Mälardalen University Press Dissertations
No. 85

PREDICTABILITY BY CONSTRUCTION

WORKING THE ARCHITECTURE/PROGRAM SEAM

Kurt C. Wallnau

2010

MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering

ISSN 1651-4238
ISBN 978-91-86135-80-5
Printed by Mälardalen University, Västerås, Sweden

Mälardalen University Press Dissertations
No. 85

PREDICTABILITY BY CONSTRUCTION
WORKING THE ARCHITECTURE/PROGRAM SEAM

Kurt C. Wallnau

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid Akademin för innovation, design och teknik kommer att offentligen försvaras torsdagen den 30 september, 2010, 14.00 i Alpha, Mälardalens högskola, Västerås.

Fakultetsopponent: Dr. Clemens Szyperski, Microsoft

MÄLARDALEN UNIVERSITY
SWEDEN

Akademin för innovation, design och teknik

Abstract

Contemporary software engineering practice overemphasizes the distinction of software *design* from software *implementation*, and *designer* (“software architect”) from *implementer* (“computer programmer”).

In this contemporary meme, software architects are concerned with large-grained system structures, the quality attributes that arise from these structures (security, availability, performance, etc.) and with tradeoffs among quality attributes; programmers are concerned with low--level algorithms and data structures, program functionality, and with satisfying architectural intent. However, software design and implementation are not cleanly separable. While architect and programmer may have many different design concerns, they also have many complementary concerns; their respective design practices must be better integrated than is the case in contemporary practice.

The research reported here defines the *Architecture/Program Seam* (“the Seam”), a region of overlap in software architecture and programming practice. The Seam emphasizes design concerns centered on predictable runtime behaviour. For behaviour to be predictable it must be described by a *computational theory*, and each such theory must provide objective evidence to demonstrate that theory predictions correspond to system observations. The validity of a theory will likely depend on invariants that can be expressed, and enforced, by means of *design rules*. A system that satisfies the design rules of a theory is then regarded as having behaviour that is *predictable by construction* with respect to that theory.

The research reported here also introduces and defines *prediction--enabled component technology* (PECT) as a foundation technology to support the Seam, and demonstrates a prototype PECT on industrial problems in electric grid substation control, industrial robot control, and desktop streaming audio. The prototype PECT extends a basic component technology of pure assembly (Pin) with theory extension points (*reasoning* frameworks) that are used to achieve predictability by construction. Reasoning frameworks for real--time performance and temporal--logic model checking are described, with statistical confidence intervals providing evidence of predictive quality for the former, and code--embeddable proof certificates providing evidence for the latter.

Finally, the research reported here defines the Seam itself as inducing a new kind of evolutionary design problem, whose solutions require the integration of programming language theory, design theory, specialized theories of system behaviour and deep systems expertise.

ISSN 1651-4238

ISBN 978-91-86135-80-5

Predictability By Construction: Working the Architecture/Program Seam

Kurt C. Wallnau,
Mälardalen University
School of Innovation, Design and Technology.
Software Engineering Division
Västerås, Sweden
and
Software Engineering Institute,
Carnegie Mellon University
Pittsburgh, USA

September 15, 2010

Acknowledgements

I am grateful for the contributions of many colleagues, without which this research would not have succeeded, and perhaps would not have been undertaken in the first place. Scott Hissam, Daniel Plakosh, Gabriel Moreno, James Ivers, Magnus Larsson and Sagar Chaki made fundamental contributions to the work reported here—to the theory of predictability by construction, the development of a prototype prediction–enabled component technology, and to its demonstration on non–trivial industrial problems. Judy Stafford, Mark Klein, Len Bass and Paul Clements provided early guidance on this research, gently steering it towards what was later recognized to be called the architecture/program seam. Mark Klein deserves special credit for also bringing his deep expertise in real–time scheduling theory to bear in the development of the $\lambda*$ reasoning framework. My acknowledgement of each of your specific technical contributions in Chapter 2 is no doubt inadequate, but to each of you I offer my sincerest personal thanks, for whatever that is worth.

I also owe a great debt of gratitude to the Software Engineering Institute, Carnegie Mellon University, for supporting this research. In particular, I wish to express my enduring thanks to Linda Northrop, my mentor and supervisor at the SEI, for supporting and encouraging me to pursue doctoral studies at MDH as an adjunct to my research tasks at the SEI, even when my academic pursuits impinged on those tasks.

Of course, as with every other PhD candidate, I owe a special debt to my faculty advisor, Ivica Crnkovic. I am sure that I have caused Prof. Crnkovic more than a little anxiety over the past few years; it is a great mystery how he managed to maintain his faith in me all this time (even when I did not).

Lastly, and most importantly, I owe more than can be expressed to my wife and best friend forever, Jeannemarie. It has been a long and circuitous road from Providence to Västerås, a journey that would not have been completed without your patience and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

All engineering disciplines use "divide and conquer" as a fundamental problem solving strategy—to decompose large problems into smaller (component) problems, and to compose (component) solutions of smaller problems into larger solutions. To the extent that software engineering is concerned with software problems and software solutions, it will also be concerned with software components. Although a more specific interpretation of software component is developed later in this thesis, for the present it is sufficient to note that by "software" I mean *computer programs*; and by "components" I mean the constituent parts of those programs.

It may seem obvious that computer programs are of essence to the discipline of software engineering. There is, however, an unfortunate and widespread tendency in software engineering theory and practice to regard programming as a routine production activity, and programmers (at best) as the equivalent of a skilled shop foremen or machinists, but nonetheless mostly concerned with filling in the details of some software engineer's design.

This research argues that disassociating programming practice from software engineering practice, and in particular from software architecture practice, is both artificial and a self-defeating. It is artificial because there are no criteria that usefully distinguish the design of computer programs from their implementation. It is self–defeating because architects and programmers, to the extent that they do have different design concerns, also have complementary concerns. If software engineers expect to produce software that has predictable and acceptable quality, then software architecture and computer programming practices must exhibit an overall integrity.

This research also provides a sound but practical demonstration that:

- A region of shared and reciprocal concerns between architectural design and program design, the *architecture/program seam* can be exposed and formalized.
- The seam can be substantially automated with *prediction-enabled component technology.*
- An automated seam permits the development of systems whose behavior is *predictable by construction.*

The remainder of this introduction is structured as follows. Section 1.1 traces the disassociation of software engineering from programming to the origins of software engineering, and discusses the symptoms of this defect in engineering practice. Section 1.2 postulates the Seam, Prediction–Enabled Component Technology, and Predictability By Construction as a way of repairing the defect (or at worst, gradually mitigating its worst symptoms). The questions addressed by this research are identified in Section 1.3, the main contributions are outlined in Section 1.4, and related work is described in Section 1.5. The approach taken to conduct the research is described in

Section 1.6, and key assumptions of the approach are detailed in Section 1.7. Finally, the structure of the remainder of the dissertation is outlined in Section 1.8.

1.1 Software Engineering's Genetic Defect

The birth of software engineering can be traced to the 1968 NATO-sponsored working conference on software engineering [114]. Given the earlier comments about divide and conquer, it is not surprising that the birth of software components can also be traced to this conference, and in particular to M.D. Mcilroy's keynote, " 'Mass Produced' Software Components." Mcilroy's remarks are interesting as an historical marker in the development of component-based approaches to software development, and also because they anticipate a number of topics that are pertinent even today on component variability, quality, testing, standardization, markets, and distribution. However, it is his remarks about the role of programmers in software engineering that are of particular interest to this thesis, because they reveal at this earliest "genetic" stage of the development of software engineering the workings of a faulty premise:

> "What I have just asked for is simply *industrialism,* with *programming* terms substituted for some of the more *mechanically* oriented terms appropriate to *mass production.*"[1]

In Mcilroy's vision of industrialized software *component factories*, a term he also coined in his remarks, programming occupies a niche somewhere between skilled craft and unskilled assembly-line work—a connotation reinforced by his association of programming with mechanization and mass production. He argued that industrial–scale production processes, and by implication Taylor*esque* theories of scientific industrial management [163, 162] that attend industrial-scale production, are the only viable way to achieve the persistent increases in programmer productivity and software quality that are needed to meet ever-increasing societal demands for software. Indeed, some have come to identify industrial software–engineering process improvement initiatives with software engineering as a whole.

The factory metaphor is without doubt compelling, and has survived and been adapted well beyond Mcilroy's original vision, as can be seen not only from Cusamano's now dated survey [47] but also by more recent publications [164, 100]. However, these most recent works on software factories are far more respectful of programming practice than Mcilroy, and bear little resemblance to the factories found in the pages of the NATO workshop proceedings. Yet the genetic defect is not located in Mcilroy's factory metaphor *per se*; that is just one symptom of the defect. Rather, the defect lies in

[1]Emphasis added.

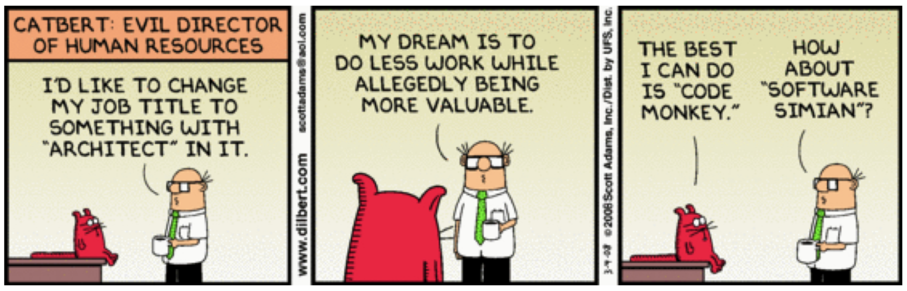

CATBERT: EVIL DIRECTOR OF HUMAN RESOURCES
I'D LIKE TO CHANGE MY JOB TITLE TO SOMETHING WITH "ARCHITECT" IN IT.
MY DREAM IS TO DO LESS WORK WHILE ALLEGEDLY BEING MORE VALUABLE.
THE BEST I CAN DO IS "CODE MONKEY."
HOW ABOUT "SOFTWARE SIMIAN"?
www.dilbert.com
scottadams@aol.com
© 2008 Scott Adams, Inc./Dist. by UFS, Inc.

Figure 1.1: Parody Reflects Reality

the implicit and yet wholly artificial distinction between the *design* of software and its *implementation*, and, by natural progression, *programmers* as *implementors* rather than as designers.

1.1.1 Consequences of the Defect

There are social as well as a technical aspects of all engineering disciplines, and software engineering's genetic defect has adverse impact on both.

Modern society, with its increasing dependence on digital technology, cannot be well served by a software engineering practice that incorrectly regards programming as a menial task best delegated (or relegated) to low-skilled assembly line workers. This artificial class structure, parodied in a widely-circulated Dilbert comic strip (Figure 1.1, reproduced with permission), risks creating an engineering culture that will lose contact with the fast-evolving software technologies that society depends upon, at which point software engineering will become an engineering discipline in name only. Conversely, the same class structure risks creating a programming culture (which arguably already exists) that regards the ethos of discipline and social responsibility that attends any engineering discipline as irrelevant and foreign. Perhaps it was just such consequences that led Edsger Dijkstra, another luminary of the NATO conference, to later disown the engineering discipline that he helped midwife:

> "Software engineering, of course, presents itself as another worthy cause, but that is eyewash: if you carefully read its literature and analyze what its devotees actually do, you will discover that software engineering has accepted as its charter 'how to program if you cannot'."[48].

Today, technologies that support architecting and programming exist in isolation of one another, with architecture description languages continuing to evolve, but without widespread impact on programming practice; the converse is true for programming languages and environments.

Table 1.1: Recent Manifestations of Software Engineering's Genetic Defect

Misconception	Seam Conception
The use of system models is the hallmark of engineering disciplines, and mature software engineering practice is concerned with models, not with computer programs. Models encode abstracted essence and their use reflects rigor and discipline; programs encode myriad fussy details, undisciplined hacks, and are generally "wrong."	All functional and non-functional runtime behavior is a consequence of computational processes, therefore all models of non-functional behavior must be an interpretation of some computer program. Computer programs are models, and language semantics provide interpretations of programs to computational models.
Architects design software systems; programmers implement those designs. Architects are concerned with design processes; programmers are concerned with production processes. Architects are polymaths and possess renaissance skills; programmers are unidimensional and possess readily substitutable skills.	Programming is fundamentally a design activity. Architects and programmers operate on a design continuum; they differ in their concerns, the criteria and theories they use to address these concerns, the design artifacts they manipulate, and the strictures they follow to maintain intellectual control of design problems.

1.1.2 Recent Symptoms of the Defect

Many attempts have been made to link software architecture to computer program; however, they have done so in a way that is generally "one-sided," i.e., in ways that tend to perpetuate the genetic defect because of the tacit devaluation of programming and programmers. *Model–based engineering* (MBE) is one widely known contemporary attempt; the emergence of "professional" software architects is another. While both attempts have merits, each is limited by the false dichotomy of "design" and "implementation."

Table 1.1 highlights, in exaggerated form, the philosophical positions adopted by advocates of MBE and architecture primacy, along with parallel concepts found in the results reported here. It is not the purpose of this research to engage in polemics, and the positions staked out in Table 1.1 are exaggerated (though not, I claim, to the point of reductio ad absurdum). Rather, the positions are constructed to highlight the "slippery slope" that leads from model–based software engineering and architecture primacy to the mythical software factory.

Automated model transformation is quite central to the research reported here, and so a few words about the first row in Table 1.1 are in order. Part of what makes this slope especially slippery is that model transformation is the essence of software development. What is sometimes forgotten is that computer programs written in *any* language other than bare machine code are themselves models, and on this ground alone the term "model–based" seems quite redundant.

However, the idea of regarding programs as "low–level" models to be automatically generated from “high–level” models is itself quite reasonable, though of course different forms of high–level model lead to different what constitutes “high–level.” Application–specific languages (also known as domain specific languages) are clearly one form that MBE can take, although the success of these approaches requires a well–defined and often quite narrow “domain of discourse.” Using design notations such as UML as “high–level” models is less well motivated because the model–to–code transformations in these cases are largely a matter of syntax.

One justification for MBE is that the vast majority of programming decisions are routine, and further that a substantial and possibly growing portion of these routine programming decisions are also mundane to the point where they can be mechanized. Automated garbage collection in place of programmer-controlled memory management is but one of many well-established examples; the advent of multi-core architecture is leading to analogous mechanization of concurrency management.

However, such arguments do not challenge the assertion of programming as a design activity. After all, a growing collection of architectural styles and patterns is evidence of routinization of architectural decisions, and adaptive middleware technologies [102] suggest that even routine architecture decisions can prove to be mundane to the point where they, too, can be mechanized. Indeed, routinization and mechanization are essential contributors to improving software engineering practice; such improvements do not reduce design to fabrication, but they do allow an incrementally sharper focus on *essential* rather than *accidental* software engineering design problems [23].

MBE approaches that use architectural design notations such as AADL [58] as syntactic scaffolding for analysis models are, in principle, quite consistent with the work reported here. Also consistent with the research reported here are approaches that extend programming languages with specialized annotations and associated static analysis tools [159], or through “first–class” extensions of the programming language syntax and semantics to incorporate architectural structures [5]. Both approaches may provide a basis for adopting the results of the work reported here, which favors neither the architecture nor program abstractions but instead seeks to find common ground for both.

1.2 Repairing the Defect

To properly motivate the research reported in this thesis, it is first necessary to justify why software design can not be cleanly distinguished from software implementation, or software designers from programmers. This, in turn, leads to notions of the Architecture/Program Seam and the technologies and practices that operate within the Seam.

1.2.1 Programmers Are Designers

Programming is problem solving, and from the initial empty edit buffer to the final compilation, successive changes to a computer program—the *design artifact*—require a programmer to choose from among many possible consequent solutions (the programs). For non-trivial programming tasks, the total set of choices that must be made—the *design space*—is vast and therefore effectively unbounded. Each choice is made to maximize the program's *fitness for use* with respect to various qualities desired of the program solution, such as functionality, efficiency and modifiability. Interactions among these qualities can be subtle, and choices have consequences on a solution's fitness that can be difficult for even the most experienced programmer to appreciate, let alone anticipate. Experienced programmers therefore employ an iterative *generate and test* problem solving strategy, generating new versions of a program and testing the fitness of these with respect to desired solution qualities. Versions that exhibit adequate fitness become the starting point for the subsequent iteration, or they (or preceding versions) may be abandoned in favor of a new approach to the problem. In effect, programmers *search a fitness landscape.* Writing computer programs, even small ones, is a design activity in every sense of the word.

To argue that programming *is* design is not the same as arguing that the techniques of computer programming are sufficient to address the design challenges posed at the scale of *systems.* By "system" I mean a design artifact that composes many computer programs, and that in the aggregate must meet the needs of many end-users and their institutions, each of which will seek different, quite possibly competing and almost certainly evolving end objectives. The traditional, and quite concrete techniques of programming are not sufficient to tackle design problems at this scale. Therefore, this research takes as one of its foundations the discipline of software architecture. Perry and Wolf [135] and Shaw and Garlan [150] are deservedly credited with laying the foundations for research in *software architecture* and in the *architectural design* of systems, and they largely agreed on several key points:

- Architects are concerned with the large-grained structure and organization of systems.

- Global system properties, variously referred to as *non*-functional or *extra*-functional qualities, or sometimes as *quality attributes*, are established by this architectural structure.

- *Architecture description languages* (ADLs) are required to express these structures and to reason about quality attributes.

- ADL abstractions must be formally linked to programming language abstractions.

Thus, the "founders" of software architecture research regarded architectural design and program design as attending to different regions of an overall shared design space. It should not be controversial to observe that these different regions require different design techniques, and that some degree of professional specialization among designers will naturally arise for these regions (as we have seen, architects and programmers, respectively), and even niches within regions (e.g., security, safety, performance experts). The image of software development that arises from this discussion is that of a design activity involving the coordination of many specialized design skills—and this is far from factory automation.

1.2.2 The Architecture/Program Seam

As previously established, architects and programmers have distinct and therefore separable concerns. However, repairing software engineering's genetic defect requires an examination of architecture and programming practice from a design-theoretic frame of reference. From this, a more detailed understanding of these distinction concerns can be obtained. Of particular interest are those design concerns that are *complementar* or *dual* architecture and program concern, because these imply a basis on which to establish common ground for architects and programmers. This common ground is called the Architecture/Program Seam (hereafter: "the Seam"), which is compactly summarized in Table 1.2.

To illustrate the seam, consider the first *Concerns* row of Table 1.2. Architects are primarily (though not exclusively) concerned with achieving *satisficing* non-functional effects. Satisficing is a portmanteau of "sufficient" and "satisfy" coined by Herb Simon; the term reflects uncertainty in the design process arising from limitations in the designer's understanding of the problem or the effect of design decisions on solutions. Programmers are primarily (though not exclusively) concerned with achieving *correct* functional effects. While "correct" can be interpreted as satisfying a specification that might also include non-functional concerns such as performance (as an arbitrary example), programmers are likely to approach the problem in terms of first achieving correct functionality, and only then ensuring that the function completes in required time. The Seam defines common ground by emphasizing *predictable* rather than *satisficing* or *correct* effects, and by restricting its focus to *computational results*, which are any observable runtime behavior however one chooses to classify them (as functional, non–functional, extra–functional, etc.).

This research offers no formal criteria with which to demonstrate that the Seam constitutes an optimal common ground for architects and programmers, i.e., in the way it reconciles each complementary architecture and programming concern, or in the overall selection of those concerns. The demonstration is, like engineering itself, pragmatic: it is grounded in practi-

Table 1.2: The Architecture/Program Seam

	Architect	Programmer	THE SEAM
Concerns	satisficing results for all attributes in all environments of use	correct computational results in the operational environment	predictable computational results in the operational environment
Theories	many attribute criteria, theories span rules of thumb to formal bases	one dominant attribute criterion, established theory of computation	extensible theories of runtime behavior that are statistically or formally validated
Rules	open-ended policy-enforced design rules; tacit or asserted intent	pre-defined language syntax and semantics; functional behavior by construction	extensible computer-enforced design rules; predictable runtime behavior by construction
Explains	explain the "why" to external stakeholders: persuasively justify major design tradeoffs	explain the "how" to internal stakeholders: concisely explain program behavior	justify significant design decisions in terms of their impact on actual or predicted runtime behavior
Abstracts	components and connectors, styles, "4+1" views, analysis and simulation models	procedures, interfaces, classes and modules; idioms, patterns and component models	software components and component models that have dual (architecture and program) meaning

cal experiences in solving non-trivial engineering design problems that clearly span architecture and program design activities.

1.2.3 Technology and Practice on the Seam

As mentioned earlier, routinization and mechanization are not antithetical to engineering design, but rather make it possible for designers to obtain more reliable outcomes as well as to focus their attention on those aspects of design that depend heavily on intuition and judgement. The Seam exposes the potential for routinizing and mechanizing a range of design activities centered on achieving predictable program *runtime* behavior. This research uses a software component model to provide a *syntax* of design, and programming language technology to provide a *semantics* for automated reasoning about designs. Figure 1.2 depicts in summary form how these technologies are combined. The following elaboration of the figure introduces terms that are defined and extensively used in the main body of this thesis; the first occurrence of terms appearing in the figure are highlighted in **boldface** to simplify the correlation of text to graphic.

The principal design artifacts are components and **component assemblies** (upper left quadrant in Figure 1.2) that conform to a component model;

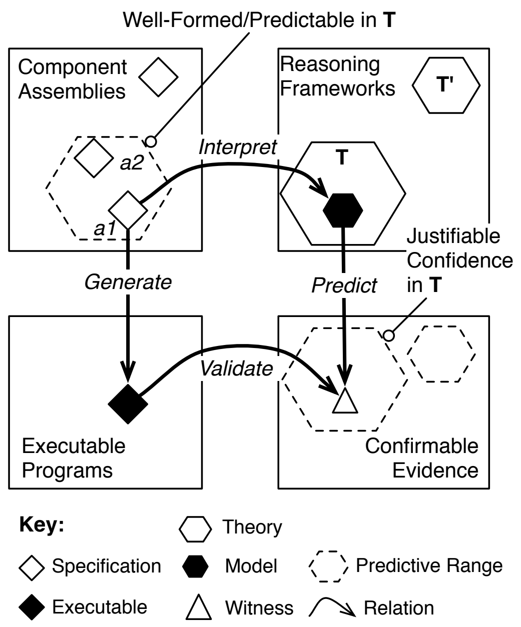


Figure 1.2: Logical Structure of Seam Technology and Practice

the Pin component technology (Chapters 6–7) defines such a component model. PCL is a specification language that formalizes the Pin component model. Component assemblies that are **well-formed** in PCL (i.e., that conform to the Pin component model) can be automatically **translated** into fully **executable programs** (lower left quadrant in the figure). The functional behavior of assemblies is defined by a traditional functional semantics (Appendix A). Every assembly that is well-formed to Pin has an **interpretation** in this semantics, and its functional (i.e., runtime) behavior is therefore predictable with respect to this semantics, by construction, i.e., by the syntactic rules that define the component model.

The functional semantics of Pin assemblies is part of the underlying Pin infrastructure. In contrast, non–traditional semantics are packaged as a new class of component called **reasoning framework** that are independently deployable extensions of a new kind of "prediction–enabled" component technology (Chapters 5). Reasoning frameworks will likely make assumptions about the environment in which component assemblies will execute beyond those made by the Pin functional semantics, for example about the Pin runtime or the computing or networking platforms on which Pin is hosted, or even about the way that component assemblies are used. Where practical, these assumptions are made explicit as **design rules**; when enforced, design rules establish runtime invariants that satisfy reasoning framework assumptions. A component assembly that satisfies the design rules of a reasoning framework is regarded as well-formed to that reasoning framework, and its runtime behavior is therefore predictable with respect to that reasoning framework, by construction.

Reasoning frameworks are used to make **predictions** about future runtime behaviors of component assemblies. The predictions of a sound semantics will correspond with the observed runtime behavior of assemblies, and independently **confirmable evidence** of this soundness is required if engineers are to have justifiable confidence in predicted behavior. Evidence can take the form of mathematical demonstrations (e.g., a formal proof), or statistical demonstrations (e.g., a confidence interval); the kind of evidence used determines the **confidence basis** for a reasoning framework. The **confirmation** of predictions made by reasoning frameworks that have a “weak” basis will require more effort than for those that have a “strong” basis; at the extremes this effort reduces to traditional testing practices for the weakest bases, to mechanical proof-checking for the strongest bases, with statistical evidence lying somewhere between these extremes.

Prediction-enabled component technology (or “PECT”) is a component technology that supports semantic extensions by reasoning frameworks (or their equivalent), makes explicit and enforces the design rules (or their equivalent) of reasoning frameworks, and produces confirmable evidence of the predictive strength of reasoning frameworks.

Predictability by construction is an engineering practice that exploits reasoning frameworks, design rules, and confirmable evidence to improve the quality of architecture and program design, and to reduce the time and effort required to obtain justifiable confidence that programs exhibit their required functional and non-functional behaviors.

1.3 Key Questions

The research questions directly correlate to the seam structure summarized in Table 1.2.

1. What makes a runtime behavior “predictable” and what constitutes “sufficiently predictable” behavior?

2. How are theories program behavior packaged as “non–traditional semantics” of programs?

3. How are “design rules” that lead to predictable behavior identified and enforced?

4. How is justifiable confidence in program behavior established, and how is it used?

5. How is software component technology used to provide substantial automation of the seam?

The main aim of this research is to demonstrate the feasibility and practicality of a substantially automated architecture/program seam. The novelty of the research arises from integration of existing technologies, rather than the development of new theories. New theories were developed in some areas, but this was not the main aim of the research.

1.4 Contribution

The key contributions made by this research are

1. It defines a region of overlapping jurisdiction between architects and programmers, called *the Seam*, and an engineering capability called *predictability by construction* that arises from the Seam.

2. It defines *prediction-enabled component technology* (PECT) as a means of achieving predictability by construction, and demonstrates its viability on non–trivial industrial engineering case studies in industrial robot control and electric grid substation automation control, as well as for desktop streaming audio manipulation.

3. It demonstrates that the Seam *itself* constitutes a new class of evolutionary design problem, whose solutions require the integration of programming language theory, design theory, specialized theories of system behavior and deep systems expertise, and whose outcome is substantially–improved software engineering practice.

Specific contributions are linked to the questions posed in Section 1.3:

- **Predictable Runtime Behavior.** This research demonstrates that *predictability* of program runtime behavior is stronger than what software architects can routinely achieve today, and better reflects what programmers can achieve today. The focus on observable runtime behavior is itself a significant step towards finding common ground for software architects and programmers; although it is a narrowing of design concern, it adds needed concreteness to architectural design practice, and expands the range of concerns that programmers can effectively address.

- **Theories of runtime behavior can be packaged as non–traditional semantics of architecture and program descriptions.** This research demonstrates that *reasoning frameworks* can be developed and independently deployed as "prediction-enabling" components of a PECT. Reasoning frameworks provide a *semantic interpretation* from specifications of component assemblies to models in a theory of runtime behavior, and a *decision procedure* to automate analysis. As such, a reasoning framework is a direct analogue to the traditional functional semantics of programming languages. It is *not* a claim of this research, however, that reasoning frameworks are themselves freely *composable*; the achievement of a modular or compositional semantics of arbitrary program behavior is left for others.

- **Design rules can be systematically defined and enforced.** This research demonstrates a *co-refinement* process for designing reasoning frameworks as a way to make explicit the *assumptions* of a reasoning framework theory, the *observations* made by that theory, and with the *strength of evidence* produced by that theory; and as a way of trading among observations, assumptions, and strength of evidence. Assumptions must be established as invariants, and an important part of tradeoff analysis is the cost associated with establishing these invariants, in terms of implementing the reasoning framework and in terms of the impact of the strictures imposed by reasoning frameworks on architects and programmers. The research demonstrates several ways of enforcing constraints, such as extending the component technology runtime, syntactic stricture enforced by interpretation, and the use of specialized component containers.

- **Objective evidence of program and predicted system behavior can be obtained and packaged.** This research demonstrates that sound and objective (empirical and formal) evidence of program behavior can be obtained and confirmed by disinterested third parties. The research demonstrates a principled way of deciding which runtime behaviors of software components require objective evidence, as well as the requisite strength of that evidence. This demonstrates the (non-circular) co-dependence of architectural analysis and grounded evidence: a theory of runtime behavior defines the evidence that is required of software components, and the evidence that can be obtained from software components constrains the theories, and hence architectural analysis, that are practicable.

- **Familiar component-based abstractions and implementation techniques can be used to implement the seam.** This research provides a proof-of-existence demonstration of the Seam. A prototype prediction-enabled component technology demonstrates all key claims made in this thesis, and is freely available for internet download.[2] The capabilities of the prototype are illustrated on challenging industrial problems described in Chapter 9.

[2]See `http://www.sei.cmu.edu/predictability/tools/starterkit/index.cfm`.

1.5 Related Work

The main contributions of this research result from integrating the results of two areas of prior research, software architecture technology and software component technology, with adaptation only as required to achieve a suitably transparent "Seam." Accordingly, the most pertinent related work can be found in these two areas of research.

Work in formal software architecture description languages (ADL) provide key foundations for this work, in particular those that emphasize component and connector abstractions. Wright [9] influenced this research in its use of a process algebra to specify the semantics of component interaction independent of the functional behavior of the components themselves. In Wright, interaction semantics are keyed to connector types, and treatment of connector types as "first class" abstractions is consistent with the goal of maximizing the expressiveness of architectural models to describe patterns of interaction [117]. In contrast, the research reported here uses a small collection of pre-defined connector types that are directly supported by familiar programming models and by Pin, although for this limited repertoire a process-theoretic semantics is also given [82]. Wright was also used to formalize *architectural style* [3, 8, 53]. However, these formalizations established only weak invariants and therefore provided insufficient grounds for the kinds of analysis required to achieve predictability by construction. In contrast, *attribute-based* architectural styles (ABAS) make use of structural invariants of more specialized architectural styles to support analysis of non-functional properties of systems [87, 88]. However, these styles were highly localized with no accommodation for their composition. The research reported here recasts Wright's global styles as a key element of a component model, and generalizes the structural invariants of an ABAS to "design rules" that can be enforced in several ways. The term "style" in ABAS is really a misnomer; more accurate would be "pattern," an idea that gained prominence with the success of design patterns for programs [52]. A number of architectural patterns were documented [25, 144], and their emphasis on software implementation is an early recognition of the Seam. However, pattern-oriented software architecture hewed too closely to programming concerns and failed to connect the structural invariants of a pattern with explicit analysis theories. The architecture analysis and description language (AADL) is a recent attempt to gain widespread adoption of a standard architecture representation, but although the notation has been used to support analysis [58], there are no provisions for extensible design rules or objective evidence.

Composition languages are intermediate between ADL and software components, and as such are candidate Seam technologies. Both Darwin [108, 109] (which is variously described even by its creators as ADL and composition language) and especially Piccola [106, 4] are representative composition languages: both use a process-algebraic approach to specify component behavior, and both use the composition operators of the process algebra to compose components into larger systems. Composition operators with nice algebraic properties are elegant and do capture the intuition of inductive composition; unfortunately theories for non-functional behavior are seldom so neat and clean. The research reported here distinguishes *constructive* and *analytic* composition, and does not provide for inductive (algebraic) constructive composition; reasoning frameworks are free to define composition as they choose, providing there is an interpretation from a constructively-composed assembly of components.

Szyperski's seminal work on component-oriented programming remains the authoritative and most comprehensive description of software component–oriented programming available[160]. His notion of component framework applies to Pin, and a PECT can be regarded as a *tiered* framework built from Pin; his notion of component markets corresponds to "implementable sets" used to define the scope of reasoning frameworks (which are components of a PECT and therefore of a tiered framework). Szyperski also addresses questions of how extra–functional behavior is specified (or at least, identifies a component's "contractual interface" as one locus of such specification). Szyperski does not, however, address the way extra–functional specifications are verified, or how they are used to reason about system–wide behavior. A number of software component technologies have been developed, however, that are strongly (if not primarily) motivated by one or more non-functional concerns [17, 18, 146, 86]. While each of these contribute to our understanding of the Seam, none of these have addressed issues of evidence described in this research, the use of design rules, or other stricture to enforce analytic assumptions, or have addressed analytic extensibility of component technology.

The research reported here also builds on related work in software model checking and real-time performance analysis. These areas of related work are themselves quite extensive, but are of secondary importance to the research reported here, which is more concerned with the way model checking and performance analysis can be packaged in reasoning frameworks of a prediction-enabled component technology. Chapter 8 provides appropriate details on related work.

1.6 Research Method

The overall research method relied on a series of *existence proofs* to demonstration that a sound and effective practice of *predictability by construction* can be established; and that a *prediction-enabled component technology* provides substantial automation and transparency to this practice.

Each existence proof is motivated by an engineering challenge that is thematic to (hence pervasive in) several industries *and* is situated in a concrete industrial setting. The combination of pervasiveness and situatedness helps to ensure that the research results are broadly applicable and suitably challenging to be of interest to researchers and practitioners alike. The essential structure of each situated engineering problem is expressed as a *model problem*, the solution of which can reasonably be argued is transferrable to the original situated problem, but generalized to a broader class of problem of which the original problem is an instance. The prototype PECT was developed to meet the demands of these situated engineering problems. The usability of the technology is also demonstrated through its packaging and free distribution as a "starter kit," the hands-on use of the starter kit by students in tutorials [72] and educators [77, 76].

Finally, the technical concepts of predictability by construction and its technology constituents are documented in refereed publications (workshops, conferences and journals), as technical reports of the Software Engineering Institute, and as tutorials.

1.7 Key Assumptions

This research makes several strong assumptions about the social, organizational, and economic context of the engineering design enterprise.

- There is a unitary design authority that can impose constraints on the design and implementation of software. Although this authority is personified as "the architect" in the work reported here, the authority might be distributed among several designers and several organizations. The important point is that an authority exists that can impose design rules, and can make the appropriate tradeoff decisions in the design of reasoning frameworks.

- There is economic value in having objective evidence of program and system behavior, and economic value in having objective evidence of the quality of architectural analysis undertaken prior to the implementing or acquiring the software components used to implement an architecture. This assumption is likely to be satisfied in safety-conscious engineering cultures, but should not be unquestionably assumed for all cultures.

- The software architecture of systems is described, at least in part, in a component and connector style of representation, and software component technology is used to implement systems specified in a component and connector style.

The first two assumptions are almost certainly necessary conditions of predictability by construction; the last is a consequence of the research approach, and no such necessity is claimed (although sufficiency *is* demonstrated).

1.8 Organization of this Thesis

The work reported here is the product of substantial collaboration among several researchers and research organizations. Chapter 2 provides the context for the research described here, identifies my role and contributions, as well as the contributions of several of research collaborators. Following chapter 2, the dissertation is organized in 4 major parts, plus appendixes.

Part I Foundations. Part I defines the domain of discourse for predictable assembly and prediction-enabled component technology. Chapter 3 describes an idealized rational software design process and defines key concepts used to describe the Seam, predictability by construction, and prediction–enabled component technology. Chapter 4 provides a closer look at the Seam, and motivates how architecture and component technologies are combined, in a prediction-enabled component technology, to provide substantial automation of the seam.

Part II A Prototype Prediction-Enabled Component Technology. Part II describes in some detail the prediction-enabled component technology (PECT) developed by this research. Chapter 5 describes the key technical features, architecture and implementation of prototype PECT. Chapters 6-8 examine the major components of the prediction-enabled component technology in turn: Chapter 6 describes the Pin Component Language, which formalizes the Pin component model and extends it in ways that make it more suitable for the Seam. Chapter 7 describes the Pin component technology itself. Chapter 8 describes the reasoning frameworks developed for this research to support real-time performance analysis and temporal–logic model checking analysis, respectively.

PART III Experiences in Use. Part III describes concrete experiences with predictability by construction and prediction-enabled component technology. Chapter 9 describes industrial case studies for electric grid substation *soft protection and control* applications and industrial robot control.

Chapter 10 steps back from technology of predictability by construction to consider how the engineering practices supported by this technology might be incrementally adopted. It describes how the seam presents a new class of design problem that requires new design strategies, and discusses how these challenges and strategies influenced the design of the prediction-enabled component technology described in this research, and how this technology might evolve in the future.

PART IV Conclusions. The main results of this research are summarized in Chapter 11.

PART V Appendices. Appendix A provides a closer look at PCL's *interaction* and *reaction* semantics. Appendix B is an assembly and component specification excerpted from the soft protection and control case study as a representative sample of scale in the prototype developed using a prototype PECT. Appendix **??** provides a list of acronyms frequently used in this dissertation.

Chapter 2

Personal Research Contribution

The contribution of this research is integrative: it draws on concepts and technologies from software architecture and software component technologies, program analysis, model checking, and performance analysis technologies, real-time operating systems, and language design and language semantics. Drawing on such a wide range of topics necessarily required the contributions of experts in these topics. Moreover, the research approach—building and testing prototype technologies in practical settings—required significant software engineering effort, far more beyond what any individual could accomplish. Indeed, the engineering effort itself provided great illumination on the kind of design and engineering tradeoffs and other such pragmatics that are implied by prediction-enabled component technology.

Accordingly, I have chosen to begin with some historical context for research in Section 2.1, and then to present my personal contribution to work in Section 2.2, as well as the contributions of collaborators in this research in Section 2.3. This allows me to place my own contributions into an appropriate context, and also gives a sense of the many and varied research results that have sprung from this work. This latter point is particularly important since an important subtext of the research is that the Seam is a locus for significant future work that will contribute to maturing software engineering practice.

2.1 Context of the Work

Software components have been a prominent area of research at Carnegie Mellon University's Software Engineering Institute (SEI). For each of the research efforts described below, I was the principal investigator (PI), a role which at the SEI is technical, not managerial. In all cases I was solely responsible for defining the overall technical vision, identifying main research questions and areas of investigation, and for carrying out the work.

The main results of the work reported here were produced by the *Predictable Assembly from Certifiable Components* (PACC) initiative (2002-2008), which involved 5-7 full-time research staff. A preliminary and smaller-scale investigation of the *Technical Concepts of Software Component Technology* (2000-2002) involved 2-3 full time research staff, and ultimately led to the outlines of PACC. My work in PACC was itself instigated by the results of the *COTS-Based Systems* (CBS) research initiative (1996-2000)[1], also involving 5-7 full-time research staff. The main results of CBS work are documented in *Building Systems from Commercial Components* [170].

PACC and CBS addressed, in different ways, fundamental questions about the role software components play in software engineering, how software components alter the software design process, and what can and cannot be predicted about systems constructed substantially from software compo-

[1]"COTS" is an acronym for *commercial-off-the-shelf*

nents. While my research in PACC and CBS share these common themes, PACC—and therefore the research reported here—can also be regarded as a *response* to the disruptions of engineering predictability that arise from using large-grained COTS software components of the sort dealt with in CBS research. The critical assumption (previously noted in Chapter 1, Section 1.7) of a unitary design authority in place of the many "invisible hands" at work in the commercial market is one part of this response. The research reported in this thesis demonstrates that, given a unitary design authority, predictability by construction can be achieved using prediction-enabled component technology.

Mälardalen University (MDH) has had a significant impact on the reported work. As part of his MdH PhD studies, Dr. Magnus Larsson made fundamental contributions in how we generated evidence for observed component execution time and predicted assembly latency. The industrial cases developed with ABB were also strongly informed by prior collaborations between MDH and ABB, and, not incidentally, by Dr. Larsson's association with ABB. In 2006 I also provided several lectures and used early versions of the technology reported here in student projects at MDH, and the results were independently used elsewhere [77, 76]. The research reported here has also had impact on several MDH research initiatives, in particular SAVE and PROGRESS in which component models for resource-constrained embedded systems were developed; Their component models were inspired by PACC notions of predictability and analytic interfaces of components.

2.2 My Contribution

I was the principal investigator and defined the overall vision and research objectives for all the work reported in this dissertation. While developing and demonstrating PECT required the contributions of many, the following contributions were uniquely mine:

- the *idea and definition of the architecture/program seam* as an overlapping jurisdiction of architecture and program design, which is described in Chapter 4 and which motivated all of the research reported here;

- the *idea and definition of predictability by construction* as a consequence of the Seam, and its basis in notions of "formal predictability," is described in Chapter 3;

- the *idea and definition of PECT* and its basis in Pin–like component technologies and language semantics, is described in Chapter 5;

- the *idea, definition and structure of reasoning frameworks*, and their role in providing non–standard semantics of component and system behavior, and are described in the introduction to Chapter 8 and in Chapter 3;

- the *idea, definition and use of co–refinement* as a way to incrementally develop reasoning frameworks, and as a way of "fitting" a reasoning framework to a class of recurring design problems, is described in Chapter 10.

While I defined the overall vision for the Seam, predicability by construction and PECT, I also made concrete contributions to the PECT prototype. I defined the syntax and semantics of PCL, described in Chapter 6 and Appendix A, respectively. I also implemented the PCL frontend and various backends used by the $\lambda*$ and ComFoRT reasoning frameworks described in Chapter 8, and contributed a substantial portion of the prototype proof–carrying code prototype, alsodescribed in Chapter 8, §8.6.3) *ComFoRT Reasoning Framework*).

2.3 My Collaborators

This research has benefitted enormously from the contributions of others, and in fact could not have been undertaken, and certainly would not have succeeded, without their strong contributions, the most significant of which are noted here, in alphabetical order.

- Jeff Hansen, Ph.D. Dr. Hansen developed simulation models for queuing theories associated with the sporadic server container and its reasoning framework.

- Scott Hissam. Mr. Hissam was chief engineer and release manager for PECT prototypes, developed thread scheduler extensions to support UML statecharts, and developed instrumentation harnesses and other tools to validate performance reasoning frameworks.

- James Ivers. Mr. Ivers worked closely with me to develop formal semantics of PCL, and to specify thread scheduler extensions to support UML statecharts. Mr. Ivers also was the lead integrator of the certifying software model checking reasoning framework.

- Mark Klein. Mr. Klein leads SEI research in software architecture technology and has provided valuable guidance on the relationship between structural invariants in architecture patterns and their associated analytical theories. Mr. Klein also developed various average-case latency theories found in the performance reasoning framework.

- Magnus Larsson, PhD. Dr. Larsson developed the measurement and sampling tools and techniques used to validate the average-case latency theories found in the performance reasoning framework. These tools and techniques formed the basis for all future empirical/statistical validation of performance reasoning frameworks, and were the focus of his PhD dissertation [98].

- Gabriel Moreno. Mr. Moreno extended the Pin component technology to support containerized components, and was also the lead integrator of the performance reasoning framework. Mr. Moreno also demonstrated advanced C++ meta-programming techniques to simplify compile-time deployment of components into their containers.

- Daniel Plakosh. Mr. Plakosh developed the WaterBeans component technology, a precursor of Pin, later extended to develop the first proof-of-concept PECT. Mr. Plakosh also developed the initial prototype of the Pin component technology and its underlying real-time operating system.

- Sagar Chaki, PhD. Dr. Chaki developed the model checker used by the performance reasoning framework, and also developed the techniques used by the model checking reasoning framework to generate proof certificates for satisfied claims.

- Natasha Sharygina, PhD. Dr. Sharygina used a model checking reasoning framework using an off-the-shelf model checker (COSPAN) to falsify behavior claims made for message-passing software used in an industrial robot controller.

- Judith Stafford, PhD. Dr. Stafford was an early and key contributor to the PECT concept, and worked closely with me to identify a useful correlation of the concepts of software architecture and software components.

- Chuck Weinstock, PhD., and John Goodenough, PhD. Drs. Goodenough and Weinstock were not directly associated with PACC research, but made valuable contributions in the use of assurance cases to construct defeasible arguments about the quality of evidence, and in particular about formal evidence; these ideas were applied to the PACC certifying code generator.

2.4 Description of Key Publications

The following publications are the basis for the thesis.

- Volume II: Technical Concepts of Component Based Software Engineering, 2nd Edition Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Seacord, R., Wallnau, K., Technical Report

CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, September 2002. Originally published in 2000, this report describes the key terminology and paradigmatic architecture of an important genre of component technology. The report also discusses the fundamental limitations of conventional approaches to component interface specification for reasoning about system level quality attributes. The description of the component technology genre is now somewhat dated, especially in comparison with recent surveys[46]), but the discussions on reasoning about quality attributes remains valid. The conclusions of this report led to the creation of the PACC research initiative. I was the lead author of this report, and defined all of the main technical concepts.

- On the Relationship of Software Architecture to Software Component Technology, Kurt Wallnau, Judith Stafford, Scott Hissam, Mark Klein, in Proceedings of the 6th ECOOP Workshop on Component-Oriented Programming, Budapest, Hungary, 2001. This paper identifies, and proposes a means for closing the gap between the dominant research agendas in the software architecture and software component research communities. The paper outlines criteria for successful integration of software architecture and software component technology. A four-level reference model that subsumes software architecture and software component technology (assembly, specification, types, metatypes) is described. Using this reference model, two paths to close the gap are detailed, one using software component technology and the other using software architecture technology as a springboard. It was sometime later that we chose a "component first" approach, leading to PECT. I was lead author of this paper, and I defined the gap, the approaches to closing the gap, and the illustrating examples.

- Hissam, S., Moreno, G., Stafford, J., Wallnau, K., Enabling Predictable Assembly, Journal of Systems and Software, Vol. 65, No. 3, 15 March 2003, pg. 185-198, North Holland. This article introduced the PECT concept, and described an initial prototype implementation based on the (non-real time) WaterBeans component model [137], extended with a performance reasoning framework for predicting worst-case latency of audio streaming and mixing applications. A shorter version of this article appeared as *Packaging Predictable Assembly* in the First International IFIP/ACM Working Conference on Component Deployment, June 20-21, 2002, Berlin, Germany, as a Springer-Verlag LNCS. I was the lead author of this article and paper, and defined the overall concept of prediction-enabled component technology.

- Volume III: A Technology for Predictable Assembly from Certifiable Components, Kurt Wallnau, Technical Report CMU/SEI-2003-TR-

009, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, April 2003. This report described in detail the architecture of prediction-enabled component technology (PECT), and the role of PECT in achieving (what was later to be called) *predictability by construction.* The report anticipated many of the practical and theoretical challenges of developing truly compositional theories of system-level quality attributes, and (properly) identified composition of these theories as a fundamental challenge—one that remains an important area of future work beyond the research reported in this thesis. I was sole author of this report, which defined the technical concepts of prediction-enabled component technology that governed PACC.

- Sagar Chaki, James Ivers, Peter Lee, Kurt Wallnau, Noam Zeilberger. Certified Binaries for Software Components (CMU/SEI-2007-TR-001). Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, April 2007. This report demonstrated the use of a PECT to achieve end-to-end automation of *proof-carrying code* (details described in Chapter 8). A shorter version of this technical report appeared as *Model-Driven Construction of Certified Binaries* in the proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS), LNCS 4735, pages 666-681, September 30-October 5, 2007. I was co-author of this report and paper. I developed the front-end (from PCL to model checker) and backends (from model checker to PCL, and then from PCL to generated code) transformations that demonstrated the end-to-end automation.

- Bass, L., Ivers, J., Klein, M., Merson, P., and Wallnau, K. 2005. Encapsulating Quality Attribute Knowledge. In Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (November 06 - 10, 2005). WICSA. IEEE Computer Society, Washington, DC, 193-194. This paper describes the concept and structure of quality attribute reasoning frameworks, and generalizes the concept of reasoning framework described originally by me in [169] but here generalized for use in architecture technologies that are not bound by the strictures of evidence, or the need to define a theory semantics in an underlying component technology. I was a co-author of this paper, and defined all of the basic underlying technical concepts of reasoning framework.

- Sagar Chaki, James Ivers, Natasha Sharygina, Kurt Wallnau. The ComFoRT Reasoning Framework, LNCS 3576, pages 164-169, 2005. This paper provides, in terms that are familiar to researchers in model checking and software model checking community, a summary description of the ComFoRT (for **Com**ponent **Fo**rmal **R**easoning **T**echnology) model checking reasoning framework. The paper was accompanied by a tool demonstration, one that by nature of this expert audience marked an important milestone in the development of PECT. I was a co-author of this paper, and defined the use of software components as a link between the automata- (or process-)theoretic notion of component extant in software model checking literature, and the computer programs that software model checkers purport to verify.

- Statistical Models for Empirical Component Properties and Assembly-Level Property Predictions: Toward Standard Labeling, Gabriel Moreno, Scott Hissam, Kurt Wallnau, in the Fifth ICSE Workshop on Component-Based Software Engineering, Orlando, Florida, May 2002. This paper describes various kinds of statistical models and how they can be used to provide evidence (i.e., *labels*) of component-level and system-level behavior. I was co-author of this paper, and defined the different forms of certification (e.g., assembly, component, normative, descriptive) to which statistical labels might apply.

- The potential for synergy between certification and insurance, Paul Luo Li, Mary Shaw, Kevin Stolarick, and Kurt Wallnau, in the First International Workshop on Software Reuse Economics, held in conjunction with the Seventh International Conference on Software Reuse, Austin, Texas, April 16, 2002. This paper describes the use of empirical, independently confirmable (and refutable) evidence of software behavior to quantify software-related risk, and how this relates to the models used in the insurance industry to define its product offerings. I was co-author of this paper, and defined aspects of statistical evidence of software pertinent to insurance and insurance underwriters; expertise in the insurance industry was provided by Dr. Stolarick, and expertise on the economics of software architecture was provided by Dr. Shaw.

- Scott Hissam, James Ivers, Daniel Plakosh, Kurt Wallnau. Pin Component Technology (V1.0) and Its C Interface (CMU/SEI-2005-TN-001). Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, April 2005. This report describes the Pin component model as seen by programmers (i.e., through its application programming interface) who choose to bypass the program generation tools provided by the PECT prototypes. I was co-author of this report, and contributed my expertise as the primary user of the technology as a target language for program generation.

- Snapshot of CCL: A Language for Predictable Assembly, Kurt Wallnau, James Ivers, Technical Note CMU/SEI-2003-TN-025, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, June 2003. This report provides a high-level language description of the Pin component model as seen by PECT users, i.e., by architects of Pin applications and by developers of Pin components. The main structural features of PCL are described, such as components, asynchronous and synchronous interaction via component *pins*, component *assemblies*, and the assembly *environment*. I was the lead author of this report, and the principal designer of PCL syntax and semantics.

- A Basis for Composition Language CL, James Ivers, Nishant Sinha, and Kurt Wallnau, SEI Technical Note CMU/SEI-2002-TN-026, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA., 2002. There are two distinct but complementary formal semantics for PCL, one that describes the semantics of component interaction (this paper), and one that describes the semantics of component behavior (this thesis, Appendix A). These are distinct from the semantics assigned to PCL by each reasoning framework. The interaction semantics is specified using Hoare's algebraic specification language, Communicating Sequential Processes (CSP), and describes the meaning of different types of pin (synchronous, mutex synchronous, and asynchronous) in terms of the composed behavior of components that interact on pins of those types. I was co-author of this report, and defined how *reaction* differs from *composition*, and an informal semantics for a repertoire of connector types used to enable reactions among components; Mr. Ivers provided expertise in CSP and formalized the semantics of interaction using CSP.

- Preserving Real Concurrency, James Ivers, Kurt Wallnau, Proceedings of the 2003 ECOOP Workshop on Correctness of Model-Based Software Composition (CMC), Technical Report 2003-13 at Universitat Karlsruhe, July, 2003. This paper motivates the interaction semantics of PCL, described in detail in [82], by showing that the semantics most frequently encountered in architecture description languages overestimates concurrency, while faithfully modeling interaction semantics in terms of an underlying software component technology produces more accurate models. I was co-author of this paper, and described the impact of over- and under-approximations of concurrency on the quality of predicted software behavior.

- Scott A. Hissam, Gabriel A. Moreno, Kurt C. Wallnau. Using Containers to Enforce Smart Constraints for Performance in Industrial Systems (CMU/SEI-2005-TN-040), Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, April 2005. This report describes the use of PECT and a specialized component container to enable safe third-party extension of a hard real-time platform, and specifically without risking scheduling deadlines. A container was developed to preserve scheduling deadlines in the platform, using the application-level *sporadic server* protocol. A new analysis theory was developed to provide performance guarantees to the developers of third party components, so that they in turn could be assured adequate computational resources for their extensions. I was co-author of the report, and defined the overall concept of analytic sandboxing with component containers.

- Scott Hissam, Mark Klein, John Lehoczky, Paulo Merson, Gabriel Moreno, Kurt Wallnau. Performance Property Theories for Predictable Assembly from Certifiable Components (PACC) (CMU/SEI-2004-TR-017),Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, April 2004. This report provides an in-depth description of a performance theory that can be used to bound the average case latency of tasks managed by a sporadic server. The theory was developed to provide performance guarantees to the developers of third party components, where those components are expected to execute in hard real-time environments. Various assumptions made by the theory, such as replenishment intervals, were later to be enforced by a type of Pin component *container*. I was co-author of this report, serving mostly in a review of details about the performance theory, while contributing basic concepts for how the design rules of these theories are enforced by containers.

- Predictable Assembly of Substation Automation Systems: An Experiment Report, 2nd Edition, Scott Hissam, John Hudak, James Ivers, Mark Klein, Marnus Larsson (ABB), Gabriel Moreno, Linda Northrop, Daniel Plakosh, Judith Stafford, Kurt Wallnau, William Wood, Technical Report CMU/SEI-2002-TR-031, revised July 2003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, September 2002. This report describes an initial experience in developing a PECT in an industrial setting (electric grid substation automation). We demonstrated, on a model problem, assembly of substation automation functionality from components that implement IEC-61850 functions, with well–formed assemblies exhibiting average case latency within a validated statistical confidence interval. The results of subsequent work on using PECT to demonstrate "soft protection and control" are described in this thesis (Chapter 9). I was the lead author of this report, and described the basic methods of *co-refinement* used to incrementally develop a performance reasoning framework.

2.5 Books

- Gorton, I., Heinemann, G. T., Crnkovic, I., Schmidt, H. W., Stafford, J. A., Szyperski, C., and Wallnau, K. 2006 Component-Based Software Engineering: 9th International Symposium, CBSE 2006, Västerås, Sweden, June 29 - July 1, 2006, Proceedings (Lecture Notes in Computer Science). Springer-Verlag New York, Inc.

- Wallnau, K., Hissam, S. A., and Seacord, R. C. 2002 Building Systems from Commerical Components. Addison-Wesley Longman Publishing Co., Inc.

2.6 Journal Articles and Book Chapters

- Crnkovic, I., Heineman, G. T., Schmidt, H. W., Stafford, J., and Wallnau, K. 2007. Guest Editorial. J. Syst. Softw. 80, 5 (May. 2007), 641-642.
- Crnkovic, I., Schmidt, H. W., Stafford, J., and Wallnau, K. Automated component-based software engineering. Journal of Systems and Software 74, 1 (2005), 1Ũ3. Automated Component-Based Software Engineering.
- Crnkovic, I., Reussner, R., Schmidt, H., Simons, K., Stafford, J., and Wallnau, K. 2005. Report of the International Symposium on Component-Based Software Engineering. SIGSOFT Softw. Eng. Notes 30, 3 (May. 2005), 1-9.
- Magnus Larsson, Anders Wall, Kurt Wallnau, Predictable Assembly - The Crystal Ball to Software, ABB Review, Journal, p49-54, Issue 2, 2005
- Crnkovic, I., Schmidt, H., Stafford, J., and Wallnau, K. 2004. 6th ICSE Workshop on Component-Based Software Engineering: automated reasoning and prediction. SIGSOFT Softw. Eng. Notes 29, 3 (May. 2004), 1-7.
- Kurt Wallnau, Judith Stafford, Dispelling the Myth of Component Evaluation, chapter 8 in Building Reliable Component-Based Software Systems, Ivica Crnkovic, Magnus Larsson, Artech House publisher, ISBN: ISBN 1-58053-327-2, 2002
- Seacord, R. C., Hissam, S. A., and Wallnau, K. C. 2000. Component Web search engines. In Handbook of internet Computing, B. Furht, Ed. CRC Press, Boca Raton, FL, 183-203.
- Alan W. Brown, Kurt C. Wallnau, "The Current State of CBSE," IEEE Software, vol. 15, no. 5, pp. 37-46, Sep./Oct. 1998.
- David Carney, Kurt Wallnau, A Basis for the Evaluation of Commercial Software, in the Journal of Information and Software Technology, Elsevier publishing, U.K., November, 1998.

- Robert C. Seacord, Scott A. Hissam, Kurt C. Wallnau, "Agora: A Search Engine for Software Components," IEEE Internet Computing, vol. 2, no. 6, pp. 62-70, Nov./Dec. 1998.

2.7 Conference and Workshop Contributions

- Stafford, J., Wallnau, K., Is Third Party Certification Necessary? , Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering, Toronto, Canada, May, 2001, pp. 13-17.
- Wallnau, K., Stafford, J., Ensembles: Abstractions for a New Class of Design Problem, Proceedings of the IEEE 27th Euromicro Conference (Euromicro 2001), Warsaw, Poland, September 2001.
- Polze, A., Plakosh, D., and Wallnau, K. 1998. CORBA in Real-Time Settings: A Problem from the Manufacturing Domain. In Proceedings of the the 1st IEEE international Symposium on Object-Oriented Real-Time Distributed Computing (April 20 - 22, 1998). ISORC. IEEE Computer Society, Washington, DC, 403.
- Andreas Polze, Daniel Plakosh, Kurt Wallnau, Real Time Computing on Off-The-Shelf Components–A Case for CORBA, in Proceedings of the International Conference on Integrated Design and Process Technology, Berlin, Germany, June 1998.
- Andreas Polze, Daniel Plakosh, Kurt Wallnau, CORBA in Real-Time Settings: A Problem from the Manufacturing Domain, in Proceedings of the 1st International Conference on Object-Oriented Real-Time Distributed Computing (ISORC98), Kyoto, Japan, 1998.
- Wallnau, K. C. 1997. Repairing coordination mismatches among legacy components. In Proceedings of the international Conference on Software Maintenance (October 01 - 03, 1997). ICSM. IEEE Computer Society, Washington, DC, 302.
- Wallace, E. and Wallnau, K. C. 1996. A situated evaluation of the Object Management Group's (OMG) Object Management Architecture (OMA). In Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (San Jose, California, United States, October 06 - 10, 1996). OOPSLA '96. ACM, New York, NY, 168-178.
- Kurt Wallnau, Fred Long, Anthony Earl, Toward a Distributed, Mediated Architecture for Workflow Management, in Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems: State of the Art and Future Directions, 1996.

2.8 Technical Reports

- Kurt C. Wallnau. Software Component Certification: 10 Useful Distinctions (CMU/SEI-2004-TN-031)., Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, April 2004
- Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long John Robert, Robert Seacord, Kurt Wallnau. Volume I: Market Assessment of Component-Based Software Engineering Assessments (CMU/SEI-2001-TN-007).Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. , 2001
- Daniel Plakosh, Dennis Smith, Kurt Wallnau. Builder's Guide for WaterBeans Components (CMU/SEI-99-TR-024)., Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA., 1999

Part I

Foundations

Chapter 3

Rational Design

To reiterate the central theme of this research:

- Software architect and programmer are distinguished only in their respective jurisdictions of a jointly-held design problem, and by the approaches they take to address design problems within their respective jurisdictions;
- These jurisdictions overlap, and shared design problems within this overlap are approached by characteristic (but often inconsistent) ways by architects and programmers;
- This overlap is consolidated in the Seam, wherein discontinuities or inconsistencies in the approaches taken by architects and programmers can be repaired or reconciled;
- Prediction-enabled software component technology defines abstractions for the Seam that permit the same software artifacts to have simultaneous and complementary meaning to architects and programmers;
- Software designed using Seam technology will exhibit runtime behavior that is predictable by construction, for runtime qualities of practical interest to architects and programmers.

This chapter lays the foundation for the Seam by establishing a basic metaphor of rational software design, and by defining terminology that situates predictability by construction in rational design. The metaphor is also used in the design of prediction–enabled component technology. Chapter 4 *The Seam* uses this foundation to characterize rational design from the architect and programmer perspectives, and to identify complements in these perspectives that can be served by the Seam.

The rest of this chapter is organized as follows. The basic "design as search" metaphor is described in §3.1; this is then made more concrete for software design in §3.2. §3.3 motivates an ideal separation of architecture and program concern, not as something that is obtainable in practice, but as a reflection of the ideals of current software engineering practice—not all of which are without merit. §3.4 then defines key terminology used throughout the remainder of this dissertation.

3.1 Metaphorical Design

Metaphor is the means by which humans extend language and thought to new and unfamiliar concepts [96]. Metaphor is expressed in poetic terms, but is a basis for highly technical concepts in for example mathematics [97] and, more to present purposes, for design [107]. Here the basic metaphor of design as a *search of a fitness landscape for forms that are most fit for use* is summarized, and the practices of software design are grounded in the metaphor and therefore in the practices addressed by the Seam. The "design as search" metaphor is widely used in design literature, and will likely be familiar to the reader, and as such no attempt is made here to justify the validity of the metaphor or to exhaust the literature that has made use of it, but to simply use it to set the stage for the Seam concepts that follow.

Herb Simon formalized "design as search" in terms of two functions, *generate* and *test*:

> "The task of the generator is to produce variety, new forms that have not existed previously, whereas the task of the test is to cull out the newly generated forms so that only those that are well fitted to the environment will survive" [152].

Simon's use of the language of biological evolution aptly expresses "fitness for use" ("fitness," or "fit," etc.) as the defining characteristic of an underlying relation of *form* and *environment.* Moreover, the relation induces an ordering on forms that permits us to *objectively* distinguish better and worse forms; in biological terms, "survival of the fittest" is an objective reality, not a subjective judgement. The ordering relation is readily expressed (and often is, in practice) as the objective function F of a multi-attribute optimization process, where F quantifies "fitness" as a function of interacting design qualities.

The metaphorical search for forms is conducted on a *fitness landscape* defined by the objective function, with better-fit forms residing on peaks, worse-fit forms in valleys [20, 168], and *hill climbing* (i.e., seek higher ground) a principal design heuristic. In a *rational* design process each design decision (equivalently, choice of form, choice of search path) is made to maximize "fit."

It is also possible that designers will seek forms that are *least misfit* to their environments. In a mathematical sense, minimizing misfit or maximizing fit can both be expressed as objective function and optimization process; for convenience we will assume the latter case applies.[1]

3.2 Rational Software Design

Simon's "rational design" ideal can be expressed in more concrete terms of software design. Without loss of generality, the following concretization assumes that the design activity will produce a single form P_f, called the *final program* that will be used to compute some function ϕ in some end-use environment E_f, and moreover that P_f has an explicit symbolic representation that is stored on a digital computer. Almost any reasonably savvy interpretation of "end-use environment," and "program" will serve; these are

[1] Christopher Alexander did, however, observe important pragmatic differences between minimizing misfit and maximizing fit [6], and this observation inspired the "Risk/Misfit" method of evaluating software components [170].

explicitly defined in subsequent sections. We will allow P and E to represent the set of all possible programs and environments, respectively, with $P_j \in P$, $E_j \in E$, etc.

The **program artifact** can be encoded as its *potential change history* $H =< P_0, \Delta >$, where P_0 represents the initial and possibly empty program, $\Delta = \{\delta(P_i, P_j) \bullet i \geq j\}$ represents a set of changes ("diffs"), such that $\forall \delta(P_m, P_n) \in \Delta \bullet P_n = P_m \circ \delta(P_m, P_n)$ for some '$\circ$' transform operator.[2] With a slight abuse of notation, we will say that $P_k \in H \Leftrightarrow P_k = P_0 \vee \exists \delta(P_x, P_k) \in \Delta$. That is, a program is in the design space if it is the initial program or can be derived from the initial program. Notationally, $P_m \hookrightarrow P_n \equiv P_m \circ \delta(P_m, P_n)$. Any sequence $P_m \overset{*}{\hookrightarrow} P_{m'} \hookrightarrow P_n$ defines a *path* from P_m to P_n, where $P_m \overset{*}{\hookrightarrow} P_{m'}$ is the (possibly empty) *prefix* of P_n. P_0 has no prefix, and every other program $P_{k \neq 0}$ has *at least one* prefix. $\Pi_{m,n}$ is the set of paths in that lead from P_m to P_n, with each $\pi_{m,n} \in \Pi_{m_n} \subseteq \Delta$.

It is assumed that each program P_j is "well-formed" in some programming language. The intuition is that each change $\delta(P_m, P_n)$ corresponds to a design decision that leads the programmer meaningfully closer to achieving some ultimate design objective. The addition of "potential" to change history emphasizes that what is being described is **not** change history in the usual "configuration management" sense of the term, but in the "design as search" sense of the term. Hence, H may be regarded as a **design space** of all possible solutions.

Criteria of fit (other than functional correctness[3]) can be represented by the *preference structure* $S =< F, \preceq >$, for an objective function F and preference relation $\preceq$. The objective function quantifies "fit" and is defined $F : P \times E \times \vec{Q}_{P,E} \rightarrow \mathbb{R}$, where $\vec{Q}_{P,E}$ is a vector of quality attribute measures with each dimension a measure of "fit," and where $\mathbb{R}$ is the domain of real numbers. The preference relation $\preceq$ is defined $\{(P_i, P_{j \neq i}) \bullet F(P_i, E) \leq F(P_j, E)\}$ as the partial order on programs induced by F, such that $P_i \preceq P_j$ means that P_j is *at least as fit* as P_i, or equivalently that P_j is *weakly preferred* to P_i. As a notational convenience, if $F(P_i, E) < F(P_j, E)$ then $P_i \prec P_j$, or P_j *is strictly more fit* than P_i, or P_j is *strictly preferred* to P_i.[4] Informally, S can be regarded as an objective statement of **designer intent**, and applying S to each $P_j \in H$ (i.e., to the design space) induces a **fitness landscape**.

The set $\Phi \in H$ of **candidate solutions** of the design problem is defined by all programs that *correctly* implement the intended function ϕ. Here the meaning of "correctness" is restricted to "computes the right answer"

[2]Note that $j \geq i$ in Δ induces a directed acyclic change history and also allows a program to be its own δ.

[3]This exclusion is useful for later discussions, but is not an intrinsic feature of preference structures.

[4]A more expressive relation can be defined (for example [143]) but the present definition is sufficient for the discussion that follows.

independent of all other considerations such as time and memory. H may contain zero or more candidate solutions; however, for present purposes we will assume that if P_j is a candidate solution, then $\neg\exists\delta(P_j, P_x) \in \Delta$, i.e., candidate solutions reside on the frontiers of H, but not every program on the frontier of H is a candidate solution. The **design solution** is a distinguished path $\pi_{0,f} \in \Pi_{0,f}$ such that $P_f \in \Phi$. Thus, functional correctness is necessary but not sufficient condition of a design solution.

A **rational designer** (equivalently, a **rational design process**) always finds a **design solution** $\pi_{0,f}$ that is at least as fit as any other program in the fitness landscape, i.e, $\neg\exists P_x \in \Phi_H \bullet P_f \prec P_x$.

3.3 Architecture and Programs

In a "theory of computation" sense it makes no difference whether the designer seeks one, two, or hundreds of computer programs in that two programs can not compute any functions that are not computable by one, and our conventional notion of "program" is a matter of packaging, or *extra-functional structuring*, much like the choice of how to present a proof in a paper is extra-logical to the formal subject or validity of the proof itself.

We know, however, that these structuring decisions have enormous practical impact on various qualities of systems composed from many program components, such as performance, modifiability and so forth.[5] This is reflected by the differentiation of the functional correctness of programs (candidate solutions) from the extra-functional attributes of programs (preference structure) in the ideal rational design process.

We can justify on practical grounds a further differentiation of the designers whose concerns are achieving functional correctness in programs—**programmers**—from designers whose concerns are achieving extra-functional attributes—**software architects**. Architects and therefore concerned with the extra-functional structures that give rise to extra-functional attributes (i.e., **software architectures**), which in turn define the functional scope of programs and their respective criteria of correctness.

The design space H can be suitable modified to accommodate an additional (and not unreasonable) assumption that software architectures can be formally encoded (i.e., there is a formal notion of well-formed architecture). For, example $H' =< A_0, \Delta_A, H^+ >$ for a new design space that is partitioned by an architecture subspace and one or more program subspaces.

[5]In fact, there are good reasons to suspect that systems at different scales will exhibit different kinds of phenomena (the above qualities, for example), for which different kinds of explanation are in order even if they can be explained in other ways from more fundamental theories [11]. However, nettlesome issues of "emergent behavior" *vis–à–vis* "predictability" seem to arise naturally in predictability by construction, and are intentionally sidestepped in this research by grounding theories of predictable behavior in theory of computation (See Defs. 3.9, 3.10, and 3.12, on or near page 58).

Stipulating that each program design solution in H^+ has A_f as a prefix results in a **continuous design space**, i.e., one in which architectural decisions "come first," but where both regions of the design space meet in some software artifact that formally belongs to both A and P, for example an interface specification that defines each $P_0 \in H^+$. A and P define a syntactic language of well-formed architecture and program artifacts, respectively; the intersection of these languages defines a syntactic language of *seam artifacts*.

Terminology: Software architecture, architecture, design. Hereafter, "software architect" and "architect" will be used interchangeably, and similarly "software architecture" and "architecture"; "software designer" or simply "designer" will be used to refer to architects and programmers as a class; and, "software artifact," "software" and "design artifact" will all be used interchangeably to denote formally-encoded architecture and program descriptions.

Practitioners will rebel at this ideal partitioning just described, and would do so fully aware that contemporary software engineering practice salutes the ideal. Nonetheless, the *ideal separation of concerns* outlined above are all well motivated: distinguishing functional correctness from extra-functional quality, attributing extra-functional quality (in non-trivial software systems) to architectural structures, distinguishing architecture design from program design, and assuming that architecture "comes first."

3.4 Technical Definitions

Before turning to the Seam, several key concepts are defined: §3.4.1 describes different kinds of environments E that Seam designers will use to assess "fit." §3.4.2 makes more explicit the notions of architecture, program and interface used in place of the schematic program artifact P. §3.4.3 describes what it means for design qualities in the Seam to be predictable, and then more precisely what it means to be formally predictable in §3.4.4. The question of how predictability is related to bounded rationality is taken up in §3.4.5, and serves as a segue to Chapter 4 *The Architecture/Program Seam.*

3.4.1 Operational and Developmental Systems

Software systems are developed for many purposes, and for different kinds of stakeholder, each of whom perceives their own needs relative to what they also perceive to be some system that defines their own context of use. Deciding where to draw the boundaries among the various stakeholder systems is therefore not a trivial undertaking. Clearly, the designer needs a conventionalized way to classify environments in terms of typical stakeholder perceptions. One conventional classification defines operational systems (Def.3.1) and developmental systems (Def.3.2):

Definition 3.1 (Operational System) *A system that relies on software to achieve one or more prescribed business objectives.*

Definition 3.2 (Developmental System) *A system that produces software for use in* operational *and* developmental *systems.*

The apparent circularity in Def. 3.2 (i.e., the use of "developmental system" to define itself) is not problematic. Indeed, a distinguishing characteristic of software design is the *strong* emphasis placed on the impact that the design of software has on its own *design* processes. In contrast, industrial design processes will strongly emphasize the impact of design artifacts on *manufacturing* (*vis fabrication, production*) processes.[6]

3.4.2 Architecture, Components and Interfaces

Definition 3.3 (Software Architecture) *The structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*

This definition is adopted "whole cloth" from a standard reference on software architecture [16].[7] Of course, the definition leaves much room for possible interpretations of "structure or structures" and "software elements." However, for the purpose of this research "software element" will be interpreted as meaning "component program":

Definition 3.4 (Component Program) *A sequence of program statements that can be executed by an abstract or real computing device, having specified interfaces and explicit context dependencies only.*

This definition is an amalgam of those provided by Szyperski [160] and Heineman and Council [65]. The purpose is not to define software component, but to use the definition of *component program* to build a bridge from the ideas of rational design, which does not depend on software component technology, to prediction–enabled component technology, which exploits software component technology in very specific ways.

Including "abstract or real" in the definition allows latitude in classifying as programs any data that is executable on a real device, such as a desktop computer, on a virtual device, such as the Java Virtual Machine, or on an

[6]Once again shining a bright light on the software engineering's genetic defect. To reiterate: programming is a design process, not a production process.

[7]The actual definition in [16] begins "The software architecture of a program or computing system" etc., which however is consistent with its use in this thesis.

ideal or abstract device, such as a Turing Machine. Note that Def. 3.4 does not require that computing devices and the symbolic languages they execute be "Turing Complete," and indeed the use of ideal computing devices that are formally weaker that Turing Machines (i.e., can compute fewer functions) is quite important for the analysis of program and system behavior. However, unless otherwise stated, "computer program" and its synonyms will assume the usual "Turing Complete" sense of the term.

Def. 3.4 relaxes Szyperski's criterion that components have *contractually-specified* interfaces to simply having *specified interfaces.* The work reported in this thesis does not strictly conform to Meyer's notion of interface contract [119], so that criterion is not included in the definition. In contemporary programming practice, a criterion of good program design is that whatever a client program needs to interact with a program is defined by its interface, and that is what is intended by "specified interfaces" in the definition.

An interface defines program invariants both by what is—and is not—specified on the interface. Parnas introduced *information hiding* as a criterion for interface design, i.e., what is not specified should not be assumed [133]. Liskov developed developed formal foundations for data abstraction [103], and later with Wing extended these notions to behavioral subtypes [104]. Meyer's more general notion of *contract* [119], which he situtated in a strongly-typed object oriented programming language, and also within an overall object-oriented software design, has become the dominant metaphor for designing software interfaces.[8]

However, a profound and highly pertinent insight of Robin Milner's was to link the concepts of *observation* and *interaction* [121] by defining interaction as mutual, simultaneous observation. Perhaps one should not read too much ontological meaning into what is after all a mathematical contrivance, but there is something quite significant in recognizing that to observe a system is to interact with it, and to interact with a system is to observe it. Nonetheless, it is clear that the notions of observation and interface, as well as notions of system (and component) boundary, and what is internal and external to systems (and components), are all closely related.

Concretely, *any* observable phenomenon of a system is a source of (intended or unintended) coupling between that system and any other system that interacts with it. It is useful to theory and practice that we distinguish between the *defined* and *potential* interfaces of software.

Definition 3.5 (Defined Interface) *A software artifact that specifies syntactic and behavioral invariants of a component program that govern permissible interactions with that component.*

[8]It should be observed that the term "contract" has become diluted, and in the literature frequently means something much weaker than Meyer intended.

This definition is more specialized than might be expected,[9] but it is consistent with the discussion about contracts, etc., above. Syntactic invariants refer to anything that can be expressed as syntactic criteria of well-formed artifacts, for example as might be expressed by the type system in a language such as Java; and behavioral invariants refer to any observable behavior of a computational process that results from executing a component program, and is not restricted to only those behaviors that can be syntactically checked and enforced.

Definition 3.6 (Potential Interface) *The set of all externally-observable phenomena of component programs when executing in some (not necessarily intended) computing environment.*

The point to note about the definition of *potential* interface is that it does not describe a design artifact, but rather gives a name to a real but at least to some extent inscrutable interface that is only approximated by a defined interface. Shaw's notion of *software credentials* [149] is one way to progressively improve the fidelity of a defined interface to a "real" interface; the notions of *analytic interface* introduced in Chapter 5 is another way that is grounded in predictability by construction.

3.4.3 Predictable Behavior

Fitness can be formally expressed as a distance measure between what stakeholders of a system *require*, and what the artifact *delivers*. Designers choose from among alternative paths in the design search based on the anticipated (predicted) *effect* that decision will have on optimizing (minimizing or maximizing) this distance. The fitness function F described in §3.2 encodes the subjective, or "evaluative judgements" of the designer, but "test" in Simon's "generate and test" requires a basis in objective, observable phenomena. Accordingly, we separate the phenomenology of "fit" (here) from its evaluation (in §3.4.5).

Documented taxonomies of system phenomena and measures of these phenomena vary considerably in terminology and emphasis. For example, one ISO standard [156] describes six "characteristics" (functionality, reliability, usability, efficiency, maintainability, and portability), each of which has a number of "sub-characteristics" (for reliability these are: maturity, fault tolerance and recoverability), numbering over twenty-seven (sub-)characteristics in total. Not to be outdone, one Wikipedia entry describes a bestiary that includes over seventy "system quality attributes."[10] In brief, there are many taxonomies; perhaps some have theoretical and practical utility.

[9]For example, "Software interface: A boundary across which two independent entities meet, and interact or communicate with each other" (`http://www.sei.cmu.edu/architecture/start/glossary`.)

[10]http://en.wikipedia.org/wiki/List_of_System_Quality_Attributes

For the purpose of this research, however, these are all simultaneously too elaborate in the phenomena they differentiate, and too vague in the criteria of their differentiation. Here, two classes of phenomena are distinguished, system phenomena (3.7) and computational phenomena (3.8):

Definition 3.7 (System Phenomena) *Any externally observable phenomena of an operational system or developmental system.*

Examples of system phenomena include: the elapsed time between the arrival of some stimulus to the system and the response generated by the system, and the total human effort required to produce that response.

Definition 3.8 (Computational Phenomena) *Any externally observable phenomena of an executing program.*

Examples of computational phenomena include: the displayed prime factors of a natural number, and the elapsed time between the start and end of that computation. The terms *computational phenomena* and *runtime phenomena* are regarded as synonyms.

Terminology: Phenomena and Behavior. To avoid cumbersome phrasing, unless otherwise stated "phenomena" will hereafter be taken to mean computational phenomena, and in the interest of style the terms "phenomena" and "behaviors" will be used interchangeably, with preference for the former in definitions, and the latter in prose.

Theories of system behavior must account for the behavior of people. Examples of these include game theory and mechanism design, which however are outside the scope of the work reported here. Theories of computational phenomena, on the other hand, need account only for the behavior of computers and the real–world effects generated by computation on other (non-human) parts of the system. It is this class of phenomena that are of interest to the research reported here. Of specific interest are behaviors that are *predictable*, and for this purpose *all* behavior will be defined in terms of some underlying computational model.

Definition 3.9 (Functional Phenomena) *All phenomena P_f that are (in principle) observable on a Turing Machine.*

The reference to Turing Machine in Def. 3.9 is to be understood as a reference to the strongly conjectured equivalence class of abstract computing devices of which Turing Machine is but one well-known member. Also, the phrase "in principle" will be taken to be implied in all further definitions of behavior.

Definition 3.10 (Extra-Functional Phenomena) *All phenomena P_{xf} that are observable on a suitably enhanced Turing Machine, or on some abstract computing device that a) can be appropriately demonstrated to be an augmentation of, or an abstraction of a Turing Machine, and b) can itself be simulated on a Turing Machine.*

The terms "appropriately demonstrated" and "augmentation or abstraction of" are intentionally suggestive of the wider latitude granted to a discipline of engineering than would be appropriate to a discipline of mathematics.

To give a concrete example, the theory that supports generalized rate monotonic analysis (RMA) [89] is used as a foundation to reason about time in many "hard real-time" systems. RMA's underlying abstract machine is not Turing complete—it describes computational states (*schedulable units*) in terms of the time they consume, and transitions between states in terms of an underlying fixed-priority scheduling discipline. For example, concurrent synchronizing pipelines of component programs can (in some cases) be given an interpretation as independently–scheduled sequences of RMA tasks [62]. The behavior of the "RMA task programs" that run on RMA abstract machines is expressed in terms of non–blocking execution time, and is quite orthogonal to the functional behavior of component programs of the original (uninterpreted) program. However, the abstraction of functional behavior to execution time, combined with formally demonstrated theorems internal to RMA theory and empirical evidence of the predictive strength of RMA for real programs, provides an appropriate basis to have confidence that there exists *some* encoding of the RMA abstract machine on, say, a Turing machine.

Definition 3.11 (Predictable Phenomena) *Given a set A of artifacts* defined by some formal invariant *inv, and a set O of observations of some phenomena $P_f \cup P_{xf}$ of programs $a \in A$; the set of* inductively predictable phenomena *$O_i \subseteq O$ is defined by a statistically-significant correlation $C(A, O)$.*

The term "formal" in Def. 3.11 makes clear that the invariants are on the syntactic structure of the software artifacts, and as such define criteria of "well-formed artifacts." Also, the term "correlation" should be interpreted broadly as any descriptive or inferential statistical measure of correspondence between structure and phenomena [156, 155, 124]. Previous research by Magnus Larsson has demonstrated that the "predictive strength" of a theory for some analytically-predictable behavior can be established as an objective property of that theory [98].

Predictable phenomena includes as a degenerate case any behavior that can be inferred as a consequence of software testing, since there is only one artifact in the set A. It excludes, however, so-called "quality metrics" such as *cohesion* and *coupling* [157] and many others, most of which purport to be predictive of how difficult it will be for programmers to understand the measured programs but lack grounding in any computational theory. It also excludes formalizable architectural styles [3] or other formalizable patterns that have a demonstrated effect on software behavior, but also lack a computational theory to express those effects.

3.4.4 Formally Predictable Behavior

Predictability is defined using terms used in the theory of programming languages, in particular the cluster *theories, models, semantics* and *interpretations.*

Definition 3.12 (Theories and Semantics) *The abstract computing devices that simulate functional and extra-functional behavior, and the collection of theorems that observe these simulated behaviors, constitute* **theories** *of those behaviors. The (abstract) programs that are executed on (simulated by) these abstract devices are* **models** *of those behaviors. A formal correspondence between a component programming language L and some theory T defines a* **semantics** *of L* **with respect to** *T. Component programs in L are said to have* **interpretations** *in T, and any such model of a program is said to be an* **interpretation** *of that program.*

As a convenient shorthand, "theories" and "semantics" will be used interchangeably despite the difference in their formal definitions. This is justified because in the research reported here, theories without an accompanying semantics are of little practical interest. Similarly, the terms "(extra-) functional semantics" and "(extra-) functional theories" will be used as synonyms. The terms "behavioral semantics" and "behavioral theories" will be used when the distinction between functional and extra-functional is neither useful nor required.

Definition 3.13 (Formally Predictable Behavior) *A phenomenon is predictable if it has a defined semantics.*

3.4.5 Standards of Fit

It is significant that even as Simon was developing theories of rational design (§3.1) he was simultaneously questioning the ability of humans to make rational decisions in the first place. His arguments were based on known limits of human cognition, on the computational complexity of linear programmed multi–attribute optimization, for the corresponding need for heuristics, and on the necessarily co-evolving solution and the designer's understanding of the problem itself. In the latter situation, the fitness landscape is co-created

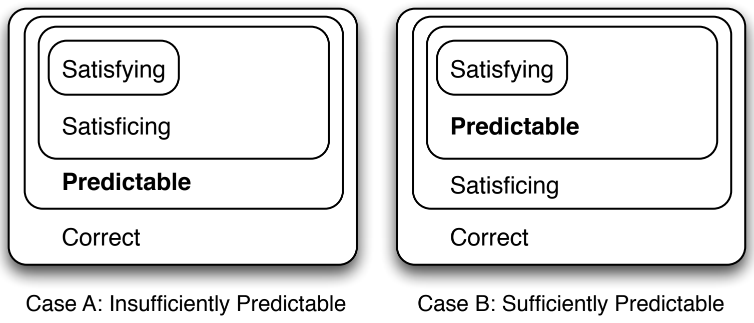


Figure 3.1: Satisfying, Predictably Satisfying, and Satisficing Designs

along with the search. There are many reasons, then, to assume that designers can not be fully rational, and that even if they could be, the design process can not be regarded as rational overall.[11]

Simon's solution was to relax the assumption of rationality to *bounded rationality*. This shifts the focus from a search for designs that maximize a measure of satisfaction of stakeholder need, to a search for designs that are *sufficiently satisfy* stakeholder needs, for which Simon coined the term "satisficing" [151]. The bounds of rationality manifest as the gap between designs that are maximally fit with respect to some criterion, and those that are sufficiently fit.

It is clearly desirable that the *predictive strength* for any behavioral semantics for this criterion (i.e., its correlation $C(A, O)$, Def. 3.11) lies somewhere within these bounds. This might seem like a contradiction in terms, in that (by definition) the upper–limit of predictive strength for a theory of some behavior would define a new bounds on rational design for that behavior. In strictly definitional terms this would be the case. However, while predictability is an objective measure of a behavioral semantics, sufficiency (as with trust) is situationally- and socially-dependent [120, 84]. Software engineers and their customers are likely to regard *all* design predictions with healthy skepticism, as they should.

Figure 3.1 provides a Venn diagram interpretation of *sufficiently satisfying* designs *vis–à–vis predictably satisfying* designs. Case A) represents the practical reality, where some designs that were predicted to satisfy stakeholder needs will turn out to not do so, while Case B) represents the (likely unobtainable) ideal that all designs predicted to satisfy needs do so. In Case A, testing, expert review and other means must be used to "close the gap" between the objective measure of a theory's predictability and the perceived

[11]Parnas and Clements argued from much less fundamental grounds but reached the same conclusion and offered a range of practical remediations, many of which remain pertinent to contemporary software engineering practice [134].

sufficiency of that theory in some social context.

While the gap between the predictive strength of a theory and its sufficiency as a definitive basis for design decisions will more than likely never be completely eliminated, we can hope to establish sufficient trust in engineering theories to justify a significant reduction of overall effort to close the gap, and it is in this way that the practical viability of predictability by construction will be established.

Chapter 4

The Seam

Previous chapters have discussed the distinct but overlapping design jurisdictions of architects and programmers. This chapter describes the nature of these overlaps in closer detail, from the perspective of architects and programmers, and then from the perspective of the Seam, an area of common ground. The Seam is defined in such a way that architects and programmers can have joint custody over a collection of design artifacts, each of which has *dual* significance—one that is pertinent to the architect's interests, and one that is pertinent to the programmer's interests. The result is (as will be demonstrated in the case studies of Chapter 9) better integration of architecture and program in systems, and a significant enhancement of rationality in software design that is enabled by predictability by construction.

Table 4.1: Overlapping Jurisdictions and Seam Consolidation

	Architect	Programmer	THE SEAM
Concerns	satisficing results for all attributes in all environments of use	correct computational results in the operational environment	predictable computational results in the operational environment
Theories	many attribute criteria, theories span rules of thumb to formal bases	one dominant attribute criterion, established theory of computation	extensible theories of runtime behavior that are statistically or formally validated
Rules	open-ended policy-enforced design rules; tacit or asserted intent	pre-defined language syntax and semantics; functional behavior by construction	extensible computer-enforced design rules; predictable runtime behavior by construction
Explains	explain the "why" to external stakeholders: persuasively justify major design tradeoffs	explain the "how" to internal stakeholders: concisely explain program behavior	justify significant design decisions in terms of their impact on actual or predicted runtime behavior
Abstracts	components and connectors, styles, "4+1" views, analysis and simulation models	procedures, interfaces, classes and modules; idioms, patterns and component models	software components and component models that have dual (architecture and program) meaning

Table 4.1 provides a concise summary of the overlapping jurisdictions, and interests, of architects and programmers, and their consolidations in the Seam. The chapter is organized by the structure depicted in Table 4.1, with each area of overlapping jurisdiction represented by each of the rows in the table described in its own section: Design Concerns (§4.1), Design Theories (§4.2), Design Rules (§4.3), Design Explanations (§4.4), and Design Abstractions (§4.5).

4.1 Design Concerns

In "design as search," what is it that software designers seek to obtain? What benchmark measure of obtainment, or confidence in future obtainment, is required by the designer to commit to the search of one part of the design space in preference to all others? These questions pertain to *design concerns*.

The Seam I (Design Concerns). Designers in the Seam seek software systems that exhibit predictable computational behavior in the operational environment.

The emergence of *structured programming* in the 1970's demonstrates that programmers have always been concerned with design qualities beyond functional behavior. For structured programs: analyzability and understandability; for object-oriented programs: modifiability and extensibility; for design patterns: reusability and composeability; for functional programming and logic programming there are others. We might argue about which qualities apply to which style of programming or to which paradigm, but the principle is surely established.

Similarly, architects are concerned with functional behavior, not only as a necessary criterion for design solutions, but as a way to define functional specifications for quality attributes. For example, for program P the architect may define *safety* properties (X will **never** hold), e.g., robot arm never moves past a certain point, unencrypted messages are never sent; or *liveness* properties (Y will **always eventually** hold), e.g., robot always eventually halts after receiving kill signal, message sends are aways eventually logged. Each of these examples is expressible in a *temporal logic* that verifies properties on infinite program traces—a purely functional concept.

Architects and programmers have the same overall goal of developing software systems that are most fit for use; they may differ in the attributes of fit they seek, and in the evaluative standards of fit they apply. Architects and programmers share quality concerns, the scope of these qualities vary—system scope for architects, and program scope for programmers, although these differences in scope are not fixed rules. Similarly, architects and programmers differ in their interests in fitness environments—operational environments for architects, developmental environments for programmers (with the same caveat about no fixed rules). Lastly, the benchmark measures of obtainment also vary, with architects concerned primarily with sufficiently satisfying customer needs and programmers concerned primarily with correctly achieving functional correctness (same caveats).

The key points worth noting about design concerns in the Seam are:

a) fitness attributes are restricted to *computational behavior* (Def. 3.8, pp.56) of operational systems (Def. 3.1, pp.53).

b) the standard of fit is *predictability of behavior* (Def. 3.11, pp.57).

There is nothing intrinsic to software design, or to the differing concerns of architects and programmers, that requires the Seam to be limited in its focus to only computational phenomena or to operational systems. This narrower focus adopted by the Seam reflects the particular research objectives of predictability by construction, and in this sense is quite arbitrary. On the other hand, Chapter 10 takes up the issue of how a technology infrastructure for the seam is itself designed, and the novel challenges posed by that undertaking. In that case, the more general class of system phenomena within developmental systems (as well as the more nebulous concerns for satisficing results, discussed next) become *dominant* concerns.

On the subject of predictability, a discipline of engineering design requires that the role of intuition and subjective judgement replaced wherever possible by objective and measurable quantities of fit. The Seam therefore relies on definitions of predictability that are based in the objective and observable runtime behaviors of software; and this in turn serves to increase the level of "rationality" in the design process. Predictability is also defined in a way that reflects common ground for the architect's native interest in *sufficiently* satisfying behavior and the programmer's native interest in *correctly* satisfying behavior. These respective interests reflect the more situational standards of fit used by architects, and the more sharply defined standards of fit imposed on programmers by the formally-defined structure and interpretation of component programs.

4.2 Design Theories

Each of the designer's navigation decisions through the design space is made with the expectation that it moves the artifact closer to the design goal—by producing a candidate solution, or by improving fit. What analyses support these judgements? What is the objective basis for these analyses? What confidence bounds or other caveats attend the use of these analyses? These questions pertain to *design theories.*

The Seam II (Design Theories). Designers in the Seam use theories of computation that can predict the runtime behavior of software systems and that have demonstrated validity.

Programmers work with highly refined symbol systems called a programming languages, each of which defines a *syntax* of well-formed programs, and a *semantics* that defines the functional behavior of well-formed programs, although extra-functional behavior of programs also results from semantics, for example on type systems that support memory safety.[1] Semantics assigns to each well-formed program an *interpretation* as an ideal program that executes on an ideal computing device (e.g., Chemical Abstract Machine [19]).[2] For general-purpose programming languages, these ideal devices belong to an equivalence class that defines a *theory of computation*.

As discussed under *design concerns*, architects also rely on well–defined programming language semantics. However, architects also make use of their own languages; the class commonly referred to as *architecture description language* (ADL) does not, however, require of its members a semantics in any theory of computation. Indeed, it might be argued that such a requirement would simply result in yet another general-purpose programming language. Instead, ADLs as a general rule tend to represent structure without behavior, with the objective of being suitable (in principle) to many theories and analyses, for any design qualities in $\vec{Q}_{P,E}$ of a design objective function F (see §3.2, pp.49). Unlike programmers, then, architects must deal with an open-ended set of theories, the vast majority of which have not been formalized as a semantics of any ADL.

To give one concrete illustration, consider real-time queuing theory (RTQT) [49] and rate-monotonic scheduling theory (RMA) [89], just two of many real-time performance theories for just one "performance" quality in $\vec{Q}_{P,E}$. RTQT and RMA are distinguished by the kinds of guarantees they provide (statistical and absolute, respectively), assumed scheduling discipline (earliest deadline first, fixed–priority), and design objectives (maximize utilization, guarantee deadlines). These are substantial theories that have not in general been used to supply ADLs with formal semantics (with the performance reasoning framework described in Chapter 8 being one exception). In addition to these and other substantial theories there are also heuristics and various "rules of thumb" that architects and even programmers on as well, for example when to use a two-tier rather than three-tier client/server pattern, when it is appropriate to use model–view–controller.

The key points worth noting about design theories in the Seam are:

a) *theories of computation* narrow the focus of the seam to *formally predictable* behavior (Def. 3.13, pp.58).

b) the behavioral theories that predict these behaviors should have an accompanying *demonstration of validity* (see discussion of Larsson's work in the context of Def. 3.11, pp.57).

[1]Very few programming languages are defined with the precision and completeness implied by this discussion.

[2]The Chemical Abstract Machine (CHAM) is "based on the chemical metaphor" (Gerard), and was used by Milner to define a semantics of $\pi-$calculus [122], which indirectly served as a basis for programming languages [136] and composition languages [106].Inverardi and Wolf made direct use of CHAM for architecture analysis [79].

The emphasis on formally predictable behaviors invites the use of "non-standard" semantics of languages.[3] The goal is *not* to achieve an equivalent level of formalization of functional and extra-functional behavior, but to lay the groundwork for a systematic treatment of how new theories of extra-functional behavior are developed, and how new theories are integrated into architecture and program abstractions. The additional stipulation that design theories be validated serves the early-stated purpose of improving the overall level of "rationality" in the software design process.

4.3 Design Rules

The rational design process sketched in §3.2, pp. 49 casually asserts the existence of a design space, but where does this come from in the first place? The use of genetic algorithms as a *meta-heuristic* to construct a design space (and simultaneously a fitness landscape) is certainly a possibility [141, 132], but it is not yet to be recommended as a general approach to software engineering design. This begs as questions: What heuristics do designers use to identify a set of successor software artifacts of any given software artifact P_m (at least two must be possible for P_m to be worth including in a design space)? How do these heuristics correspond to design theories, i.e., what quality attributes do designers believe are maximized (minimized) by the successor forms, and what design theories (if any) justify these beliefs? In what ways (if any) do heuristic-generated successors expand or restrict downstream design choices, i.e., the design space that can be constructed from each successor? If there are restrictions, how are they enforced? These and similar questions pertain to *design rules.*

The Seam III (Design Rules). Designers in the Seam define design rules that, when enforced by a computer, satisfy the assumptions of analysis theories, and as a consequence software systems exhibit behavior that is predictable for these theories, by construction.

If there is a nexus of creativity, invention, codification and automation in design, it resides in design rules. Contemporary architects and programmers use remarkably similar language to talk *about* design rules—chiefly in terms of *architectural styles* [135, 2, 116, 51] and *design patterns* [52, 45, 25, 144]; these however differ more in terminology than in substance, and for this reason the term "pattern" will be preferred. Generally speaking, patterns codify best practice rather than invention, and accordingly are sometimes

[3] A core functional semantics of imperative programming languages is sufficiently common that it can been referred to as *standard semantics*[128].

justified by the heuristic that a pattern must have been used in three distinct contexts to qualify as a pattern.

However, with few exceptions (for example attribute-based architectural styles [88]) there has been little overt emphasis on grounding specific patterns in any specific theory of program behavior. This is not surprising given the number of possible design theories that might give rise to specialized patterns, with each $< theory, pattern >$ pair based in different and possibly inconsistent assumptions. The potential for inconsistency is especially problematic to achieving the "Alexandrian" [7] ideal of pattern composition that inspired much of the early work in object-oriented design patterns (which is also one of the few genuine differences between architectural styles and design patterns).

Architects and programmers differ in rules they use to achieve desired effect on attributes of fit, how strongly correlated these rules are to the desired attribute effects, and how the rules are enforced, but the Seam provides a consistent theme to consolidate the treatment of design rules:

a) design rules *satisfy assumptions* of theories of runtime behavior

b) from this predictability of behavior follows *by construction*

Programmers make good use of the formal strictures of modern programming languages, such as Java and C#. They might not appreciate, however, that the unforgiving syntax of these languages is defined in large part to permit an automated theorem prover called "the type checker" to prove theorems in a higher-order logic called a *type theory*, and that language syntax, type theory, and type checker combine to form what the programmer sees as the *type system* of the programming language. The type system's sole purpose is to establish *invariants* on program executions; from these various important and useful behavioral properties can be proven, for example memory safety. Type theories establish strong but quite specific properties of program behavior; semantic theories establish weaker but more general properties of program behavior, and in addition make various assumptions about the runtime environment of programs, about heap store, thread scheduler, etc. The same principle applies to programming language type theories and semantics: well-formed programs have predictable functional behavior (for type systems: provable behavior), by construction.

The Seam anticipates that the analogous principle applies also to extra-functional behavior, such that type systems or semantics of extra-functional behavior can be made predictable by construction.

4.4 Design Explanations

The success of the design depends on more than the qualities of the software (its correctness and its fit)—it depends as well on the explanation to stakeholders of the software design, as well as of the software design process. Why does the software exhibit one structure and not another? What is the objective basis for designer's belief that the software satisfies its requirements? Why does the software exhibit more of one quality attribute Q_0 and less of another Q_1 in $\vec{Q}_{P,E}$, and might some other tradeoff have been made, possibly involving some other quality attribute Q_2? These and similar questions pertain to *design explanations.*

The Seam IV (Design Explanations). Designers in the Seam provide their stakeholders with objective justification for significant design decisions, based on confirmable evidence of actual or predicted software runtime behavior.

Without elaborating a theory of stakeholder, we observe that a software design process has internal stakeholders (architects and programmers) and external stakeholders (end customers). Both classes (and we can define many such classes) have need for explanations. External stakeholders are generally interested in explanations of the design process, and in particular answers to "why" questions, as in "why this design (or this design *decision*) and not some other?"; answers to these questions are sometimes referred to as *design rationale*, and there are various ideas about how the architect might construct rationale [12, 24, 161]. Internal stakeholders are generally interested in explanations of the software artifacts, and in particular answers to the "how" questions, as in "how does this software work, and how does this working satisfy its requirements?"; answers to these questions are sometimes referred to as *technical documentation*, and there are various ideas about how the programmer might construct technical documentation, including various widely-used tools (Doxygen[4], JavaDoc[5]) as well as Knuth's more inspired (and largely unrealized) idea of "literate programming" [167, 153].

Of course, "internal" and "external" are relative terms, and architects and programmers are stakeholders of one another, and might require of one another various kinds of explanations. To illustrate one case, programmers for example might require from architects a form of design rationale that is slightly different than that required by end customers, what we might call *architectural intent*, to answer "what/where" questions, as in (metaphorically) "what region of the design space should be explored, what regions should be avoided, and where is the intended destination?"

Architects and programmers must provide explanations and justifications for critical design decisions:

a) designers justify key design decisions

[4]http://www.stack.nl/~dimitri/doxygen/
[5]http://java.sun.com/j2se/javadoc/

b) justifications are based on confirmable, objective evidence

These points are made concrete in the following definition of *objective justification*:

Definition 4.1 (Objective Justification) *An argument of the form "The decision to change the artifact a_1 to a_2 is justified by the corresponding change $p(a_1)$ to $p(a_2)$ that will be, or already has been, observed for some predictable behavior P." Formally: $\delta(a_1, a_2) \hookrightarrow \delta(p(a_1), p(a_2))$.*

Note that "phenomenon" refers to an *formally predictable* behavior (Def. 3.13). In an ideal design process, each design decision (i.e., each decision to search one region of the fitness landscape it preference to others) would be accompanied by an objective justification; the set of these justifications could be regarded as the documented *design rationale.*[6]

The architect will attend to external stakeholder needs at the earliest stages of design and throughout: when the scope of a software product is established; as stakeholder needs are *formalized* as system requirements; as each requirement, in turn, is *reduced* to testable criteria of fit; and as tradeoffs among requirements are made as a consequence of multi-criteria optimization. If success were determined exclusively by designer judgements or objective measures of fit, then perhaps this is the best the architect can do. However, key stakeholders, such as investors, might be concerned less with the software product than in the rationale for specific design tradeoffs. They might require a *justification* of why the product is more fit in some ways and less fit in others, or might go further by requiring an *explanation* of which *specific* design decisions have as their consequence some aggregated or criteria-specific measure of fit.

Programmers generally have stakeholders who are internal to the design activity, in particular other programmers and the architect. Peer programmers are more likely to be concerned with a concise and accurate account of how various internal parts of a program work, and with an explanation of any subtle interactions among those parts that are not readily apparent from program source code, or are not expressible in the language syntax. Programmers may also be required to provide to their architect stakeholders evidence, or an explanation of how their programs satisfy architectural intent. For example, the architect might require a demonstration that a program conforms to call-level protocol of allowable calling sequences on various defined interfaces [22] (Def. 3.5, pp.54). Conformance also extends to a program's *real* interface (Def. 3.6, pp.55), in particular to any observable phenomena of programs on which the architect's decisions depend, for

[6]One interesting speculation is that the extent of an architecture is defined by those design decisions for which explicit explanation and justification are likely to be required by stakeholders of current and future developmental environments.

example the observed average-case or worst-case runtime performance of a program in a performance-critical system.

4.5 Design Abstractions

If software engineering has any claims of supremacy among other engineering disciplines, it is that it is *non plus ultra* in defining and using abstraction.

Two points are worth emphasizing. First, the kinds of abstraction of foremost interest in this research are *formal* abstractions—those that are defined in terms of a formal symbol system and by (one or more) interpretations from their symbol systems to *other* formal abstractions (e.g., programming language to machine language, programming language to transition system). Ultimately, these are grounded in some *informal* (i.e., empirical) domains of observable runtime behavior (e.g., transition system to execution time).

Second, software engineering (and engineering in general) is a *practical* discipline—it is concerned as much (and more) with what can and does work *in practice* than with what could and should be done *in theory*, with an ill-defined but changeable boundary between these laying somewhere in the penumbra of "good enough." Vanishingly few[7] programming languages used in practice have a thoroughly defined formal semantics, but most software engineering problems can be solved as if they do.

This research attends to software engineering practice, and accordingly defines abstractions that are *sufficiently* formalized. Indeed, what it is that defines "sufficiently" in this context is itself a matter of interest in this research, as previously discussed (§3.4.5, pp.58) and more extensively in Chapter 10 under the general heading of "co–refinement."

The Seam V (Design Abstractions). The Seam uses software components and component models that have dual and simultaneous meaning in software architecture and computer program.

Architects and programmers require abstractions that address their concerns, that are based in validated theories, that have well-defined rules that govern their use, and that provide an objective basis for making and justifying design decisions based on their use. Achieving an effective integration of architecture and program design also requires that these design abstractions be not biased towards architecture or programs, but instead reflect common ground for both. This research offers a proof of existence in the form of *prediction-enabled component technology* (PECT) that such abstractions can be defined, and can be substantially supported by automation. Figure 4.1

[7]The definition of SML [123] is a noteworthy and illustrative exception to the rule.

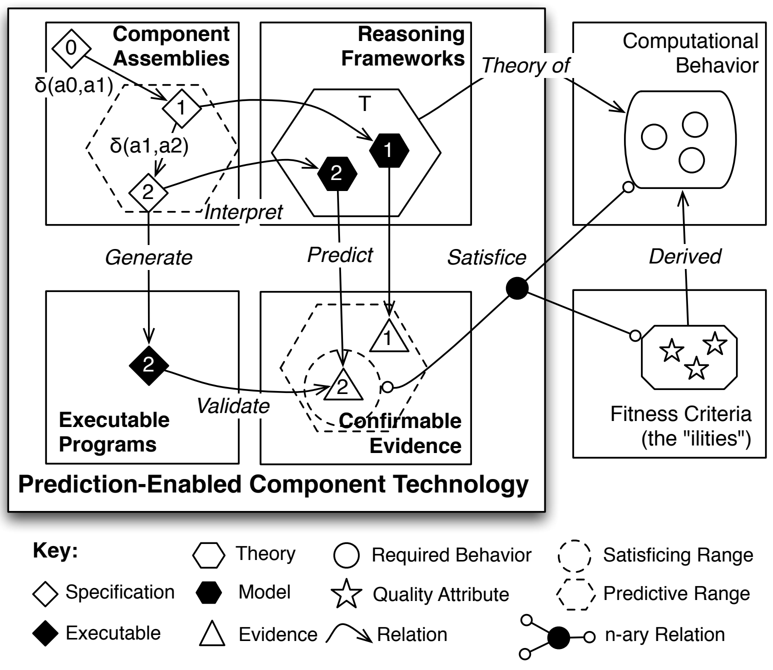


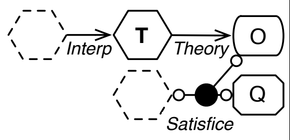
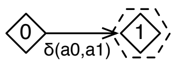
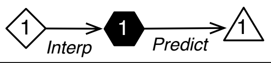
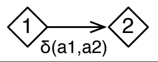
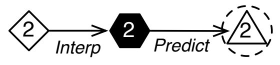
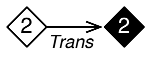
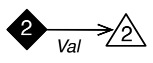
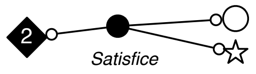
Figure 4.1: Prediction-Enabled Component Technology in Context

graphically depicts the large-scale abstractions of PECT—assembly specifications, assembly implementations, reasoning frameworks, confirmable evidence, and various relations that each correspond to some automated transformation of design artifacts of one abstraction to another.

At this point the discussion shifts in focus from metaphor, definition and conceptual framework ("The Seam") to prototype technology. The main ideas of PECT are more naturally introduced, then, in terms of a simple and generic vignette that illustrates how a designer uses PECT to search a small region of the fitness landscape, using validated extra-functional semantics, to produce software that "predictably satisfices" its requirement, and to objectively justify the design decisions leading to this particular design solution. This vignette is summarized in Table 4.2 as a sequence of design steps, each of which highlights the portion of Figure 4.1 it explains. Part II provides a more detailed view of PECT.

To ease the transition from abstract concepts to concrete prototype, each step in Table 4.2 is described using metaphors and definitions developed earlier, but each also introduces PECT-specific terminology, highlighted in **boldface** font, and pointers to the chapters that pertain to these terms.

Table 4.2: Seam Scenario using PECT

Step	Index to Figure 4.2	Key Ideas
-1		Quality attribute Q with runtime behaviors O recurs as a critical requirement. A **reasoning framework** F (Ch.8) is designed (**co-refinement**: Ch. 10), based originally in theory T, that is **validated** (Ch. 8) as sufficiently predictive of O, and which defines design rules (**analytic constraints**) that permit a sufficient range of systems (**assemblies**) to be designed.
0		A new system is to be developed with quality requirement q' with behavior o'. The **assembly** a0 in the **Pin Component Language and Technology** (Ch. 6-7) is the starting point.
1		The designer attempts to predict the behavior of a0 using F, but F reports constraint violations on the **analytic interface** of a components in a0. The designer produces new assembly a1.
2		a1 is **well-formed** in F, and its o' behavior is **predictable** in F. F constructs as an **interpretation** of a0 and invokes a (semi-) decision procedure to **predict** o'.
3		The behavior of a1 is **predictable by construction**, but its predicted behavior does not satisfy the requirement, leading the designer to produce a new assembly a2.
4		a2 is also well-formed in F, and its predicted behavior falls within the normative range of the acceptable values for the requirement.
5		The designer generates an implementation of a0 for the **Pin runtime** (Ch. 6-7).
6		Predictions do not provide absolute certainty, even when backed by **proof certificates** (Ch. 8). The designer **spot validates** a2 to see if actual and predicted behaviors of a2 correspond.
7		Because F has a **validated confidence interval** of its predictive quality (Ch.8), the designer can justify the design of a2, and reduce testing effort by some margin within the confidence of F.

Part II

Technologies

Chapter 5

Prediction-Enabled Component Technology

Part II of this thesis describes a prototype prediction-enabled component technology (hereafter, PECT). This prototype is freely available for download.[1] The technology at this download location is called the "PACC Starter Kit," or "PSK." See Ch.2 §2.1 for a discussion of the relationship between PACC and PECT. In brief, PSK is an instance of a general class of PECT. To maintain this distinction, I will use PSK to refer to the prototype, and PECT to refer to the general idea.

The PSK is simple in concept but requires the integration, and in several instances modification, of several non-trivial technologies. The overall PSK is packaged as a *PECT Perspective* in Eclipse; reasoning frameworks are implemented as *plug-ins* to the PECT Perspective. Figure 5.1 provides a summary view of the PSK, again in terms of the "quadrant" depiction (see Fig. 4.1, page 71), but this time highlighting technologies rather than the theory concepts:

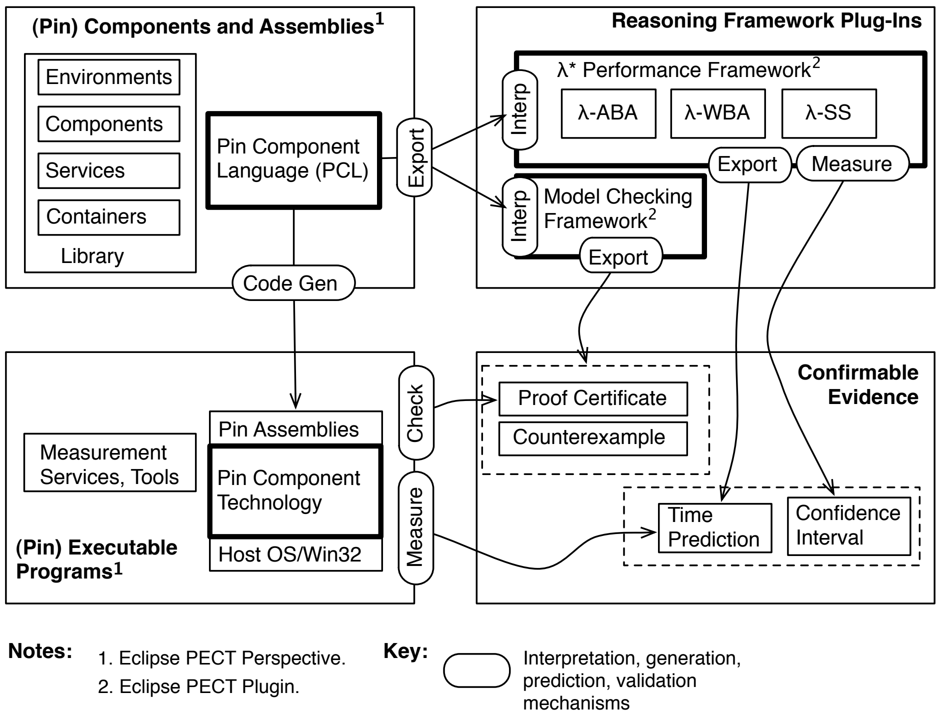


Figure 5.1: The PSK: A Prototype PECT

Pin Component Language. The Pin Component Language (PCL) (bold box in upper-left quadrant of Figure 5.1) is the constructive basis of the PSK. It formalizes the Pin component model, and extends it in various ways to accommodate larger-scale abstractions such as *component assemblies* and

[1]http://www.sei.cmu.edu/predictability/tools/starterkit/index.cfm.

environments.[2] PCL is a language processor in the traditional sense. The "front-end" implements a syntax for the Pin component model (substantially based on "components and connectors") and semantics described in Appendix A). The "middle-end" performs static analysis and generates an annotated abstract syntax tree, which is exported for use by reasoning frameworks. The "back-end" is a code generator (the "Generate" relation in Figure 4.1), with ANSI-C and the Pin runtime environment as target language and target machine, respectively.

Pin Component Technology. Pin (bold box, lower-left quadrant of the figure) provides the runtime environment and services used by components, for example directory services, distributed message queues, container management, and life-cycle management for components and their assemblies. Pin also defines a binary standard for Pin components (WIN32 dynamically linked libraries), and a defined C application programming interface (API) for implementing components and for component access to runtime services. The Pin runtime environment implements a variant of real-time POSIX threads, with fixed-priority scheduling, 128 thread scheduling priorities, optional runtime support for priority inheritance protocol, and various extensions to support specialized Pin event types defined by PCL. Pin has a portability layer, and has been hosted on WindowsCE and WindowsXP and the RTX real–time extension of Windows; only WindowsXP is supported by the PSK.

Reasoning Frameworks. Reasoning frameworks are independently packaged and deployed as plug-in extensions to the PSK. The PSK includes several reasoning frameworks: for performance analysis, model checking analysis, buffer overflow analysis and memory footprint analysis; only the first two are discussed in this research. The performance reasoning framework is based substantially in generalized rate monotonic scheduling theory [89] and can be used to predict average–case and worst–case execution time in systems that have periodic and stochastic event arrivals. The model checking reasoning framework uses automatic abstraction–refinement to construct a sound but finite over–approximation of component and assembly behavior; the checker then exhaustively searches the finite model to confirm or falsify some behavior.

Confirmable Evidence. Reasoning frameworks announce their predictions by writing something to the computer display, or to a file, or by launching an external tool, such as a simulator. Each reasoning framework defines how predictions are confirmed. The performance reasoning framework uses direct observation; the PSK provides a measurement infrastructure with "hooks"

[2]In the PSK distribution, PCL is known instead as CCL, for "construction and composition language." PCL is preferred in the text to emphasize its basis in the Pin component model, and to deemphasize "composition," a term with too many (mostly overambitious) connotations

into the Pin runtime environment and tools to gather runtime data. The model-checking reasoning framework provides a counterexample if an assembly fails to exhibit a required behavior, and a proof certificate if it does exhibit required behavior; the PSK provides an interactive explorer for counterexamples and an automated proof checker.

The remainder of this chapter is structured as follows. Each of the main elements of the PSK (bold boxes depicted in Figure 5.1) are described: the Pin component language (§5.1), the Pin component technology (§5.2), the $\lambda*$performance reasoning framework (§5.3), and model-checking reasoning framework (§5.4). Because these elements are further elaborated in separate chapters, so the emphasis here is on key ideas. Each summary also includes an "In Action" illustration of the key ideas using examples that are available in the PSK. An important theme of the research is that properly locating the Seam requires that genuine attention be paid to the practical aspects of tooling and scale; the illustrations provide a sense of what was required to achieve the goals of this research.

5.1 Pin Component Language (PCL)

PCL combines elements of architecture description language (components and connectors), software component technology (Pin) and programming languages (UML statecharts, ANSI-C).

5.1.1 Components and Connectors

The choice of "components and connectors" as a syntactic core in PCL is not surprising—it is a choice made by many ADLs (Garlan makes the general case [54]) and component models ([59, 86]), it provides a natural correspondence of component to process algebraic interpretations ([106, 109]) and it is regarded as a fundamental view type for documenting software architectures ([44, 93]).

The interface of a Pin component is specified in terms of *pins*, which as a first approximation can be regarded as equivalent to *ports* in other notations, though pins are intentionally primitive in some respects—for example, they can not be aggregated, and there is no means to define pin-specific protocols. The *type* of a pin is a composite $< D, P, S >$ where D is either "sink" (inbound requests to a component arrive on these) or "source" (outbound requests of a component are made on these), P is either "asynchronous" or "synchronous" protocol, and S is a signature of data types transmitted on the pin.

PCL does not support connectors as "first-class" abstractions—new types of connector can not be specified directly in the language.[3] Instead, PCL de-

[3]Of course, the same effect can be emulated by defining components whose only task is

fines two basic kinds of connector, one each for *synchronous* and *asynchronous* interactions, that are used to connect synchronous and asynchronous pins, respectively, to one another.

5.1.2 Reactions and Interactions

Components in PCL are *reactive*—they begin their execution by *listening* for the arrival of stimulus on a sink pin, and *react* to any stimulus by performing some task, and when finished return to the listening state. As part of performing its task, a component may issue requests on its source pins, and these in turn become stimulus for any component with sink pins that have been *connected* to those source pins; their connection enables *interactions* among component reactions. A component may define one or more reactions, and each reaction may be *threaded* or *unthreaded*.

Since components are reactive, a reasonable question to ask is "where is the first event?" Components interact with their *environments* by interacting with a special class of components, called *services*, for which requirement for strictly reactive behavior is relaxed. This class of component can interact with the external world, for example on low–level devices. However, aside from this, and the use of the keyword **service** rather than **component**, services are defined in exactly the same way as components.

5.1.3 Reactions and Pincharts

PCL uses a subset of UML statecharts, called Pincharts, to specify the behavior of components. The subset is consistent with UML version 2.0 both in terms of semantics, and in resolution of choice points. One UML choice point is the choice of "action" language, and for this PCL uses a subset of ANSI C.

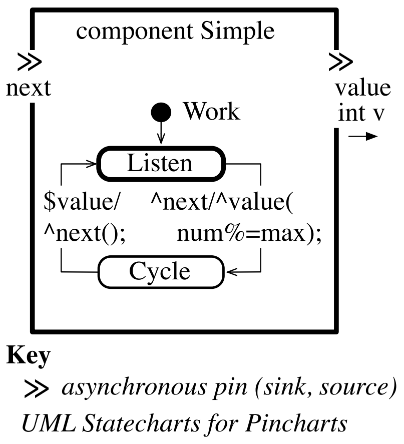
Example 5.1 shows a completely specified component `simple` (text on the left) and a UML diagram of the reaction `R` (on the right); the component responds to a request on its sink pin next by producing a value modulo a value max supplied at instantiation time. The correspondence should be easy to see for anyone familiar with UML notation; not surprisingly, the diagramming notation (which is *not* supported by the PSK) is somewhat easier to understand for simple to moderately complex reactions.

One important detail emphasized in Example 5.1 is the interpretation of pins p as pairs ($\hat{}p$, $\$p$) of UML events, where $\hat{}p$ is pronounced as "initiate interaction on p" and $\$p$ as "complete interaction on p". This interpretation allows PCL specifications to preserve the asymmetry of caller and receiver that is especially important to synchronous interactions that may exhibit blocking behavior.

to mediate interactions among other components. This was a much-debated topic in the early days of ADL research.

Example 5.1: Simple Component.

```
component Simple (int max)
{
 int num=0;
 sink asynch next();
 source asynch
   value(produce int v);
 threaded react R (next, value)
 {
  start -> Listen {}
  Listen -> Cycle {
   trigger ^next;
   action ^value(num %= max)}
  Cycle -> Listen {
   trigger $value;
   action $next() }
 } // R
} // Simple
```



5.1.4 Constructive and Analytic Interfaces

Pins define the *constructive* interface of components—the external structure of components from which, with connectors, a system of components can be composed.[4] This constructive interface is not sufficient, however, for describing the observable behavior of components, or therefore for predicting the behavior of systems constructed from components. The *analytic interface* of components are specified as *annotations* on reactions. One reaction annotation is so generally important that it has its own keyword in PCL—“threaded.” Another annotation is equally important, and has its own complex syntax and semantics based in UML statecharts and ANSI-C, as previewed in example in Figure 5.1.

PCL also has a generalized *annotation mechanism* for tagging any *named* construct in PCL (environments, pins, variables, states, etc.) with whatever information is required by a reasoning framework, provided that information can be expressed as a value in the type system of the action language. Annotations are therefore one way that reasoning frameworks specify analytic constraints on components, environments, and systems.

[4]Strictly speaking, because components may have one or more reactions, pins describe the constructive interfaces of reactions. Such details are reserved to Chapter 6.

5.1.5 Composition and Assembly

The constructive interfaces of components are denoted by their pins, in the example asynchronous '$\ll$' pins, and interactions are expressed by composition operators $\rightsquigarrow$. Together they are used to express *constructive compositions* of components. In this research, the term "composition" is reserved for cases where a composite has analytically-predictable behavior. The term *constructive composition* is not an oxymoron, however, because PCL defines a functional semantics of Pincharts and connectors (see Appendix A). Because each reaction must specify a Pinchart, the functional behavior of a constructive composition is semantically well-defined. The semantic interpretation defines composition operators; we might denote such an operator as '$\overset{\ll}{\rightsquigarrow}$' for the case where asynchronous pins are connected.

Each reasoning framework F also defines a semantics, and therefore also defines one or more composition operators '$\overset{F}{\rightsquigarrow}$' that express *analytic compositions* of components. Here, analytic composition refers to case where a composite has analytically-predictable, but possibly extra-functional behavior (see Def. 3.10 on 57 for the definition of extra-functional behavior). However, '$\overset{F}{\rightsquigarrow}$' will *in practice* differ substantially from '$\overset{\ll}{\rightsquigarrow}$' in the numbers and types of operands they take, and in the types of their results. For example, while $\overset{\ll}{\rightsquigarrow}$ can, for most intents, be regarded as a dyadic operator, the analogous operator $\overset{\lambda}{\rightsquigarrow}$ in the $\lambda*$ reasoning framework is intrinsically polyadic because it composes *all* components that are scheduled on a single processor.

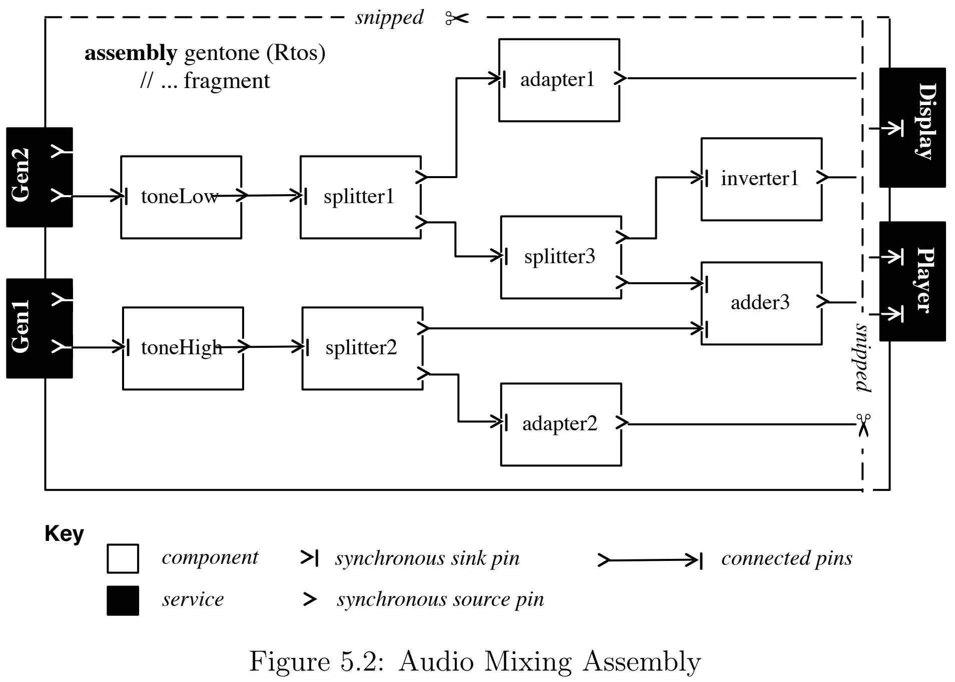


Figure 5.2: Audio Mixing Assembly

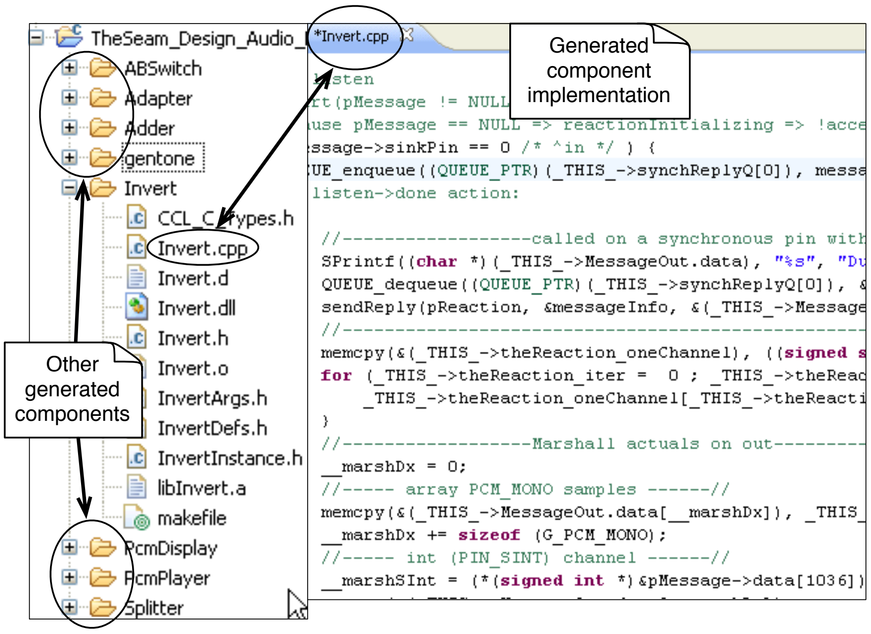
PCL semantics defines a semantics of constructive composition for $\overset{\ll}{\leadsto}$ using Hoare's process algebra, CSP [75] (and for other operators as well). However, constructive compositions do not result in new component types; in PCL one can not say something like C12 = C1:r ~>C2:s, for example. The different properties of constructive and algebraic composition, and the different hierarchies they induce, make such a notion less useful than one might expect. Instead, a PCL *assembly* aggregates arbitrarily many components and singly constructively-composed components; assemblies may themselves be instantiated in hierarchical fashion, but this amounts to something more akin to macro expansion than composition (although it is not in substance different than the elaboration semantics defined for Darwin [108]).

5.1.6 PCL in Action

Although the PSK was developed and validated on design problems posed by electric grid substation automation and industrial robot control, it turns out that desktop streaming audio provides a ready source of examples with which to demonstrate, and to teach, the concepts of predictability by construction (in particular for hard ad soft real time behavior). The PSK includes a collection of components and services that can be used to build interesting applications for mixing, playing and visualizing digital audio data, and several pre-built audio applications are provided in the PSK, and are used to demonstrate the features of several reasoning frameworks.

Figure 5.2 depicts a fragment of one pre-built application that receives two constant tone signals from the environment (the `Gen1` and `Gen2` services depicted on the left border of the assembly), and plays these back to an audio output service (`Player`, on the right border of the assembly) and transmits the audio signals to external viewers (`Display`). In between there are more than twenty instances of various component types (splitters, adders, inverters, filters, etc.) executing at different priorities, which must synchronize their behavior and sustain a 44Hz sampling rate. More interesting examples can be constructed using music media which for licensing reasons can not be distributed in the PSK, but end–users are free to use their own media files.

Figure 5.3 is a screen capture of a portion of the PSK Eclipse environment that shows a number of folders that contain generated C code that implements each component *type* used in the assembly, as well as a top-level assembly controller (`gentone`) that manages the lifecycle of the assembly. The generated implementation of component type (`Invert`) has been expanded to reveal a portion of its reaction in the Pin component technology. An important principle governing this research is that PECT abstractions must have simultaneous meaning for architect and programmer. Reasoning frameworks address the needs of the architect, and their predictive strengths are validated. The practical "programmability" of the PSK is how we validate that the solution attends to the needs of the programmer. Indeed, this grounding is also essential to find the right balance, and to make the right design tradeoffs, between the needs of the architect and programmer.



```
listen
rt(pMessage != NULL
use pMessage == NULL => reactionInitializing => !acce
ssage->sinkPin == 0 /* ^in */ ) {
UE_enqueue((QUEUE_PTR)(_THIS_->synchReplyQ[0]), messa
listen->done action:

 //------------------called on a synchronous pin with
 SPrintf((char *)(_THIS_->MessageOut.data), "%s", "Du
 QUEUE_dequeue((QUEUE_PTR)(_THIS_->synchReplyQ[0]), &
 sendReply(pReaction, &messageInfo, &(_THIS_->Message
 //--------------------------------------------------
 memcpy(&(_THIS_->theReaction_oneChannel), ((signed s
 for (_THIS_->theReaction_iter =  0 ; _THIS_->theReac
     _THIS_->theReaction_oneChannel[_THIS_->theReacti
 }
 //------------------Marshall actuals on out---------
 __marshDx = 0;
 //----- array PCM_MONO samples ------//
 memcpy(&(_THIS_->MessageOut.data[__marshDx]), _THIS_
 __marshDx += sizeof (G_PCM_MONO);
 //----- int (PIN_SINT) channel ------//
 __marshSInt = (*(signed int *)&pMessage->data[1036])
```

Figure 5.3: Audio Mixing (Generated Code)

5.2 Pin Component Technology

The Pin component technology was developed expressly for use in the PSK, and to support the kinds of performance-critical, real-time and embedded software systems that are the subject of the case studies in Chapter 9. The major design goals for Pin included *simplicity* in its programming model (for the programmer) and *extensibility* and *adaptability* to new reasoning framework-imposed analytic constraints (for the architect) [69]. Pin is therefore a quintessential component technology for the Seam.

Pin is a standalone component technology, and as such it has its own Pin component model. This component model is, however, significantly more flexible than the component model that is formalized by PCL. For example, standalone Pin allows, among other things, distributed component assemblies, runtime adaptation of assembly topology, and extensibility to new component interaction policies (as with PCL, connectors are implicit). This does not point to a design flaw in PCL; it reflects the governing philosophy of *co-refinement* (Chapter 10): start with predictable even if highly restricted design rules, and progressively relax these rules while maintaining required prediction quality.

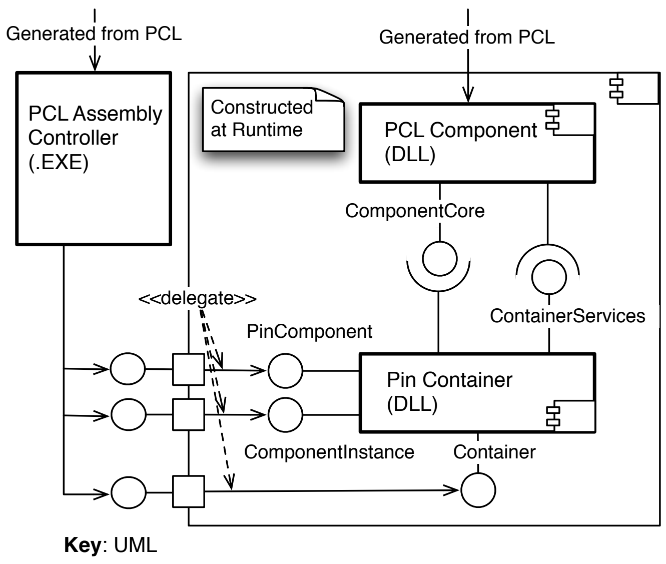


Figure 5.4: Pin Components and Containers

The architecture of Pin will be examined in close detail in Chapter 7. The feature of Pin that is of particular interest in the context of the earlier description of PCL is Pin *containers*.[5] Pin's basic container architecture is depicted in Figure 5.4. The key points to note are those that pertain to *isolation*, *runtime assembly*, and *analytic sandboxing*:

Isolation: PCL components are isolated from their execution environments, and from other PCL components, by Pin containers. All interactions among PCL components are mediated by containers. Containers control the main event loop of PCL components, and transparently (to the PCL component) manage interactions across threaded and unthreaded reactions, and handle various failure conditions such as message timeouts. While this constrains what PCL component designers can do, the result is significantly simpler and more easily analyzed PCL component behavior.

Runtime assembly: Pin components are composed at runtime from PCL components and Pin containers.[6] A top-level assembly controller is generated by PCL to perform the runtime composition of Pin components, and to manage the Pin component and assembly lifecycles. Containers can manage one or more PCL components, and PCL com-

[5]Containers are treated as *annotations* in PCL.

[6]The notions of "isolation" and "sandboxing" also make it possible to regard PCL components as being *deployed* into containers, but it is probably best to restrict the meaning of deployment to market distribution mechanisms.

ponents may be relocated at runtime to other containers. Containers can be *nested* (not evident in Figure 5.4), which is a particularly useful feature for analytic sandboxing.

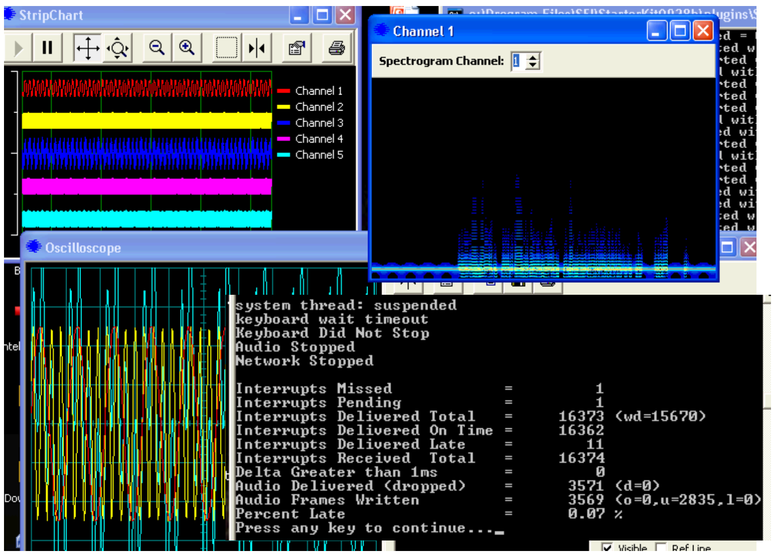
Analytic sandboxing: The term "sandboxing" is usually associated with container-like mechanisms that enforce security policies on programs, for example Java applets execute in a "sandbox" provided by a browser. Pin allows the development of containers that implement *analytic sandboxes* for qualities not limited to just security, and provides guarantees to both the environment and the sandboxed component. For example, λ_{SS} uses a specialized container to make bi-lateral guarantees of bounded temporal intrusiveness of PCL components the environment *and* service time of PCL components managed by these containers.

5.2.1 Pin runtime environment in Action

The Pin runtime environment is designed to support a variety of hard and soft real-time applications, and to support the development of new performance reasoning frameworks, possibly requiring a mix of scheduling disciplines. Various commercial real-time operating systems are available that would provide a solid basis for further development along these liens. At the same time, the PSK was not intended only for real-time applications; that scope would have been too narrow to demonstrate the applicability of PECT, and would have also restricted the availability of the PSK to a small and highly specialized installed base of host platforms. The Pin runtime environment strikes a balance between the need to support real-time predictability, generality, and applicability by implementing a virtual real-time operating system as a layered extension of Microsoft's WindowsNT and WindowsXP platforms.

Figure 5.5 is a screen capture of the `gentone` assembly (Figure 5.2) at runtime. Three graphics-intensive displays have been attached to the `Display` service in Figure 5.2: strip chart, oscilloscope, and spectrograph. Each of these programs are external to the Pin runtime environment—they are executing in the native WindowsNT or WindowsXP host. The Pin runtime environment runs as a real–time Windows process so that it can provide fixed-priority scheduling of component threads, but these external displays are operating at normal (default) Windows application priority.

The practical significance of the demonstration is that the PSK can provide real-time guarantees to component assemblies, and at the same time have bounded impact on the external host environment. Pin assemblies can therefore be safely "embedded" in a non–real–time host, and can access external devices through PCL environments.



```
system thread: suspended
keyboard wait timeout
Keyboard Did Not Stop
Audio Stopped
Network Stopped

Interrupts Missed                 =          1
Interrupts Pending                =          1
Interrupts Delivered Total        =      16373 (wd=15670)
Interrupts Delivered On Time      =      16362
Interrupts Delivered Late         =         11
Interrupts Received  Total        =      16374
Delta Greater than 1ms            =          0
Audio Delivered (dropped)         =       3571 (d=0)
Audio Frames Written              =       3569 (o=0,u=2835,l=0)
Percent Late                      =       0.07 %
Press any key to continue..._
```

Figure 5.5: Audio Mixing (Pin Runtime)

5.3 Performance Reasoning Framework ($\lambda *$)

The performance reasoning framework in the PSK can be used to predict the real-time behavior of assemblies. It applies to systems that execute on a uniprocessor, use a fixed-priority preemptive scheduling discipline, have periodic and aperiodic tasks with hard (worst case) and soft (average case) deadlines.

The performance reasoning framework is actually three distinct but related reasoning frameworks λ_{ABA}, λ_{WBA}, and λ_{SS} that are packaged together in a single PECT plugin:[7]

$\lambda_{ABA}, \lambda_{WBA}$: These reasoning frameworks predict the average– and worst–case performance, respectively, of assemblies that have periodic inter-arrival rates for work. Both reasoning frameworks can be used on assemblies that have components with threaded and unthreaded reactions, where reaction threads execute at different priorities, and the assembly has a mix of asynchronous and synchronous interactions among components.

λ_{SS}: This reasoning framework predicts the average–case latency of assemblies that have aperiodic (stochastic) inter-arrival rates for work. However, even though λ_{SS} predicts average-case latency, these assemblies

[7]λ stands for **latency**, A/W stands for "**average**–case" and "**worst**–case" respectively, B stands for "**blocking** effects are considered," and A stands for "**asynchronous** interactions are permitted."

can still be used in systems that have hard periodic deadlines because the invasiveness of these assemblies is strictly bounded by the λ_{SS} container.

The following sections provide a summary of the theory, interpretation and validation of these reasoning frameworks.

5.3.1 Theory

Each of the $\lambda*$ frameworks is based in generalized rate–monotonic analysis (GRMA) [89]. $\lambda_{ABA,WBA}$ make use of GRMA to schedule tasks with varying execution priorities [62]; this prior work in the applied GRMA provides a natural interpretation of a "synchronizing concurrent pipeline" pattern that is easily constructed in PCL (and in Pin). In $\lambda*$ jargon this pattern is called an "HKL pattern" from the initials of the authors of [62].

The λ_{SS} container enforces the application–level sporadic–server algorithm [63]; this prior work in applied GRMA also has a natural interpretation in containerized components. λ_{SS} also broke new ground in applied GRMA by defining a queuing-theoretic solution to predict the average-case execution time of PCL components managed by the λ_{SS} container [67]. This solution provides closed-form upper and lower bounds on managed execution time, and a simulation model if more precise predictions are required.

5.3.2 Interpretation

The $\lambda*$ frameworks impose various design rules (constraints), and these are summarized in Chapter 8. Some of these design rules are enforced implicitly by the Pin runtime environment (e.g., thread scheduler, λ_{SS} container), while others are enforced statically by the $\lambda*$ interpretations (e.g., priority ceiling, acyclic topology). The analytic interfaces on components and environment are another source of statically-enforced design rules, and these are specified as PCL annotations (e.g., non-blocking execution time of component reactions, statistical distribution of event inter-arrival rates from the environment).

If a PCL assembly is "well-formed" to the specific $\lambda*$ reasoning framework, the interpretation will generate a *model* that is well-formed in a syntax defined by a *performance meta-model* (PMM), and which has its own semantics in the equations of GRMA, and in simulations that can be executed on these models using one of the simulators packaged with $\lambda*$. Some technical ingenuity was required to interpret acyclic but quite complex PCL component topologies as a collection of PMM tasks that are each composed from a linear sequence of PMM subtasks, but the interpretation is otherwise straightforward because of the close correspondence of the HKL pattern to the Pin component model.

5.3.3 Validation

The predictive quality of the $\lambda*$ frameworks have been validated by extensive model simulation as well as by targeted "spot checks" of $\lambda*$ predictions in non-trivial industrial cases (see Chapter 9). The sufficiency of these spot checks is justified by an earlier and more extensive validation of the λ_{ABA} framework using techniques developed by Larsson [98] and Moreno et al [124]. These techniques established a well-founded *confidence interval* for λ_{ABA} predictions.

Because the other members of $\lambda*$ are based in substantially the same underlying theory, it is reasonable, if somewhat optimistic, to conjecture that the confidence interval applies to these other members as well. Optimistic or otherwise, spot validations (sometimes involving more substantial testing than credited by the term "spot") have shown the original λ_{ABA} confidence interval to be valid and even conservative for all of $\lambda*$. The conjecture is further sustained by the fact that the confidence interval was established on a version of Pin that was hosted on a commercial real-time extension of Windows-NT, but has remained valid despite numerous and significant changes to Pin, including its re-hosting to a real-time kernel developed for this research and used in subsequent industrial cases.

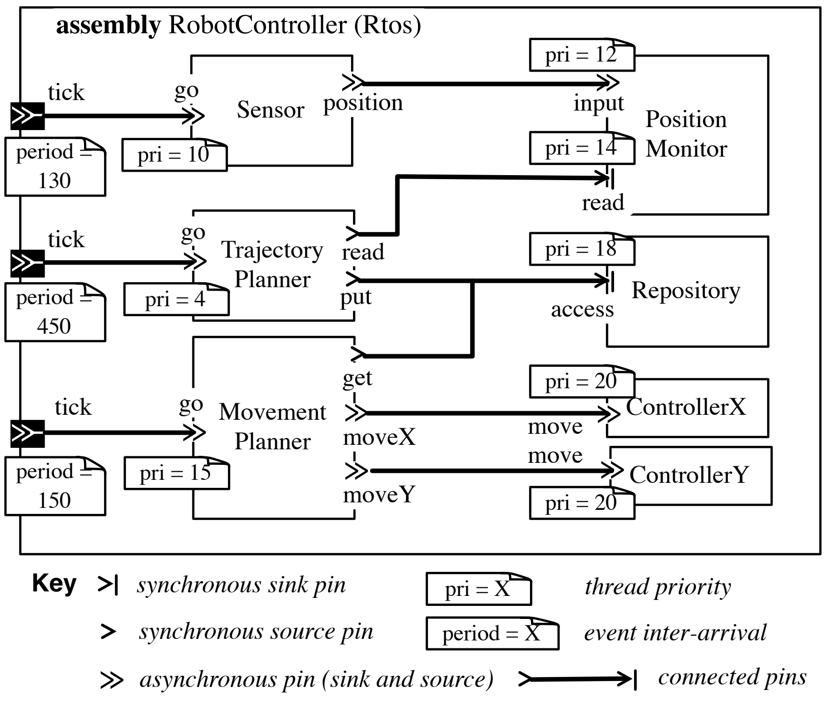


Figure 5.6: $\lambda*$ Robot Controller

It is worth remarking at this point that in $\lambda *$, predictability applies to the behavior of PCL *assemblies*, not to the behavior of PCL *components*; this is not the case for the model checking reasoning framework (described in §5.4). The portion of a component's $\lambda *$ analytic interface that describes, for example, its unblocking execution time is assumed by the reasoning frameworks to be ground truth. It is also a general rule of reasoning framework design that all parts of a component's analytic interface be obtainable and confirmable by third parties using a mechanism defined by, if not provided by, the reasoning framework. The meaning of "unblocking component execution time" is defined by $\lambda *$, and the PSK provides a measurement infrastructure with which to obtain its value for a component.

5.3.4 $\lambda *$ in Action

Figure 5.6 depicts a robot controller that, although highly simplified, accurately models the basic coordination scheme used by a robot controller in one of the case studies described in Chapter 9. It also introduces several complications not present in the audio example discussed earlier: the presence of non-harmonic task periods, and a mixture of synchronous and asynchronous task interaction.

The `TrajectoryPlanner` component receives work orders every 450ms, which is simulated in the assembly by pulse from a clock service. which it reads from `Repository`. On receipt of a work order, `TrajectoryPlanner` consults the current robot state by issuing a synchronous read on `PositionManager`. The `PositionManager` is updated by sensors every 130ms. Based on the current robot position, `TrajectoryPlanner` generates a series of subwork orders, which it then synchronously deposits in `Repository`, at which point `TrajectoryPlanner` waits for another work order. The `MovementPlanner` component operates on a 150ms period, and at each period it read an subwork order from the repository and generates movement commands for two control axes. If `MovementPlanner` encounters an empty repository (i.e., no subwork orders available), the robot must abort. Thus both `TrajectoryPlanner` and `MovementPlanner` must meet their respective deadlines.

Figure 5.7 shows $\lambda *$'s interpretation of the `RobotController` assembly as an "HKL" assembly. The $\lambda *$ interpretation has translated an acyclic graph of component compositions, containing a mix of synchronous and asynchronous interactions, into three distinct task/subtask chains that will be "executed" by the HKL equations (or simulators, depending on which decision procedures are selected) to determine worst–case and average–case assembly latency. The effects of blocking on the Repository component have been accounted for by replicating its shared reaction on two different tasks; analogously, PositionMonitor also shows up on two different tasks, though for different reasons. What is important to note is that the component topol-

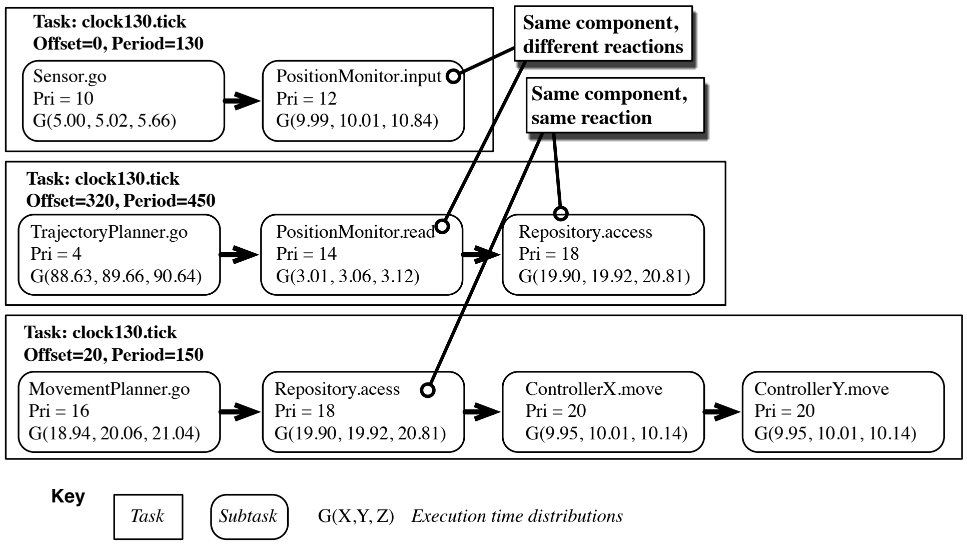


Figure 5.7: $\lambda*$ Interpretation of Robot Assembly

ogy and scheduling topology are semantically related but quite distinct, as was discussed in §5.1.5 on the differences between constructive and analytic composition.

5.4 Model-Checking Framework (ComFoRT)

The software model checking reasoning framework in the PSK, ComFoRT,[8] can be used to verify that a component or assembly satisfies behavior requirements expressed in a temporal logic that describes sequences of future execution states. Model checkers work by exhaustively checking all possible execution paths to verify specified behavior.

ComFoRT introduced a new form of linear temporal logic (LTL) called *state/event*-LTL [30]. SE–LTL specifications can be composed from propositions on states (PCL variables) and events (PCL begin/end events on pins). This leads to simpler and more intuitive specifications of component-based behavior when compared with the tradition LTL approaches that allow propositions on either states or events, but not both. For example, the following annotation:

```
annotate simple  {
   "comfort",
   const string LivenessAndSafety =
   "G((^next => F $next) & ([R:num < Simple:max])"}
```

[8]**Com**–ponent **F**–ormal **R**–easoning **T**–echnology.

when added to the PCL fragment shown in Example 5.1, pp.80, specifies that it is `G` globally (i.e., always) the case that if an incoming request arrives on the `^next` pin, the request is `F` finally (i.e., eventually) completed `$next`, and the value of reaction R's local variable `R:num` is always strictly less than the instantiation parameter `max`. ComFoRT successfully verifies this claim. It should be noted, however, that SE-LTL is strictly no more expressive than an LTL based exclusively in states or events.

5.4.1 Theory

Clarke and Emerson are generally regarded as having introduced the idea and name "model checking" [42]. Model checking adopts a model-theoretic (i.e., semantic) rather than proof-theoretic (i.e., syntactic) approach to verifying that software satisfies its specification. To *model check* a system, the following steps are performed:

1. The system is modeled as M, using the description language of a model checker.

2. The claim Φ to check is defined using the specification language of the model checker, typically a temporal logic formalism.

3. The model checker examines each state in M to demonstrate that M "satisfies" Φ, expressed formally: $M \models \Phi$.

4. The model checker reports "Yes" if $M \models \Phi$ and "No" otherwise.

When a claim is not satisfied, most model checkers also produce a *counterexample* that documents system behavior that causes the failure. Counterexamples are one of the most useful features of model checking, as they allow users to quickly understand why a claim is not satisfied.

All model checkers (software or otherwise) suffer from "state-space explosion." A precursor to ComFoRT was used to verify a small portion of an interprocess communication library (IPC) in an industrial robot control system [80]. The IPC software was manually abstracted into executable UML [118], and the reasoning framework produced an interpretation [173] from UML to S/R, the input language of the COSPAN model checker [64]. However, its estimated 2.35×10^{1932} number of states arising from the S/R interpretation of the original UML model was well beyond all practical limits. The considerable manual effort required to produce a "checkable" UML model is what led us to investigate automated abstraction refinement and ultimately to ComFoRT. See the case study description in Chapter 9, §9.3.3, pp. 197.

ComFoRT attacks state space explosion with a combination of predicate abstraction [57] and counter-example guided abstraction refinement (CEGAR) [43]. Predicate abstraction produces a conservative over-approximation

$\Psi(p)$ of the behavior of program p, i.e., every specified behavior p is exhibited by $\Psi(p)$, but not every behavior in $\Psi(p)$ can be exhibited by p. Thus, if $\Psi(p) \models f$, then $p \models f$ and the model checker terminates. However, a counterexample that falsifies f in $\Psi(p)$ may not be possible in p—it may be *spurious*. CEGAR uses a theorem prover to check if a counterexample is spurious. If it is not, the model checker reports a genuine error in p and terminates. Otherwise, CEGAR uses the spurious counter-example to refine abstraction $\Psi(p)$ to $\Psi(p)'$.

There is no decidable way to construct a provably sound (if conservative) finite abstraction of an arbitrary program. Thus, ComFoRT implements a semi-decision procedure: if produces a correct result if it terminates, but there is no guarantee that it will terminate. Nonetheless, ComFoRT does reasonably well in practice—which is not to say that it has solved the problem of state–space explosion.

5.4.2 Interpretation

PCL defines a process-algebraic semantics [82] of constructive composition (using CSP [75]), along the same lines of Wright [3] (also using CSP), Darwin [109] (using FSP [111]), and Piccola [106] (using π–calculus[122]). PCL differs in one important way: by differentiating *real concurrency* as threaded reactions and *potential concurrency* as unthreaded reactions, it permits a *faithful interpretation* of concurrency in the real system. Threaded reactions represent real concurrency, and as such are naturally modeled as processes (in the CSP, FSP and $\pi-$Calculus sense). Unthreaded reactions represent potential concurrency—depending on how they are composed with other reactions, they may execute on one or arbitrarily many threads.

Assigning a semantic interpretation of unthreaded reactions as processes would be a sound but conservative over-approximation, and would almost certainly result in spurious counterexamples that could not be detected by CEGAR. Further, it would exacerbate state space explosion, as the size of the state space can be exponential in the number of parallel-composed processes. ComFoRT defines a faithful interpretation of concurrency by using information available in the global assembly topology to allocate potential concurrency to real concurrency. As a consequence, its interpretations are smaller (in potential state space) and more accurate than more conservative approaches. However, the reliance on global analysis leads to a non-compositional interaction semantics, and the resulting formalization is not as clean or elegant as the semantics of Wright, Darwin, Piccola or others that are more conservative and, we argue, simplistic.

5.4.3 Validation

If a component or assembly fails to satisfy its SE-LTL specification f, ComFoRT (and this is true of all model checkers) will produce a counterexample that demonstrates the violation of f as an execution trace (not necessarily the shortest!) through M. Each counterexample is thus a "witness" to the cause of the failure. As implied in the earlier description of CEGAR, the witness can be interrogated, i.e., checked for its veracity, and as already mentioned, ComFoRT only reports counterexamples that it has established (by proof) to be real. This makes ComFoRT useful for developers because it provides a tool for "debugging" systems at even early stages of development, for example when high-level interaction protocols among components are defined.

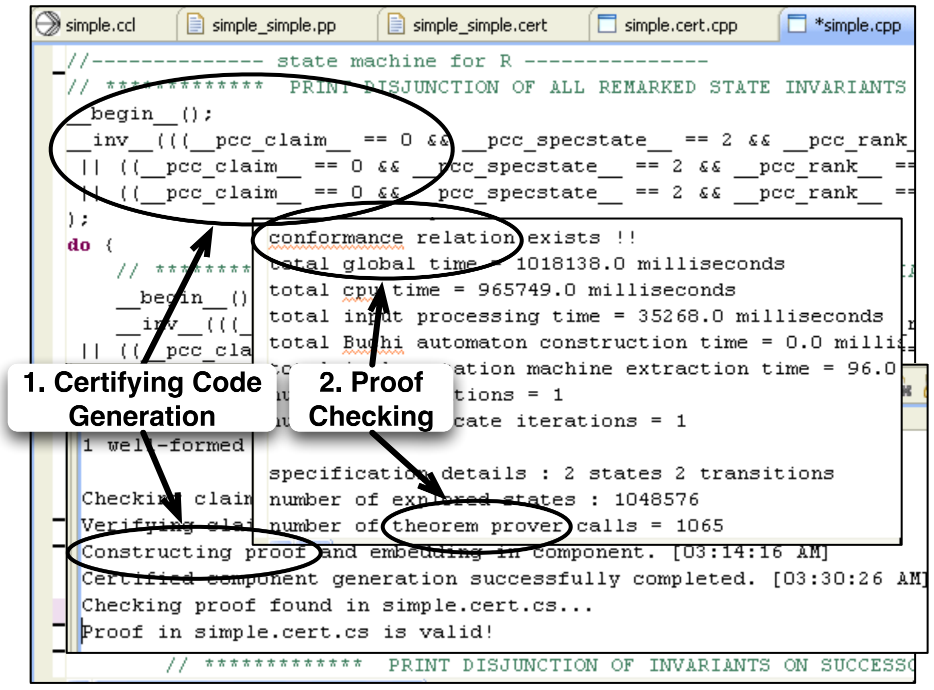
While all model checkers provide counterexample witnesses, almost none offer proof witnesses to support $M \models f$. Instead, they answer "yes" if no counterexample has been discovered. This is not a principled "yes" in the axiomatic sense that $\neg M \not\models f \Leftrightarrow M \models f$. For example, "yes" might be the result of a "bug" in the model checker itself, hardly an unthinkable possibility given the complexity of these tools. The explanation for this asymmetry may well be explained by the greater importance that designers and programmers place on finding errors than generating evidence of the absence of error—although this emphasis may be changing. ComFoRT can be asked to generate a *proof certificate* to serve as a witness that $M \models f$ [28].

The idea of a certifying model checker did not originate in with ComFoRT, see for example Namjoshi [129]. ComFoRT breaks new ground by concretizing proof certificates into executable code, thereby providing a fully automated end–to–end capability to generate *proof-carrying code* (PCC) [130]. One argument made in support of PCC is that it reduces the "trusted computing base" (TCB) that is required to trust code. Because the PSK embeds a proof witness in executable code, it is possible to mount a principled argument that none of the components on the tool chain leading to that code (ComFoRT, PCL program generator, even the GNU C compiler) need be trusted. This is a strong claim, and not one to be dismissed—or accepted—lightly. As part of this research we also demonstrated the use of traditional safety–case analysis to argue not about the end–system behavior guaranteed by the certified code, but about the quality of the evidence itself [31] (see Chapter 8).

5.4.4 ComFoRT In Action

ComFoRT witnesses are constructed from their CEGAR abstractions. To make these witnesses confirmable or falsifiable requires they be concretized in some way. The PSK concretize counterexample witnesses via a reverse-interpretation from ComFoRT to PCL, which of course requires a bit of bookkeeping since semantic interpretation is (in general) not bijective. A interactive explorer is provided to assist the PSK user to examine what might be quite long and involved counterexamples; this is a practical necessity given the ability of model checkers to uncover deep concurrency defects.

PCC in PSK is implemented as a succession of concretizations: from ComFoRT to PCL source via reverse interpretation; from PCL source to Pin "C" source code via program generation, and from Pin source code to assembly language via the GNU C compiler [33, 32].



```
//-------------- state machine for R ---------------
// **************  PRINT DISJUNCTION OF ALL REMARKED STATE INVARIANTS
__begin__();
__inv__(((__pcc_claim__ == 0 && __pcc_specstate__ == 2 && __pcc_rank_
 || ((__pcc_claim__ == 0 && __pcc_specstate__ == 2 && __pcc_rank__ ==
 || ((__pcc_claim__ == 0 && __pcc_specstate__ == 2 && __pcc_rank__ ==
);
do {
    // *********
      __begin__()
      __inv__(((_
 || ((__pcc_cla
```

```
conformance relation exists !!
total global time = 1018138.0 milliseconds
total cpu time = 965749.0 milliseconds
total input processing time = 35268.0 milliseconds
total Buchi automaton construction time = 0.0 millis
        ation machine extraction time = 96.0
        tions = 1
        cate iterations = 1

specification details : 2 states 2 transitions
number of explored states : 1048576
number of theorem prover calls = 1065
```

```
1 well-formed

Checking claim
Verifying clai
Constructing proof and embedding in component. [03:14:16 AM]
Certified component generation successfully completed. [03:30:26 AM]
Checking proof found in simple.cert.cs...
Proof in simple.cert.cs is valid!
```

```
        // **************  PRINT DISJUNCTION OF INVARIANTS ON SUCCESS
```

Figure 5.8: Certified Code

Figure 5.8 provides a screen capture of two steps in the process of generating a certified PSK component:

1. Generating a certified proof-carrying component by embedding invariants in generated Pin–C component source.

2. Checking that the proof is valid on the PowerPC assembly code generated by the GNU compiler from the Pin–C component.

In this example we certify that variable `i` of the `Simple` component shown in Example 5.1 is always bounded by `min` and `max`.

5.5 Summary of Key Points

The PSK is a proof–by–existence that PECT can be implemented, and that predictability by construction can be obtained for an interesting class of systems and system behavior. This summary has emphasized those features of the PSK that demonstrate practical and useable automation. These aspects of the PSK are not merely "nice to have," but are essential to demonstrating the feasibility of the overall approach. Without such a grounding it would be impossible to locate the Seam in practice, and to make the kinds of tradeoff decisions that are required that balance the restrictiveness of design rules, strength and quality of analytic predictions, and scaleability of solutions required by engineering practice.

The remaining chapters of Part II examine in closer detail each of the main components of the PSK technology. Part III then turns to practical experiences in using these technologies on non-trivial industrial problems.

Chapter 6

Pin Component Language

The Pin Component Language (PCL) combines a structure notation based in the Pin component technology with a behavior notation based in a subset of UML Statecharts called PinCharts. The syntax and semantics of PCL formalizes the Pin component model, and extends it in ways that make it suitable as a Seam technology.

PCL notation mimics C syntax, and the action language of PinCharts is a reasonably complete but pointer–free subset of C; an escape mechanism is also provided to access full ANSI–C. The Pin–based structures allow *components* to be specified in terms of their *sink pins*, which components use to receive requests from client components, and *source pins*, which components use to make requests of client components. Although the underlying communication model and semantics are event–based, PCL allows pins to be declared as using either synchronous or asynchronous communication protocols; components can use synchronous pins to send ("produce") or receive ("consume") data, while components may only produce data on asynchronous pins.

Component behavior is specified in their *reactions*, which may optionally be declared as executing on their own independent thread of control. Reactions, in turn, are specified as *PinCharts*, which is used to define state machines that link behavior specified in the C–based action language with the sending and receiving of pin (and other types of) *events*. Component interaction is enabled when the source pins of one component are connected to the sink pins of other components. Components are aggregated into *assemblies*, and the pins of aggregated components are selectively exposed to allow hierarchical assembly. An annotation mechanism is provided to allow any storable expression value to be associated with any named PCL construct; this allows code–generators, reasoning–frameworks and other "back-end" clients of the PCL processor to obtain, or to require, additional information of components and their assemblies.

Organization of this Chapter

Section 6.1 introduces the notational conventions used to describe PCL. Section 6.2 introduces the basic structures of PCL, and is primarily concerned with the C–based action language; readers familiar with C can safely skip the details and move directly to Section 6.3, which describes the *structural* aspects of PCL, in particular *components*, *assemblies* and *environments*. Section 6.4 then turns to the *behavioral* aspects of PCL, and in particular the use of *PinCharts* to specify *reactions*. Section 6.5 gives a brief overview of the denotational definition of interaction and the operational definition of reaction, details of which are provided in Appendix A. Section 6.6 discusses language pragmatics, with a particular emphasis on the balance PCL strikes between expressiveness and restrictiveness in its choice of interaction primitives. Finally, section 6.7 summarizes the key points of the chapter and points the way to the Pin Component Technology described in Chapter 7.

6.1 Notational Conventions

The syntax of PCL is defined using a variant of Backus–Naur Form (EBNF):

Extended Backus–Naur Form (EBNF)

BNF	Meaning
$N = M$	Production rule: N is a non-terminal.
$Q\ R$	Concatenation: Q is followed by R.
$Q \mid R$	Alternation: Q or R.
$[Q]$	Optional Q: zero or one Q.
$Q^{+\mathrm{c}}$	One or more Q. ‘c’ is a separator if present.
$Q^{*\mathrm{c}}$	Zero or more Q. Equivalent to [$\mathrm{Q}^{+\mathrm{c}}$].
(P)	Grouping.
keyword	Keywords are shown in **boldface**.
Id_{R}	Identifiers are *italicized*. R is a context clue if present.
T_{LIT}	Literals are formed from their type name T.
‘c’	Character literals are single quoted.

Example fragments of PCL are provided to illustrate the major abstractions, along with their associated graphical notation.

6.2 Basic Elements

The basic lexical structure of PCL is strongly shaped by the C programming language, and C is the basis for the PCL action language. Familiarity with the basic syntax of C will be both useful and assumed; however, a detailed understanding of C is not required to understand the main ideas.

Tables are used to identify elements that are defined exclusively in PCL, those that have overloaded meaning in PCL and C, and those that retain their original meaning in C. Text appearing in these tables *in small italics* provides hints or pointers to where in the chapter the text entry is discussed in more detail.

6.2.1 Lexical Structure

Table 6.1 defines the structure PCL literals, and Table 6.2 lists the PCL keywords. Note that PCL is more restrictive than C in forming numeric and string literals and identifiers. PCL supports two familiar form of comments: `‘//’` which denotes a comment until the end of the line of text, and `‘/*’` which denotes a (possibly multi-line) comment until a matching `‘*/’` is encountered. As usual, multi-line comments do not *nest*. The PCL processor supports the #**include** pre–processor directive; this is the only pre–processing directive supported.

Table 6.1: PCL Literals

Char	=	ASCII US keyboard
Letter	=	(‘a’–‘z’) | (‘A’–‘Z’)
Digit	=	(‘0’–‘9’)
$Integer_{\text{LIT}}$	=	$Digit^{+}$
$Float_{\text{LIT}}$	=	$Digit^{*}$ ‘.’ $Digit^{+}$
$String_{\text{LIT}}$	=	‘"’ $Char^{*}$ ‘"’
Id	=	*Letter* (*Letter* | *Digit* | ‘_’)*

Table 6.2: PCL Keywords

Added by PCL	Inherited from C
action after alert annotate as assembly assume asynch boolean byte component consume environment expose from guard proc produce react service singleton sink source start state string synch threaded trigger when	**break const continue do double else enum extern false float for if int return short true typedef unsigned void while**

6.2.2 Types

PCL diverges sharply from C by excluding explicit use of *address* types, i.e., C pointers. C without pointers is a bit like Java without classes, but nonetheless the restriction was an important compromise to ensure strict separation of components at runtime. A base language other than C might have been a more principled choice, but (as of this writing, in mid–2010) C and C++ remain *de facto* standards for building embedded, real–time software. Table 6.3 summarizes the types that are supported by PCL.

Table 6.3: PCL Types

	Added by PCL	Supported C
Basic Type	**byte** (signed, unsigned) **boolean** **string**	**short** (signed, unsigned) **int** (signed, unsigned) **float**, **double**
Constructed Type	***event*** (see §6.4.1)	**typedef** ***array*** (by typedef) ***enum*** (by typedef)

PCL supports multi-dimensional arrays. As with C, PCL array types have no explicit type name. However, because C array types and indexing are defined in terms of address arithmetic, there are a few noticeable differences in the way PCL treats arrays and C's treatment. In particular, PCL array types do not support operators such as ==, !=, which would be meaningless without address types and expensive if implemented as byte-wise comparison (which the programmer can always choose to do). As with C, type conformance between two PCL array variables is defined on the number and size of array indexes, and on conformance of the base types (using C's rather complicated rules for implicit conversion).

PCL also makes a compromise with C address types by introducing an explicit type **string**. As with C, a string is essentially a one-dimensional array of characters. However, PCL strings can be constructed only from literals or by parameter passing; they can not be subscripted and so do not reveal their base type values. The **byte** type is used as a syntactic replacement for C's *char* type to avoid any confusion between the PCL notion of string and the C notion of array of *char*. Comparison operators ==, >= etc., are defined for strings, and are defined in terms of lexicographical order.

PCL does not support C *struct* or *union* types. This is a limitation in the current implementation.

6.2.3 Expressions

PCL supports a reasonably complete set of C operators, and full C expression syntax, with the exception of operations on arrays and strings, as noted above. Table 6.4 lists the operators supported by PCL, which conform to C precedence and associativity. Table 6.5 provides an abstract syntax for expressions. The abstract expression operator '$\circ$' corresponds to one of the prefix, postfix, or infix operators, where infix is to be inferred if an operator is neither prefix or postfix.

The two prefix operators introduced by PCL ('^' and '$') (see *PinEvent* and *PinData* productions in Table 6.5) operate on pins rather than program values, and evaluate to values of UML event type; these operators are described in §6.4.2. The *ScopedId* syntax reflects the hierarchical namespace introduced by the PCL structural abstractions discussed in §6.3.

6.2.4 Declarations

PCL inherits C's rather baroque declaration syntax, as summarized in Table 6.6. There are a few divergences from C worth noting. First, PCL requires that enumerator types and array types be introduced by an explicit **typedef**; enumeration and array variables must be declared using a "typedef–ed" type identifier. Example 6.1 illustrates various forms of declaration and partial (static) evaluation.

Table 6.4: PCL Expression Operators

Op	Description	Syntax[a]
()	function call, pin event	*Call*, *PinEvent*
.	Pin data selection	*PinData*
[]	array subscript	$E \circ$
$--\ \ ++$	decrement, increment	$E \circ$
$--\ \ ++$	decrement, increment	$\circ E$
$-\ \ +$	sign	
!	logical negation	
(Id_{typecast})	typecast	
$*\ \ /\ \ \%$	multiply, divide, modulus	$E \circ E$
$+\ \ -$	add, subtract	$E \circ E$
$<\ \ >$	less than, greater than	
$<=\ \ >=$	less than equal, greater than equal	$E \circ E$
$!=\ \ ==$	relational (in)equality	$E \circ E$
$\|\|$ &&	logical or, and	$E \circ E$
$=$	assignment	$E \circ E$
$+=\ \ -=$ /= %= <<= >>= ^= &= |=	compound assignment	
,	expression sequence	$E \circ E$

[a]See PCL Expressions, Table 6.5

Table 6.5: PCL Expressions

Exp	=	$\circ$ *Exp* | *Exp* $\circ$ *Exp* | *Exp* $\circ$ | *Call* | *PinEvent* | *PinData* | *Lit*
Call	=	$ScopedId_{\text{Proc}}$ ‘(’ $Exp^{*,}$ ‘)’
PinEvent	=	[‘**$**’ | ‘^’] Id_{Pin} ‘(’ $Exp^{*,}$ ‘)’
PinData	=	$ScopedId_{\text{Pin}}$ ‘**.**’ $Id_{\text{FormalParam}}$
Lit	=	*ScopedId* | $string_{\text{LIT}}$ | int_{LIT} | $float_{\text{LIT}}$ | **true** | **false**
ScopedId	=	[‘**:**’] $Id^{+:}$

Example 6.1: Example declarations.

```
typedef enum {white=0, blue=2, red=3} TColor;
typedef TColor I3[2][2] ;
I3 myI3 = { {white,red}, {blue,blue} };
I3 theirI3 , yourI3 = myI3;
const int peek = (int) yourI3[0][1];
```

PCL allows functions to be declared and defined in C-style, when pro-

Table 6.6: PCL Declarations

Declaration	=	*ProcDecl* | *VariableDecl* | *TypeDecl*
VariableDecl	=	[**const**] *TypeSpec* $InitDecl^{+,}$ ‘**;**’
TypeSpec	=	*BasicType* (see Table 6.3) | $Id_{(\text{typedef})}$
InitDecl	=	*Id* [‘=’ *Exp*] | *PinInstance*
PinInstance	=	*Id* ‘**(**’ $Exp^{*,}$ ‘**)**’ [*Provision*] (see Table 6.13, pp.112)
TypeDecl	=	**typedef** *TypeDeclSpec* $InitDecl^{+,}$ ‘**;**’
TypeDeclSpec	=	*TypeSpec* | *ArraySpec* | *EnumSpec*
ArraySpec	=	*Id* $IndexExp^{+}$
IndexExp	=	‘**[**’ *Exp* ‘**]**’
EnumSpec	=	**enum** ‘**{**’ $Enumerator^{+,}$ ‘**}**’
Enumerator	=	*Id* [‘=’ *Exp*]
Exp	=	(*See Table 6.4.*)
ProcDecl	=	(*See discussion later in this section.*)

ceeded by the keyword **proc**. A few minor adjustments to C’s basic syntax are made to accommodate the lack of address type, in particular borrowing from C++ notion of *reference type* on formal parameters of procs (the only place in PCL that permits the use of explicit reference types). The following fragment illustrates the main idea:

Example 6.2: Example **proc** declaration and definition.

```
extern boolean proc strcmp(string s1, string s2);
int proc times2add1(int &param);
void proc addone (int v1, int v2, int &v3)
{
  if (v3 > 0) v3 = v1 + v2;
}
```

As with C, PCL has a “pass by value” semantics. In the above example, strcmp(**string** s1, **string** s2) incurs the overhead of an explicit copy of the string arrays, which is worse than it sounds because strings are represented internally by PCL as fixed–length character arrays (due to the fixed–length message format used by Pin). Better would have been to declare both formal parameters as reference types, as was done for the &v3 parameter of **proc** addone.[1]

[1]To paraphrase former SEI colleague Mark Graham: you might not *see* C’s pointer semantics in PCL, but you can *smell* it.

6.2.5 Statements

PCL supports the major C statement constructs (with the exception C's *switch*) as summarized in Table 6.7. The symbol '$\triangleq$' denotes the union of simple assignment and compound assignment operators.

Table 6.7: PCL Statements

Stmt	=	*AsgnStmt* | *IfStmt* | *IterStmt* | *BreakStmt* | *CpmdStmt*
AsgnStmt	=	*Exp* '$\triangleq$' *Exp*
IfStmt	=	**if** '(' *Exp* ')' *Stmt* [**else** *Stmt*]
IterStmt	=	*ForStmt*| *WhileStmt*| *RepeatStmt*
ForStmt	=	**for** '(' *Exp* ';' *Exp* ';' *Exp* ')' *Stmt*
WhileStmt	=	**while** '(' *Exp* ')' *Stmt*
RepeatStmt	=	**repeat** *Stmt* '(' *Exp* ')'
BreakStmt	=	**break**| **continue**| **return** [*Exp*]
CpmdStmt	=	'**{**' $Stmt^{*;}$ '**}**'

6.2.6 Annotations

PCL uses an annotation mechanism to associate a *(name,value)* pair with any *named* construct. Annotations can be used to communicate information to the runtime environment, code generators, or any other "backend" tools of the PCL processor. Reasoning frameworks use annotations to specify their analytic interfaces. For example, the performance reasoning framework in the publicly released PSK requires that each sink pin be annotated with its *execution time*, which is defined as the latency of a component to react to an event on the sink pin when executing in isolation, that is without any preemption or blocking effects, and no execution time expended on interactions through any of the component's source pins.

Table 6.8: PCL Annotations

Annotate	=	**annotate** *Id* '**{**' $string_{\text{LIT}}$ ',' **const** *TypeSpec* $InitDecl^{+,}$ '**}**'

In the *Annotate* production in Table 6.8, *Id* is the name of the construct being annotated, $string_{\text{LIT}}$ is a tag used to define a class of annotations, and the constant declaration assigns one or more $(name, value)$ pairs to *Id*. The following fragment illustrates several annotations:

Example 6.3: Example annotations.

```
annotate compon {"Pin", const boolean gencode = false}
annotate compon:theReaction {"Pin", const int timeout = 60}
annotate clock:tick {"lambda*", const int period = 450}
```

The code generator looks for "Pin" annotations. In the example above the first annotation tells instructs the code generator to skip code generation on component compon; the second instructs it to change the default timeout value on that component's reaction (compon:theReaction) message handler. The "lambda∗" annotation is one of the annotations defined by the $\lambda*$ reasoning framework (described in Chapter 8).

6.2.7 Verbatim

PCL is fairly expressive, but on occasion programmers require direct access to devices or other low-level platform services, and this often requires access to C libraries or the ability to access C's address types. For these situations, PCL provides a mechanism to escape into C.

The following example illustrates the technique to define the times2add1 function that was previously declared in Example 6.2. The implementation returns one more than the value of the parameter, but then has a side-effect on the parameter. This is not an example of good programming style, but demonstrates several aspects of PCL, such as the representation of reference types as C pointer types and how to bridge the PCL and underlying C namespaces.

Example 6.4: Example verbatim code.

```
int proc times2add1(int &param);
{
  %{ int *loc = $ccl$param; return *loc++; %}
}
```

Everything within the verbatim start '**%{**' and end '**%}**' delimiters is written directly to the output stream (the generated code file). Symbols that are prefixed with '\$ccl\$' are mapped by the code generator to their mangled internal names.[2]

6.3 Structural Elements

The major structural abstractions of PCL are now presented, along with their informal graphical notation.

At the top-level (Table 6.9), a PCL specification consists of a (possibly empty) sequence of annotations, constant declarations, function (proc)

[2]This fragment also demonstrates the kinds of program that PCL tries to discourage by excluding C pointer types. Sometimes, however, such code is necessary.

Table 6.9: PCL Top–Level Structure

TopLevel	=	(*BasicUnit* | *PinUnit*)*
BasicUnit	=	*Annotate* | *Declaration*
PinUnit	=	*Component* | *Environment* | *Assembly*

declarations, type definitions (as described in §6.2), *components* (§6.3.1), *assemblies* (§6.3.2), and *environments* (6.3.3).

6.3.1 Components

The syntax of PCL components is introduced in Table 6.10. PCL components conform to the Pin component model [69], and are independently deployed as Win32 dynamically linked libraries (DLLs).

Table 6.10: PCL Components and Services

Component	=	(([**singleton**] **service**) | **component**) *CompDef*
CompDef	=	*Id* '(' *FormalParm**, ')' *CompBody*
CompBody	=	'**{**' *Declaration**| *PinDef*$^+$| *ReactionDef*$^+$ '**}**'
PinDef	=	*PinModes Id* '(' *PinParam**, ')' '**;**'
PinModes	=	(**sink** | **source**) (**asynch** | **synch**)
PinParam	=	(**produce** | **consume**) *FormalParam*
FormalParm	=	*TypeSpec Id*
ReactionDef	=	See Table 6.14, pp.113

Component behavior is accessed exclusively by other components through *pins*. As a first approximation, pins are analogous to *ports* in Darwin [108], Acme [54] and SAVE [86]. However, pins have several distinctive features and so care must be taken not to overstate the analogy.

Each pin p has three constituent parts, $p =< D, P, S >$, where D is *direction*, P is *protocol* and S is *signature*.

Direction. Pin *direction* is specified by one of the keywords **sink** or **source**. Events arrive at components on their sink pins, and leave through their source pins. The set of sink pins on a component is sometimes referred to informally as the component's *stimulus interface*, and its set of source pins as its *response interface*. However, it is not accurate to think of these as *provides* and *requires* interfaces; this point is elaborated in the discussion of *assemblies* in §6.3.2.

Protocol. Pin *protocol* is specified by one of the keywords **synch** or **asynch**. These correspond to *synchronous* and *asynchronous* interaction protocols, respectively. On a surface level, these correspond to function-based interaction (**synch**) and event-based interaction (**asynch**). It is worth noting here, however, that all pin interaction is asynchronous, i.e., is event based. The deeper significance of **synch** and **asynch** is deferred to the discussion of PCL's behavioral elements in §6.4.

Signature. Pin *signature* is specified as a pin identifier followed by a (possibly empty) sequence of pin formal parameters. The data flow direction of each pin formal parameter is declared by one of the keywords **consume** or **produce**, depending on whether the parameter is read by the component or written by the component, respectively.

PCL defines two kinds of components, specified by one of the keywords **component** or **service**. There are no differences in how components and services are implemented or composed. However, PCL enforces different design rules for components and services. The most important of these is that components must be purely reactive, while services may be either reactive or self–stimulating. In concrete terms, components must have at least one sink pin, and all component behavior is triggered as a consequence of events arriving on sink pins, or from the runtime environment (§6.3.3). Service reactions need not specify any sink pins, and in addition to the triggers available to components, services may be triggered by the Pin real–time operating system, for example device interrupts.

Examples 6.5 and 6.6 illustrate the syntax and informal graphical notation for components and services, respectively. Note that these examples are **not** syntactically valid because all components and services must have at least one reaction, and these have not yet been introduced (see §6.4). Without reactions specifications there is no way to know what **component** FIFOQ and **service** Clock will do, but for the moment let as assume that FIFOQ implements a bounded buffer and Clock implements an event generator.

Note that Clock does not specify a sink pin; it is self–stimulating, and it specifies one source pin Clock:tick that it uses to deliver events, presumably at some known rate. LFIOQ however is a component, and its behavior is exclusively triggered by events arriving on its sink pin, Q:enq (line 3).

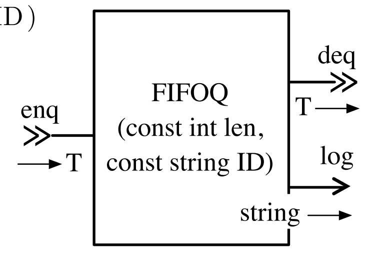
Terminology: The term "component" will be used to refer to both components and services where the distinction is unimportant.

6.3.2 Assemblies

Components in PCL may interact with each other only when their pins have been *connected* by an interaction *operator* '$\leadsto$'. The interaction operator provides for *constructive composition* of components. However, PCL does

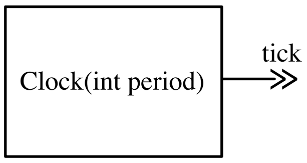
Example 6.5: Component Fragment.

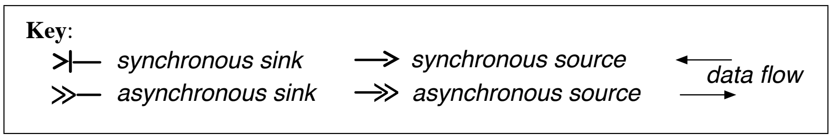
```
typedef unsigned byte T[32];
component FIFOQ (
  const int len, const string ID)
{
  sink asynch enq(
    consume T in);
  source asynch deq(
    produce T out);
  source synch log(
    produce string msg);
  threaded react Work ...
  // continued in Example 6.10, pp.116
}
```



Example 6.6: Service Fragment.

```
service Clock(int period)
{
  source asynch tick();
} // reaction omitted
```





not provide a way to *name* the resulting composition. This is because PCL components are defined in terms of units of deployment rather than units of composition. The significance of this, and the closely related subject of interaction semantics (i.e., constructive composition), are deferred to the discussion of *semantics* in §6.5.

PCL does however provide a way of creating denotable hierarchies called *assemblies*, using the syntax defined in Table 6.11.

Rather than being defined in terms of constructive composition, assemblies are defined in terms of *aggregation* and *restriction*:

Assembly aggregation: PCL assembly *types* define an aggregation of (possibly connected) component and assembly *instances*. The discussion of components has (so far) been in terms of their types; components, environments and assemblies may also be *instantiated.*

Assembly restriction: PCL assembly types restrict (hide) the pins of all

Table 6.11: PCL Assemblies

Assembly	=	**assembly** *Id* '(' *FormalParm**, ')' [*Env*] *AsmbDef*
Env	=	'(' Id_{Env} ')'
AsmbDef	=	'**{**' *Assumes* (*Declaration*| *Interaction*)* *Exposes* '**}**'
Assumes	=	**assume** '**{**' *Declaration** '**}**'
Interaction	=	*CompSrcPin* '⇝' *CompSnkPin*
CompSrcPin	=	$ScopedId_{\mathrm{Pin}}$
CompSnkPin	=	$ScopedId_{\mathrm{Pin}}$| '**{**' $ScopedId_{\mathrm{Pin}}{}^{+,}$ '**}**'
Exposes	=	**expose** '**{**' *ExposedPin**, '**}**'
ExposedPin	=	$ScopedId_{\mathrm{Pin}}$ [**as** *Id*]

component instances. These may be selectively *exposed* by the assembly type, in which case they become *pseudo*–pins of the assembly type.

This basic ideas are illustrated by Examples 6.7 and 6.8, which assembles larger buffered queues from the mystical **component** Q type introduced in Example 6.5. The important point to note is that assemblies do not define behavior.

Example 6.7 defines **assembly** QQ to compose two instances of Q. The meaning of the **assume** clause (line 2) is described later, in Example 6.8, pp.110. On line 4, two instances q1(i), q2(i) of Q are created using QQ's formal parameter **int** i, presumably to create two queues of length i. These component instances are then connected, source–pin to sink–pin, on line 6. The **expose** clause on lines 8–9 is required if QQ is to serve a useful purpose as a queue. Without these lines, no events could arrive on q1, and events leaving on q2 would have no place to go. Note that the synchronous sink pins of q1 and q2 are now hidden by QQ; in and out are pseudo–pins that are really just aliases of q1:enq and q2:deq, respectively.

Note Graphical Convention: Component pins are sometimes depicted with stalks, as in examples 6.5 and 6.6, pp.108, or without stalks, i.e., with the pin symbol directly on the border of components, assemblies or environments, as in example 6.7 and in most of the assembly examples found in this thesis. The line connecting QQ:in to FIFOQ:enq is *not* a connector; it is a pin aliasing from the **expose** clause. It is generally easy to distinguish connectors from aliasing, since the latter are always drawn from an enclosing assembly to an enclosed component or assembly instance.

It is worth noting that source pin q1:log is not only hidden, but it is also not connected. In the earlier discussion of pin direction in §6.10 it was observed that it would be incorrect to regard source pins as a kind

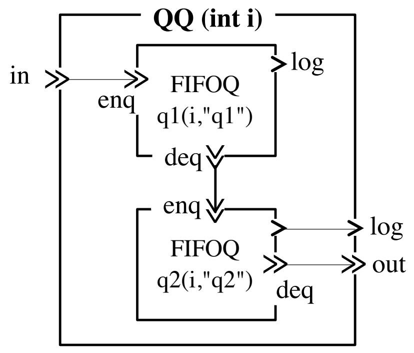
Example 6.7: Assembly.

```
assembly QQ(int i){
 assume{}

 FIFOQ q1(i,"q1"),
       q2(i,"q2");

 q1:deq ~> q2:enq;

 expose {q1:enq as in,
  q2:deq as out,
  q2:log as log}
}
```



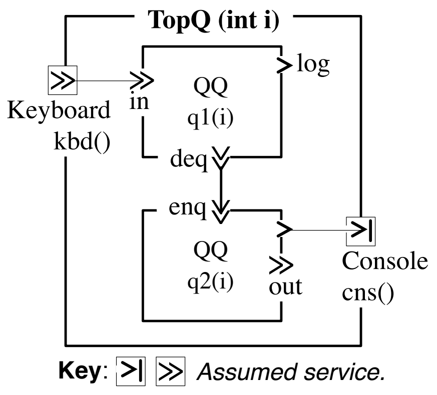
Example 6.8: Top-Level Assembly.

```
assembly TopQ()(Rtos){
 assume{
  Keyboard kbd();
  Console  cns(); }

 QQ qq1(1), qq2(1);

 kbd:type ~> q1:in;
 q1:out   ~> q2:in;
 q2:log   ~> cns:write;

 expose {}
}
```



of "requires" interface; clearly, the log source pin can not be both required and unconnected. Instead, we can regard a component's set of synchronous source pins *that consume data* as defining a "requires" interface (recall that asynchronous source pins may not consume data). An example of this is shown at the end of this chapter (Example 6.13, pp.128).

There are a few basic conformance rules that govern the connection of all pins C1:r ~>C2:s:

- r (any pin on the left side of ~>) must be a **source** pin and s (any pin on the right side) must be a **sink** pin.
- the signatures of r and s must agree in the number and type of formal parameters.

- each corresponding pair of formal parameters (r.p, s.p) must have complementary **produce** or **consume** direction.
- C1 $\neq$ C2. Services may be self–stimulating, but no component or service may be self–connected.

The TopQueue assembly defined in example 6.8 differs from QQ in several important ways. The first difference is the appearance of the additional parenthetical expression "(Rtos)" on line 1. Assemblies ultimately will be instantiated within a runtime environment (the subject of §6.3.3), and TopQueue will be instantiated in *an instance of* the Rtos environment. Lines 2–4 specify that instances of two Rtos services Keyboard kbd() and Console cns() are assumed. These lines do *not* create these service instances, but merely give them local names. Line 6 creates two instantiations of QQ, and provides to each instance the queue length (corresponding to the formal parameter **component** FIFOQ (**const int** len) from Example 6.5). Lines 8–10 establish all the connections. There is no need to expose any pins to the runtime environment, although there is no harm in doing so.

6.3.3 Environments

Environments represent the runtime environment of components and assemblies. As discussed in Chapter 5, reasoning frameworks may make assumptions about the runtime environment, and different reasoning frameworks might be valid for different runtime environments. A distributed or heterogeneous system might have components executing in, and interacting across, several quite distinct runtime environments.

The syntax of environment specification, shown in Table 6.12, is quite simple when compared with components and assemblies, and consists mainly as a sequence of constant declarations, annotations and services.

Table 6.12: PCL Environments

Environment	=	**environment** *Id* '**{**' *EnvironPart* '**}**'
EnvironPart	=	(*Declaration*| *Component*| *Annotate*)*

An environment is likely to provide services that access platform devices and other platform–dependent details. The Rtos environment referred to in the following examples is part of the PSK distribution (see Chapter 5 for details on the PSK), and includes as basic services a variety of clocks, a network gateway, keyboard and console; and extensions for audio mixing that include various decoders, inverters, adders, etc. Many of these services require direct access to platform devices, and their development in PCL is a good test of its expressiveness and programmability.

6.3.4 Instantiation

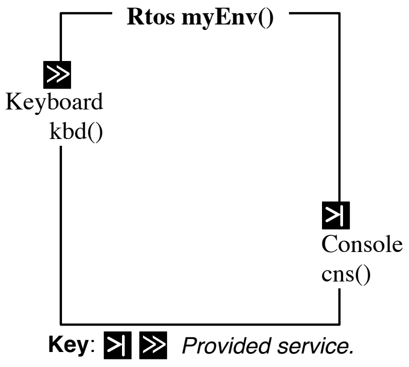
Returning to Example 6.8, one important step remains: to *instantiate* the Rtos runtime environment and TopQ top–level assembly. Unlike the earlier instantiations of components and assemblies, these instantiations must ensure that whatever services are assumed by the top–level assembly (TopQ in this case) are provided by the runtime environment. For this we need to complete the syntax for *PinInstance* that was begun earlier (see Table 6.6, pp.103). The complete syntax is defined in Table 6.13.

Table 6.13: PCL Provisionings

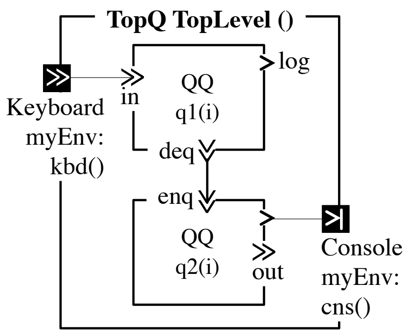
PinInstance	=	*Id* '(' *Exp**, ')' [*Provision*]
Provision	=	'{' *ServiceInstance** | *AsmbUses** '}'
ServiceInstance	=	*ScopedId*$_{\text{Service}}$ *Id* '(' *Exp**, ')' ';'
AsmbUses	=	*ScopedId*$_{\text{Assumed}}$ '=' *ScopedId*$_{\text{ServiceInstance}}$ ';'

Example 6.9: Instantiation.

```
1  // instantiate Rtos
2  Rtos myEnv ()
3  {
4   /**** services ****/
5   Rtos:Keyboard kbd();
6   Rtos:Console cns();
7  };
```



```
8  // instantiate TopQ
9  TopQ TopLevel()
10 {
11  /**** assumptions ****/
12  TopQ:kbd = myEnv:kbd;
13  TopQ:cns = myEnv:cns;
14 };
```



Example 6.9 illustrates the two–step process for instantiating environments and top–level assemblies. Line 2 instantiates the Rtos environment

as myEnv, and lines 5–6 provision myEnv with two service instances. Line 9 instantiates the TopQ assembly as topLevel. You might recall from lines 1–4 from Example 6.8, pp.110 that TopQ "knows" it will be deployed in an instance of Rtos and expects that its assumed services will be provided by that environment instance. In lines 12–13 these topLevel assumptions are satisfied by myEnv. The topLevel assembly instance is deployed into the myEnv environment instance by means of a controller (a "main program") that executes as a high–priority thread in the Pin real–time operating system. (See Chapter 7, §7.3.2, pp. 138 for additional detail about controllers.)

6.4 Reactions

The behavior of components resides exclusively in their *reactions*, and all component types define *one or more* reactions. The syntax of reactions is described in Table 6.14.

Table 6.14: PCL Reactions

ReactionDef	=	[**threaded**] **react** *Id ReactAlphabet ReactPart*
ReactAlphabet	=	'(' $Id_{\text{Pin}}{}^{+,}$ ')'
ReactPart	=	'**{**' *Declaration** *Statechart* '**}**'
Statechart	=	See Table 6.15.

Reactions are defined as *threaded* or *unthreaded* by using the optional keyword **threaded**. Threaded reactions execute on their own independent thread of control, while unthreaded reactions execute on the thread of control of some other reaction (of some other component), which itself might be unthreaded and hence executing on yet another reaction thread, and so on. Explicit threading is an unusual but not novel feature in architecture description languages (see Koala [166] for example). Making concurrency explicit is necessary on practical grounds; any reasonably complex system will require at least some concurrency, and making all behavior concurrent will not scale. Explicit concurrency is also necessary for analysis, and both PSK reasoning frameworks (Chapter 8) make use of this extra information.

Reactions are parameterized by a set of pins; these define the stimulus and response interfaces of the reaction. Reactions also specify a state machine of the *reactive behaviors* of components, and when composed with '$\rightsquigarrow$>' specify *interactive behaviors* of the composed components. There are a few design rules that govern the allocation of pins to reaction R of a component C, i.e., C:R:

- Each sink pin C:Snk is allocated to *exactly one* reaction.

- Each source pin C:Src is allocated to *at least one* reaction
- Asynchronous sink pins may only be allocated to threaded reactions.

Thus, sink pins are uniquely associated with reactions, but source pins may be shared by reactions.

6.4.1 Pin Events

As suggested by their keywords, reactions are intended to be *reactive*—they respond to changes in the environment; in PCL these are delivered as events on sink pins. Each reaction, then, is implicitly endowed with an event handler, although for reasons of accident these have come to be called *reaction handlers* and this is the term used here. If a reaction is **threaded**, its reaction handler is executed on the reaction thread, otherwise (i.e., if it is unthreaded), its reaction handler is executed on its *caller*'s thread. Thus, unthreaded reactions are effectively library functions, and they can be access without explicit synchronization or message queuing.

Reaction handlers can handle different types of events, and by far the most important of these are *pin events*. Each declared pin P induces a pair of event types ^P and $P, pronounced "begin P" and "end P," respectively. Each of these event types has a *signature* that is defined as an *order-preserving* projection of P's **consume** parameters onto ^P and **produce** parameters onto $P. To understand how these event types are used to define the semantics of component interaction, and how different interaction policies can be defined by components or containers (which are described in Chapter 7), it is necessary to describe how the behavior of reactions is specified using PinCharts.

6.4.2 PinCharts

PCL adopts a subset of UML statecharts for describing reactions. The PinCharts syntax is defined in Table 6.15. The syntax and semantics of PinCharts has a wholly consistent interpretation of all UML "semantic variation points" (a term defined by the UML standard).

A brief and informal summary of PinChart semantics is a useful preliminary:

Current State. A Pinchart is said to be "in" a *current state* at any given instant, beginning with the initial *start* state. The transition from the start state to an initial state occurs when the Pinchart is activated. In PCL , this happens when the component is instantiated.[3]

Event Trigger. Behavior is *triggered* by the arrival of an event (PCL keyword **trigger**). All transitions with matching triggers become *activated*;

[3]A more complete description of the component lifecycle is provided in Chapter 7.

Table 6.15: PCL Statecharts

Statechart	=	(*StateDecl* | *TransDecl*)$^+$
StateDecl	=	**state** *Id* '**{**' *Statement* '**}**'
TransDecl	=	*FromState* '–>' Id_{State} '**{**' [*Trigger*] [*Guard*] [*Action*] '**}**'
FromState	=	**start**| Id_{State}
Trigger	=	**trigger** (*PinTrigger*| *TimeTrigger*| *ChangeTrigger*)
PinTrigger	=	*PinEvent* '**;**' (See Table 6.5, pp.102)
TimeTrigger	=	**after** Exp_{int} '**;**'
ChangeTrigger	=	**when** Exp_{boolean} '**;**'
Guard	=	**guard** Exp_{boolean} '**;**'
Action	=	**action** *Statement*

the event is discarded if no transition has that type of event as a trigger. PCL supports three kinds of events: *pin events*, *timer events* and *change events.* The example shows only the use of pin events.

Guard Evaluation. The *guards* of all activated transitions are evaluated (PCL keyword **guard**). If no guard is *satisfied* the event is *silently discarded*; it is good practice to avoid this situation. If no guard is *specified* the transition is treated as if it had specified a guard that is always satisfied.

Transition Firing. A non-deterministic choice is made of one transition from the set of transitions whose guards are satisfied, and this transition is said to have *fired.* Nondeterminancy is sometimes useful in design, but seldom useful in programming.

Execution Order. The transition *actions* of the fired transition are executed (PCL keyword **actions**), the target state actions are executed[4], and the target state becomes the current state; the cycle repeats from here.

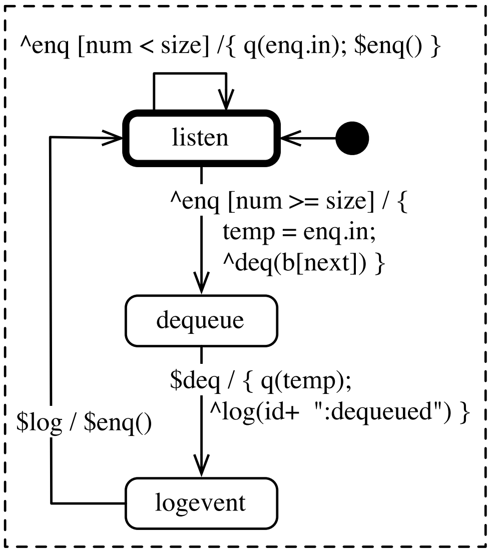
Example 6.10 completes the specification of **component** FIFOQ that was begun in Example 6.5. The reaction is also described using standard UML graphical notation; accepting states (defined, below) are shown in bold outline.

In this example, **component** FIFOQ defines one threaded reaction **threaded react** Work (lines 3–29); Work must be threaded because FIFOQ:enq is an asynchronous pin. Each reaction must define a transition from the **start** state (line 8). There are restrictions on the actions that may be performed

[4]UML defines both entry and exit actions on states; however, PCL makes do with only entry actions, as Pin provides no mechanism to interrupt a state's actions.

Example 6.10: Statechart Reaction.

```
 1 component FIFOQ (const int len, const string id) {
 2   // ...continued from Example 6.5, pp.108
 3   threaded react Work (enq, deq, log) {
 4     int num = 0, next = 0; // could initialize at start
 5     T b[MAX], temp;
 6     proc void q(T &it){ b[next++] = it; next %= len;}
 7
 8     start->listen {}
 9 //-- ^enq, !full, finish
10     listen->listen{
11       trigger ^enq;
12       guard num < size;
13       action {
14         q(enq.in);
15         $enq()}}
16 //-- ^enq: full, continue
17     listen->dequeue{
18       trigger ^enq;
19       guard num >= size;
20       action {
21         temp = enq.in;
22         ^deq(b[next]);}}
23 //-- $deq: continue
24     dequeue->logevent{
25       trigger $deq;
26       action {q(temp); ^log(id + ":dequeued");}}
27 //-- $log: finish
28     logevent->listen{trigger $log; action $enq();}
29   } // end Work
30 } // end FIFOQ
```



on this initializing transition; for example, no interactions with other components are permitted. The start transition tends to be superfluous because PCL allows static initialization of variables (line 4).

A state is said to be an "accepting state" if it has event–triggered transitions, and a "reacting state" otherwise. PCL imposes two rules concerning accepting states:

1. If a state has *any* event–triggered transition, then *all* of its transitions must be event–triggered.

2. For components, but not for services, it is required that all target states from the **start** transition be accepting, and further that all execution paths in a reaction complete in an accepting state.

This latter constraint can not be enforced by the PCL processor, but it can be checked with the model checking reasoning framework described in Chapter 8. To anticipate just a bit, the following claim would be appropriate:[5]

```
annotate FIFOQ {
  "comfort", const string FullyReactive =
   "G((^enc => F $enc))"
}
```

In the example, FIFOQ:Work *implicitly* defines three states: listen (an accepting state), dequeue and logevent (reacting states).

Lines 10–22 define the reactive behavior of FIFOQ:enq. There are two cases: lines 10–15 cover the first case, when a new item arrives to a non–full queue (line 12, num < size), while lines 17–22 cover the second case, when the queue is full (line 19, num >= size). It might have been possible to write the guard for latter case as num==size, and then leave it to the model checker to establish that num > size can never hold. However, it is always risky to under–specify guards because UML semantics (which are honored by PCL) requires that unhandled events be silently *discarded*.

Case 1: Queue has room. The incoming item is placed in the buffer (line 14, q(enq.in)), where **proc void** q(T &it) is defined locally (line 6) to make the reaction somewhat easier to read. The reaction then *generates* an "end interaction event" for the reaction (line 15, $enq()). Having completed the transition action for the transition listen –>listen introduced on line 12, the new current state becomes (once again) listen; had this state been explicitly defined its actions would have been executed.

Case 2: Queue is full. This is the more interesting case, as it requires FIFOQ to interact with other components. Because the scope of a pin parameter is the transition, not the reaction, a temporary copy of the pin parameter is made on line 21 (temp = enq.in), and then on line 22 a *begin interaction event* is generated by ^deq(b[next]) to forward the oldest item in the queue on the source pin C:deq. PCL requires that reactions immediately wait on the completion of interactions. For this reason, the transition relation on line 17 listen –>dequeue specifies a reacting target state dequeue (defined on lines 24–26) that only triggers on ^deq(b[next])'s matching *end interaction event* $deq. A similar chain of begin/end interactions is initiated on line 28, which is triggered by a $log event, and whose final action is to generate the end reaction event $enq().

[5]This is not *quite* strong enough as it assumes the environment is also reactive.

6.4.3 Other Event Types

PCL supports two other UML event types: *timer events* and *change events*, which are briefly summarized here. The semantics of events (in UML and PCL), is quite tricky—when timers are started, when they are cleared, when change conditions are evaluated, precedence, etc., and is discussed in §6.5.2, pp.121.

Timer events and time–triggered transitions. A *time trigger* is an integer-valued, side effect free expression that causes a timed event to be generated *no sooner than* the amount specified by the time trigger expression (in milliseconds). It is important to note that timed events are not clocks: specifying wait (100) means that the environment will generate a timer event *no sooner than* 100ms, and incidentally will attempt to provide that event as soon as possible thereafter. Clock capabilities are provided by services, and indeed several kinds of clocks, each with its own *guaranteed* inter-arrival distributions (e.g., uniform and exponential distributions) are provided with the public release of the PSK.

Timer events are useful mainly for handling communication failures or other disruptions to component interaction.[6] For example, time–triggered transitions from the listen state could be used to signal that the queue component is waiting longer than expected on the arrival of a new item, or analogously on the dequeue and logevent states that an interaction is taking longer to complete than expected.

Change events and change notification–triggered transitions. Components may have more than one reaction, and PCL allows state (constants, variables, functions) to be defined at component scope and shared by reactions. Change events are used to communicate changes in state shared by reactions within a component. Component– and reaction–scoped variables can both appear in change trigger expressions. However, if no component–scoped variables appear in a trigger, then the trigger is semantically equivalent to a guard, and the language processor is free to use guards (which are far more efficient) if it chooses. The condition is evaluated when the change trigger is created, and it is evaluated subsequently whenever the value of any relevant (i.e., "watched") component–scoped variable changes.

In practice, components with multiple reactions have proven to be less useful than originally anticipated, and as a rule a component with N reactions is just as easily implemented as N components, each having one reaction, and thereby sidestepping the need for inter–reaction synchronization. On the other hand, there is also additional code and code management overhead associated with each additional component.

[6]The Pin runtime environment provides an alternative *timeout* mechanism that can be manipulated via PCL annotations. As always, there is a fine line in deciding what to specify at a design level and what to leave implicit.

6.4.4 Controller Alerts

Components may initiate interaction with their environments using the built–in **alert** mechanism, which has the following definition:

```
extern proc alert (string &msg, int status);
```

The controller will display msg on the Pin runtime console (which is *not* the same console as provided by the Rtos console service). If status is a nonzero value, the controller will initiate shutdown procedure, and report *abnormal termination* if status < 0. If status $== 0$ the controller will return control to the issuing component.

Using environment–provided services rather than alerts to interact with environments is a valid design choice that is available to developers. However, a direct channel between component and environment is a great convenience and results in simpler assemblies.

6.5 Semantics

The previous discussions have defined the formal syntax of PCL, and have provided an informal description of its *constructive semantics* by way of a few very simple examples. A more formal treatment of PCL semantics is provided in Appendix A. Here only the basic schemas used to define this semantics is summarized, and there are two: a denotational semantics [128] of PCL interaction terms of Hoare's algebra of Communicating Sequential Processes (CSP) [75], and an operational semantics [138] of reactions. The aim here is to convey the intuition of interaction and reaction semantics rather than the gory details, which are safely hidden in Appendix A.

6.5.1 Interaction Semantics

The aim of *interaction semantics* is to give an accounting of the behavior of the 'C1:r~>C2:s' operators that are used to connect component instances. As mentioned several times already, we adopt a process–algebraic approach to defining interaction behavior, both in the definition of PCL interaction semantics and in the ComFoRT interpretation, though these differ in approach as discussed further ahead.

It is certainly possible to define a *naive* semantics ("minimalist" might be less pejorative) that interprets component instantiations C c1(),C c2() as CSP process P1, P2, respectively, and interpret C1:r~>C2:s as the parallel composition P1 || P2. However, this would not observe certain behaviors that are quite important contributors to extra–functional behaviors such as performance, for example FIFO queueing (or other) policies for events arriving on a component sink pin, or blocking behavior for reactions initiating interactions on source pins. To observe such behaviors a somewhat more elaborate interpretation is required, the basics of which are illustrated graphically in Figure 6.1.

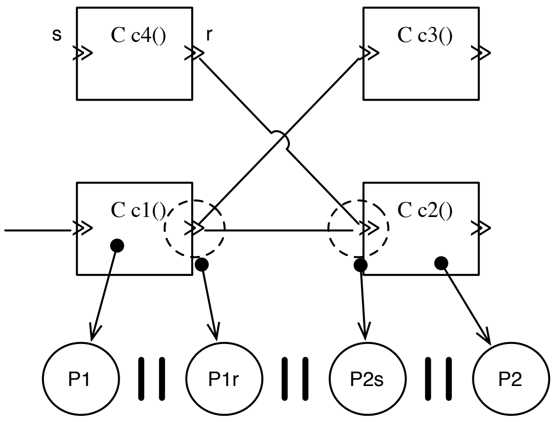


Figure 6.1: Interaction Semantics: Schema

Consider the small arrangement of component instances in Figure 6.1, and the question of what processes will model the behavior of C1:r⤳C2:s, which in this case is one of two interactions in the serial unicast (the other being C1:r⤳C3:s initiated by C1. As with the naive semantics, component instances are assigned CSP processes.

Here, though, two auxiliary processes P1r and P2s are also constructed, called *source glue* and *sink glue* processes, respectively. The source glue process P1r defines where blocking occurs in the initiating reaction, and the order in which events are queued to c2:s and c3:s. The definition of "glue" processes depends on details of connection topology, and in this example P1r is constructed from, and it's *alphabet* is defined by, C1:r⤳C2:s and C1:r⤳C3:s, and similarly P2s is constructed from and alphabet defined by C1:r⤳C2:s and C4:r⤳C2:s. The CSP process defined by P1r || P2s observes the behavior of an "asynchronous connector." An analogous semantic interpretation for synchronous interactions likewise observes the behavior of "synchronous connectors."

The ComFoRT interpretation uses a similar construction, but is optimized for model checking. In particular, the interpretation sketched above *over–approximates* the potential concurrency in the assembly, since the implementation of the assembly need not (and in the current implementation of Pin, does not) have separate Pin threads for both glue processes. Since state space explosion in model checking arises from process composition, it pays to minimize the number of distinct CSP processes used to model behavior. ComFoRT also avoids constructing CSP processes for *unthreaded* reactions (see [83, 81]). A closer look at PCL interaction semantics can be found in Appendix §A.1.

6.5.2 Reaction Semantics

While interaction semantics describes the *external* behavior of components, reaction semantics defines their *internal* behavior. There are two aspects of reaction semantics:

- semantics of the imperative action language
- semantics of event handling and the *internal* view of interaction

The first is quite routine, involving order of expression evaluation, environments (mappings from names to locations) and stores (mappings from locations to values), control flow, etc. No semantics for this part of PCL is provided here. Several have been defined at various points, but because PCL's action language is so basic have never proven to be useful, or at least worth keeping up–to–date with PCL as it evolved. The second, however, while not particularly complex (when compared with interaction semantics) requires explicit treatment because it defines the PCL interpretation of the subset of UML statecharts used by PinCharts, and also formalizes the relationship between reactions defined in PCL and the *Pin containers* that manage the execution of reactions.

In brief, each PCL reaction is implemented as a callback function called the *reaction handler*. The reaction handler is invoked by a Pin container with the next FIFO–ordered event when the reaction has inbound events on its event queue (sink pin events, time events, change events, and various "undocumented" events such as measurement and other instrumentation events). The PCL reaction semantics defines what reactions do with these events, how Pin mechanisms are used to construct time events and change events, how and when they are constructed and deleted, how transitions are enabled and fired—in general how Pin implements PCL PinCharts.

Several formalisms have been used (denotational, small–step structural operational) but the most practical and useful by far has been proven to be the *pseudo–code* of a generic reaction handler. This can be found in Appendix §A.2.

6.6 Pragmatics

Syntax and semantics are concerned with structure and meaning of a language; pragmatics is concerned with its use. A discussion of the Seam as a problem of language design, and therefore the role of pragmatics in defining the fitness of the Seam, is taken up in Chapter 10, *Theories and Co-Refinement*. Here several aspects of PCL pragmatics are discussed in convenient proximity with its syntax and semantics. Because the action language is, with the inclusion of *verbatim* syntax, essentially as expressive as C, few pragmatic arise in the description of component behavior. Also, the Pinchart subset of statecharts appears from experience to be sufficiently expressive if component behavior can be conveniently described by UML statecharts.[7]

Instead, issues of pragmatics tend to center on the expressiveness of PCL to different coordination schemes among components. For example, what kinds of topologies can be constructed? Are two kinds of connector protocols sufficient? Can different coordination schemes be defined even without "first class" connector types? The following discussion touches on some of these issues.

6.6.1 Reactivity and Immediacy

PCL constrains the behavior of component (but not service) reactions in two ways: reactivity and immediacy; though it is admitted that these names are not particularly helpful. The design rules are better described in terms of their CSP formalization.

Given a reaction R with sink pin s and source pin r (signature and protocol does not matter at this point), the following behavioral scheme demonstrates the two constraints:

$$R = s \xrightarrow{\tau} r \rightarrow \bar{r} \xrightarrow{\tau} r \rightarrow \bar{r} \xrightarrow{\tau} \bar{s} \rightarrow R \tag{6.1}$$

R is a reaction that accepts an interaction on channel (sink pin) s and becomes a process that can perform actions that are not visible to the environment (that is the meaning of $\xrightarrow{\tau}$), and then becomes a process that can interact on channel (pin) r, etc., as described in §6.5.1.

Components are reactive ("reactivity"). The pairing of s as the first accepting event in a reaction with a matching $\bar{s}$ as the last accepting event defines a *reactive* process. PCL requires that all component reactions are reactive in this sense.

Interactions are immediate ("immediacy"). The pairing $r \rightarrow \bar{r}$ describes an *immediate interaction*; once it starts an interaction on r, the R process can do nothing but wait for the interaction on the source pin to complete with $\bar{r}$ before moving on. PCL requires that *all* interactions be immediate.

Although it is a slight abuse of CSP notation, immediacy is much more clearly shown using the following scheme, which will be the preferred form henceforth:

[7]Statecharts are not suitable for all kinds of programs; they are well suited to event–driven and reactive programs.

$$R = s \xrightarrow{\tau} r\bar{r} \xrightarrow{\tau} r\bar{r} \xrightarrow{\tau} \bar{s} \rightarrow R \tag{6.2}$$

It was mentioned earlier that PCL can not directly enforce reactivity. However, PCL can and does enforce immediacy: target states of transitions whose actions (either a transition action or state action) generate a ^sourcePin (...) "begin interaction" must end with that action and all outgoing transitions must trigger on a matching $sourcePin "end interaction" event (see for example lines 25 and 28 of Example 6.10, pp.116).[8]

Of course, reactivity brings with it all the advantages of finite termination, and also brings along all the attendant issues of decidability. Immediacy means that components are not free to defer or delay completing an interaction once requested. Together these rules achieve a reasonably strong but by no means complete decoupling of reaction from interaction. A reactive component will complete what it has been requested to do; and immediate interaction means that components will obtain results from requests they initiate in a standard way.

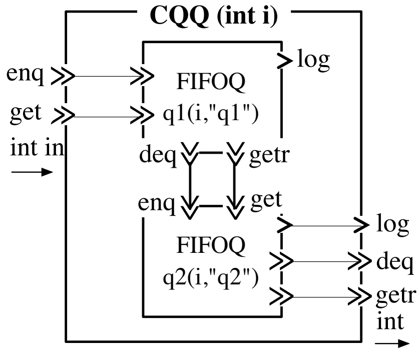
6.6.2 Coordination Expressiveness

Reactivity and immediacy are strong constraints—but are they *too* strong, or perhaps even too *weak*? The answer will depend on many factors: the kind of problem being solved; whether it is a recurring problem and if so its intrinsic variety; the need (if any) for objective evidence of predicted system behavior; and so on (see Chapter 10). Seam abstractions need to provide programmers and architects with coordination primitives that are "satisficingly" varied and expressive. Too much of either will make it difficult to establish invariants that reasoning frameworks may require; too little and design space may become overly–constrained, and programming too awkward, to be of interest to practitioners. It is the nature of the Seam that there be no clear dividing line between too much and too little of either, and certainly there is no stable line where customer needs or engineering practices change.

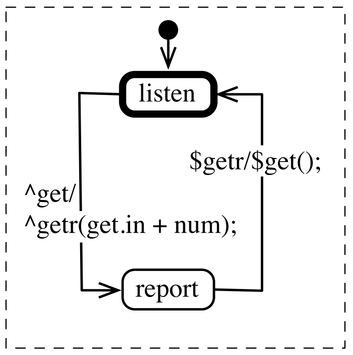
The following examples, although simple, show where the original design of PCL drew the line, and how its location has already shifted in response to changing needs, and might therefore change in the future. The examples pose a simple question in the use of the FIFOQ component: *how many items are in the queue?* Some degree of coordination of FIFOQ instances is required to answer the question. Two cases will be considered, which will lead to three examples. In the first case, **component** FIFOQ uses asynchronous pins to communicate the number of elements it has (Examples 6.11, pp.124 and

[8]It might be considered that interactions on asynchronous source pins should not require an explicit trigger on the matching end interaction event, but this turns out to be a minor inconvenience when compared with the uniformity it brings in the way reactions are specified.

Example 6.11: Reactive Asynchronous Coordination.



component FIFOQ has been extended by **sink asynch** get(**int** in) and **source asynch** getr(**int** out). Note that the position of pins on the bounaries of components is freely changed to suite the needs of convenient connection topology.



The original FIFOQ:Work reaction has one new state and two transitions. The incoming ^get may be coming from another queue so its data parameter is added to the local length and then passed to the (possibly) next queue with ^getr.

6.12, pp.126), while in the second and substantially simpler case, it uses synchronous pins (Example 6.13, pp.128).

Reactive Asynchronous Coordination

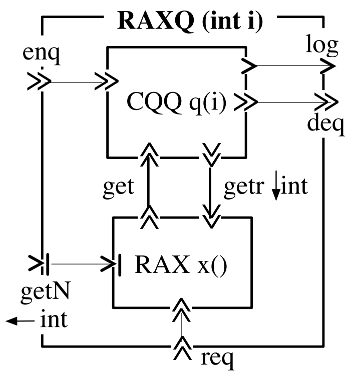
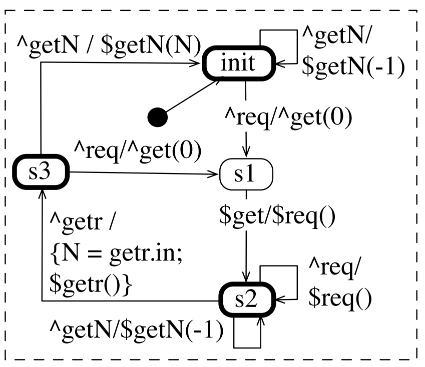
Example 6.11 introduces the first case by making the appropriate changes to the FIFOQ component developed in earlier examples.

A downside of the design in Example 6.11 of course is that all queue instances most communicate even if only a subset of them have items, but our concern is with coordination, not with efficiency in this example. Example 6.11 (cont), illustrates what a component must do if it wishes to obtain the number of items in a queue from q(i) *and* satisfy the requirement that it be *reactive*. For this purpose a coordinator **component** RAX has been introduced (for *reactive, asynchronous coordination*).

Eventually–Reactive Asynchronous Coordination

In this case, requiring *reactivity* introduces considerable complexity in doing something (obtaining a queue length) that ought to be simple to do; the **component** RAX reaction would have become unmanageable had it been required to coordinate with several internal queues.

Example 6.11: Reactive Asynchronous Coordination (cont).

 RAXQ (int i) enq log CQQ q(i) deq get getr int RAX x() getN int req The reactive asynchronous coordinator exposes two sink pins, an asynchronous pin (req) to initiate a request, and a synchronous pin (getN) on which to await a response. Two internal pins **asynch sink** getr and **asynch source** get coordinate with like–named queue pins. Two exposed pins are required because RAX must be *reactive* with its external client *and* with its internal client q(i).	 ^getN / $getN(N) init ^getN/ $getN(-1) ^req/^get(0) ^req/^get(0) s3 s1 ^getr / {N = getr.in; $getr()} $get/$req() s2 ^req/ $req() ^getN/$getN(-1) This is the Pinchart for RAX. Each of the accepting states **init**, **s2** and **s3** must be prepared to accept requests on both ^getN and ^req, including duplicates (recall that UML semantics will "silently discard" inbound events that are not handled). Here, \$getN(−1) signals premature synchronization by returning an illegal number of elements.

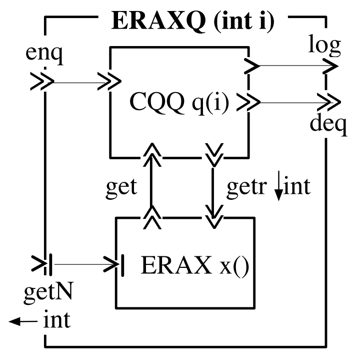
It is interesting that this particular coordination pattern did not arise in any of the industrial cases from electric grid substation control or industrial robot control (see Chapter 9). It *did* arise, however, in streaming audio applications (as well as in model checking the industrial robotics communication library, discussed in Chapter 9).

This forces a relaxation of reactivity to *eventually reactive.* The schema for eventually reactive components is shown in 6.3 and 6.4, and can be contrasted with the stronger form of reactivity shown earlier in Eq. 6.1. In an eventually–reactive component, a reaction need not immediately react to a pin request before moving on to other requests, but it should do so eventually.

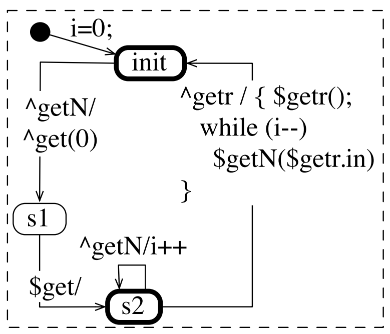
$$R = s \xrightarrow{\tau} r\bar{r} \xrightarrow{\tau} R' \xrightarrow{\tau} r\bar{r} \xrightarrow{\tau} \bar{s} \rightarrow R \tag{6.3}$$
$$R' = s' \xrightarrow{\tau} r\bar{r} \xrightarrow{\tau} r\bar{r} \xrightarrow{\tau} \bar{s}' \rightarrow R \tag{6.4}$$

Example 6.12 shows an eventually reactive coordinator **component** ERAX.

Example 6.12: Eventually Reactive Asynchronous Coordination.



This is an eventually–reactive coordinator. It is no longer necessary to require clients of the coordinator to manage a two–step coordination protocol because it is permitted for this component to accept intermediate interactions on its sink pin ^getr before responding to ^getN.



In contrast to RAX, the accepting state **s2** can keep count of how many requests have arrived on getN, and wait until the arrival of stimulus from the queue on ^getr before responding to *all* requests on getN. What is *not* shown is the queuing policy—does the first $getN correspond to the first or last ^getN?

As expected, the guarantee of *eventual* reactivity allows the coordinator to expose just one synchronous getN pin.

However, this coordination pattern introduces a number of difficulties for PCL, not all of which it was equipped to address in its original form. As discussed in §6.6.1, a design goal of PCL is to separate reaction from interaction, and for this purpose component programmers were given limited access to ^p/$p() events. Technically, event types in PCL are denotable and expressible (can be named and used in expressions), but are not *storable.* Were events to also be storable, component programmers could develop arbitrary coordination schemes—this is too much expressiveness to serve the Seam.

As a consequence of having denotable, expressible, but not storable event types, the arriving ^getN events on the ERAX coordinator in Example 6.12 can not be stored in an array (for example). There is no way to tell from the Pinchart what policy is being used; the $getN events being generated might correspond FIFO or LIFO to the previously arrived ^getN events. In fact, the current policy implemented by PCL is *not* what one would expect—it uses FIFO.

PCL adopts a FIFO policy because doing so ensures that components can not change the FIFO policy one expects when communicating on pins—i.e., calls to an interface are answered in the order they arrive. However, why PCL has adopted a FIFO policy is not at issue in the present discussion; it is a simple–enough matter to add an annotation to instruct PCL to adopt another policy. More interesting is the question where to draw the line between expressiveness and restrictiveness. The original requirement for reactivity (Eq. 6.1) was too restrictive; eventual reactivity is as undecidable as reactivity, but it adds useful expressiveness while still being suitably constrained.

Reactive Synchronous Coordination

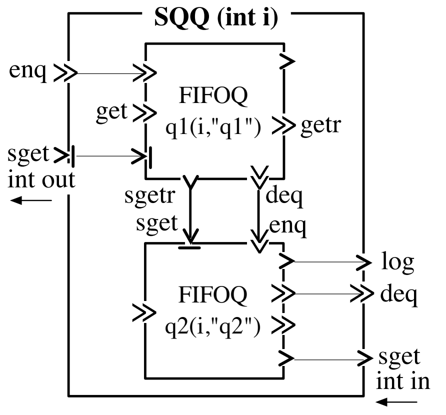
To complete the discussion about coordination expressiveness, we consider the case that FIFOQ uses a synchronous rather than asynchronous pin to communicate its number of elements. Besides being simpler as a coordination scheme, synchronous communication may also be required to remove cycles in a component connection topology; both of the coordination schemes shown above have cycles in their connection graphs. PCL has no difficulties with cycles, and the Pin components generated from a PCL specification can be connected in arbitrary topology, subject only to pin conformance and the rules against self–connection. However, the $\lambda*$ performance reasoning framework described in Chapter 8 imposes acyclic interaction as an analytic constraint.

Example 6.13 shows the synchronous coordination solution, which be done entirely without an external coordinator component. The only point worth remarking is that source pins that consume data (which means they must also be synchronous pins) are regarded as *required* interfaces—they must be connected. There is no shortcut in PCL to this requirement, and for this purpose a **component** K(**const int** v) is specified whose instances always and only returns the value v from an unthreaded reaction that handles K:val requests. This involves quite a bit of overhead to produce zero results, and perhaps this is an area where a judicious use of annotations might be useful.

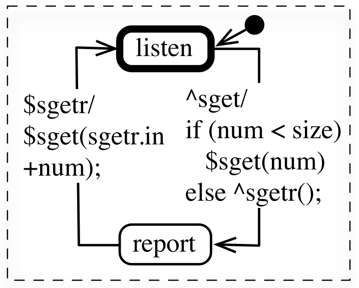
6.7 Summary

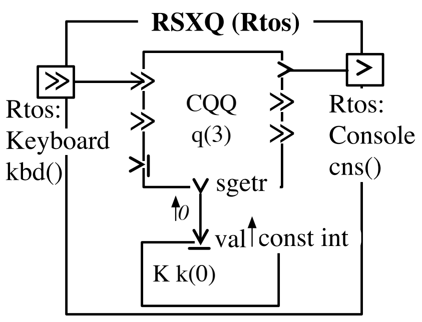
PCL is a specification language that is more detailed in the behaviors that it describes than architecture description languages, and less flexible (but not less expressive, functionally) than programming languages—it occupies the Seam. It provides a basis for substantial automation, for program generation and for program and architecture analysis by reasoning frameworks. PCL is by no means a perfect language, but its current syntax and semantics—warts and all—are a result of its evolution to different kinds of design problems.

Example 6.13: Reactive Synchronous Coordination.



component FIFOQ is extended by **sink synch** sget(**produce int** out) and **source synch** sgetr(**consume int** in). Its reaction can return its length immediately if it is not full. However, because sgetr *consumes* data it must be connected, in this case to constant **component** K k(0) which always returns 0.





PCL formalizes the Pin component model, and makes use of the real–time platform services of the Pin Component Technology, described in Chapter 7.

Chapter 7

The Pin Component Technology

This chapter describes the Pin component technology. While Chapter 6 describes a logical view (one might say "semantic" view but for the overloading of that term with reasoning framework *interpretation*) of Pin, this chapter describes its implementation.

§7.1 discusses the major design objectives for Pin. The remaining sections provide a closer look at the implementation. §7.2 describes the logical and layered architectural views of Pin. §7.3 describes component *containers* (§7.3.1) and assembly *controllers* (§7.3.2), both of which are largely implicit in PCL. §7.4 describes the "real time operating system" (RTOS) layer of the Pin runtime environment. Finally, §7.5 summarizes how Pin satisfies the design objectives sketched in §7.1.

7.1 Pin Design Objectives

Before developing Pin we defined several top–level design objectives that Pin must satisfy if it were to serve *as a foundation for PECT*.

The emphasis on "as a foundation, etc." serves to highlight an important point: Pin is not intended to be used directly by architects and programmers (though it could be), but indirectly through PCL or some equivalent language (or other "frontend") veneer. In this regard Pin can be thought of as defining a "machine code" for PCL. In short, our ambition was not to design the most sophisticated component technology, but rather a core upon which to build a new kind of component technology called PECT.

We had formed definite opinions about what characteristics of a component technology were desirable for PECT based on our first prototype PECT[71, 70]. From these experiences we decided that Pin must:

1. Provide only those features needed for predictability by construction.
2. Provide a *simple* programming model and an execution model.
3. Provide multiple ways of enforcing design constraints.
4. Be adaptable to new target environments.
5. Be freely distributable and installable on conventional PC desktops.

Objectives (1) and (2) are intended to simplify automation: (1) to simplify the development of code generators for components and assemblies; (2) to simplify the development of sound interpretations. Objective (3) is intended to ensure that Pin supports, if not simplifies, the development of reasoning frameworks by providing various "hooks" to satisfy reasoning framework constraints. Objective (4) is intended to ensure that a PECT can be sustained through new operating system releases, and to maximize the potential to re–target any PECT to different deployed–system platforms.

Objective (5) is intended to facilitate the transition of PECT technology to university or industry users.

7.2 Pin Architecture

Figure 7.1 provides a logical, layered view of Pin.

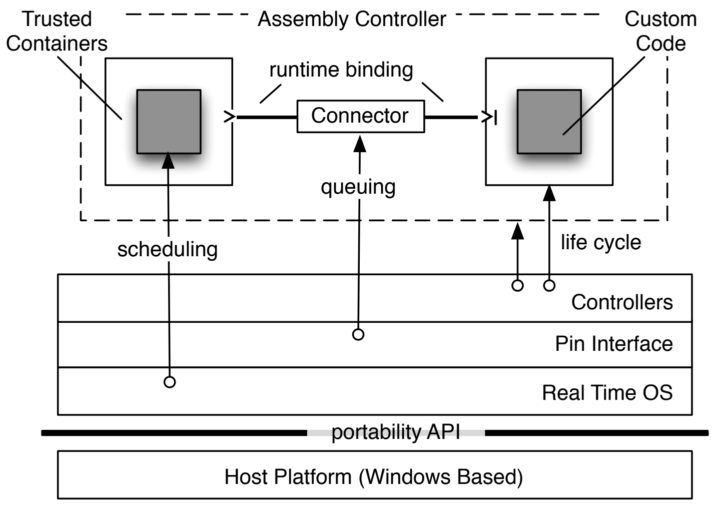


Figure 7.1: Pin Architecture (Logical View)

Working from the bottom to top of Figure 7.1:

- *Portability API.* Although the currently released version of Pin (in the PSK) supports only Windows platforms (WindowsNT, WindowsXP, WindowsVista), it can be easily re–hosted to a conventional Unix variant. The host computer need not be a desktop–class machine, and indeed an earlier (and not currently supported) version of Pin was hosted on WindowsCE in anticipation of its use in embedded environments. However, re–hosting Pin to an embedded or highly resource–constrained platform will likely require more effort than for desktops, and this has not yet been attempted.

- *Real Time OS.* Pin provides its own real–time operating system (RTOS, described in more detail in §7.4). RTOS provides a conventional preemptive priority–based scheduler with 128 thread priorities, and in most respects adheres to the POSIX specifications for real–time extensions,[1] with a few exceptions to support various UML–extensions used by PCL.

[1]POSIX.1b, Real-time extensions (IEEE Std 1003.1b-1993)

- *Pin Interface.* The Pin interface layer provides application–level libraries for building and executing Pin components and assemblies of components. The application programming interfaces (API) of the Pin component model are implemented in this layer.

- *Controllers.* The Controller layer is not, strictly speaking, part of Pin, but rather it is the design pattern adopted by the PSK to build Pin applications. Controllers are generated from the specification of top–level PCL assemblies; they are "main" programs responsible for managing the life cycle of Pin components and assemblies.

The dashed–line box labeled "Assembly Controller" in Figure 7.1 highlights a few additional points of interest about Pin:

- *Components and Trusted Containers.* Pin components are formed from a composite of *trusted* containers and *custom* code. Here "trusted" refers to containers that are provided in the Pin distribution, which have known behaviors that can be exploited by code–generators and reasoning frameworks; the λ-SS container is an example of the latter. However, the Pin interface layer does permit the development and use of untrusted containers. Custom code executes component type–specific behavior, possibly but not necessarily generated by PCL.

- *Runtime Composition.* Pin components use run–time binding to form connections with other components. Although the connection topology of Pin assemblies can change during runtime, PCL supports static topologies only. This PCL restriction is not "principled" but expedient; it reflects the restrictions imposed by the two reasoning frameworks ($\lambda *$ and ComFoRT) developed in tandem with Pin.

- *Restricted Range of Interaction (Connector) Mechanisms.* Pin connector semantics for asychronous and synchronous connections are implemented in the Pin Interface layer, but the basic queueing mechanism for connectors is provided by RTOS. Pin does *not* make it easy to add new connector types; this is an intentional design decision rather than a limitation in the component model.

7.3 Pin Component Model

The simple PCL specification shown in Example 7.1 (the PCL equivalent of "Hello, World") and the generated code that implements the assembly will be used to illustrate key ideas about Pin component and its correspondence to PCL. The assembly consists of one component instance of **component** Tap and one instance each of **service** Keyboard and **service** Console. If Keyboard were connected directly to Console, keystrokes would appear directly on the console. In this assembly, Tap listens in on the conversation by interposing itself between keyboard and console.

Example 7.1: Simple PCL Assembly.

```
component Tap (string p) {
  sink asynch in(consume string s);
  source asynch out(produce string s);
  threaded react Work (in, out) {
    start     -> listen {}
    listen    -> send {trigger ^in; action ^out(in.s);}
    send      -> listen {trigger $out; action $in();}}
}
assembly Tapped () (Rtos) {
  assume { Rtos:Keyboard keyb(); Rtos:Console outp();}

  Tap tap(".");

  keyb:keyed  ~> tap:in;
  tap:out     ~> outp:write
  expose {}
}

Rtos env()
  {Rtos:Keyboard kb();Rtos:Console cns();};

Tapped tapped()
  {Tapped:keyb = env:kb;Tapped:outp = env:cns;};
```

7.3.1 Components and Containers

Figure 7.2 summarizes the structure of Pin components. A Pin component instance is constructed at runtime from two separately and independently deployed binary constituents: a *container*, and *custom* code (called the "Nub"). All interactions between Nub and its external environment (i.e., the runtime environment, other Pin components) are mediated by the container Clients of a component see a component instance that supports three interfaces, PinComponent, ComponentInstance, and Container, each of which is delegated by a the container.

Different container types may be defined that introduce container–specific interaction policies. For example, the λ-SS reasoning framework comes bundled with a λ-SS container that mediates between Nub and the environment to manage Nub execution priority, execution budget and replenishment schedule. (See §8.5 pp. 156 for details about λ-SS.) Moreno demonstrated a technique based on C++ template metaprogramming to generate custom containers [125].

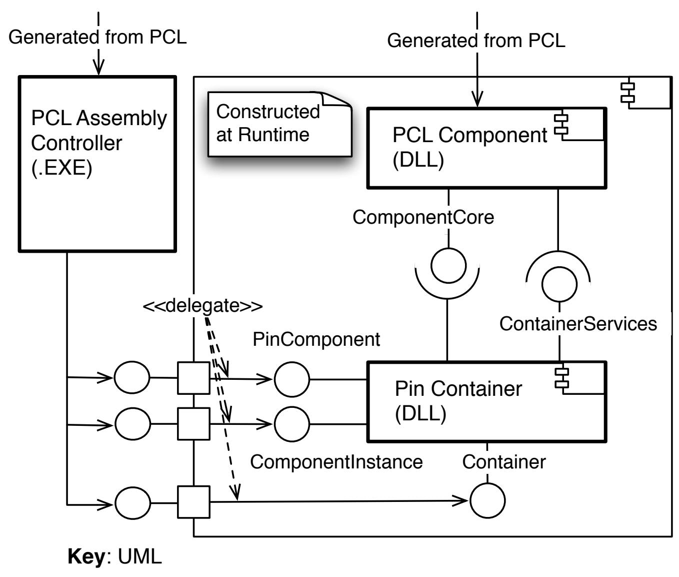


Figure 7.2: Pin Component Interfaces

There is no analogous notion of *assembly container*, although the controller pattern might be a reasonable place to start should such a concept be found useful.

Table 7.1 presents a brief digest of the ComponentCore and ContainerService interfaces:[2]

- *ComponentCore* defines "hooks" for managing the component lifecycle (create, delete, initialize) and a callback function invoked by the container when new messages arrive on a reaction event queue (TCommonHandler).
- *ContainerServices* is used by the Nub to initiate interactions on a synchronous (SendOutSourcePinWait) and asynchronous (SendOutSourcePin) source pin, and to generate an "end interaction event" on sinkPin event along with any **produce** parameters on that event.

Only the target of the operation is included in the signatures[3]; however, in conjunction with their brief descriptions in Table 7.1 and the examples that follow should be sufficient to demonstrate the main ideas.

Compiling the Pin specification in Example 7.1 will produce a number of C++ source files. One of these (Tap.cpp) implements Nub. Excerpts of

[2]The complete documentation of Pin interfaces is available in the online help feature of the PSK.

[3]The Pin API is defined in ANSI–C, so they simulate object–oriented concepts by passing the target object instance as the first parameter of a function call.

***Container* Interface**
PinComponent ∗ loadComponent (**char** ∗componentName, Container ∗) *Loads a component factory (a DLL) into memory.*
BOOL unloadComponent (PinComponent ∗) *Unloads a component factory (a DLL).*
***ContainerServices* Interface**
BOOL sendOutSourcePin (Reaction ∗...) *Sends an asynchronous message out a component instance's source pin.*
BOOL sendOutSourcePinWait (Reaction ∗...) *Sends a synchronous message out a component instance's source pin.*
BOOL sendReply (Reaction ∗...) *Sends a reply to a received synchronous message.*
IpcPort_Message ∗ parseUserMessage (Reaction ∗...) *Parses a PIN_MSG received by a reaction handler.*
int notifyController (ComponentInstance ∗...) *Sends a notification to the controller.*

***ComponentCore* Interface**
BOOL createComponentInstance (ComponentInstance ∗...) *Allocate and Initialize the internal state of a component instance.*
BOOL deleteComponentInstance (ComponentInstance ∗...) *Deletes the internal state of an instance being deleted.*
void reactionInitialize (Reaction ∗...) *Initialization hook for the reaction of a component instance.*
void reactionTerminating (Reaction ∗...) *Termination hook for the reaction of a component instance.*
typedef ReactionStatus TCommonHandler (Reaction ∗...) *Callback invoked by container when events arrive to a reaction.*

Table 7.1: Container, ContainerService and ComponentCore Interfaces

this implementation are discussed in a running illustration beginning with Example 7.2:

Example 7.2: Tap Component Implementation.

```
57 // earlier code omitted...
58 ReactionInfo reactionsInfo[ 1 ] =
59 {
60   {
61     REACTION_Work_NUM_SOURCES,   //number of source pins
62     REACTION_Work_NUM_SINKS,   // number of sink pins
63     REACTION_Work_SOURCE_ARRAY, //ordered array of pin nums
64     REACTION_Work_SINK_ARRAY, //ordered array of pin nums
65     PIN_TRUE, //is threaded
66     DEFAULT_QUEUE_SIZE,
67     REACTION_DEFAULT_PRIORITY,
68     IPCPORT_WAITFOREVER,         // default timeout
69     FALSE,          // default measurement flag
70     REACTION_Work_PIN_GENERAL_EVENT_HANDER // for all events
71   }
72 };
73 // later code omitted...
```

Lines 58–72 show a component defining an array of ReactionInfo structures; each of these corresponds to a "Reaction *" parameter in the ContainerServices and ComponentCore interfaces. The last element of the structure is a pointer to the event handler for the reaction, and these handlers implement the PinChart defined for that reaction (in this case, for the Tap:Work reaction). The Pin Interface defines a number of structures that are analogous to ReactionInfo; these interfaces allow Pin to provide a "poor man's" form of run–time introspection of components and assemblies.

Example 7.2 Tap Component Implementation (Cont.)

```
85 // earlier code omitted...
86 BOOL createComponentInstance(ComponentInstance *pInstance,
87     void *State, unsigned int SizeOfState)
88 {
89   int i = NUM_SOURCE_PINS;
90   INSTANCE_DATA *pData;
91   COMPONENT_Tap_ARGS *pp = (COMPONENT_Tap_ARGS *)State;
92   if (State != NULL) {
93     pInstance->pInstanceData =
94       Malloc( sizeof (INSTANCE_DATA) );
95     if (pInstance->pInstanceData == NULL) { return FALSE; }
96 
97     pData = (INSTANCE_DATA *)pInstance->pInstanceData;
98 
99     memcpy(&(pData->args),
100            State, sizeof (COMPONENT_Tap_ARGS) );
101       pData->synchReplyQ[0] =
102          (QUEUE) QUEUE_new(pData->args.numConnectedSources[0]);
103       // --- initialize component local variables ---
104       // --- initialize reaction local variables ---
105       pData->Work_CURRENT_STATE = 0;
106    }
107
108    return PIN_TRUE;
109 }
110 // later code omitted...
```

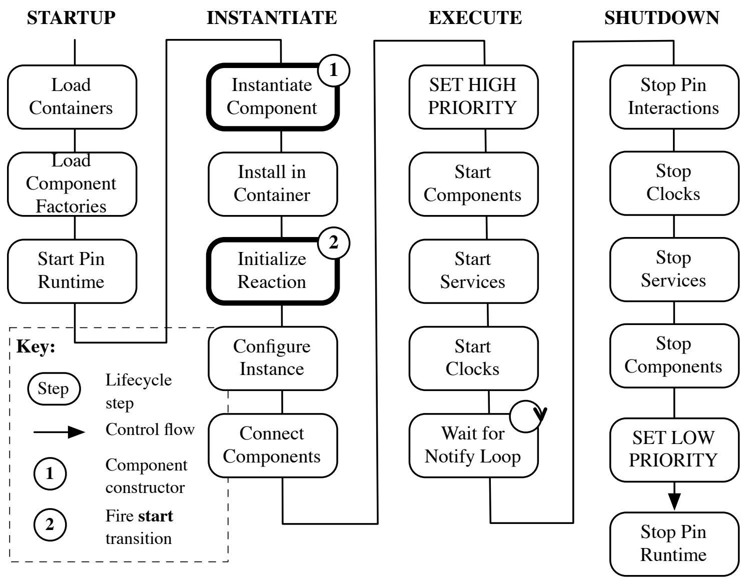
Lines 85–109 show the implementation of Tap's constructor, *createComponentInstance*. The Nub is passed a pointer to a *ComponentInstance* structure, one member of which is the array *reactionsInfo* of reactions defined earlier. In Pin, all heap memory is allocated at component instantiation; the call to *Malloc* on line 94 does this for Tap's **string** instance parameter, and had there been **string** variables defined by Work they would have been allocated between lines 104–105.

Example 7.2 Tap Component Implementation (Cont.)

```
168 // earlier code omitted...
169    case 1: // listen
170       if (pMessage->sinkPin == 0 /* ^in */ ) {
171          __marshDx = 0;
172          strncpy(
173             (char *) &(_THIS_->MessageOut.data[__marshDx]),
174             ((char *)(&pMessage->data[0])) /* in.s */,
175             PIN_MAX_STRING_LENGTH);
176          __marshDx += PIN_MAX_STRING_LENGTH;
177          //--- Call asynchronous IPC mechanism ---//
178          if (!sendOutSourcePin( /* ^out() */
179               pReaction, 0,
180               &(_THIS_->MessageOut),
181               (short) (sizeof(_THIS_->MessageOut.data)),
182               IPCPORT_WAITFOREVER,
183               &messageInfo))
184          {
185             // error handling, abort reaction
186          }
187          _THIS_->Work_CURRENT_STATE = 2;
188 // later code omitted...
```

Finally, the Nub invokes the *sendOutSourcePin* from its reaction handlers to initiate interactions on its source pins. Lines 168–187 shows the code corresponding to the listen -> send transition from Example 7.1. Had the source pin been synchronous, *sendOutSourcePinWait* would have been used

Figure 7.3: Assembly Controller Pattern



on line 178, and the reaction would have blocked until the reaction of the corresponding (connected) component had completed.

Therefore, the *sendOutSourcePin* and *sendOutSourcePinWait* correspond to ^sourcePin(...) "begin interaction" events, and the code immediately following these calls corresponds to **trigger** $sourcePin "end interaction" transitions. Although PCL semantics regards $sourcePin events as "first class" events that arrive on the message queue, the implementation handles these as conventional return values of a *remote function call.*

7.3.2 Assembly Controllers

Another C++ file generated by compiling the Pin specification in Example 7.1 is the assembly controller *tapped.cpp.* The controller corresponds to the instantiation of the Tapped assembly on line 22 of Example 7.1. Controllers use the *Container* interface summarized in Table 7.1 and the *PinComponent* and *ComponentInstance* interfaces summarized in Table 7.2 to implement the component and assembly lifecycle summarized in Figure 7.3. The code fragments beginning with Example 7.3 show how this is accomplished.

Example 7.3: Tapped Assembly Implementation.

```
46 // earlier code omitted...
47 // load containers used in this assembly...
48   if (!(pStandardContainer =
49     loadContainer("StandardContainer.dll")))
```

***PinComponent* Interface**
unsigned int getNumSourcePins (PinComponent ∗) *Gets the number of component source pins.*
unsigned int getNumSinkPins (PinComponent ∗...) *Gets the number of component sink pins.*
unsigned int getNumReactions (PinComponent ∗...) *Gets the number of component reactions.*
SourcePinInfo ∗getSourcePinInfo (PinComponent ∗...) *Gets information about a source pin.*
SinkPinInfo ∗getSinkPinInfo (PinComponent ∗...) *Gets information about a sink pin.*
ReactionInfo ∗getReactionInfo (PinComponent ∗...) *Gets information about a reaction.*
ComponentInstance ∗createInstance (PinComponent ∗...) *Creates a Pin component instance.*

***ComponentInstance* Interface**
BOOL configureInstance (ComponentInstance ∗) *Configures a newly created instance of a component.*
BOOL configureContainer (ComponentInstance ∗...) *Configures the component container for an instance.*
BOOL setReactionPriority (ComponentInstance ∗...) *Sets the priority of a component instance's reaction.*
BOOL setReactionQueueLength (ComponentInstance ∗...) *Sets message queue length of instance reaction.*
BOOL setReactionTimeout (ComponentInstance ∗...) *Sets the timeout of instances reaction.*
BOOL setMeasureExecutionTime (ComponentInstance ∗...) *Enables or disables measurement trace events.*
BOOL startInstance (ComponentInstance ∗...) *Starts an instance of a component.*
BOOL stopInstance (ComponentInstance ∗...) *Stops an instance of a component.*
BOOL deleteInstance (ComponentInstance ∗...) *Deletes an instance of a component.*

Table 7.2: PinComponent and ComponentInstance Interfaces

```
50      {
51        Printf("Failed␣to␣load␣standard␣container\n");
52        return EXIT_FAILURE;
53      }
54
55   // load factories for components used in this assembly...
56      factories[0] =
57        loadComponent("Tap.dll", pStandardContainer);
58      if (factories[0] == NULL) {
59        Printf("Failed␣to␣Load␣Tap\n");
60        return EXIT_FAILURE;
61      }
62      else {
63        Printf("Tap␣Load␣Successful\n");
64      }
65   // later code omitted...
```

Lines 46–65 illustrate the first two steps in the assembly lifecycle, loading containers and component factories. Line 57 associates the component factory for **component** Tap with the standard Pin container; all instances of Tap will be managed by a single instance of this standard container. Had we wished to have Tap instances execute in the λ-SS container, we would have specified that as an annotation in the PCL specification. In that case, the controller would still load the standard container (line 48), but then would have followed by loading the λ-SS container (also a DLL) *into* the standard container. In this way, Pin allows containers to be "nested," possibly in order to implement interaction constraints required by several reasoning frameworks.

Example 7.3 Tapped Assembly Implementation (Cont.)

```
132  // earlier code omitted...
133  strncpy(Tap_args.s, ".", PIN_MAX_STRING_LENGTH);
134     Tap_args.numConnectedSources[0] = 0;
135     if ((instances[2] = createInstance(
136        factories[0], "tap",
137       &Tap_args,
138        sizeof (Tap_args) )) != NULL)
139     {
140        Printf("tap␣instantiated\n");
141     } else {
142        Printf("tap␣FAILED␣TO␣BE␣instantiated\n");
143        return EXIT_FAILURE;
144     }
145     if (!configureContainer(instances[2], NULL)) {
146        Printf("%s␣container␣configuration␣FAILED\n",
147           instances[2]->uniqueName);
148     }
```

```
149 // later code omitted...
```

Lines 132–147 instantiate the Tap component. The call to *configureContainer* permits different instances managed by the container to have distinct container–specific properties. Note that the "0" on the right hand side of the assignment statement on line 134 denotes the integer that identifies a connected source pin.

Example 7.3 Tapped Assembly Implementation (Cont.)

```
187 // earlier code omitted...
188 if (sourceAddSinkPin(instances[1], 0,
189          instances[2]->uniqueName, 0) == FALSE)
190       {
191          Printf("keyb:keyed ~> tap:in Failed\n");
192          return EXIT_FAILURE;
193       }
194
195       if (sourceAddSinkPin(instances[2], 0,
196          instances[0]->uniqueName, 0) == FALSE)
197       {
198          Printf("tap:out ~> outp:write Failed\n");
199          return EXIT_FAILURE;
200       }
201 // later code omitted...
```

Lines 187–199 show the use of *sourceAddSinkPin* to build the assembly topology. The first invocation implements the keyb:keyed ~>tap:in connection, while the second implements the tap:out ~>outp:write connection. As can be seen, the decision to enforce assemblies to static topologies is one that could be revisited should a reasoning framework for e.g., mobile computing be developed in the future. The remaining steps in the assembly lifecycle are implemented analogously to the examples above, and no further details are required to understand the basic ideas of the Pin component model.

7.4 Pin Runtime Environment

The case studies reported in Chapter 9 used an earlier version of Pin than the one available in the PSK. That version used a commercial real–time extension of Windows (RTX[4]) as the Real–Time OS layer (refer to Figure 7.1). Because this restricted the availability of Pin to users of this commercial product, we developed a non–proprietary replacement. Figure 7.4 adds detail to the bottom two layers of the Pin architecture (*Real Time OS* (RTOS) and *Pin Interface*) to show the essential elements of this replacement.

[4]See `http://www.advantech.com/`, last accessed 26 August, 2010. An early technical description of this product was published 1997 [27].

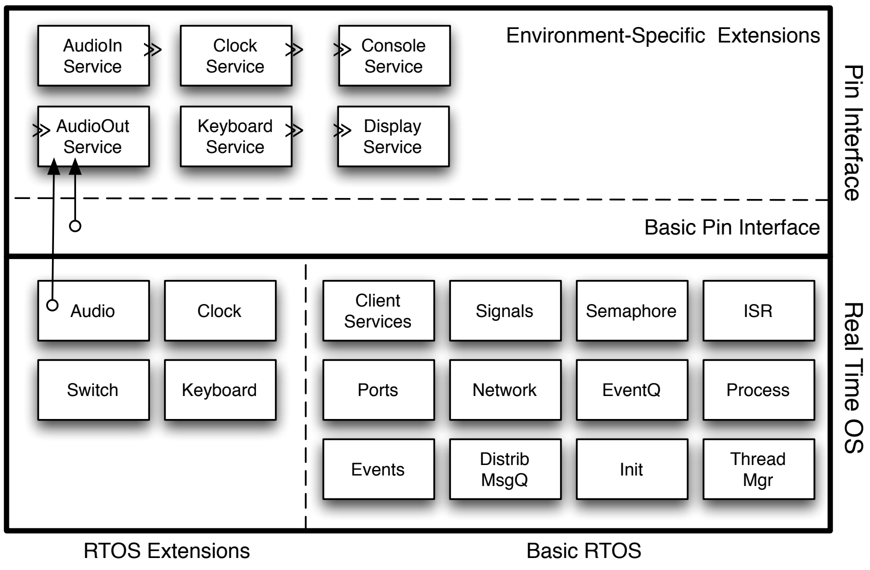


Figure 7.4: Pin Kernel (Layered View)

Basic and Extended RTOS

The RTOS layer is logically partitioned into a Basic RTOS and extensions:

- The Basic RTOS provides many of the essential services of a real–time operating system, including not just thread scheduling but networking, distributed message queues, signal handling, etc. Basic RTOS is the non–proprietary replacement to RTX.
- The RTOS extensions layer includes device drivers and other host platform mechanisms that are used to implement PCL *services*. For example, the *Switch* extension to RTOS was used to implement the custom switch device used in the first stage of the substation control case study (§9.2.3, pp. 180).

RTOS is implemented as a single executable Windows process that uses a small number of real–time *Windows* threads. One Windows thread executes the RTOS thread manager (*ThreadMgr* in Figure 7.4), which itself implements a fixed–priority scheduler, with a 10 ms scheduling quantum, for an arbitrary number of *Pin threads*, each of which corresponds to a *threaded reaction*.

Other real–time Windows threads are used by RTOS to implement networking services and timer services. RTOS extensions that require interacting with platform devices (such as audio drivers) also use a real–time Windows thread. This imposes a practical limitation on the number of such device extensions that can be added to RTOS, because Windows provides only a small number of real–time priorities, and because each such Windows thread becomes a potential source of scheduling interference for Pin's scheduler.

It is worth mentioning that RTOS does not provide a device driver architecture that permits third–party plugins. As a consequence, extending RTOS requires the equivalent of a "kernel mod." This is a limitation in the implementation of RTOS that could of course be remedied.

Basic and Extended Pin Interface

The *Pin Interface* layer is likewise partitioned into a basic and extended form, with the *Basic Pin Interface* implementing the component model described earlier in §7.3, while the *Environment Specific Extensions* provides the various PCL **environment** specifications that make use of the **service** extensions alluded to above.

7.5 Summary of Pin

Pin does a reasonably good job of meeting its original design objectives.

Objective 1: Provide only those features needed for predictability by construction.

It is difficult to establish that Pin has *only* those features required by the Seam. However, the case studies in Chapter 9 demonstrate that it has *at least* those features required by several non–trivial demonstrations of predictability by construction.

Moreover, the Pin component model, as specified by *Pin Interfaces* surveyed in §7.3, is quite compact, providing primitive but flexible mechanisms for managing the runtime lifecycle of components and assemblies. It is difficult to identify any element of *Pin Interfaces* that could be eliminated without adversely affecting one the above mentioned case studies—although this is not a definitive proof.

Overall, Pin satisfies the "spirit" of the design objective.

Objective 2: Provide a *simple* programming model and an execution model.

The Pin programming model is remarