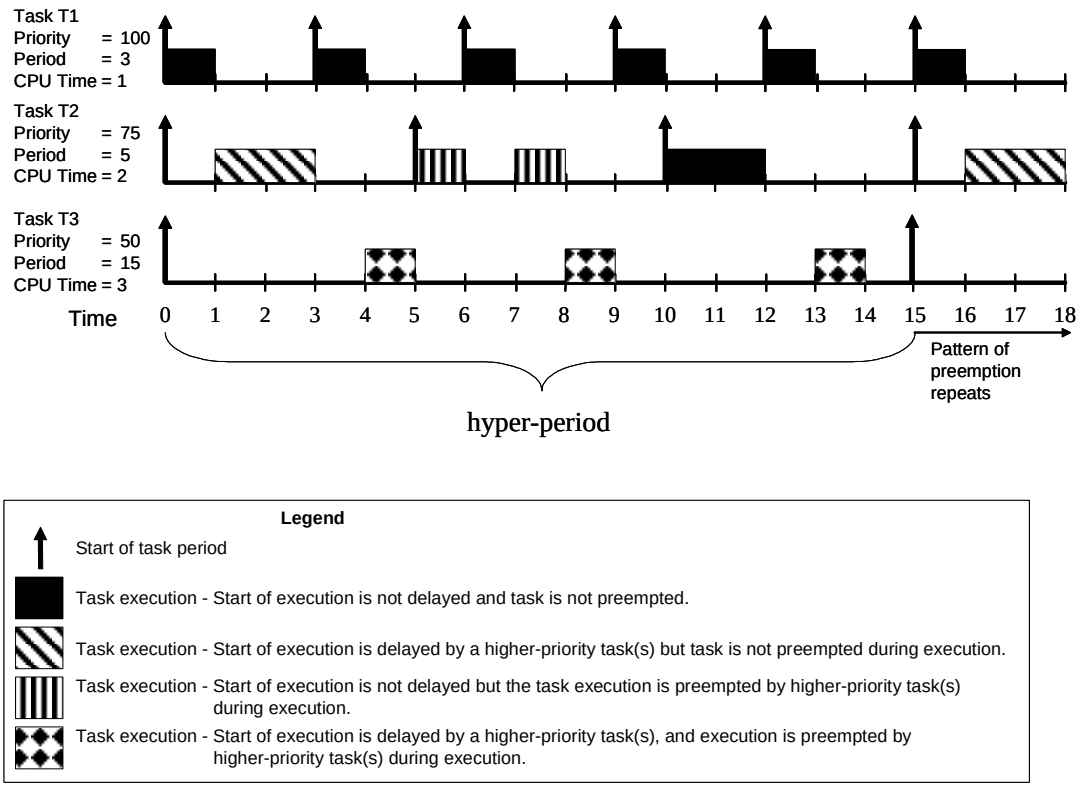

Figure 22: Timeline Showing Task Phasing and Hyper-Period

Given N tasks T_i, we define the *hyper-period*[33] (HP) as the least common multiple (LCM) of the periods of all tasks. As shown in Figure 22, the hyper-period is the amount of time until the pattern of execution for a set of tasks repeats. We only need to analyze the hyper-period, because it covers all the possible latencies that a task will exhibit. After a hyper-period, the pattern of latencies repeats, meaning that, for each task, the number of latency predictions can be computed as follows:

$$NP = \left\lceil \frac{HP}{T_i.period} \right\rceil$$

Besides the period, a task T_i has a property, T_i.offset, that indicates the arrival time of the task's first job. In Figure 22, all tasks have offset zero, but in most assemblies analyzed using λ_{ABA}, tasks are ready to run their first job at different times after the first job of the first task starts. In this case, the tasks have different offsets. Period and offset are required, fixed-value properties of a task.

33. Sometimes also referred to as *cycle time* in real-time literature.

Each task T_i consists of a sequence of subtasks, T_i.subtasks, that perform the actual computation. A subtask corresponds to the execution that takes place when a sink pin of a component is activated. Each subtask s_j has a priority, s_j.priority, and an execution time, s_j.C. These concepts and their relations are depicted in Figure 23. The total execution time of a task can be calculated as the sum of the execution time of its subtasks.

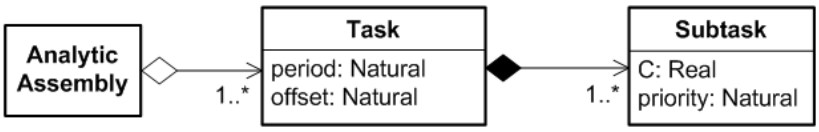

Figure 23: Elements of the λ_{ABA} Analysis Model

The following are the major assumptions of the λ_{ABA} property theory:

- A task is the only unit of concurrency. In fact, tasks are partitioned in subtasks, but subtasks within a task run sequentially, not concurrently.
- A subtask corresponds to the activation of a component and, because different components may have different priorities, each subtask runs at a different priority. Therefore, the same task can run at different priorities depending on which subtask of that task is running at the moment.[34]
- A task can be preempted only by a higher-priority task.[35]
- If no higher-priority task is ready, a running task runs up to completion.
- Subtasks do not yield the CPU until they complete their execution (i.e., tasks do not suspend themselves).
- Subtasks do not block on I/O.

λ_{ABA} does not address the situation where tasks can be blocked waiting on an I/O request, but it can handle blocking on a semaphore to enter a critical section. If two or more tasks share a critical section, that section is modeled as a subtask in each task. In Figure 24, the shared critical section is subtask C. Task A consists of subtasks A_1, A_2, C, and D; task B consists of subtasks B_1, C, and D.

34. Priorities are fixed and predefined by the assembly developer; priorities are **not** assigned using rate monotonic assignment.

35. Based on Figure 23, one can argue that tasks do not have priority. Indeed, when we say that a task T is running at priority X, we implicitly mean that the subtask of T that is currently running has priority X. Also, when we say task T1 preempts task T2, it is the current subtask of T1 that is preempting the current subtask of T2.

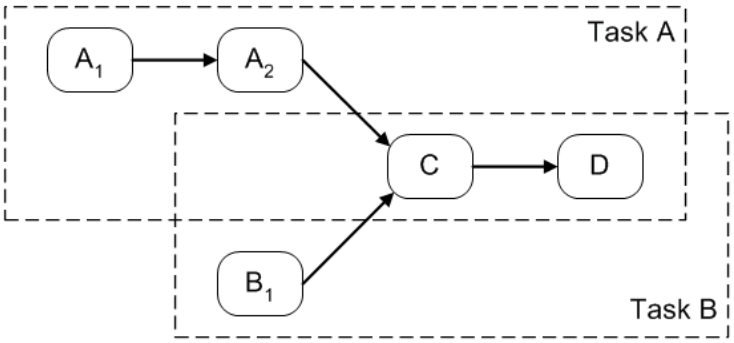

Figure 24: Blocking Example

The priority of critical section C must be assigned according to the priority ceiling protocol.[36] That is, its priority has to be as high as the highest priority of all the subtasks immediately preceding the critical section. For example, if the priority of A_2 is 2 and the priority of B_1 is 4, the priority of C has to be at least 5. In Section A.3, we will see how interpretation handles priorities.

A.2 λ_{ABA} Predictor

As explained in Chapter 5, the initial λ_* property theories were purely analytic. When we wanted to know not only how the latency of a task would be affected by blocking, but also when blocking would occur, the amount of information in the model made the formulation of a purely analysis model quite awkward. Thus, we developed a simulator that uses the tasks, subtasks and their properties to simulate the execution of one hyper-period, keeping track of the state of all the tasks while advancing time.

36. The use of a priority ceiling has several virtues, including the blocked-at-most-once property of assemblies that adhere to this restriction. Nonetheless, there are alternatives; for example, allowing priority inheritance if it is supported by the runtime environment.

Some analytic computations are used by the simulator, thus making λ_{ABA} a hybrid analytic-simulation theory.[37] For example, the end of the hyper-period is computed as

$$HPEnd = \max_{i=1..N}(T_i.offset) + LCM_{i=1..N}(T_i.period)$$

Also, the arrival time of the next task at or after time *t* for task T_i is computed as

$$a_i(t) = \left\lceil \frac{t - T_i.offset}{T_i.period} \right\rceil (T_i.period) + T_i.offset$$

The simulation is event driven (commonly referred to as discrete-event simulation) and controlled by a main loop that advances the time from one scheduling event to the next. A scheduling event is a point in time when a task must be stopped, started, or resumed.

Figure 25 shows the pseudocode of the `predict` function. It takes as input a vector of tasks; each task has a period, an offset, and an ordered list of subtasks. Tasks have also two dynamic properties that are relevant when we run the prediction: state and current subtask. The state indicates whether the task is ready to run, running, or sleeping. In Figure 22 for example, task T2 is ready from time 0 to 1, running from 1 to 3, sleeping from 3 to 5, running from 5 to 6, ready from 6 to 7, running from 7 to 8, sleeping from 8 to 10, and so forth. The current subtask points to the subtask of that task that will execute if the task has the CPU.

Each subtask has priority and execution time. The `predict` function simulates the execution of the tasks through a loop that advances time from instant zero to the end of the hyper-period. In each iteration of the loop, we inspect the state of all tasks to determine the next scheduling event for each task (see Figure 26 for the pseudocode of `getNextEvent`). Then, based on time, priority, and type, one event is selected to be handled next. Tasks are told to advance their internal clock to the point in time when the selected event starts, and to update their internal states accordingly (see Figure 27 for the pseudocode of `advanceClock`). Upon an event, each task knows what it has to do next: start, run, or stop. At any point, only one task has the CPU; when this task advances its internal clock, it assumes it has executed for that amount of time.

37. Models can be analytic, simulation based, or a hybrid of both. An analysis model is based on mathematical relations among variables and intended to be solved mathematically. A simulation model is a computer program that reproduces, to some level of detail, the behavior of a real system. Simulation models are generally used instead of analysis models when it is impossible, or very difficult, to create an analysis model. λ_{ABA} is a hybrid analytic-simulation model.

```
predict(tasks) {
  hyperPeriodEnd = lcmOfPeriods(tasks) + maxPeriod(tasks)
  // put all tasks in the sleeping queue
  for (int i = 0; i < tasks.length; ) {
    tasks[i].state = SLEEPING
    tasks[i].currentSubtask = tasks[i].subtasks[0]
  }
  t = 0      // time in the execution timeline
  taskWithCPU = null
  while (t < hyperPeriodEnd) {
    // Each task can be: ready to run, running, or sleeping (has
    // executed all subtasks and is waiting for the next period).

    // Check what would be the next event of each subtask
    Vector events
    for (int i = 0; i < tasks.length; ) {
      events[i] = getNextEvent(tasks[i], t)
    }
    sort events vector by timeItHappens and priority (in this order)
    // The event in events[0] is the next to occur (ordering by time and
    // priority). It will indeed be the next if:
    //   - it's START or STOP
    //   - it's RUN and has higher priority than the priority of the subtask
    //     currently running. If it has lower priority, we check the
    //     following events to see if any of them is able to preempt the
    //     currently running subtask.
    j = 0
    while (events[j].type == RUN &&
           events[j].priority < taskWithCpu.currentSubtask.priority) {
      j++
    }
    // Next event to happen is events[j] and we have to "advance the clock"
    // accordingly
    delta = events[j].timeItHappens - t
    // For each task, we check what happens when we advance the clock
    for (int i = 0; i < tasks.length; ) {
      advanceClock(tasks[i], t, delta, taskWithCpu)
    }
    t = t + delta
    if (events[j] == STOP) {
      taskWithCpu = NULL
    } else if (events[j] == RUN) {
      taskWithCpu = events[j].task
    }
  }
}
```

Figure 25: Prediction Pseudocode

```
Event getNextEvent(task, currentTime) {
  if (task.state == SLEEPING) {
    // task is sleeping: next thing is to start when the next period arrives
    nextEvent.type = START
    nextEvent.priority = priority of the 1st subtask
    nextEvent.timeItHappens = getNextPeriod(task, currentTime)
  } else if (task.state == READY) {
    // task is ready to run: next thing is to run as soon as now
    nextEvent.type = RUN
    nextEvent.priority = task.currentSubtask.priority
    nextEvent.timeItHappens = currenTime   // task is ready to run now
  } else if (task.state == RUNNING) {
    // task is running: next thing is to stop when the current subtask ends
    nextEvent.type = STOP
    nextEvent.priority = task.currentSubtask.priority
    nextEvent.timeItHappens = currentTime +
                              amount of time to the end of current subtask
  }
  return nextEvent
}


Time getNextPeriod(task, currentTime) {
  nextPeriod = task.offset +
    ceiling((currentTime - task.offset) / task.period) * task.period
  return nextPeriod
}
```

Figure 26: Pseudocode of getNextEvent and getNextPeriod

Figure 28 is a statechart diagram using UML notation that represents a task as a composite state. There are two concurrent substates, one that tells if the task is ready to run, running, or sleeping, and the other that indicates whether the task has the CPU.

When `predict` ends, we have recorded the exact moment when each subtask started and finished, and hence we know how long it took to run each job of each task. That would give us the average latency of the task, accounting for the execution times and delays due to blocking and preemption. However, in reality, the execution time of a subtask is not an exact number as we have used it so far. It comes from the execution time of a previously measured certified component and consists of a mean value and respective standard deviation. Therefore, in order to gain statistical certainty that average job latency is estimated correctly, we should run the prediction several times, each time picking a value for the subtask execution time that is

bounded by the respective standard deviation. Thus, the definite prediction comes from the average of the results of many executions of `predict`, each execution using random execution times for subtasks. Such procedure follows the Monte Carlo simulation method, as illustrated in the pseudocode in Figure 29. The inputs to the `montecarlo` function are a vector of tasks and the number of iterations. In each iteration, the function creates a clone of each task and their subtasks, with the single difference that the execution time of each subtask is reassigned to a random value following a normal distribution.

To this point, we have seen how the property theory handles tasks and subtasks in a hybrid analytic-simulation model to predict average job latency. What we have not seen so far is how to obtain tasks and subtasks from a Pin assembly. That is the subject of the next section.

```
advanceClock(task, currentTime, delta, taskWithCPU) {
  if (task.state == SLEEPING) {
    nextPeriod = getNextPeriod(task, currentTime)
    if (currentTime + delta == nextPeriod) {
      task.state = READY
    }
  } else {
    if (task.state == READY && task == taskWithCPU) {
      task.state = RUNNING
    }
    if (task.state == RUNNING
      if (task == taskWithCPU) {
        if (currentTime + delta == time when current subtask is complete) {
          if (task.currentSubtask == task.lastSubtask) {
            // finished task
            task.state = SLEEPING
            task.currentSubtask = task.subTasks[0]
            record execution time for the job just completed
          } else {
            // There are more subtasks to run within this task. The state
            // changes to READY and when the next subtask can be scheduled
            // to run the state we'll change to RUNNING again.
            task.currentSubtask = task.nextSubtask
            task.state = READY
          }
        }
      } else {
        // Task was running but lost CPU to a higher-priority task
        task.state = READY
      }
    }
  }
}
```

Figure 27: Pseudocode of advanceClock

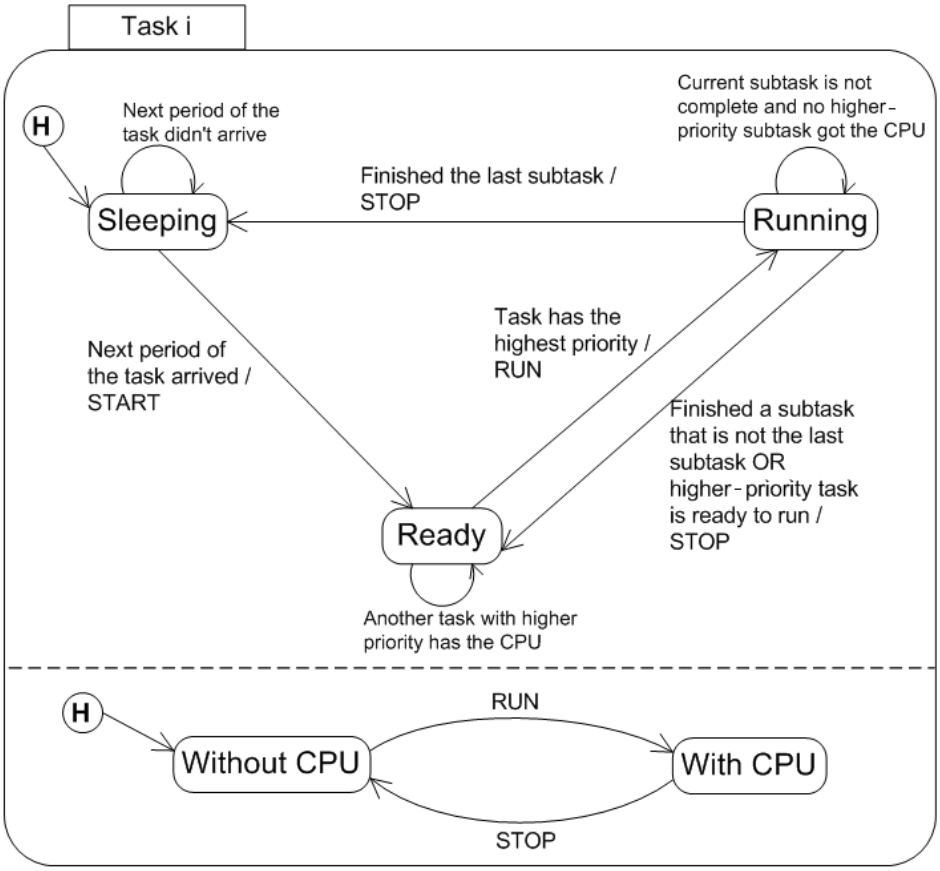

Figure 28: Task Statechart Diagram

```
montecarlo(tasks, numberOfIterations) {
  for (sample = 1; sample <= numberOfIterations; sample++) {
    Vector simulatedTasks
    for (int i = 0; i < tasks.length; ) {
      //Create a clone of each task
      simulatedTasks[i] = tasks[i]
      for (int j = 0; j < tasks[i].subtasks.length; ) {
        //The clone has the same subtasks of the cloned task
        simulatedTasks[i].subtask[j] = tasks[i].subtask[j]
        //The only difference is that the execution time of a cloned subtask
        //is random. The random value is calculated based on the average
        //execution time and stddev of the corresponding component.
        simulatedTasks[i].subtask[j].executionTime =
              random(tasks[i].subtask[j].executionTime,
              tasks[i].subtask[j].stddev)
      }
    }
    predict(simulatedTasks)
    store results of prediction
  }
}
```

Figure 29: Pseudocode of montecarlo

A.3 Pin-λ_{ABA} Interpretation

The property theory models every assembly as a set of periodic tasks. Each task comprises a sequence of subtasks, which execute at specified priority levels. The goal of Pin-λ_{ABA} interpretation is to translate a well-formed assembly into a group of tasks, each consisting of a linear sequence of subtasks. The tasks can then be analyzed by the λ_{ABA} property theory, using the algorithms described in Section A.2.

A.3.1 Clock Components and Task Period

The concept of task and subtask does not exist in Pin. Given that a task in the analysis model performs a computation periodically, it is intuitive to associate clock components with tasks. A clock component is provided by an environment; it has no sink (i.e., it doesn't execute anything) and has one asynchronous source pin that is activated periodically.

In λ_{ABA}, each task corresponds to exactly one clock component and all components that get called when that clock is activated.

A.3.2 Restrictions on Reactions

The Pin assembly shown in Figure 30 has a clock component with a period of 60 milliseconds, and two subtasks, C_1 and C_2. Period is a property of clock components and, in the SAS prototypes, this and other properties were encoded directly in an XML description of the assembly.

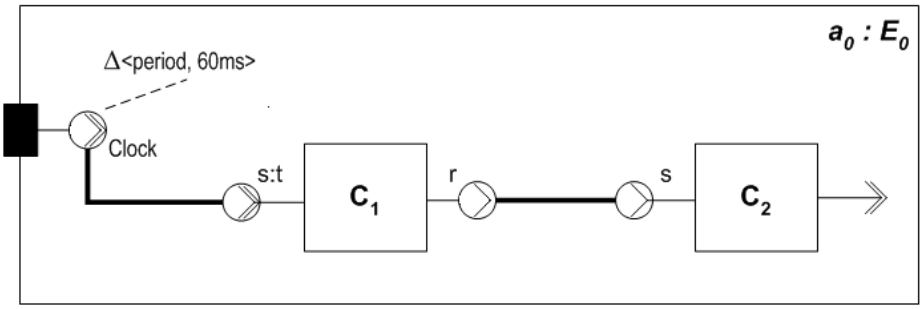

Figure 30: Simple Constructive Assembly

We impose restrictions on the form that reactions take within components. In our interpretation, sink pins will be mapped to subtasks. If we allow components to have the following code structure (as reflected in the reaction specification of components), we would generate a subtask timeline along the lines of the one shown in Figure 31.

```
C1.s() {
   Do some computation
   Activate pin C1.r
   Do some more computation
}
```

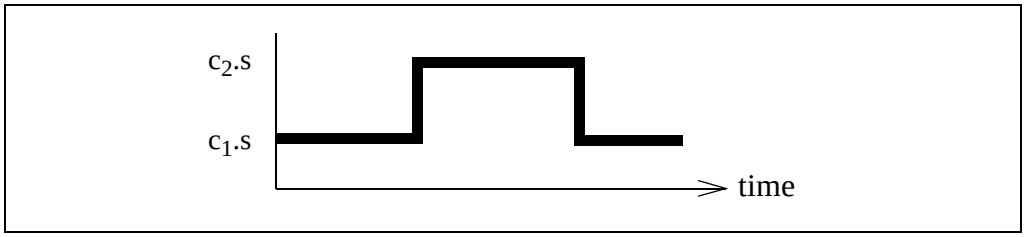

Figure 31: Timeline for a Sink Pin with Interactions in the Middle of Its Execution

This would, however, violate our analysis assumption that a subtask has to finish before its successor can start. To avoid this situation, we introduce a constructive constraint requiring that reactions occur only at the end of a sink's execution. With this constraint, the code for the previous example follows this pattern:

```
C1.s() {
   Do all computation
   Activate pin C1.r
}
```

Its execution timeline is shown in Figure 32.

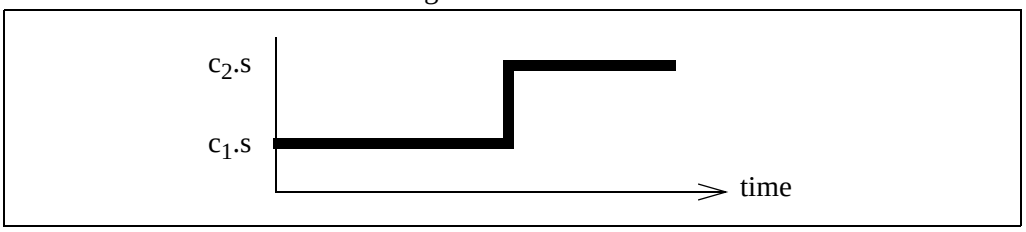

Figure 32: Timeline for a Sink Pin with Interactions at the End of Its Execution

The first step of interpretation is to identify clock components. In Figure 30, we have one clock and hence one task. Each activation of a component corresponds to a subtask, so we identify subtasks with the name of the sink pin that is called (e.g., c_1.s is the subtask that is executed when the homonymous sink pin is activated). Thus, the result of interpretation for the assembly in Figure 30 is <c_1.s, c_2.s>. We use the notation <a, b> to represent a sequence that is interpreted as *a executes before b.*

A.3.3 Interpreting Synchronous Interactions

The assembly Figure 30 has a linear topology, making it straightforward to get the corresponding sequence of subtasks. However, Pin allows one source pin to be connected to multiple sink pins and, within a component, one sink pin to react to multiple source pins. Thus, it is possible—and perhaps the most common situation—to have assemblies with tree-like topologies. The challenge then is to linearize that tree into a sequence of subtasks that can be analyzed by the property theory.

Figure 33 shows an assembly with a more interesting topology. Again, there is one clock and hence one task. The reaction on c_1.s, denoted R_s in Figure 33, defines the order of the interactions associated with c_1.s. Based on R_s, we can determine that source pin $c_1.r_1$ is activated before $c_1.r_2$ and therefore, that $c_1.r_1 \rightsquigarrow c_2.s$ occurs before $c_1.r_2 \rightsquigarrow c_3.s_2$. What is not clear though is whether $c_1.r_1$ interacts first with c_2.s or $c_3.s_1$. The order of interactions can be specified

using the appropriate annotation, but hereafter we consider the order of interaction in diagrams to be from top to bottom. So, in Figure 33, $c_1.r_1 \rightsquigarrow c_2.s$ occurs before $c_1.r_1 \rightsquigarrow c_3.s_1$.

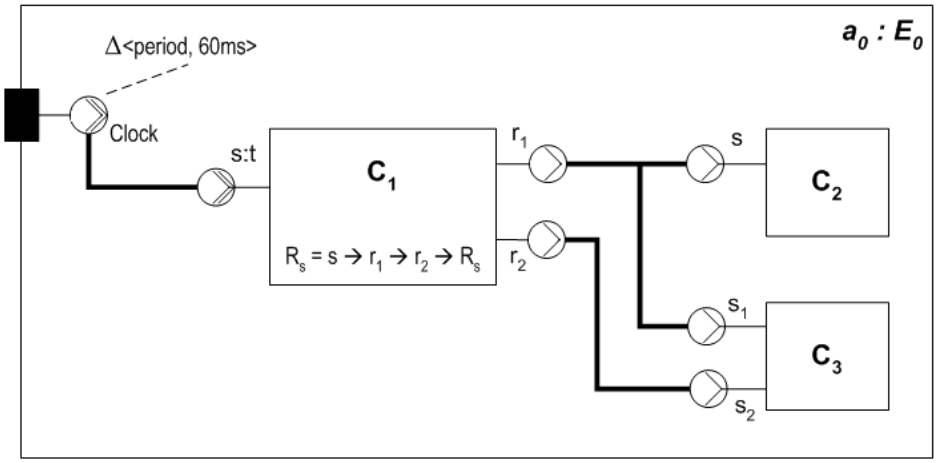

Figure 33: Assembly with Synchronous Pins and Non-Linear Topology

The assembly in Figure 33 has only one thread, which is associated with $c_1.s$. Sinks $c_2.s$, $c_3.s_1$, and $c_3.s_2$ execute in that same thread, because they are synchronous reentrant pins, which are always non-threaded. If $c_2.s$, $c_3.s_1$, and $c_3.s_2$ were synchronous mutexed pins, they could possibly be threaded (i.e., execute in a separate thread). However, in the examples of this appendix, as well as in all assemblies used in the experiments documented by this report, synchronous sink pins are non-threaded. Nonetheless, λ_{ABA} and the runtime environment support threaded synchronous pins, and they shall be used in the future. Figure 34 depicts all existing types of sink pins, highlighting the ones used in the experiments conducted with λ_{ABA} so far.

The fact that only one thread is created for this assembly makes it easy to figure out the order in which sinks and, consequently, subtasks will execute. The corresponding analysis task for this assembly is $<c_1.s, c_2.s, c_3.s_1, c_3.s_2>$. As we will see soon, when multiple threads exist, preemption makes interpretation more difficult.

For assemblies that use a single thread, the sequence of subtasks can be obtained algorithmically by doing a pre-order traversal of the tree. Another algorithm starts from the leaves and works towards the root. Every time we get to a node where two branches are synchronously connected, we prepend the subsequence for the branch that executes first to the subsequence for the branch that executes second. For example, from the C_1 branch, we obtain the subsequences $x = <c_2.s, c_3.s_1>$ and $y = <c_3.s_2>$. When we get to $c_1.s$, subsequence *x* is prepended to subsequence *y*, and then $c_1.s$ is prepended to the result, obtaining, as before, $<c_1.s, c_2.s, c_3.s_1, c_3.s_2>$.

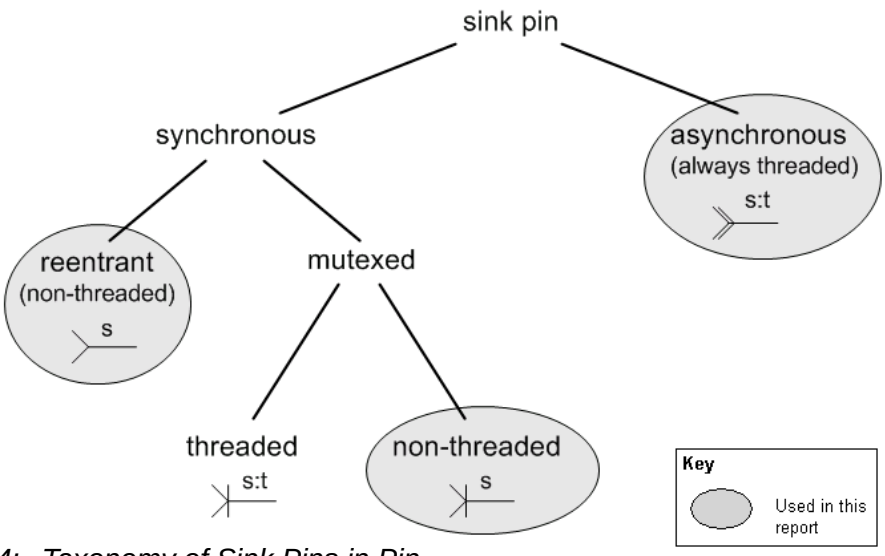

Figure 34: Taxonomy of Sink Pins in Pin

A.3.4 Interpreting Asynchronous Interactions

When we introduce more asynchronous connections to the assembly, we also introduce more threads, because asynchronous sink pins have associated threads. The problem that arises is that now we have many concurrent elements in the construction model (i.e., threads), and they must somehow be mapped to concurrent elements in the analysis model. But in the analysis model, the unit of concurrency is task and we do not have one task per thread: we have one task per clock component. So, to solve this problem, we eliminate real concurrency within the constructive assembly. We achieve this by assigning unique priorities to asynchronous sink pins. Figure 35 shows an example in which concurrency is handled via priorities.

Sink pins c_1.s and c_3.s have threads; therefore, the top and bottom paths of execution could, in principle, run concurrently. Priorities allow us to interpret this subassembly into a linear sequence of subtasks.

The asynchronous calls to c_1.s and c_3.s are done before either of the two sink pins has a chance to execute, meaning that both C_1 and C_3 are ready to run right after the clock call. Pin c_3.s will execute first, because it has a higher priority than c_1.s. Then, even though c_1.s is ready to execute, c_4.s will execute because it has a higher priority. When c_4.s finishes, it activates c_5.s, but c_1.s (which has a higher priority) goes first. Then c_2.s executes and lastly c_5.s does. The resulting analysis task is <c_3.s, c_4.s, c_1.s, c_2.s, c_5.s>. If we try to interpret this assembly by doing a prepend of branches as we did in Section A.3.3, we will obtain the incorrect sequence <c_1.s, c_2.s, c_3.s, c_4.s, c_5.s>.

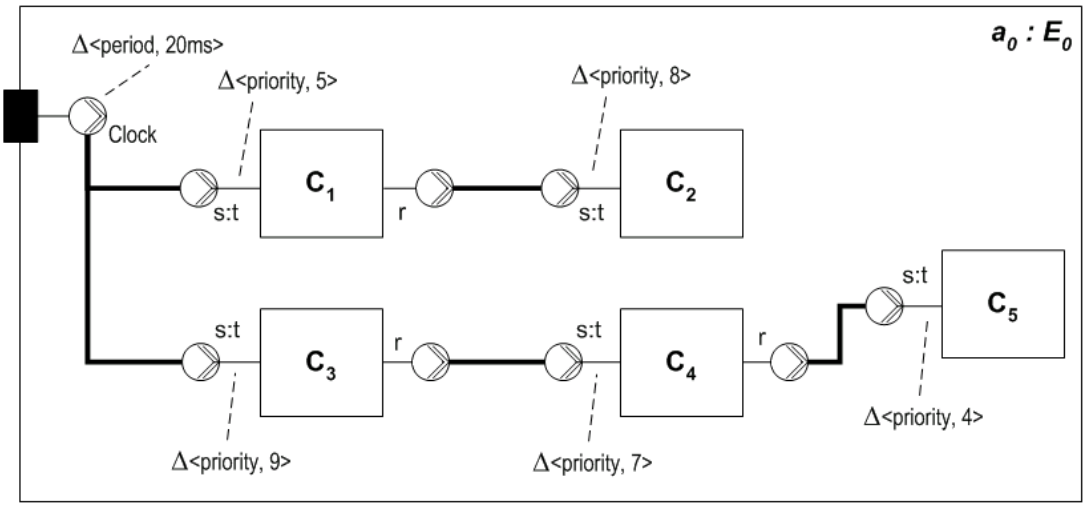

Figure 35: Constructive Assembly with Asynchronous Connections

For assemblies like the one in Figure 35, the right sequence of subtasks can be obtained algorithmically. The sequence of subtasks for the two concurrent branches are x = <c_1.s, c_2.s> and y = <c_3.s, c_4.s, c_5.s>.[38] When two execution paths are initiated asynchronously, we *merge* the two sequences. This merge operation consists of creating a new sequence of subtasks by appending to it the highest priority subtask from the head of the sequences being merged and repeating this step until both sequences have been consumed. Table 5 shows the state of the sequences during the merge operation in the example.

Table 5: Merge Operation Example

Clock0.subtasks	**x**	**y**
<>	<c_1.s, c_2.s>	<c_3.s, c_4.s, c_5.s>
<c_3.s>	<c_1.s, c_2.s>	<c_4.s, c_5.s>
<c_3.s, c_4.s>	<c_1.s, c_2.s>	<c_5.s>
<c_3.s, c_4.s, c_1.s>	<c_2.s>	<c_5.s>
<c_3.s, c_4.s, c_1.s, c_2.s>	<>	<c_5.s>
<c_3.s, c_4.s, c_1.s, c_2.s, c_5.s>	<>	< >

The merge operation requires only that the two subtasks in the head of the sequences do not have the same priority. If they did, the algorithm would not be able to determine which one would execute first. It is important to note that, as long as the previous condition is met, differ-

38. In Figure 35, you can easily see what the sequences are because both branches are very simple, with no further branches. In general, to compute the sequences algorithmically, it is necessary to apply the same operation starting from the leaves.

ent sink pins in one task could share the same priority level. This property will be helpful when we try to interpret synchronous sink pins, as we will see in the next section.

A.3.5 Synchronous Sink Pins and Priorities

If a synchronous sink pin is reentrant (non-mutexed), in λ_{ABA}, it will run at the caller's priority. Otherwise, if a reentrant synchronous sink pin were assigned a priority, two separate tasks could call that sink pin concurrently, and there would be two threads running at the same priority—a situation we should avoid.

If a synchronous sink pin is mutexed and non-threaded, in λ_{ABA}, it will run at the maximum priority. A mutexed sink pin is a "shared resource." So, we need to guarantee the blocked-at-most-once property. One alternative would be to check the priority of all callers of a mutexed synchronous pin and use the highest value to define that pin's priority, like in the priority ceiling protocol described in Section 5.4.4. A simpler approach, which gives satisfactory practical results, was chosen—for any mutexed synchronous sink pin, the priority is raised to the *maximum* value, which we call "super ceiling."

Synchronous sink pins that are mutexed and threaded were not used in the experiments discussed in this report.

Figure 36 shows the same assembly represented in Figure 35, where the sink pins of C_2, C_4, and C_5 were changed from asynchronous to synchronous. We use the same merge operation described earlier to define the sequence of subtasks. But, because synchronous sink pins use the caller's or the maximum priority, the resulting sequence of subtasks is $<c_3.s, c_4.s, c_5.s, c_1.s, c_2.s>$. In Figure 30, all the synchronous sink pins are mutexed and have maximum priority. However, in this case, the merge operation would produce the same sequence if any of the synchronous pins were reentrant.

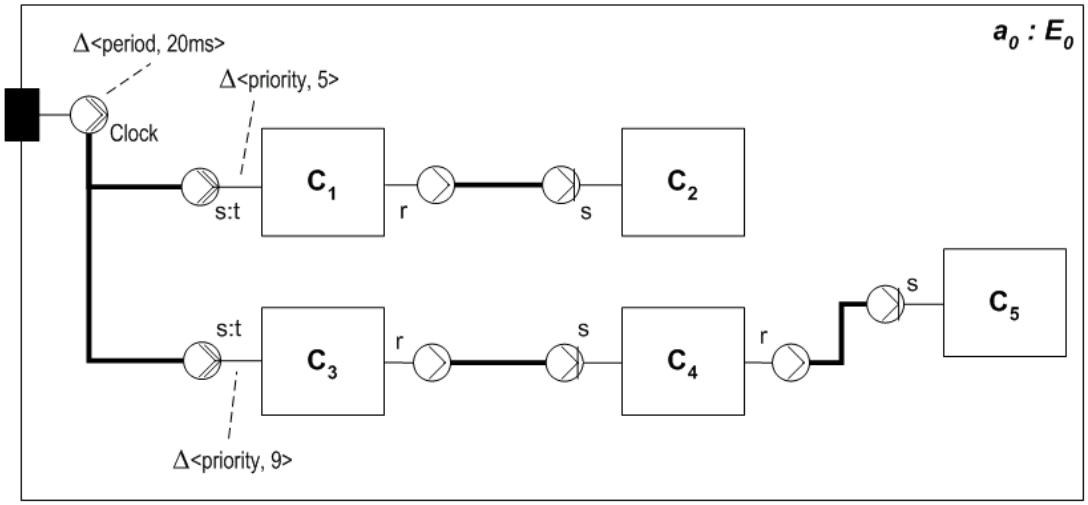

Figure 36: Synchronous Connections Following Asynchronous Connections

A.3.6 Asynchronous Connections Following Synchronous Connections

It may seem that applying the prepend and merge operations will produce the correct interpretation of any assembly with mixed synchronous and asynchronous connections. However, this strategy will not render the right analytic interpretation if synchronous connections are followed (from root to leaves) by asynchronous connections, as is the case in the assembly shown in Figure 37.

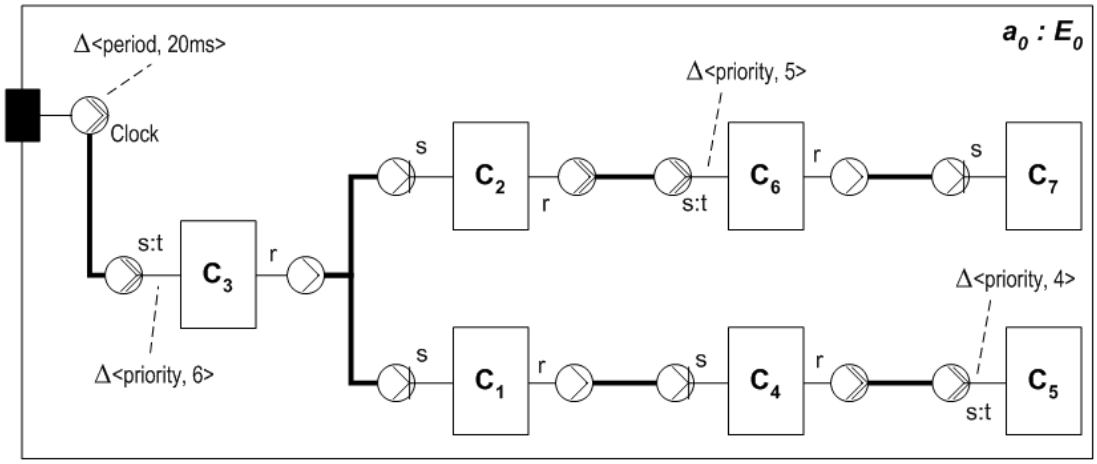

Figure 37: Asynchronous Connections Following Synchronous Connections

The correct interpretation of this subassembly is <c_3.s, c_2.s, c_1.s, c_4.s, c_6.s, c_7.s, c_5.s>. If we work from the leaves toward the root, we get the sequences x = <c_2.s, c_6.s, c_7.s> and y = <c_1.s, c_4.s, c_5.s>. If we apply the prepend operation as suggested for the earlier interpretations (see

Section A.3.3), we would get a different sequence, <c_3.s, c_2.s, c_6.s, c_7.s, c_1.s, c_4.s, c_5.s>, which would be correct if the sink pins of C_5 and C_6 were synchronous and not asynchronous. Furthermore, we cannot apply the merge operation as described in Section A.3.4 because we should only merge execution paths that are initiated asynchronously, and we have synchronous connections after C_3.

Asynchronous interactions activate new threads, allowing more than one component to be ready simultaneously. The asynchronous connections to C_6 and C_5 in Figure 37 are preceded by synchronous connections. To handle this situation, we use a modified version of the merge operation. The algorithm is modified as follows:

- When we obtain a sequence that corresponds to a branch, if the subtask is activated asynchronously, an overbar (¯) is used over the name of the asynchronous sink pin. Thus, in Figure 37, the two sequences after C_3 are x = <c_2.s, $c_6.\bar{s}$, c_7.s> and y = <c_1.s, c_4.s, $c_5.\bar{s}$>.
- The operation behaves as a prepend as long as no asynchronously activated subtasks are in the head of either sequence. When one is, the operation behaves as a merge from that point on.

Table 6 shows the steps to combine the two sequences in Figure 37. Note that the synchronous sink pins of C_1, C_2, C_4, and C_7 are mutexed and hence run at maximum priority (see Section A.3.5). Thus, for example, in the second step of the algorithm, when $c_6.\bar{s}$ and c_1.s are the heads of sequence x and y respectively, c_1.s is chosen because it has maximum priority, whereas the priority of $c_6.\bar{s}$ is 5.

Table 6: Merging the Sequence in Figure 37

Clock0.subsequence	x	y	Mode
<>	<c_2.s, $c_6.\bar{s}$, c_7.s>	<c_1.s, c_4.s, $c_5.\bar{s}$>	prepend
<c_2.s>	<$c_6.\bar{s}$, c_7.s>	<c_1.s, c_4.s, $c_5.\bar{s}$>	merge
<c_2.s, c_1.s>	<$c_6.\bar{s}$, c_7.s>	<c_4.s, $c_5.\bar{s}$>	merge
<c_2.s, c_1.s, c_4.s>	<$c_6.\bar{s}$, c_7.s>	<$c_5.\bar{s}$>	merge
<c_2.s, c_1.s, c_4.s, $c_6.\bar{s}$>	< c_7.s>	<$c_5.\bar{s}$>	merge
<c_2.s, c_1.s, c_4.s, $c_6.\bar{s}$, c_7.s>	<>	<$c_5.\bar{s}$>	merge
<c_2.s, c_1.s, c_4.s, $c_6.\bar{s}$, c_7.s, c_5.s>	<>	<>	merge

The results would be different if the synchronous sink pins were reentrant instead of mutexed. Figure 38 shows the same assembly using reentrant pins. We also changed the priority of c_3.s from 6 to 3. The interpretation for this assembly is <c_3.s, c_2.s, c_6.s, c_7.s, c_1.s, c_4.s, c_5.s>, and the iterative merge operation is described in Table 7. Because reentrant sink pins run at the caller's priority, c_1.s, c_2.s, and c_4.s use priority 3 (the priority of c_3.s), and c_7.s uses priority 5

(the priority of $c_6.s$). Therefore, when $c_6.\bar{s}$ and $c_1.s$ are the heads of the sequences, $c_6.\bar{s}$ is chosen.

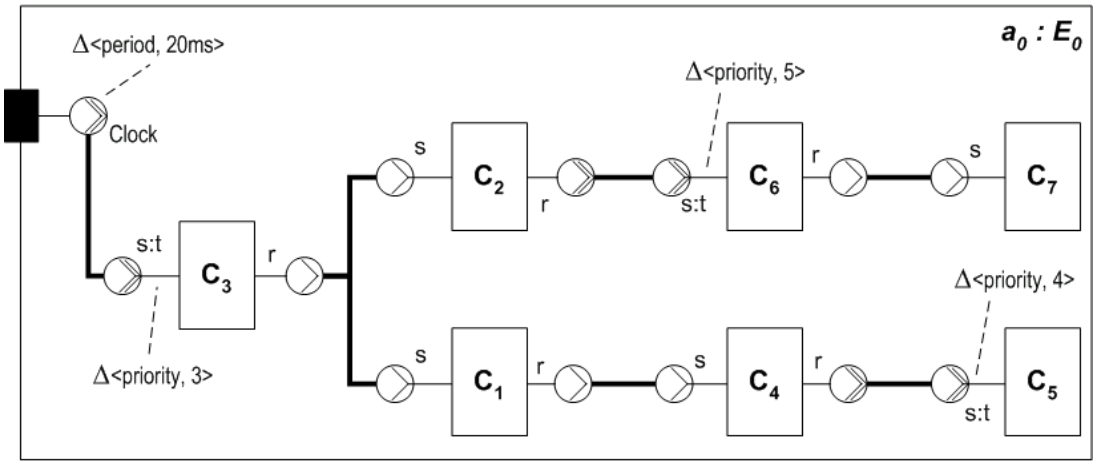

Figure 38: Asynchronous Connections and Reentrant Synchronous Connections

Table 7: Merging the Sequences in Figure 38

Clock0.subsequence	x	y	Mode
<>	<$c_2.s$, $c_6.\bar{s}$, $c_7.s$>	<$c_1.s$, $c_4.s$, $c_5.\bar{s}$>	prepend
<$c_2.s$>	<$c_6.\bar{s}$, $c_7.s$>	<$c_1.s$, $c_4.s$, $c_5.\bar{s}$>	merge
<$c_2.s$, $c_6.\bar{s}$>	<$c_7.s$>	<$c_1.s$, $c_4.s$, $c_5.\bar{s}$>	merge
<$c_2.s$, $c_6.\bar{s}$, $c_7.s$>	<>	<$c_1.s$, $c_4.s$, $c_5.\bar{s}$>	merge
<$c_2.s$, $c_6.\bar{s}$, $c_7.s$, $c_1.s$>	<>	<$c_4.s$, $c_5.\bar{s}$>	merge
<$c_2.s$, $c_6.\bar{s}$, $c_7.s$, $c_1.s$, $c_4.s$, >	<>	<$c_5.\bar{s}$>	merge
<$c_2.s$, $c_6.\bar{s}$, $c_7.s$, $c_1.s$, $c_4.s$, $c_5.s$>	<>	<>	merge

A.3.7 Multiple Connections on One Source Pin

When two or more sink pins are connected to the same source, at runtime they are activated in the order specified in the assembly—in our examples, this order is assumed to be from top to bottom. If one of the activated sinks is asynchronous or leads to the activation of an asynchronous sink having a priority greater than that of the activating source, the source will be preempted, possibly before it finishes activating the remaining sinks. However, the merge operation described in previous sections assumes that these multiple connections will be activated one after the other without interruption, that is, without the activating source itself being preempted. For that reason, the actual execution sequence of components may differ from the sequence of subtasks interpreted by the merge operation.

Although this problem may arise with both multiple asynchronous and multiple synchronous connections, it is easier to see its effect when a source pin has multiple asynchronous connections. Figure 39 depicts an assembly with this problem. The interpretation algorithm described so far renders the incorrect sequence <c_3.s, c_1.s, c_4.s, c_2.s, c_6.s, c_5.s>, when the actual behavior is <c_3.s, c_2.s, c_6.s, c_1.s, c_4.s, c_5.s>. The difference stems from the fact that the merge operation assumes that both c2.s and c1.s are activated at the same time, before either has a chance to start executing. The actual behavior differs because sink c_2.s preempts the thread on the reaction (source) c_3.r before it activates pin c_1.s.

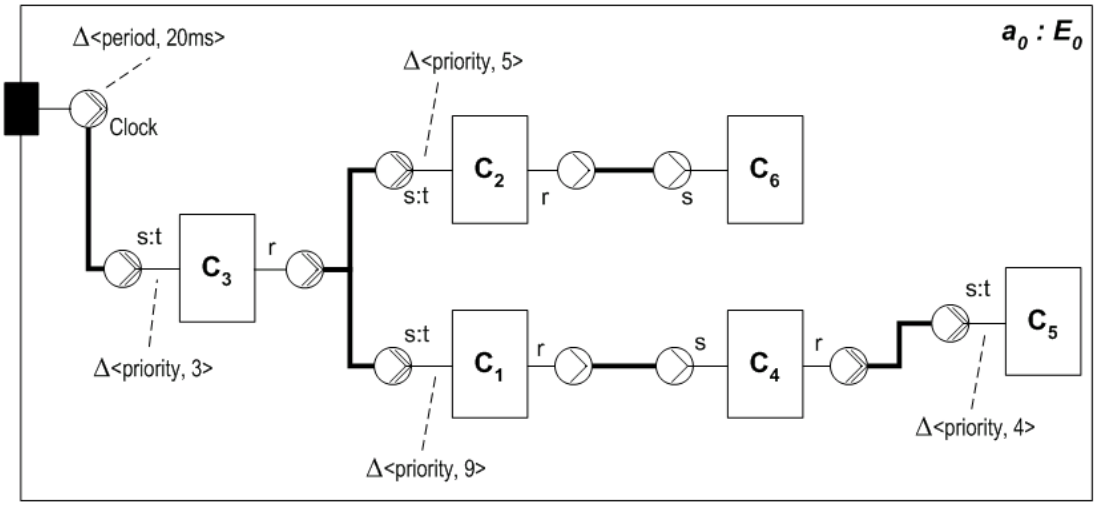

Figure 39: Multiple Synchronous Connections Problem

We solve the problem by applying a rewrite rule on the constructive assembly before interpreting it. This preprocessing consists of rewriting all asynchronous connections by interposing a *proxy component* between each source and the asynchronous connector. The proxy has a reentrant (and therefore unthreaded) sink that is connected to the original caller component, but using a synchronous instead of the original asynchronous source. The proxy also has a source pin that is connected to the asynchronous sink of the original called component. The proxy component has no execution time, and because it is unthreaded, it executes at the same priority as its caller. Figure 40 shows this rewrite rule applied to the assembly in Figure 39.

After preprocessing, the interpretation renders <c_3.s, pc_2.ps, c_2.s, c_6.s, pc_1.ps, c_1.s, c_4.s, c_5.s>. The proxy components pc_n have no execution time, so they can be removed from the sequence. Finally, we obtain the correct execution sequence <c_3.s, c_2.s, c_6.s, c_1.s, c_4.s, c_5.s>. The problem just described would not occur if C_3 had a priority greater than the priority of C_1 and C_2 in Figure 39. In this case, during the actual execution, the source (c_3.s) would activate the two sinks (c_2.s and c_1.s) without being preempted, and the actual sequence would be <c_3.s,

c_1.s, c_4.s, c_2.s, c_6.s, c_5.s>. Interpretation using the merge operation would produce the same sequence without requiring proxy components.

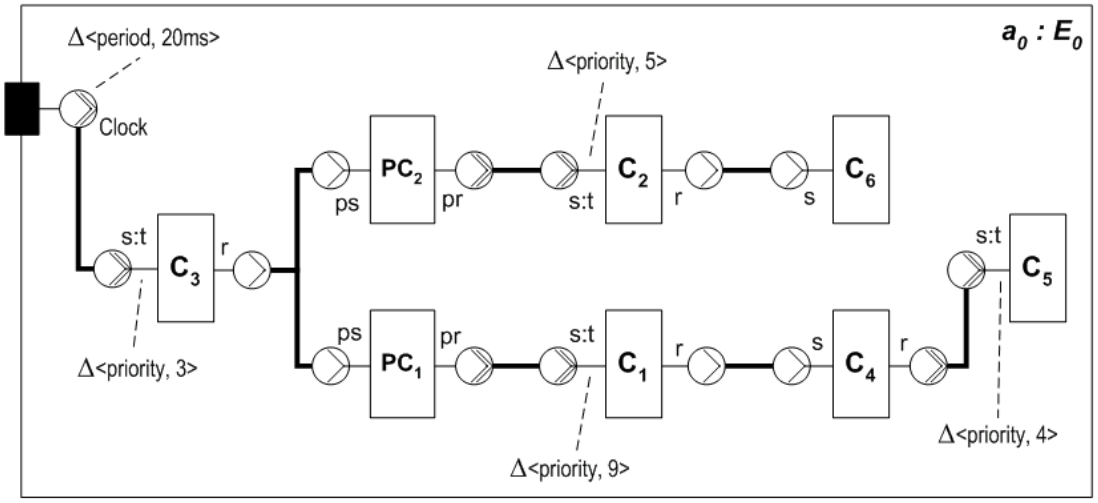

Figure 40: Multiple Asynchronous Connections Rewrite Rule Using Proxy Components

Therefore, proxies are not necessary when the caller component has a priority higher than the priority of the called components. Nonetheless, the addition of proxies does not affect the correct interpretation in this case and, indeed, proxies are created in λ_{ABA} for *all* asynchronous connections. In Figure 40 for example, if the priority of C_3 were higher, say 20, interpretation would produce <c_3.s, pc_2.ps, pc_1.ps, c_1.s, c_4.s, c_2.s, c_6.s, c_5.s>, which can be factored to <c_3.s, c_1.s, c_4.s, c_2.s, c_6.s, c_5.s>, the same as the actual sequence.

In the case of multiple synchronous connections, an analogous technique is used, and proxy components have a synchronous source.[39]

A.4 Syntax-Directed Interpretation

The discussion so far has been informal. Table 8 summarizes the key elements of Pin and their corresponding elements in λ_{ABA}. To enable automated interpretation, we must complete the formalization. We chose to use syntax-directed translation to implement the interpretation

39. In the interim between the first and second edition of this report, Pin was further restricted to disallow 1:N synchronous interactions.

[Aho 87]. This requires an input grammar for the constructive assembly and a set of semantic reductions that render the interpretation.

Table 8: Mapping Pin to λ_{ABA}

Constructive Framework	**Reasoning Framework**	**Comment**
clock component	task	The execution of all components that occur after a given clock is activated comprise a task.
clock period	task period	
—	job	A job is one complete execution of a task.
clock period minus the period of the clock with the smallest period in the assembly	task offset	For the clock with the smallest period, the corresponding task has offset zero.
assembly	set of concurrent tasks	
environment	—	The environment is abstracted from the property theory.
sink pin of a component	subtask	A sink pin (more accurately, the reaction associated with it) is mapped to a subtask.
• *asynchronous sink pin*: value of the “priority” property; • *reentrant synchronous sink pin*: priority of the caller; • *mutexed synchronous sink pin*: maximum priority	subtask priority	
value of the “execution time” property of a component	subtask execution time	
thread	—	During interpretation, the existence of multiple threads, and mutexed and reentrant pins in the assembly governs preemption and affects the sequence of subtasks, but no element of the reasoning framework represents these elements directly.
mutex	—	
reentrancy	—	

The grammar for constructive assemblies is as follows:

asm → *src* | *asm* & *src*
src → *pin* * *subasm_set* | *pin* + *subasm_set*
pin → *component.pinId*
subasm_set → (*subasm_list*)
subasm_list → *subasm* | *subasm_list* , *subasm*
subasm → *snk* | *snk* – *src_set*
snk → *pin*
src_set → (*src_list*)
src_list → *src* | *src_list* , *src*

where an asterisk (*) denotes asynchronous interaction, a plus sign (+) denotes synchronous interaction, a dash (–) denotes reaction, and a comma (,) separates source pins in a reaction and sink pins in a 1:N interaction. Note that reactions are not expressed in CSP, but rather only in the restricted form of call dependency.

Notes:

- The interpretation assumes that all λ_{ABA} constructive constraints are honored. That is why we make no distinction in this language for mutex sink pins—their blocking effect is addressed by the constructive constraint for the priority ceiling.
- To make the grammar simpler, we have disregarded the fact that the *src* sources in the first production have to be different, because their pins must have an implicit periodic stimulus. This is the case for the clock component.

The syntax-directed definition for obtaining the analytic assembly for a constructive assembly is shown in the following table. Both synthesized and inherited attributes are used. The pseudocode for the types and functions used in the syntax-directed translation are shown in Sections A.4.1 and A.4.2.

To keep the syntax-directed definition presented here as simple as possible, some liberties have been taken. The `src.period` and `pin.priority` attributes are used without first being assigned anywhere. Assigning them would have required adding a declaration construct to the grammar and a symbol table to the syntax-directed definition. Also, the `src.period` attribute should have been synthesized from the `pin.period` attribute, which isn't expressed in the semantic rules either.

Table 9: Syntax-Directed Definition for the Analytic Interpretation

Production	Semantic Rules
asm → src	addtask(src.period, src.subtasks) src.priority = 0
asm → asm & src	addtask(src.period, src.subtasks) src.priority = 0
src → pin * subasm_set	src.subtasks = subasm_set.subtasks subasm_set.priority = src.priority subasm_set.mode = async
src → pin + subasm_set	src.subtasks = subasm_set.subtasks subasm_set.priority = src.priority subasm_set.mode = sync
pin → component . pinId	pin.name = component.lexicalvalue() + pinId.lexicalvalue()
subasm_set → (subasm_list)	subasm_set.subtasks = combine(subasm_list.stseq) subasm_list.priority = subasm_set.priority subasm_list.mode = subasm_set.mode
subasm_list → subasm	subasm_list.stseq = seqadd(null, subasm.subtasks) subasm.priority = subasm_list.priority subasm.mode = subasm_list.mode
subasm_list → subasm_list1 , subasm	subasm_list.stseq = seqadd(subasm_list1.stseq, sub- asm.subtasks) subasm.priority = subasm_list.priority subasm.mode = subasm_list.mode
subasm → snk	snk.epriority = getpriority(subasm.priority, snk) subasm.subtasks = prependsubtask(<>, snk.name, snk.epriority, subasm.mode)
subasm → snk - src_set	snk.epriority = getpriority(subasm.priority, snk) src_set.priority = snk.epriority subasm.subtasks = prependsubtask(src_set.subtasks, snk.name, snk.epriority, subasm.mode)
snk → pin	snk.priority = pin.priority snk.name = pin.name snk.mutex = pin.mutex snk.threaded = pin.threaded
src_set → (src_list)	src_set.subtasks = combine(src_list.stseq) src_list.priority = src_set.priority
src_list → src	src_list.stseq = seqadd(<>, src.subtasks) src.priority = src_list.priority
src_list → src_list1 , src	src_list.stseq = seqadd(src_list1.stseq, src.subtasks) src.priority = src_list.priority

A.4.1 Types

```
// type definitions
t_seq : sequence of subtasks
t_stseq : sequence of sequences of subtasks
t_mode: mode of activation of a sink: sync, async
```

A.4.2 Functions

The functions used in the semantic rules of the syntax-directed translation are shown here in pseudocode. The notation for sequences uses angle brackets. The empty sequence is denoted by `<>`, and `<x,y,z>` is a sequence containing the objects x, y, and z in that order. If x is a sequence, then `<x>` is a sequence of sequences. The binary operator + applied to sequences means concatenation. The function `head(s)` returns the first element of the sequence s, and the function `tail(s)` returns a sequence containing the subsequence of s that follows its first element. For example, if `s=<x,y,z>`, then `first(s)=x`, and `tail(s)=<y,z>`.

```
addtask(int period, t_seq subtasks) {
   adds a periodic task with the specified period and subtasks
   to the analytic assembly
}

t_seq seqadd(t_stseq a, t_seq b) {
   t_seq r
   r = a + <b>
   return r
}

int getPriority(int callerPriority, snk sink) {
   // determine the effective priority of a sink
   if (sink.mutex) {
      return SUPER_CEILING_PRIORITY
   }
   if (sink.threaded) {
      return sink.priority
   }
   return callerPriority
}

t_seq prependsubtask(t_seq a, char name, int priority, t_mode mode) {
   t_subtask st
   st.name = name
   st.priority = priority
   st.mode = mode
   t_seq b = <st> + a
   return b
}

t_seq combine(t_stseq a) {
   t_seq b = <>
   for (each t_seq c it a)
      // combine t and c into b
      b = <>
      while (t.size > 0 && first(t).mode == sync) {
         b = b + <head(t)>
         t = tail(t)
      }
      while (t.size > 0 && c.size > 0) {
         if (head(c).priority > head(t).priority) {
            b = b + <head(c)>
            c = tail(c)
         } else {
            b = b + <head(t)>
            t = tail(t)
         }
}
if (t.size > 0) {
   b = b + t
} else {
   b = b + c
}
   }
   return b
}
```

Appendix B λ_{ABA} Empirical Validation

B.1 Introduction

Validation for the controller PECT consists of empirically quantifying the accuracy and confidence of the predictions based on λ_{ABA}. This quantification is done by statistically comparing predictions with actual measurements of assembly properties and establishing confidence in the results, which, in turn, increase confidence in the analysis model.

This appendix describes in detail the experiment to empirically validate λ_{ABA}.

B.2 Empirical Validation

Sections B.2.1 - B.2.6 correspond to steps 1 - 6 in Figure 15 on page 47.

B.2.1 Define Validation Goal

The goal for the controller PECT is that the latency for a job within the hyper-period can be predicted with an MRE ≤ 0.05 (5%) with a confidence level γ =0.99 (99%). A minimum acceptable p = 0.80 (80%) was established for a pass/fail condition. Latency is defined as the average latency of each job in each period of the assembly hyper-period.

B.2.2 Define Measures

B.2.2.1 Time and Pins

Which time is measured and how is fundamental to this empirical validation, the controller PECT, and λ_{ABA}. Although the specific *measure* for time differs between the property for components (Section B.2.2.2) and assemblies (Section B.2.2.3), the *units* of time are the same, and the same apparatus is used.

The measurement infrastructure put into place is hosted on a Microsoft Windows platform and is conformant to the VenturCOM RTX clock counter, which has a resolution on the order of

< 1µs.[40] Time recorded by the measurement infrastructure using this clock counter is in milliseconds (ms) with eight digits of precision.

A time-stamped event is generated when a pin is entered (activated) or exited (deactivated). When such an event is generated, the measurement infrastructure reads the clock counter's value and records it with the event. Doing so provides two pieces of information—a time-ordered sequence of pin events and a means of determining the time interval between any two events.

B.2.2.2 Component Execution Time

When a pin (source or sink) is entered, it is considered *activated*. Activation occurs immediately after a

- sink pin *enter* event. The component is in receipt of a message or request (stimulus). This is the start of a reaction.
- source pin *enter* event. The component is making a request or sending a message (response). This is the start of an interaction.

When a pin (source or sink) is exited, it is considered *deactivated*. Deactivation occurs immediately after a

- sink pin leave event. The component has satisfied the request. This is the end of a reaction.
- source pin leave event. The component has completed making the request or sending the message. This is the end of an interaction.

The above time-recorded events define measurement points. For example, consider the component shown in Figure 41.

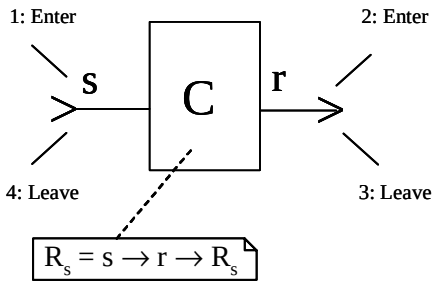

Figure 41: Pin Component Enter and Leave Events

The reaction rule for this simple component could be expanded from that illustrated in Figure 41 to show the specific enter and leave events as $R_s = s \rightarrow r \rightarrow \bar{r} \rightarrow \bar{s} \rightarrow R_s$, where $\bar{x}$ represents

40. RtGetClockTime() reports time in 100 nanosecond units.

the specific leave event for a given sink or source pin. Figure 42 illustrates six time measurements on a component based on enter and leave events for sink and source pins.

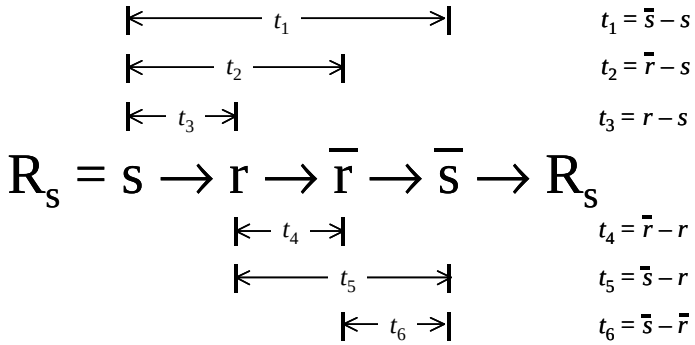

Figure 42: Time Measurements on Component Sink and Source Reactions

When two or more components are assembled, the number of possible measurement points (due to the increase in events generated by additional components) allows additional calculations to be made across component interactions.

For λ_{ABA}, the total execution time of reactions is of interest—specifically, t_1 in Figure 42, where $t_4 = 0$ for all source pins in the reaction (see Figure 43).

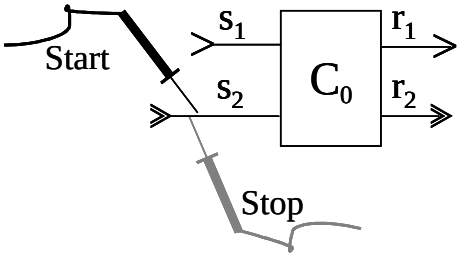

Figure 43: Measuring the Execution Time of a Pin Component

B.2.2.3 Assembly Latency

An interaction between components exists when the source pin of one component is connected to the sink pin of another. Further, interactions between components can continue as a

- linear interaction (as shown in Figure 44)
- multilinear interaction (as shown in Figure 45)
- one-to-many interaction (as shown in Figure 46)
- many-to-one interaction (as shown in Figure 47)
- many-to-many interaction (as shown in Figure 48)[41]

41. Since the writing of the first edition of this report, 1:N synchronous interactions have been disallowed by Pin.

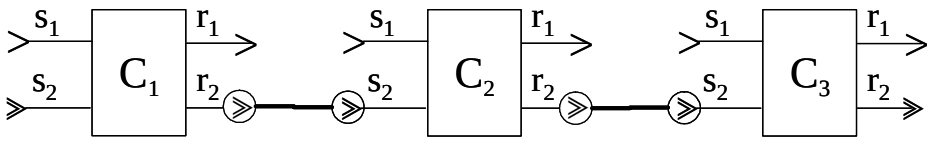

Figure 44: A Simple Linear Interaction

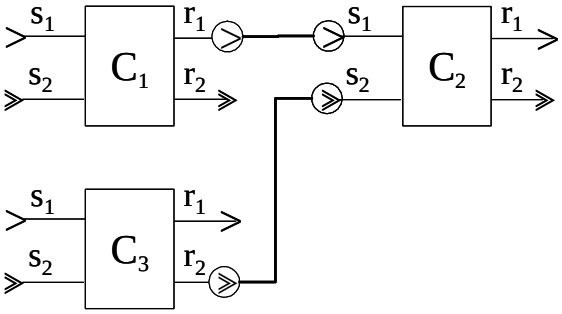

Figure 45: A Simple Multilinear Interaction

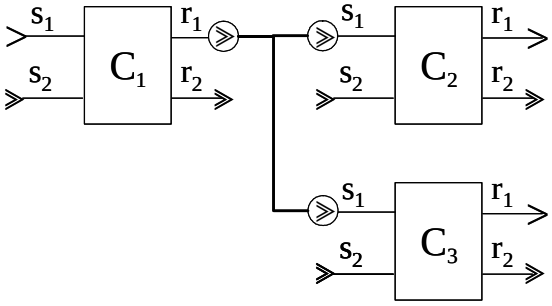

Figure 46: A Simple One-to-Many Interaction

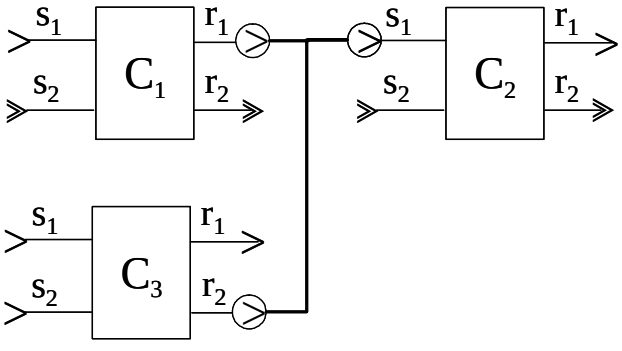

Figure 47: A Simple Many-to-One Interaction

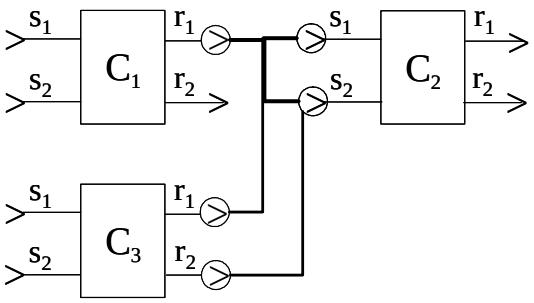

Figure 48: A Simple Many-to-Many Interaction

An assembly is a set of one or more interactions. A task is defined as the set of interactions starting with one component's source pin and terminating at another component's sink (or source) pin. Consider the small assembly depicted in Figure 44. The set of interactions from $c_1.r_2 \rightsquigarrow c_3.s_2$ (i.e., $\{c_1.r_2 \rightsquigarrow c_2.s_2, c_2.r_2 \rightsquigarrow c_3.s_2\}$) is considered a task. A job is the scheduled periodic execution of a task; a task has one or more jobs in a hyper-period. For λ_{ABA}, latency for any job in any period is of interest.

Latency for a job is measured from the time the first source pin is activated to the time the last sink pin is deactivated ($c_3.\bar{s}_2$), as illustrated in Figure 49.

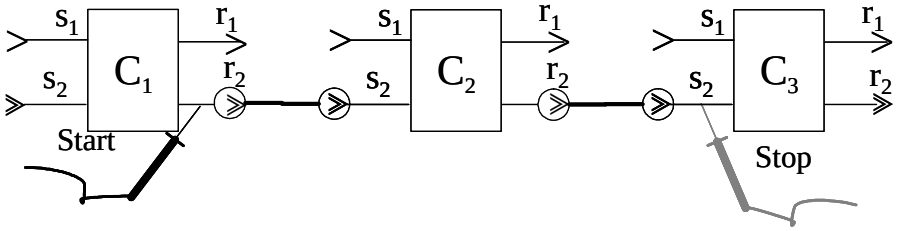

Figure 49: Measuring the Latency for a Simple Assembly

B.2.3 Define Sampling Procedure

The overall process for validating λ_{ABA} was to

1. Predict the latency of the longest path jobs in a sample of assemblies.
2. Measure the actual latencies of those jobs.
3. Compare predicted and actual latencies.

To perform the steps above, we needed the following:

1. a population of components (with measured execution times)
2. a representation population and sufficient quantity of assemblies built from those components

The components and their assemblies did not exist—a situation that may arise frequently in the early stages of PECT development. We therefore performed an analytic rather than enumerative study. For this we developed synthetic components and the means to generate synthetic assemblies.

B.2.3.1 Component Sample

To create a viable population of components to choose from, 15 classes of synthetic components were created. The classes were named sequentially from synthetic5 through synthetic20, where the numerical postfix indicated the number of milliseconds for which the component was designed to execute. The calculation that was performed by this synthetic component is shown in Figure 50.

```
Line. 1 double t = execution;
Line. 2 for (int i = INT_MIN / 76500; i < 0; i++) {
Line. 3  for (int j = 0; j < execution * 4; j++) {
Line. 4   t = j + i * j + i;
Line. 5  }
Line. 6 }
Line. 7 t = t * t;
```

Figure 50: Execution Calculation Made by the Synthetic Component

The variable `execution` in Line 3 of Figure 50 is initialized for all instances for each class to a value that approximates the desired execution delay (e.g., execution = 5 ms for synthetic5 and 6 ms for synthetic6).

The interface and reaction rules for each synthetic component were identical. Each synthetic component had one synchronous (s_1) and one asynchronous (s_2) sink pin, as well as one synchronous (r_1) and one asynchronous (r_2) source pin. The interface for each pin (named signal) was identical, taking no `in` or `in/out` arguments and having no return value, which is equivalent to the C language specification `void signal (void)`. The asynchronous sink pin (s_2) was handled by one thread. Finally, the reaction rules for the synthetic component were

$$RS1 = s_1 \rightarrow r_1 \rightarrow \bar{r}_1 \rightarrow r_2 \rightarrow \bar{r}_2 \rightarrow \bar{s}_1 \rightarrow RS1$$
$$RS2 = s_2 \rightarrow r_1 \rightarrow \bar{r}_1 \rightarrow r_2 \rightarrow \bar{r}_2 \rightarrow \bar{s}_2 \rightarrow RS2$$

The reaction rule for either sink pin would be fired only after the internal calculation in Figure 50 was performed.

The execution time (in milliseconds) for each reaction was measured to a 95% confidence interval. The number of samples was chosen to achieve that target precision. In most cases, that precision was achieved before 30 samples. For each reaction, the average execution time and standard deviation were recorded.

In addition to synthetic components, the environment Clock component was developed. This component is used to periodically generate a signal on a source pin for a specific period (e.g., every 200 ms). The period can be configured prior to the execution of any instance of that class, and the signal can be generated on either a synchronous or asynchronous source pin.

For the validation of λ_{ABA}, it was only necessary to measure the execution time of synthetic components. The Clock component has no "internal" calculation and is used as the stimulus for various jobs within the assembly. The remainder of this section discusses the procedure for measuring synthetic components.

The steps for sampling the execution time of the synthetic component are

1. Create a test Pin component assembly for each synthetic component to be measured.
2. Execute each synthetic component through the runtime environment and capture its measured execution time.
3. Update the Pin component description (registry) for each component tested with the captured measures from the runtime environment.

Each step is described in detail below.

Step 1: Create a test Pin component assembly for each synthetic component to be measured.

This step is done using a series of MS-DOS batch (.BAT) files. The createsinktest.bat script is used to generate a component description for each synthetic component class having an approximate designed execution time of 5 ms (i.e., synthetic5) to 20 ms (i.e., synthetic20). That script calls the createonesinktest.bat script to generate a single component description for the synthetic component enumerated as a parameter. The output from this initial step is a sequence of synthetic component classes, each with its own unique execution time in the general form shown in Figure 51.

```
Line. 1 <?xml version="1.0" ?>
Line. 2  <Assembly xmlns="PinTekXML.xsd">
Line. 3   <Components>
Line. 4    <Component name="clock1" type="clock">
Line. 5      <Property propId="measurement" value="true" />
Line. 6      <Property propId="period" value="100" />
Line. 7    </Component>
Line. 8    <Component name="task1" type="synthetic5">
Line. 9      <Property propId="measurement" value="true" />
Line. 10     <Property propId="priority" value="10" />
Line. 11     <Property propId="loadFactor" value="5" />
Line. 12   </Component>
Line. 13   <Component name="task2" type="syntheticN">
Line. 14     <Property propId="measurement" value="true" />
Line. 15     <Property propId="priority" value="15" />
Line. 16     <Property propId="loadFactor" value="N" />
Line. 17   </Component>
Line. 18  </Components>
Line. 19  <Connectors>
Line. 20   <Connector>
Line. 21     <Source component="clock1" pin="0" />
Line. 22     <Sink component="task1" pin="1" />
Line. 23   </Connector>
Line. 24   <Connector>
Line. 25     <Source component="task1" pin="2" />
Line. 26     <Sink component="task2" pin="0" />
Line. 27   </Connector>
Line. 28   <Connector>
Line. 29     <Source component="task1" pin="3" />
Line. 30     <Sink component="task2" pin="1" />
Line. 31   </Connector>
Line. 32  </Connectors>
Line. 33 </Assembly>
```

Figure 51: General Assembly Topology Description for Benchmarking Synthetic Component Classes

In lines 13 and 16 of Figure 51, where **N** was the configured execution time for the instances of synthetic components being tested, **N** for this validation ranged from 5 to 20, inclusive. Figure 52 shows the general topological form for each synthetic component tested.

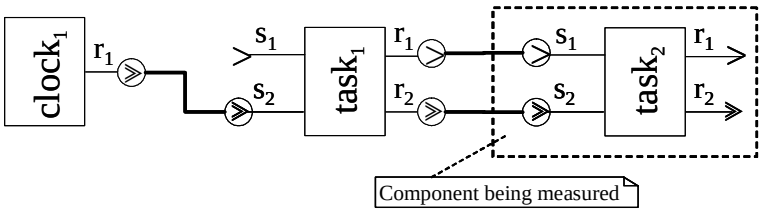

Figure 52: General Assembly Topology for Benchmarking Synthetic Components

Step 2: Execute each synthetic component through the runtime environment and capture its measured execution time.

The synthetic components are run using the sinktestscript.bat and measure2.bat scripts.

These scripts invoke the Pin runtime environment and assembly measurement environment using the test Pin component assemblies built in the previous step. Each test component assembly is run twice: once for the synchronous pin of the synthetic component (task2.$s_1 \rightsquigarrow$ task2.$\bar{s}_1$) and once for the asynchronous pin of the same synthetic component (task2.$s_2 \rightsquigarrow$ task2.$\bar{s}_2$). The test assembly is run until it is stopped by the assembly measurement environment. The measurement environment will cease measuring once it has achieved a 95% confidence interval in the latency of the measured reaction. In all cases, the 95% confidence interval was achieved before the minimum number of samples, which was set to 30.

Step 3: Update the Pin component description (registry) for each component tested with the captured measures from the runtime environment.

As the measurement environment measures the component from the runtime traces emanated from the runtime environment, it is continuously computing the average elapsed execution time (in milliseconds) and standard deviation. Once the 95% confidence interval is met, the measurement environment records the measures for the component and exits (thus causing the runtime environment to cease). The form of the results for each recorded assembly is shown in Figure 53.

```
Line. 1 <ComponentProperties
Line. 2  name="sinktest\sinktest10.xml.cfg"
Line. 3  pin="1" samples="30"
Line. 4  avgElapsedTime="10.03265333" stdDevElapsedTime="0.001559"/>
Line. 5 <ComponentProperties
Line. 6  name="sinktest\sinktest10.xml.cfg"
Line. 7  pin="2" samples="30"
Line. 8 avgElapsedTime="9.83988000" stdDevElapsedTime="0.003972"/>
```

Figure 53: Example Measures for Component Execution Time

In Figure 53, component synthetic10 has an average execution time for synchronous pin s_1 of 10.03 ms with a standard deviation of 0.002 ms and an average execution time for asynchronous pin s_2 of 9.84 ms with a standard deviation of 0.004 ms. The actual execution time for each synthetic component class is captured in Table 17 on page 122.

The measures collected by the measurement environment were transferred to the Pin component description for each synthetic component class tested in this procedure, essentially establishing those measures as the “certified” execution time properties for those components, under the runtime environment described in Section B.2.4.2.

B.2.3.2 Assembly Sample

Rather than arbitrarily selecting and manually building assemblies from the population of components (possibly introducing bias into the selection process), assemblies were randomly selected from a population of possible assemblies defined by assembly variation points. The variation points serve to define an assembly design space (an n-dimension coordinate system) of possible assemblies. For example, no assembly was considered for λ_{ABA} if it had more than 50 components. The following variation points were defined:

- number of clocks—the total number of Clock components discussed above that are used as a stimulus to other components within an assembly
- number of components—the total number of components, in this case, synthetic components, that make up an assembly
- number of connections per source pin—the maximum number of connections allowed for a single component's source pin to be connected to other components' sink pins
- minimum load factor—the minimum execution time for a component in the assembly. This number could range from 5 to 20 and had to be less than or equal to the maximum load factor.
- maximum load factor—the maximum execution time for a component in the assembly. This number could range from 5 to 20.
- harmonic period—determination of whether the clocks in the system have the same period
- minimum clock period—the minimum clock period for the clocks in the assembly. This value must be less than or equal to the maximum clock period.
- maximum clock period—the maximum clock period for the clocks in the assembly
- communication type—determination of whether the connections between the components in the assembly are asynchronous ('A'), synchronous ('S'), or heterogeneous ('M,' a mix of synchronous and asynchronous) type communications
- percent blocking—percentage of the total number of synchronous pins used in the assembly that must be blocked or mutexed

Tuples $A' = <v_0, v_1, ..., v_n>$ are constructed from variation points, where v_k is a binding of the k^{th} variation point. Each tuple A' represents, in some way, a stereotypical assembly. Thirteen such tuples were defined, again using judgment. An example tuple is shown in Figure 54; the complete set of tuples is shown in Table 10.

```
Line. 1 <Description name="1"
Line. 2   nofClocks="1" nofComponents="2"
Line. 3   nofConnectionsPerSrc="1"
Line. 4   minLoad="5" maxLoad="5"
Line. 5   harmonicPeriods="true"
Line. 6   minPeriod="20" maxPeriod="100"
Line. 7   connectionType="A"
Line. 8   percentblocking="50" />
```

Figure 54: Variation Points for One Assembly Design Space

Table 10: Variation Points for All Assembly Design Spaces

Assembly Design Space Tuple	# of Clocks	# of Components	# of Connections	Minimum Load (ms)	Maximum Load (ms)	Harmonic Period	Minimum Period (ms)	Maximum Period (ms)	Communication Type	% Blocking
1	1	2	1	5	5	Y	20	100	A	50
2	4	50	2	5	20	Y	500	1000	A	50
3	1	2	1	5	5	N	20	100	A	50
4	1	2	1	5	5	Y	20	100	M	50
5	4	50	2	5	20	Y	500	1000	M	50
6	1	2	1	5	5	N	20	100	M	50
7	2	4	1	5	10	N	100	500	M	50
8	2	4	1	5	10	Y	100	500	M	0
9	2	8	1	5	10	N	20	100	A	100
10	2	8	1	5	10	N	20	100	M	0
11	4	35	2	15	20	Y	500	2000	A	50
12	4	35	1	5	5	Y	500	2000	M	100
13	2	4	1	5	5	Y	200	400	M	25
14	3	10	1	5	5	N	200	400	M	25
15	4	15	1	5	5	Y	200	400	M	25
16	2	15	3	20	20	Y	400	1200	M	75

Each tuple (one row of Table 10) was given to the SEIAssemblyGeneratorController tool to randomly select an assembly from the design space. Each generated assembly was tested for λ_{ABA} well-formedness per the rules described in Appendix A.

Figure 55 is one of the assemblies generated randomly from tuple #1 (i.e., the first row in Table 10).

```
Line. 1 <?xml version="1.0" encoding="utf-8" standalone="yes"?>
Line. 2 <Assembly xmlns="PinTekXML.xsd">
Line. 3   <Components>
Line. 4     <Component name="Clock0" type="clock">
Line. 5       <Property propId="period" value="40" />
Line. 6     </Component>
Line. 7     <Component name="C1" type="synthetic5">
Line. 8        <Property propId="measurement" value="true" />
Line. 9        <Property propId="priority"    value="68" />
Line. 10       <Property propId="loadFactor"  value="5" />
Line. 11       <Property propId="blocking"    value="true" />
Line. 12     </Component>
Line. 13     <Component name="C2" type="synthetic5">
Line. 14       <Property propId="measurement" value="3" />
Line. 15       <Property propId="priority"    value="124" />
Line. 16       <Property propId="loadFactor"  value="5" />
Line. 17       <Property propId="blocking"    value="false" />
Line. 18     </Component>
Line. 19   </Components>
Line. 20   <Connectors>
Line. 21     <Connector>
Line. 22       <Source component="Clock0" pin="0" />
Line. 23       <Sink   component="C1" pin="1" />
Line. 24     </Connector>
Line. 25     <Connector>
Line. 26       <Source component="C1" pin="3" />
Line. 27       <Sink   component="C2" pin="1" />
Line. 28     </Connector>
Line. 29   </Connectors>
Line. 30 </Assembly>
```

Figure 55: Example of a Randomly Selected Assembly

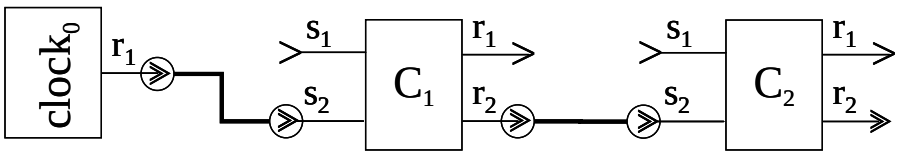

Figure 56: Topology of an Example of a Randomly Selected Assembly

λ_{ABA} predicts the average latency of the longest path of an assembly of components from the "certified" execution time of components (see Section B.2.5.1). The latency computed for the job becomes the prediction for the latency of that job for that assembly. For the assembly

shown in Figure 56, the calculated prediction (based on the component property for pin 1 of the synthetic5 component [see Table 17 on page 122]) is shown in Table 11.

Table 11: Predicted Latency for Synthetic5's Job

Task	Job	Predicted Latency (ms)	Standard Deviation (+/- ms)
1	1	10.447651	0.001320

Task #1 (from Table 11) refers to the job beginning with component instance clock0 in the assembly. Job #1 is the "longest path" emanating from that clock. Since, in this assembly, there is only one "path" (i.e., clock0.$r_1 \rightsquigarrow c_1.s_2$, $c_1.r_2 \rightsquigarrow c_2.s_2$), it is, by default, the "longest path" and therefore the only job—hence one prediction. Predicted latency is the prediction for the average number of milliseconds it will take to complete one period in the hyper-period for the job, which for Task #1, Job #1 is 10.45 ms. The standard deviation is the computed standard deviation for the prediction (in milliseconds).

With a prediction for each job in each defined and randomly selected assembly, each assembly was run using the assemblytestscript.bat and measureAssembly.bat scripts. These scripts invoke the Pin runtime environment and assembly measurement environment using the randomly selected component assemblies. Each random component assembly is run once for each job found in the assembly so that it can be measured independently from any other executing jobs in the assembly. Each assembly job is run until it is stopped by the assembly measurement environment. That environment will cease measuring once it has collected *N* samples (in this experiment, *N*=30).

The measurement environment computes continuously, from time-stamped Pin events, the average elapsed time (in milliseconds), and the standard deviation of assembly execution. Upon completion of the measurement process, the measurement environment records the measures for the assembly and exits. The form of the results for each recorded assembly is shown in Table 12.

Table 12: Observed Average Latency for an Assembly

Task	Job	Samples	Average Latency (ms)	Standard Deviation (+/- ms)
1	1	30	10.464237	0.007309

Table 12 is very similar to the table described above for recording the prediction. The differences are that after successfully measuring a job for an assembly, the total number of latency samples observed are recorded (in this case, Samples = 30), and the computed arithmetic mean (i.e., Average Latency = 10.46 ms) is calculated.

The actual assembly latencies observed for all tasks (i.e., average job) are captured in Table 18 on page 123.

B.2.4 Develop Measurement Infrastructure

The infrastructure described here was developed to support the sampling procedure outlined in Section B.2.3. We expect a large portion of this infrastructure to be reusable by other property theories, not necessarily time-based ones.

B.2.4.1 Measurement Tool Suite

The complete measurement infrastructure required to obtain measurement data on the components and assemblies used in the empirical validation is discussed briefly below.

The overall control and data flow for the tools created for the empirical validation is shown in Figure 57. Component execution time is measured by placing components in a test harness—essentially an assembly occupied only by the component (the Composer tool). An assembly generator uses variation points (the tuples described earlier) to generate well-formed assemblies from measured components. The generated assemblies are then executed in an instrumented runtime that records actual job latencies. This data is then compared with predictions to produce statistical labels for λ_{ABA}.

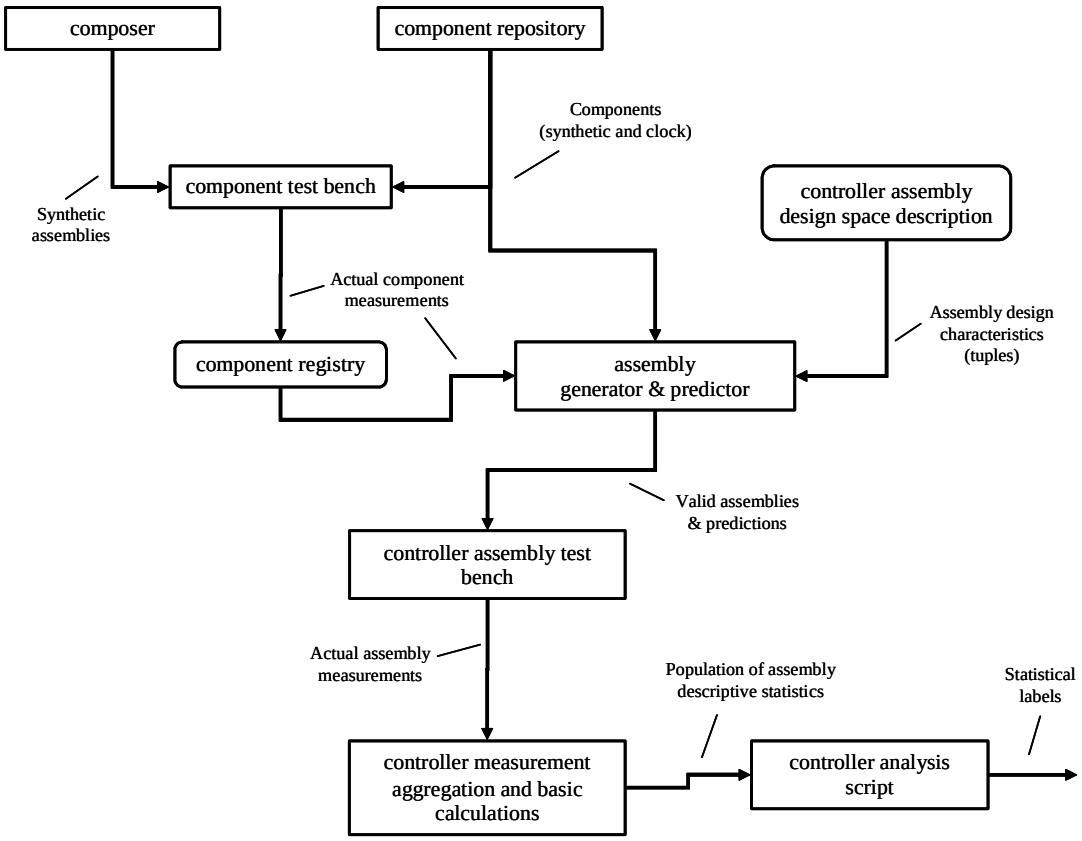

Figure 57: Measurement Infrastructure Tool Workflow

The specific tools that were created and their purposes are shown in Table 13.

Table 13: Measurement Infrastructure Tool Suite

Workflow Element/Tool	Purpose
Component repository	
reg.bat	Registers RTX components (.rtdlls) with the RTX Runtime executive
*.rtdll	RTX components
*.dll	WIN32 components
Composer	
createsinktest.bat	Outer control loop, iterating from 5 to 20 times as arguments for the component assembly generator script
createonesinktest.bat	Component assembly generator script, creating a simple assembly for testing a single component. The name of the component is passed as an argument.
XML2CFGsinktest.bat	Outer control loop, iterating over the assemblies generated by the component assembly generator and passing the generated XML file to the XML-to-CFG translator
SEITranslateAssembly-fromXML.exe	XML-to-CFG translator that converts an assembly generated by the component assembly generator from an XML representation to a CFG representation suitable for consumption by the RTX Pin Runtime
Component test bench	
sinktestscript.bat	Outer control loop, iterating over the assemblies generated by the component assembly generator and passing the converted CFG file to the main measurement script
measure2.bat	Main measurement script overseeing the execution of the assembly in the RTX Pin Runtime and measured by the RTX Assembly Measurement tool. When the RTX Assembly Measurement tool is complete, the script kills the RTX Pin Runtime and exits.
results2.xml	Component measurements (results) of actual values captured by the RTX Assembly Measurement tool
Component test bench OR controller assembly test bench	
Runtime.rtss	RTX Pin Runtime. Besides executing the Pin assembly, the RTX Pin Runtime generates measurement events that are consumed by the RTX Assembly Measurement tool.
assemblymsrmt.rtss	RTX Assembly Measurement tool. Reads measurement events generated by the RTX Pin Runtime and calculates the job execution time for a hyper-period based on command line arguments.

Table 13: Measurement Infrastructure Tool Suite (Continued)

Workflow Element/Tool	Purpose
mywait.bat	Used to pause the execution of the main measurement script until the RTX Pin Runtime has been signaled by the RTX Assembly Measurement tool that it is complete. The mywait.bat script and the RTX Assembly Measurement tool do this signaling through the creation and removal of a file in the file system.
wait.exe	A tool designed to block (wait) for *N* seconds using native WIN32 calls for sleep() (a non-busy wait)
killruntime.rtss	Used to signal the RTX Pin Runtime to suspend execution of the assembly and exit
rtx-hack.exe	A tool that is used to aid the continuous, non-human interactive execution of the RTX Pin Runtime. A bug in the RTX executive causes the generation of an RTX exception that can be dismissed only by pressing a button on the exception dialog. This WIN32 application looks for that dialog and dismisses the dialog automatically.
Component registry	
components.xml	Registration information about components available for use within the Pin Runtime environment. This file contains path locations to the components in the file system as well as pin configurations and measurement data for those components.
components.cfg	RTX-Pin-Runtime-translated version of the XML representation of the same name. This file is suitable for use with the RTX Pin Runtime.
XML2CFGcomponents.bat	Simple one-line script for running the XML-to-CFG translator for the component registry
SEITranslateComponentsXML.exe	XML-to-CFG translator that converts a component registry from an XML representation to a CFG representation suitable for consumption by the RTX Pin Runtime
Controller assembly design space description	
ControllerValidationDescr.xml	XML file that describes a number of valid design spaces from which assemblies can be selected randomly
Assembly generator and predictor	
SEIAssemblyGeneratorController.exe	Outer control loop, iterating over randomly selected assemblies from a predetermined assembly design space. The selected assembly is passed to the Pin Runtime simulator for rejection (a non-schedulable assembly) or prediction (a schedulable assembly).
AsyncGen.exe	Pin Runtime simulator. If the assembly is schedulable, a prediction is made for each job within the assembly. If it is not schedulable, the assembly is rejected.

Table 13: Measurement Infrastructure Tool Suite (Continued)

Workflow Element/Tool	Purpose
SEI_#_*.xml	Assembly that is generated by the SEIAssemblyGeneratorController.exe script and deemed schedulable by the Pin Runtime simulator
SEI_#_*.cfg	CFG representation of the XML assembly generated by the SEITranslateAssemblyfromXML.exe script
SEI_#_*.csv	Prediction made for each job in the assembly by the Pin Runtime simulator
SEI_#_*.lpi	"Longest path" information for each job within an assembly
XML2CFGassemblies.bat	Outer control loop, iterating over the assemblies generated by the SEIAssemblyGeneratorController.exe script assembly generator and passing the generated XML file to the XML-to-CFG translator
SEITranslateAssembly-fromXML.exe	XML-to-CFG translator that converts an assembly generated by the component assembly generator from an XML representation to a CFG representation suitable for consumption by the RTX Pin Runtime
sed	Tool used to patch the generated CFG file due to a bug in the Pin Runtime simulator
Controller assembly test bench	
doemall.bat	Outermost control loop to repeat execution of all the assemblies generated by the SEIAssemblyGenerator-Controller.exe script and the Pin Runtime simulator. As configured, this script will repeat the run of the entire set of assemblies 40 times.
2assemblytestscript.bat	Support script for the outermost control loop
assemblymeasure.bat	Inner control loop to manage the execution of all generated assemblies
measureAssembly.bat	Main assembly measurement script overseeing the execution of the assembly in the RTX Pin Runtime and measured by the RTX Assembly Measurement tool. When the RTX Assembly Measurement tool is complete, the script kills the RTX Pin Runtime and exits.
Controller measurement aggregation and basic calculations	
pcfu_analysis_v3.xls	Excel spreadsheet to collect all the measurements recorded by the Pin Measurement tool into one place. Performs basic descriptive statistics on the data and aggregates (such as Average, MRE, and standard deviation). Other sheets within the Excel workbook include those on graph generation and other manual aids for analysis.
Module1.gatherpredictions()	Excel.VBA script for automating the inclusion and aggregation of recorded measurement data

Table 13: Measurement Infrastructure Tool Suite (Continued)

Workflow Element/Tool	Purpose
Module1.paccCompare()	Excel.VBA script for automating the separation of actual latency records from the bulk of other collected data. Also aids in analysis.
Module1.paccCompareAVG()	Excel.VBA script for automating the separation of Actual MREs calculated from the bulk of other collected data. Also aids in analysis.
Controller analysis script	
pect1.exa	Script used for automating the analysis that determines the statistical labels from the experiment
StInt.exe	Tool created by Hahn and Meeker that comes complete with a rudimentary statistical scripting language [Hahn 91]

B.2.4.2 Test Environment

Execution time properties for components, as well as assembly job latency, are directly affected by the runtime environment. Such properties include both hardware (e.g., processor speed) and software (e.g., operating system) properties. The significant hardware and software features and configurations are listed in Table 14 and Table 15, respectively.

Table 14: Hardware Characteristics

Item	Value
System Manufacturer	Dell Computer Corporation
System Model	DIM4400
Processor	x86 Family 15 Model 1 Stepping 2 GenuineIntel ~1595 Mhz
Physical Memory	1,047,856 KB 184-pin DIMM PC2100
Virtual Memory	3,570,352 KB
Hard Disk 1	WDC WD400BB-75CAA0 40 GB 7200 RPM 2 MB Buffer 8.9 ms Average Read Seek Time

Table 15: Software Characteristics

Item	Value
Operating System Name	Microsoft Windows 2000 Professional 5.0.2195 Service Pack 2 Build 2195
Compiler	Microsoft Visual C# .NET 1.0 Build 3705
Compiler	Microsoft Visual Studio C++ 6.0 Service Pack 5
Real-Time Extensions	VenturCom RTX 5.1.1 Build 3517

Additionally, the property values collected can be affected by other operating and user processes currently running on the machine. All opportunities were taken to "quiet" the runtime environment. Table 16 lists all the residual processes running at test time.

Table 16: Processes (Running Tasks)

Name	Version
smss.exe	5.00.2195.2901
csrss.exe	5.00.2195.2581
winlogon.exe	5.00.2195.2953
services.exe	5.00.2195.2780
lsass.exe	5.00.2195.2964
svchost.exe	5.00.2134.1
spoolsv.exe	5.00.2161.1
svchost.exe	5.00.2134.1
mdm.exe	7.00.9466
regsvc.exe	5.00.2195.2104
mstask.exe	4.71.2195.1
winmgmt.exe	1.50.1085.0029
logon.scr	5.00.2195.2104

B.2.5 Collect Validation Data

B.2.5.1 Component Execution Times

Table 17 lists the actual execution times recorded for each synthetic Pin component.

Table 17: Recorded Actual Execution Time Per Synthetic Pin Component

Component	Pin	Samples	Average Execution (ms)	Standard Deviation (+/- ms)
synthetic5	s1	30	5.418953330	0.0015290
	s2	30	5.223903330	0.0009710
synthetic6	s1	30	6.339036670	0.0011540
	s2	30	6.150383330	0.0010060
synthetic7	s1	30	7.262256670	0.0010040
	s2	30	7.069983330	0.0012750
synthetic8	s1	30	8.186626670	0.0011700
	s2	30	7.992573330	0.0012050
synthetic9	s1	30	9.108433330	0.0012930
	s2	30	8.916026670	0.0011840
synthetic10	s1	30	10.032653330	0.0015590
	s2	30	9.839880000	0.0039720
synthetic11	s1	30	10.956230000	0.0012750
	s2	30	10.758920000	0.0013960
synthetic12	s1	30	11.878773330	0.0035830
	s2	30	11.680800000	0.0017240
synthetic13	s1	30	12.797450000	0.0013040
	s2	30	12.607413330	0.0011450
synthetic14	s1	30	13.722073330	0.0016360
	s2	30	13.527796670	0.0018540
synthetic15	s1	30	14.647590000	0.0010480
	s2	30	14.450010000	0.0015210
synthetic16	s1	30	15.567820000	0.0016890
	s2	30	15.371700000	0.0012900
synthetic17	s1	30	16.491470000	0.0012860
	s2	30	16.291336670	0.0012010

Table 17: Recorded Actual Execution Time Per Synthetic Pin Component (Continued)

Component	Pin	Samples	Average Execution (ms)	Standard Deviation (+/- ms)
synthetic18	s1	30	17.415736670	0.0020440
	s2	30	17.214350000	0.0013260
synthetic19	s1	30	18.334366670	0.0013310
	s2	30	18.141623330	0.0012160
synthetic20	s1	30	19.257180000	0.0014420
	s2	30	19.062330000	0.0015600

B.2.5.2 Assembly Latencies

Table 18 lists the predicted and actual latencies recorded for each Task.

Table 18: Recorded Predicted and Actual Latencies Per Task

Sample	Assembly	Task	Predicted Latency (ms)	Average Latency (ms)	Average MRE (AVGMRE)
1	1	1	10.447651000	10.464237000	0.001585018
2	2	1	10.447651000	10.464103000	0.001572232
3	3	1	10.447651000	10.468303000	0.001972813
4	4	1	10.447651000	10.481843000	0.003262022
5	5	1	10.642405000	10.647977000	0.000523292
6	6	1	5.223772000	5.242617000	0.003594579
7	7	1	10.642405000	10.649987000	0.000711926
8	8	1	10.642405000	10.648773000	0.000598003
9	9	1	10.466214200	10.434522200	0.003037226
10		2	26.211976000	26.010260000	0.007755247
11	10	1	16.908883000	16.938880000	0.001770896
12		2	39.233906000	39.198687000	0.000898474
13	11	1	21.285280000	21.317490000	0.001510966
14		2	15.866593500	15.845152000	0.001353190
15	12	1	26.899670000	26.476400000	0.015986690
16		2	64.441911500	63.444630000	0.015718927
17		3	86.117799000	84.727607000	0.016407781
18		4	96.760863000	95.213423000	0.016252330

Table 18: Recorded Predicted and Actual Latencies Per Task (Continued)

Sample	Assembly	Task	Predicted Latency (ms)	Average Latency (ms)	Average MRE (AVGMRE)
19	13	1	31.343418000	31.345887000	0.000078766
20		2	10.447805000	10.486627000	0.003702048
21	14	1	32.319165000	31.876143000	0.013898231
22		2	70.056397000	68.859087000	0.017387829
23		3	51.089765500	50.240897000	0.016895966
24		4	77.989699000	76.642826500	0.017573367
25	15	1	5.223772000	5.240607000	0.003212414
26	16	1	10.447651000	10.467927000	0.001936964
27	17	1	10.447651000	10.483470000	0.003416712
28	18	1	5.223772000	5.241083000	0.003302943
29	19	1	10.642405000	10.645567000	0.000297025
30	20	1	5.223772000	5.242970000	0.003661665
31	21	1	10.642405000	10.644620000	0.000208086
32	22	1	31.040545514	30.827478573	0.006911592
33		2	28.740783575	28.494810878	0.008632193
34	23	1	5.223772000	5.245897000	0.004217582
35	24	1	12.486407000	12.507233000	0.001665116
36		2	13.313233500	13.312455000	0.000058479
37	25	1	54.716278000	54.752170000	0.000655536
38		2	5.223817000	5.243300000	0.003715790
39	26	1	20.895468000	20.905221500	0.000466558
40		2	10.447968000	10.486070000	0.003633582
41	27	1	54.686602000	54.454630000	0.004259913
42		2	14.143044000	14.143053000	0.000000636
43	28	1	17.302273710	17.261133611	0.002383395
44		2	22.866187841	22.621172793	0.010831227
45		3	11.003177993	11.022679949	0.001769257
46	29	1	203.385312000	201.129930000	0.011213557
47		2	55.102592500	53.494173500	0.030067181
48		3	123.077102000	121.362487000	0.014128048
49		4	17.559318500	16.247148500	0.080763095
50	30	1	15.866163000	15.851590000	0.000919340
51		2	23.897367500	23.910218000	0.000537448

Table 18: Recorded Predicted and Actual Latencies Per Task (Continued)

Sample	Assembly	Task	Predicted Latency (ms)	Average Latency (ms)	Average MRE (AVGMRE)
52	31	1	204.195290000	201.662288833	0.012560609
53		2	344.016553333	338.834986333	0.015292302
54		3	58.958727333	58.155217833	0.013816636
55		4	48.185407000	47.485040750	0.014749198
56	32	1	5.223772000	5.239153000	0.002935780
57	33	1	214.891847000	214.506840000	0.001794847
58		2	34.433039000	34.449800000	0.000486534
59		3	88.236056000	88.217465000	0.000210741
60		4	148.640039500	148.530475000	0.000737657
61	34	1	10.447651000	10.464323000	0.001593223
62	35	1	7.069980000	7.089007000	0.002684015
63		2	13.220204000	13.212337000	0.000595428
64	36	1	437.787672000	437.825835000	0.000087165
65		2	17.214288000	17.253165667	0.002253364
66		3	33.505698000	33.522708667	0.000507437
67		4	68.248844667	68.215058667	0.000495287
68	37	1	10.447651000	10.466423000	0.001793545
69	38	1	202.290165000	202.417706500	0.000630091
70		2	229.650747000	229.393180000	0.001122819
71		3	12.293624000	12.336327000	0.003461565
72		4	39.963095500	39.935103500	0.000700937
73	39	1	10.447651000	10.463353000	0.001500666
74	40	1	7.992571000	8.012658500	0.002506971
75		2	8.814414333	8.819739000	0.000603722
76	41	1	222.578150000	222.664175000	0.000386344
77		2	98.374782000	98.301925000	0.000741155
78		3	114.665828000	114.596211500	0.000607494
79		4	94.683699000	94.630223000	0.000565105
80	42	1	10.447651000	10.469507000	0.002087586
81	43	1	10.642405000	10.647427000	0.000471663
82	44	1	229.516865000	228.742650000	0.003384655
83		2	23.361351000	22.890378500	0.020575129
84		3	31.353861000	31.342447000	0.000364171
85		4	33.507831000	33.512895000	0.000151106

Table 18: Recorded Predicted and Actual Latencies Per Task (Continued)

Sample	Assembly	Task	Predicted Latency (ms)	Average Latency (ms)	Average MRE (AVGMRE)
86	45	1	5.223772000	5.240627000	0.003216218
87	46	1	41.155253000	40.868471500	0.007017182
88		2	26.821569000	26.696627000	0.004680067
89	47	1	9.225493000	9.226523500	0.000111689
90		2	6.150224000	6.163042286	0.002079863
91	48	1	6.453807375	6.467737125	0.002153729
92		2	9.839250000	9.856764333	0.001776885
93	49	1	473.421081167	470.870088333	0.005417615
94		2	5.223854000	5.254280750	0.005790850
95		3	6.965180333	6.980025333	0.002126783
96		4	9.577441000	9.584958167	0.000784267
97	50	1	85.142083000	84.681448667	0.005439613
98		2	49.101117500	48.649770000	0.009277485
99		3	90.366019000	89.437177667	0.010385405
100	51	1	95.980318000	95.206460000	0.008128209
101		2	5.223967000	5.252733500	0.005476482
102		3	26.314324000	26.267070000	0.001798983
103		4	21.090584000	21.061773000	0.001367929
104	52	1	362.962624000	363.222297000	0.000714915
105		2	343.900028000	343.661253000	0.000694798
106	53	1	10.447651000	10.466990000	0.001847618
107	54	1	211.515258000	211.171632000	0.001627236
108		2	233.034612000	232.761743000	0.001172310
109		3	53.188201000	53.152377000	0.000673987
110		4	27.981434000	28.033153500	0.001844940
111	55	1	10.447651000	10.476167000	0.002721988
112	56	1	15.867223000	15.900327000	0.002081970
113	57	1	194.014049000	192.847623000	0.006048433
114		2	26.899835500	26.926842000	0.001002958
115		3	129.438334000	128.484298500	0.007425308
116		4	148.500717500	148.038875000	0.003119738
117	58	1	10.447651000	10.461457000	0.001319701
118	59	1	38.628699250	38.482468250	0.003799938
119		2	15.254885000	15.268325800	0.000880306

Table 18: Recorded Predicted and Actual Latencies Per Task (Continued)

Sample	Assembly	Task	Predicted Latency (ms)	Average Latency (ms)	Average MRE (AVGMRE)
120	60	1	9.839871000	9.851990600	0.001230168
121		2	11.989716500	11.993116500	0.000283496
122	61	1	6.150381000	6.167381000	0.002756437
123		2	8.551507545	8.568310364	0.001961042
124	62	1	189.479111833	189.407213833	0.000379595
125		2	132.194130500	132.131390000	0.000474834
126		3	123.174457667	123.090005333	0.000686102
127		4	25.416587333	25.455879000	0.001543520
128	63	1	237.923872000	235.610187000	0.009819970
129		2	62.941314500	62.047110750	0.014411690
130		3	5.224044000	5.252087000	0.005339401
131		4	6.530178250	6.548859750	0.002852634
132	64	1	31.538572000	31.547863000	0.000294505
133		2	10.447630000	10.484670000	0.003532777
134	65	1	172.340149000	171.930870000	0.002380486
135		2	38.124446000	38.165202000	0.001067884
136	66	1	10.447651000	10.474927000	0.002603932
137	67	1	171.854499000	171.951639750	0.000564931
138		2	217.354521000	217.203003000	0.000697587
139		3	41.199258000	41.228013000	0.000697463
140		4	215.511509500	215.330905000	0.000838730
141	68	1	10.642405000	10.651257000	0.000831076
142	69	1	10.642405000	10.649523000	0.000668387
143	70	1	32.553767000	32.335025889	0.006764835
144		2	14.091335800	13.996779000	0.006755611
145	71	1	396.218259000	393.783429000	0.006183170
146		2	436.698304333	433.988472333	0.006244018
147		3	219.274297167	217.936968500	0.006136309
148		4	142.896621250	142.249186000	0.004551416
149	72	1	10.642405000	10.649430000	0.000659660
150	73	1	10.642405000	10.648750000	0.000595845
151	74	1	32.553767000	32.335675222	0.006744618
152		2	14.091335800	14.149474800	0.004108916

Table 18: Recorded Predicted and Actual Latencies Per Task (Continued)

Sample	Assembly	Task	Predicted Latency (ms)	Average Latency (ms)	Average MRE (AVGMRE)
153	75	1	396.218259000	393.627532333	0.006581670
154		2	436.698304333	433.993043333	0.006233420
155		3	219.274297167	237.628940500	0.077240774
156		4	142.896621250	142.245071500	0.004580473

B.2.6 Analyze Results

The goal of the analysis study for empirically validating the property theory (in Appendix A) was to support or refute the hypothesis that λ_{ABA} would predict the latency for any job within the hyper-period with an MRE ≤ 0.05 with a confidence level $\gamma = 0.99$. A minimum acceptable p = 80% was established for a pass/fail condition.

B.2.6.1 Computations

The MRE for each recorded and observed average latency was calculated for each task using this equation:

$$\mathrm{MRE} = \left| \frac{\mathrm{Predicted} - \mathrm{Actual}}{\mathrm{Actual}} \right|$$

This resulted in having one MRE computed for each task executed. The result of that computation is found in Table 18.

B.2.6.2 Descriptive Statistics

Table 19 includes some basic descriptive statistics for the assembly measures recorded in Table 18.

Table 19: Descriptive Statistics for Job Assembly Measurements

Basic Statistic	Value
Samples (N)	156
Mean MRE (mAVGMRE)	0.005081224
Standard Deviation (s)	0.009841344
Spearman rank correlation of Predicted Latency and Average Latency[a]	0.998 p-Value < 0.001

a. A high correlation (closer to -1.0 or 1.0) is an indication of correlation between the predicted latency and average latency. P-value is the probability that any such correlation is a coincidence. For a small p-value, reject the hypothesis that the correlation is a coincidence. Spearman's correlation is a non-parametric correlation and does not make assumptions about the data (i.e., normality).

B.2.6.3 Normality Test

The Shapiro-Wilk normality test was used to support (or refute) the hypothesis that the populations of job AVGMREs was normal. Depending on the result, different forms of intervals must be used. The results from the Shapiro-Wilk normality test are shown in Table 20.

Table 20: Results from the Shapiro-Wilk Normality Test on AVGMRE

***W* statistic**	**p-value**
0.4403	2.2e-16

Since the p-value is less than the generally accepted critical p-value of 0.05, the hypothesis that the MREs for the jobs are normally distributed was rejected.

Next, a logarithmic transformation was applied to the AVGMREs with the intent to make them normally distributed. That is, each AVGMRE in the population was transformed using LOG(AVGMRE), and a new population was created. Again, the Shapiro-Wilk normality test also refuted that the new, transformed population was normal because the p-value, although much improved, was still smaller than the critical p-value of 0.05.

Table 21: Results from the Shapiro-Wilk Normality Test on LOG(AVGMRE)

***W* statistic**	**p-value**
0.9427	5.77e-06

Similarly, a Box-Cox transformation also failed to produce a normal distribution (varying $\lambda = \{ 5.0, 0.5, 0.3, 0.1, 0.01\}$[42]). Histograms of the AVGMRE and LOG(AVGMRE) population are shown in Figures 58 and 59 below.

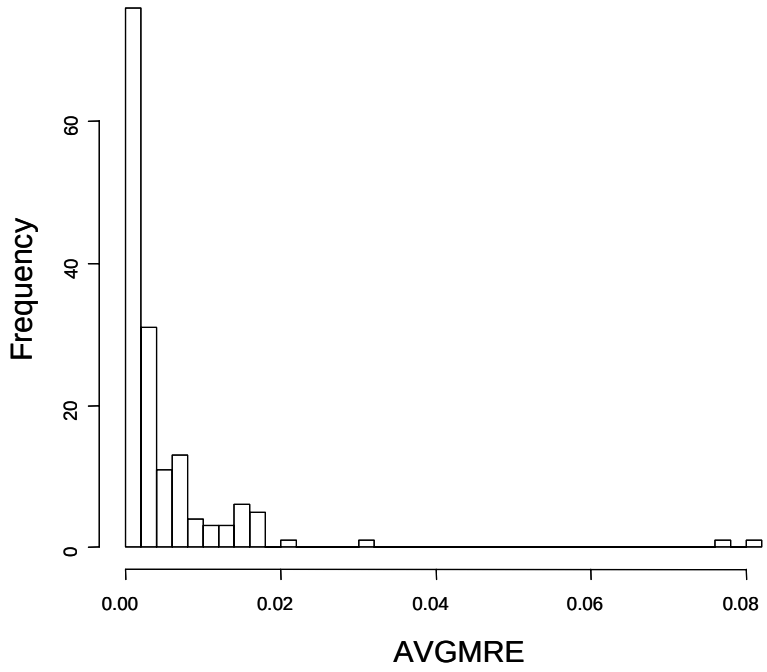

Figure 58: Histogram of AVGMRE

42. As the λ-values approach 0, the transformation approaches the logarithmic transformation performed earlier. This is not surprising, because the Box-Cox transformation for λ-value 0 equals LOG().

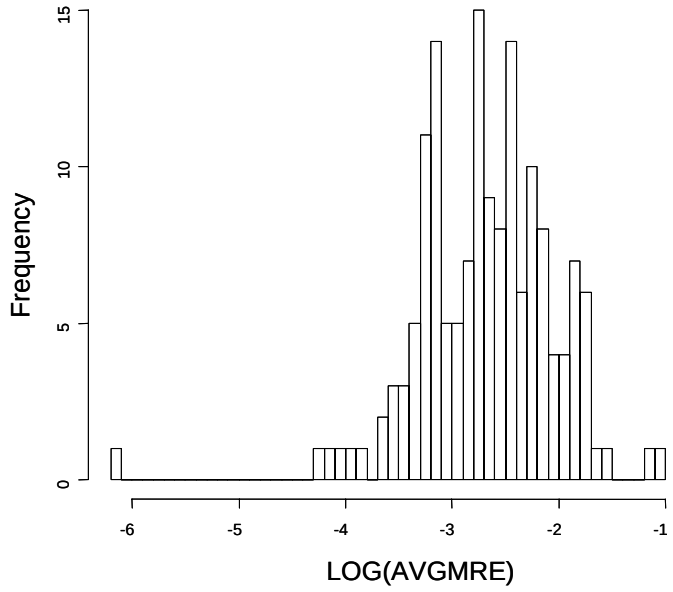

Figure 59: Histogram of LOG(AVGMRE)

Given that the population of the AVGMRE job data is not normal, statistical intervals that assume normally distributed data cannot be used. Therefore, distribution-free statistical intervals had to be used to compute the tolerance and confidence intervals.

B.2.6.4 One-Tail Distribution-Free Statistical Intervals

The upside to using distribution-free statistical intervals is that there are no assumptions about the data's normality. Given that the population of the AVGMREs recorded is not normal, this upside is an incentive for using distribution-free intervals. However, the downside is that it is difficult to achieve, precisely, the desired confidence intervals. Further, the intervals that are determined tend to be longer than those derived from normal distributions. One reason for this difference is that such statistical methods do not require parameters to the formulae and table lookups (such as standard deviation) that are used to determine either tolerance or confidence intervals for distribution-free populations[43] [Hahn 91].

The goal for this analysis is to support or reject the hypothesis that λ_{ABA} will predict the latency of a job to have an MRE ≤ 0.05. This is the upper bound for an MRE interval representing situations where predictions would be *no worse* than the μMRE for a job. Notice that the lower bound for an MRE interval would represent situations where the predictions would

43. Distribution-free statistical intervals are also referred to as non-parametric statistical intervals.

be *no better* than the μMRE for a job. Thus, knowing both ends of the interval (a two-tail interval) is not the goal of this analysis study, as only the upper-bound interval (a one-tail interval) is of interest.

StInt, a tool that performs a number of different kinds of statistical intervals (including distribution-free intervals), was used to determine the tolerance interval for the population of job AVGMREs [Meeker 93]. The first calculation performed is shown in Figure 60 where

- `p=0.80` is the proportion of the percentile of the population to compute the interval.
- `kside=2` is the upper tolerance bound.
- `conlev=0.99` is the desired confidence level.

```
Line. 1 stint> dfti p=0.80 kside=2, conlev=0.99
Line. 2  finding a nonparametric  99.0% upper confidence bound
Line. 3    for the  80.0 percentile.
Line. 4  this is also a 99.0% upper tolerance bound to exceed
Line. 5    at least 80.0% of the population.
Line. 6
Line. 7  the interval (bound) is based on  156 observations.
Line. 8  the desired upper bound is x(137).
Line. 9  the actual confidence level is= 0.9929 (99.29%)
Line. 10
Line. 11  ordered observation number   137 is    0.01 (1% MRE)
```

Figure 60: Distribution-Free Calculation Using StInt

The result of this initial calculation is that our hypothesis for achieving an MRE ≤ 0.05 was supported. The upper-bound tolerance we could expect to achieve based on this population (`N=156`, line 7 above) was 0.01 (or 1% MRE, line 11 above) for 80% of the population, with an actual confidence level of 99.29% (line 9 above).

Table 22: Results from the Initial Calculation for a One-Sided Distribution-Free Tolerance Interval

Part of Interval	Value
N = 156	sample size
$\gamma = 0.9929$	confidence level
$\rho = 0.80$	proportion
μ_{MRE} = ~0.0051 (or 0.51%)	mean MRE
UB = 1%	upper-bound MRE

Based on trial and error (iterating though different values of ρ), it was determined that an upper-bound tolerance of 2% MRE could be achieved for 90% of the population with an actual confidence of 99.04%.[44]

B.2.7 Final Statistical Labels

Table 23 is a compilation of the data from the previous sections. All calculations assume a distribution-free tolerance interval.

Table 23: Final Statistical Label

Descriptive Statistics	
Basic Statistic	**Value**
Samples (N)	156
Mean MRE (μ_{MRE})	< 0.51%
Standard deviation (s)	< 0.01
Spearman rank correlation	0.998 p-Value < 0.001
One-Tail Confidence Interval	
Part of Interval	**Value**
confidence level (γ)	99.29%
proportion (ρ)	80%
upper bound (UB)	1%

B.3 Conclusions

The goal for the λ_{ABA} property theory that the latency for a job within the hyper-period can be predicted with an MRE ≤ 0.05 with a confidence level $\gamma = 0.99$ for at least 80% of the population was supported thorough empirical validation.

B.4 StInt Program

The StInt program contains the following code:

```
Line. 1 batch
Line. 2 c
Line. 3 c  # AVGMRE dataset
Line. 4 c
```

44. `dfti p=0.90 kside=2, conlev=0.99`

```
Line. 5 read 156 observations
Line. 6 0.001585018, 0.001572232, 0.001972813, 0.003262022,
Line. 7 0.000523292, 0.003594579, 0.000711926, 0.000598003,
Line. 8 0.003037226, 0.007755247, 0.001770896, 0.000898474,
Line. 9 0.001510966, 0.001353190, 0.015986690, 0.015718927,
Line. 10 0.016407781, 0.016252330, 0.000078766, 0.003702048,
Line. 11 0.013898231, 0.017387829, 0.016895966, 0.017573367,
Line. 12 0.003212414, 0.001936964, 0.003416712, 0.003302943,
Line. 13 0.000297025, 0.003661665, 0.000208086, 0.006911592,
Line. 14 0.008632193, 0.004217582, 0.001665116, 0.000058479,
Line. 15 0.000655536, 0.003715790, 0.000466558, 0.003633582,
Line. 16 0.004259913, 0.000000636, 0.002383395, 0.010831227,
Line. 17 0.001769257, 0.011213557, 0.030067181, 0.014128048,
Line. 18 0.080763095, 0.000919340, 0.000537448, 0.012560609,
Line. 19 0.015292302, 0.013816636, 0.014749198, 0.002935780,
Line. 20 0.001794847, 0.000486534, 0.000210741, 0.000737657,
Line. 21 0.001593223, 0.002684015, 0.000595428, 0.000087165,
Line. 22 0.002253364, 0.000507437, 0.000495287, 0.001793545,
Line. 23 0.000630091, 0.001122819, 0.003461565, 0.000700937,
Line. 24 0.001500666, 0.002506971, 0.000603722, 0.000386344,
Line. 25 0.000741155, 0.000607494, 0.000565105, 0.002087586,
Line. 26 0.000471663, 0.003384655, 0.020575129, 0.000364171,
Line. 27 0.000151106, 0.003216218, 0.007017182, 0.004680067,
Line. 28 0.000111689, 0.002079863, 0.002153729, 0.001776885,
Line. 29 0.005417615, 0.005790850, 0.002126783, 0.000784267,
Line. 30 0.005439613, 0.009277485, 0.010385405, 0.008128209,
Line. 31 0.005476482, 0.001798983, 0.001367929, 0.000714915,
Line. 32 0.000694798, 0.001847618, 0.001627236, 0.001172310,
Line. 33 0.000673987, 0.001844940, 0.002721988, 0.002081970,
Line. 34 0.006048433, 0.001002958, 0.007425308, 0.003119738,
Line. 35 0.001319701, 0.003799938, 0.000880306, 0.001230168,
Line. 36 0.000283496, 0.002756437, 0.001961042, 0.000379595,
Line. 37 0.000474834, 0.000686102, 0.001543520, 0.009819970,
Line. 38 0.014411690, 0.005339401, 0.002852634, 0.000294505,
Line. 39 0.003532777, 0.002380486, 0.001067884, 0.002603932,
Line. 40 0.000564931, 0.000697587, 0.000697463, 0.000838730,
Line. 41 0.000831076, 0.000668387, 0.006764835, 0.006755611,
Line. 42 0.006183170, 0.006244018, 0.006136309, 0.004551416,
Line. 43 0.000659660, 0.000595845, 0.006744618, 0.004108916,
Line. 44 0.006581670, 0.006233420, 0.077240774, 0.004580473
Line. 45 c
Line. 46 c  # Calculate mean and stdev
Line. 47 c
Line. 48 cndata
Line. 49 c
Line. 50 c  # distribution free one-tail tolerance interval
Line. 51 c  # for upper bound for 80% population.
Line. 52 c
Line. 53 dfti p=0.80 kside=2, conlev=0.99
Line. 54 c
Line. 55 c
Line. 56 c  # distribution free one-tail tolerance interval
```

```
Line. 57 c  # for upper bound for 90% population.
Line. 58 c
Line. 59 dfti p=0.90 kside=2, conlev=0.99
Line. 60 stop
```

Appendix C NuSMV Model of CSWI

This appendix contains the NuSMV model for the CSWI component described in Chapter 7.

```
MODULE main
IVAR
  input : {_opsel_on, _opsel_off, _oppos_open, _oppos_close,
sbosel, sbopos, none};
VAR
  output : {_sbosel_on, _sbosel_off, _sbopos_open,
_sbopos_close, opsel, oppos, none};
  state : {waiting, selecting, selected, opening, closing,
deselecting, unselecting};

ASSIGN
  init(input) := none;
  -- next(input) is unconstrained/non-deterministic

  init(output) := none;
  next(output) :=
    case
      state = waiting     & input = _opsel_on    : _sbosel_on   ;
      state = waiting     & input = _opsel_off   : _sbosel_off  ;
      state = selecting   & input = sbosel       : opsel        ;
      state = selected    & input = _opsel_on    : _sbosel_on   ;
      state = selected    & input = _opsel_off   : _sbosel_off
;
      state = selected    & input = _oppos_open  : _sbopos_open
;
      state = selected    & input = _oppos_close :
_sbopos_close;
      state = opening     & input = sbopos       : _sbosel_off  ;
      state = closing     & input = sbopos       : _sbosel_off  ;
      state = deselecting & input = sbosel       : oppos        ;
      state = unselecting & input = sbosel       : opsel        ;
      1 : none;
```

```
    esac;

  init(state) := waiting;
  next(state) :=
    case
      state = waiting     & input = _opsel_on    : selecting  ;
      state = waiting     & input = _opsel_off   : unselecting;
      state = selecting   & input = sbosel       : selected   ;
      state = selected    & input = _opsel_on    : selecting  ;
      state = selected    & input = _opsel_off   : unselecting;
      state = selected    & input = _oppos_open  : opening    ;
      state = selected    & input = _oppos_close : closing    ;
      state = opening     & input = sbopos       : deselecting;
      state = closing     & input = sbopos       : deselecting;
      state = deselecting & input = sbosel       : waiting    ;
      state = unselecting & input = sbosel       : waiting    ;
      1 : state;
    esac;

SPEC EF(state = waiting)
SPEC EF(state = selecting)
SPEC EF(state = selected)
SPEC EF(state = opening)
SPEC EF(state = closing)
SPEC EF(state = deselecting)
SPEC EF(state = unselecting)

SPEC AG((state = waiting) & (input = _opsel_on) -> AX(output =
_sbosel_on))
SPEC AG((state = selected) & (input = _opsel_off) -> AX(output
= _sbosel_off))
SPEC AG((state = selected) & (input = _oppos_open) -> AX(output
= _sbopos_open))
SPEC AG((state = selected) & (input = _oppos_close) -> AX(out-
put = _sbopos_close))

SPEC !E[!(output = _sbosel_on) U (output = _sbopos_open)]

-- the following claim is expected to fail
SPEC AG(input = _opsel_on -> AG output = none)
```

Appendix D Switch Schematic

The schematic for the SEI switch is shown in its entirety in Figure 61. Figures 62 through 65 show the same schematic in larger scale split into four quadrants for easier viewing. To get a copy of the original schematic, contact Kurt Wallnau via email at kcw@sei.cmu.edu.

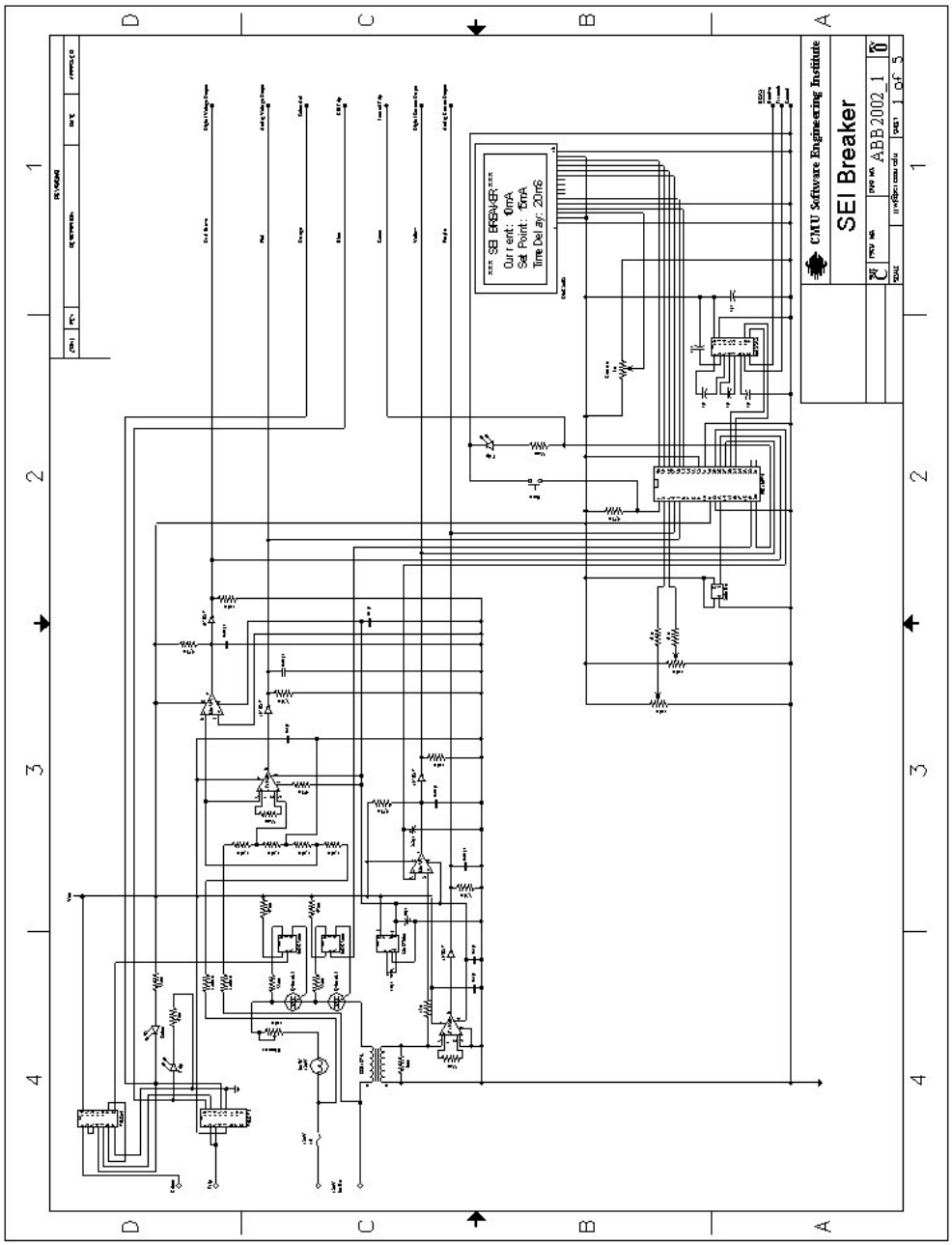

Figure 61: SEI Switch: Full Schematic

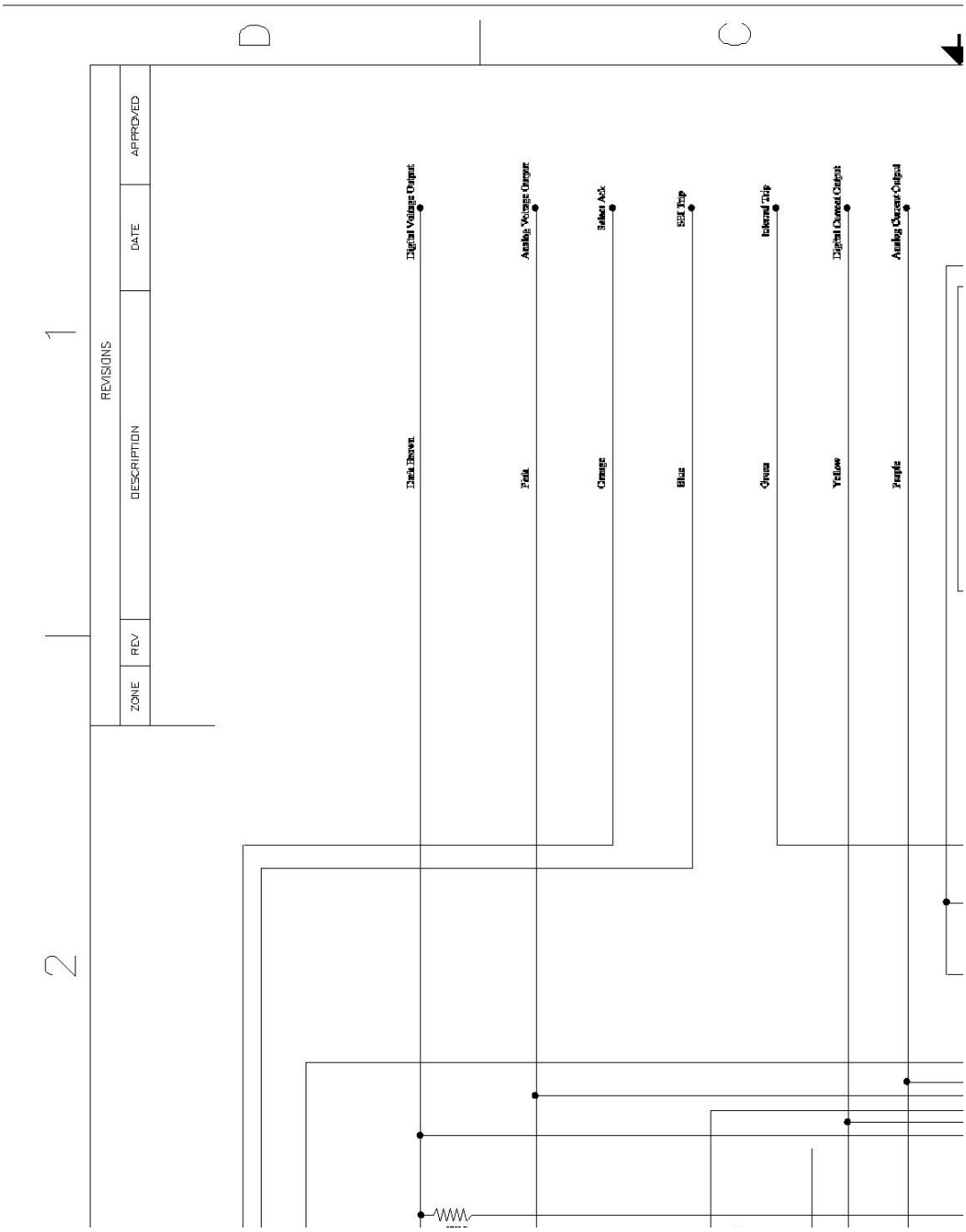

Figure 62: SEI Switch: Upper Right Quadrant

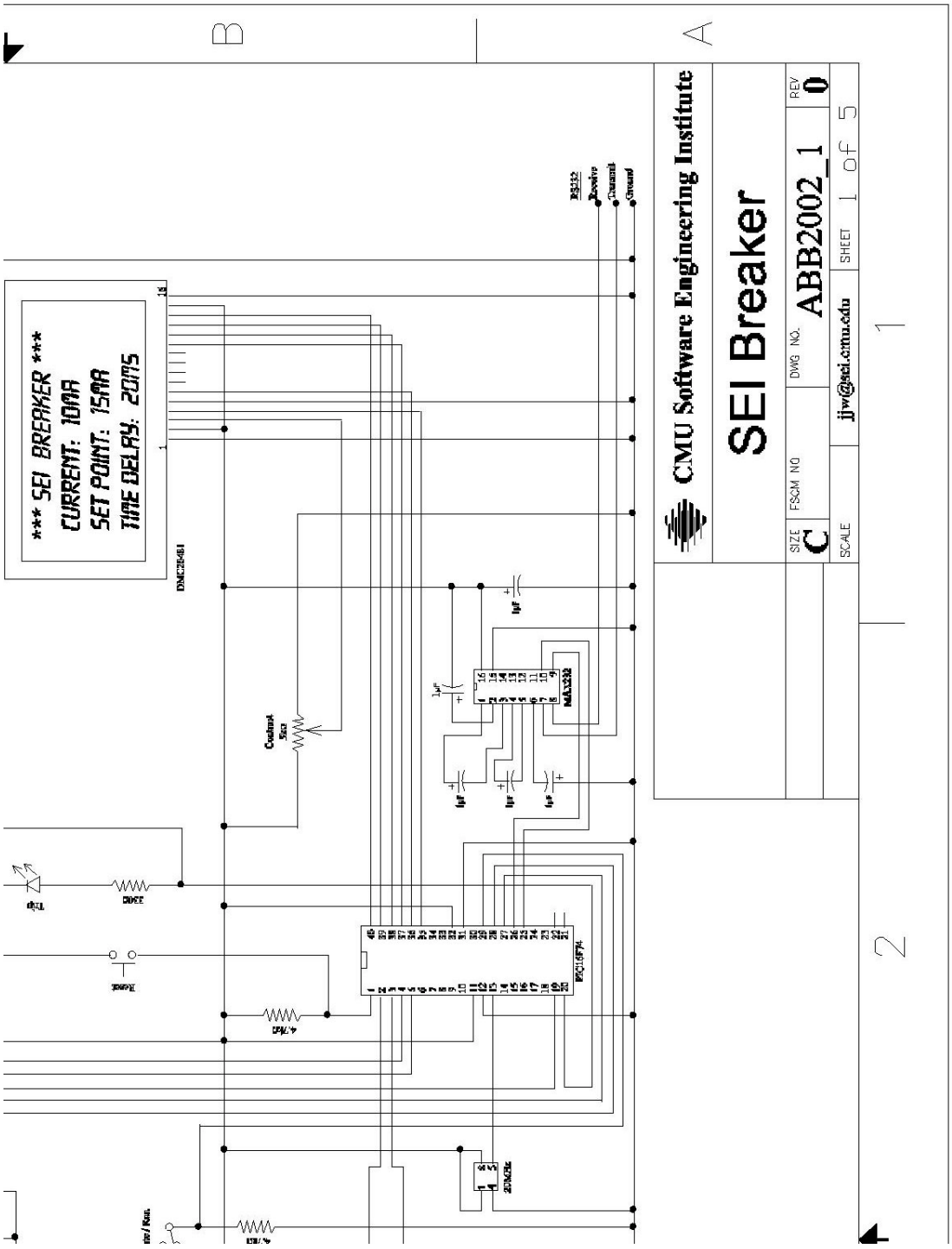

Figure 63: SEI Switch: Lower Right Quadrant

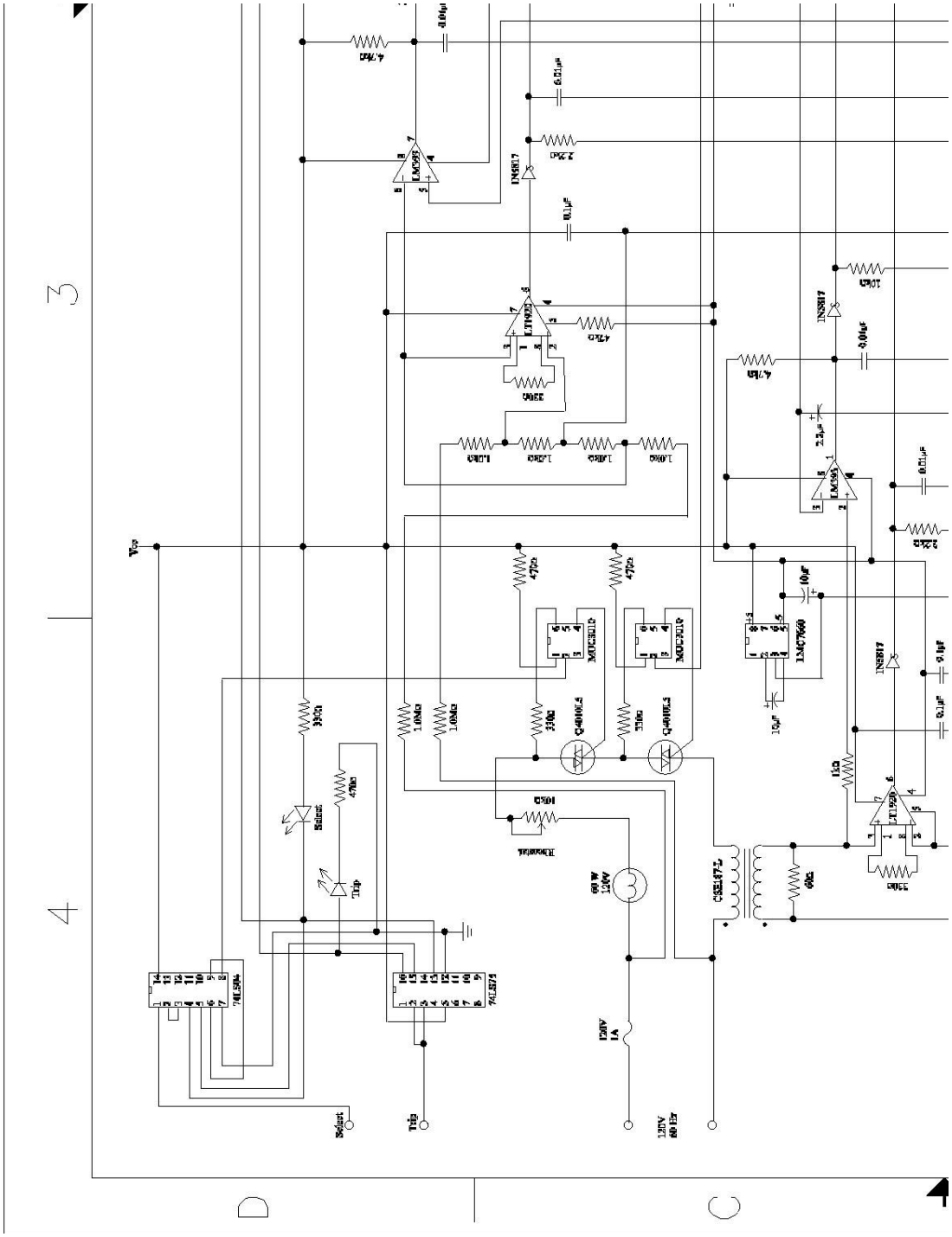

Figure 64: SEI Switch: Upper Left Quadrant

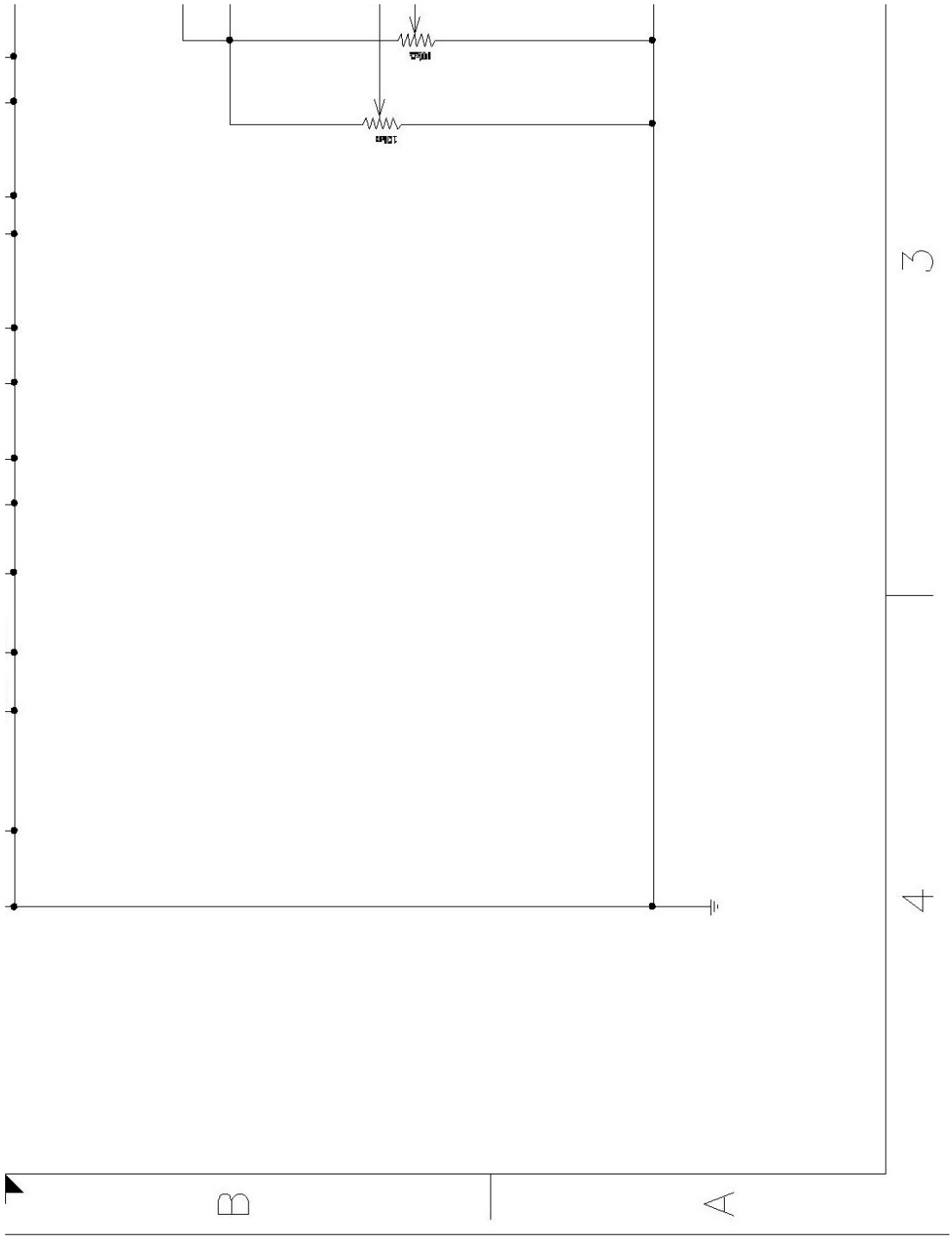

Figure 65: SEI Switch: Lower Left Quadrant

Acronym List

ABB/CRC	ABB Ltd. Corporate Research Center
AIP	Aspect Integration Program
API	application program interface
COTS	commercial off-the-shelf
CPU	central processing unit
CTL	computational tree logic
DoD	Department of Defense
EMS	energy management system
FIFO	first in, first out
HP	hyper-period
IEC	International Electrotechnical Commission
IED	intelligent electronic device
I/O	input/output
LCM	least common multiple
LIFO	last in, first out
LN	logical node
LS-SDE	language-specific software development environment
LTL	linear temporal logic

MRE	magnitude of relative error
MS	milliseconds
NS	nanoseconds
OLE	object linking and embedding
OPC	OLE (object linking and embedding) for process control
OS	operating system
PACC	predictable assembly of certifiable components
PC	physical connection
PD	physical device
PECT	prediction-enabled component technology
RMA	rate monotonic analysis
RUP	Rational Unified Process
SAS	substation automation system
SCADA	supervisory control and data acquisition
SDE	software development environment
SEI	Software Engineering Institute
TBD	to be decided
UDP	Universal Datagram Protocol
UML	Unified Modeling Language
XML	Extensible Markup Language

Bibliography

URLs valid as of the publication date of this document

[Aho 87] Aho, A.; Sethi, R.; & Ullman, J. *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1987.

[Allen 97] Allen, R. "A Formal Approach to Software Architecture." PhD diss. (CMU-CS-97-144), Carnegie Mellon University, May 1997.

[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>.

[Ball 02] Ball, T. & Rajamani, S. K. "Automatically Validating Temporal Safety Properties of Interfaces," 103-122. *Proceedings of the Workshop on Model Checking of Software* (LNCS 2057). Toronto, Ontario, Canada, May 19-20, 2001. New York, NY: Springer-Verlag, 2002.

[Barnett 01] Barnett, M. & Schulte, W. "Spying on Components: A Runtime Verification Technique," 7-13. *Proceedings of the Workshop of Specification and Verification of Component-Based Systems (SAVCBS) at OOPSLA*. Tampa, Florida, October 14, 2001. New York, NY: Association for Computing Machinery, 2001. <http://www.cs.iastate.edu/~leavens/SAVCBS/papers-2001/barnett-schulte.pdf> or <http://research.microsoft.com/foundations/savcbs.pdf>.

[Bass 98] Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*. Boston, MA: Addison-Wesley, 1998.

[Buskins 02] Buskins, V. *Social Networks and Trust*. Boston, MA: Kluwer Academic Publishers, 2002.

[CDIF 97] Electronic Industries Association. *CDIF CASE Data Interchange Format—Overview.* <http://www.eigroup.org/cdif /electronic-extracts/OV-extract.pdf> (1997).

[Cimatti 00] Cimatti, A.; Clarke, E.; Giunchiglia, F.; & Roveri, M. "NuSMV: A New Symbolic Model Verifier." *International Journal on Software Tools for Technology Transition 2*, 4 (April 2000): 410-425. <http://nusmv.irst.itc.it/NuSMV/papers/sttt_j/html/index.html>.

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

[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.

[Davies 93] Davies, J. *Specification and Proof in Real-Time CSP.* New York, NY: Cambridge University Press, 1993.

[Fenton 97] Fenton, N. E. & Pleeger, S. L. *Software Metrics A Rigorous and Practical Approach.* Boston, MA: PWS Publishing Company, 1997 (ISBN 1-85032-275-9).

[Gardiner 00] Gardiner, P.; Goldsmith, M.; Hulance, J.; Jackson, D.; Roscoe, B.; & Scattergood, B. *Failures-Divergence Refinement: FDR2 User Manual*. Oxford, England: Formal Systems (Europe) Ltd., 2000.

[Garlan 93] Garlan, D. & Shaw, M. "An Introduction to Software Architecture," 1-39. *Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering, Advances in Software Engineering and Knowledge Engineering*. Capri, Italy, June 15-20, 1992. River Edge, NY: World Scientific Publishing Company, 1993.

[Giannakopoulou 99] Giannakopoulou, D. “Model Checking for Concurrent Software Architectures.” PhD diss., Imperial College of Science, Technology, and Medicine, University of London, 1999.

[Gluch 02] Gluch, D.; Comella-Dorda, S.; Hudak, J.; Lewis, G.; Walker, J.; Weinstock, C.; & Zubrow, D. *Model-Based Verification: An Engineering Practice* (CMU/SEI-2002-TR-021, ADA407768). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. <http://www.sei.cmu.edu/publications /documents/02.reports/02tr021.html>.

[Gorton 02] Gorton, I. & Liu, A. “Software Component Quality Assessment in Practice: Successes and Practical Impediments,” 555-558. *Proceedings of the 24th International Conference on Software Engineering*. Orlando, Florida, May 19-25, 2002. New York, NY: Association for Computing Machinery, 2002.

[Hahn 91] Hahn, G. & Meeker, W. *Statistical Intervals: A Guide for Practitioners*. New York, NY: John Wiley & Sons, Inc., 1991.

[Havelund 00] Havelund, K. & Pressburger, T. “Model Checking Java Programs Using Java PathFinder.” *International Journal on Software Tools for Technology Transfer (STTT) 2,* 4 (April 2000): 366-381.

[Hissam 02] Hissam, S.; Moreno, G.; Stafford, J.; & Wallnau, K. “Packaging Predictable Assembly,” 108-124. *Proceedings of the 1st ACM/ IFIP Working Conference on Component Deployment* (LNCS 2370). Berlin, Germany, June 20-21, 2002. Berlin, Germany: Springer-Verlag, 2002.

[Hoare 85] Hoare, C. A. R. *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice Hall International, 1985.

[Holzman 97] Holzman, G. J. “The Model Checker Spin.” *IEEE Transactions on Software Engineering 23*, 5 (May 1997): 279-295.

[IEC 02] International Electrotechnical Commission. Communications Networks and Systems in Substations, Working Draft for International Standard IEC 61850-1..10. Geneva, Switzerland: International Electrotechnical Commission, 2002.

[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>.

[Ivers 03] Ivers, J. & Wallnau, K. "Preserving Real Concurrency," 15-22. *Proceedings of the Correctness of Model-Based Software Composition (CMC) Workshop* (2003-13), in conjunction with the 17th European Conference on Object-Oriented Programming (ECOOP). Darmstadt, Germany, July 22, 2003. Karlsruhe, Germany: Universitat Karlsruhe, 2003.

[Kazman 00] Kazman, R.; Klein, M.; & Clements, P. *ATAM: Method for Architecture Evaluation* (CMU/SEI-2000-TR-004, ADA382629). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. <http://www.sei.cmu.edu/publications/documents/00.reports/00tr004.html>.

[Klein 93] Klein, M.; Ralya, H.; Pollack, B.; & Obenza, R. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston, MA: Kluwer Academic Press, 1993.

[Kruchten 00] Kruchten, P. *The Rational Unified Process An Introduction,* 2nd edition. Boston, MA: Addison-Wesley, 2000.

[Lamport 77] Lamport, L. "Proving the Correctness of Multiprocess Programs." *IEEE Transactions on Software Engineering SE 3,* 2 (March 1997): 125-143.

[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).

[Magee 93] Magee, J.; Dulay, N.; & Kramer, J. “Structured Parallel and Distributed Programs.” *Software Engineering Journal 8*, 2 (March 1993): 73-82.

[Magee 99] Magee, J. & Kramer, J. *Concurrency State Models & Java Programs*. New York, NY: Wiley, 1999.

[Mason 02] Mason, D. “Probabilistic Analysis for Component Reliability Composition.” *Proceedings of the 5th International Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly*, in conjunction with the 24th International Conference on Software Engineering (ICSE2002). Orlando, Florida, May 19-20, 2002. <http://www.sei.cmu.edu/pacc/CBSE5/Mason-cbse5-final.pdf>.

[Meeker 93] Meeker, W. & Chow, I. *StInt—A Computer Program for Computing Statistical Intervals*. <http://www.public.iastate.edu/~wqmeeker/StInt/StInt.pdf> (1993).

[Meyer 98] Meyer, B.; Mingins, C.; & Schmidt, H. “Providing Trusted Components to the Industry.” *Computer (IEEE) 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, 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>.

[Naumovich 00] Naumovich, G. & Clarke, L. “Classifying Properties: An Alternative to the Safety-Liveness Classification,” 159-168. *Proceedings of the Eighth International Symposium on the Foundations of Software Engineering.* San Diego, California, November 8-10, 2000. New York, NY: Association for Computing Machinery, 2000.

[Plakosh 99] Plakosh, D.; Smith, D.; & Wallnau, K. *Builder's Guide for Water-Beans Components* (CMU/SEI-99-TR-024, ADA373154). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. <http://www.sei.cmu.edu/publications/documents/99.reports/99tr024/99tr024abstract.html>.

[Pnueli 85] Pnueli, A. & Harel D. *On the Development of Reactive Systems* (Technical Report CS85-02). Rehovot, Israel: Weizmann Institute of Science, Department of Computer Science, 1985.

[Preiss 01] Preiss, O. & Wegmann, A. *Towards a Composition Model Problem Based on IEC61850*. <http://www.sei.cmu.edu/pacc/CBSE4_papers/PreissWegmann-CBSE4-4.pdf> (2001).

[Prior 62] Prior, Arthur N. *Formal Logic*, 2nd ed. Oxford, UK: Clarendon Press, 1962.

[Rabinovich 00] Rabinovich, S. G. *Measurement Errors and Uncertainties*, 2nd ed. New York, NY: Springer-Verlag, 2000 (ISBN 0-387-98835-1).

[Reed 88] Reed, G. M. “A Uniform Mathematical Theory for Real-Time Distributed Computing.” PhD diss., Oxford University, 1988.

[Roscoe 98] Roscoe, A. W. *The Theory and Practice of Concurrency.* New York, NY: Prentice Hall, 1998.

[Rumbaugh 00] Rumbaugh, J.; Jacobson, I.; & Booch, G. *The Unified Modeling Language Reference Manual.* Reading, MA: Addison-Wesley Longman, 2000.

[Schubert 99] Schubert, P. & Ginsburg, M. “Virtual Communities of Transaction: The Role of Personalization in Electronic Commerce,” 646-663. *12th International Bled Electronic Commerce Conference: “Global Networked Organizations.”* Bled, Slovenia, June 7-9, 1999. Bled, Slovenia: Moderna Organizacija, 1999.

[Schwetman 78] Schwetman, H. “Hybrid Simulation Models of Computer Systems.” *Communications of the ACM 21*, 9 (September 1978): 718-723.

[Sharygina 02] Sharygina, N. “Model Checking of Software Control Systems.” PhD diss., The University of Texas at Austin, 2002.

[Stafford 01] 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. D. “Issues in Predicting the Reliability of Composed Components.” *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 /StaffordMcGregor-cbse5.pdf>.

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

[Taylor 94] Taylor, B. & Kuyatt, C. *Guidelines for Evaluating and Expressing the Uncertainty of NIST Measurement Results* (microform, NIST Technical Note 1297). Gaithersburg, MD: U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1994.

[Theil 02] Thiel, S. & Hein, A. “Modeling and Using Product Line Variability in Automotive Systems.” *IEEE Software, Special Issue on Software Product Lines 19,* 4 (July/August 2002): 66-72.

[van Glabbeek 01] van Glabbeek, R. Part 1, Ch. 1, "The Linear Time-Branching Time Spectrum I. The Semantics of Concrete, Sequential Processes," 3-99. *Handbook of Process Algebra*. New York, NY: Elsevier, 2001 (ISBN 0-444-82830-3). <http://adam.wins.uva.nl/~alban/Handbook-free>.

[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 02] Wallnau, K.; Hissam, S.; & Seacord, R. *Building Systems from Commercial Components*. Boston, MA: Addison-Wesley, 2002.

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

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

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

[Woodcock 97] Woodcock, J. & Davies, J. *Using Z: Specification, Refinement, and Proof*. New York, NY: Prentice Hall, 1997.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

10. AGENCY USE ONLY (leave blank)	11. REPORT DATE	12. REPORT TYPE AND DATES COVERED
	September 2003	Final

13. TITLE AND SUBTITLE

Predictable Assembly of Substation Automation Systems: An Experiment Report, Second Edition

14. FUNDING NUMBERS

F19628-00-C-0003

15. AUTHOR(S)

Scott Hissam, John Hudak, James Ivers, Mark Klein, Magnus Larsson, Gabriel Moreno, Linda Northrop, Daniel Plakosh, Judith Stafford, Kurt Wallnau, William Wood

16. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

17. PERFORMING ORGANIZATION REPORT NUMBER

CMU/SEI-2002-TR-031

18. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

19. SPONSORING/MONITORING AGENCY REPORT NUMBER

ESC-TR-2002-031

20. SUPPLEMENTARY NOTES

12.a DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified/Unlimited, DTIC, NTIS

12.b DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

The Predictable Assembly from Certifiable Components (PACC) Initiative at the Software Engineering Institute (SEI[SM]) is developing methods and technologies for predictable assembly. A software development activity that builds systems from components is *predictable* if the runtime behavior of an assembly of components can be predicted from known properties of components and their patterns of interactions (connections), and if these predictions can be objectively validated. A component is *certifiable* if these known properties can be obtained or validated by independent third parties. The SEI's technical approach to PACC rests on prediction-enabled component technology (PECT). At the highest level, PECT is a scheme for systematic and repeatable integration of software component technology, software architecture technology, and design analysis and verification technology. This report describes the results of an *exploratory* PECT prototype for substation automation, an application area in the domain of power generation, transmission, and management. This report focuses primarily on the methodological aspects of PECT; the prototype itself was only a means to expose and illustrate the PECT method.

14. SUBJECT TERMS

component-based software, predictable assembly, component technology, certification, component framework, CBSE, prediction-enabled component technology

15. NUMBER OF PAGES

178

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL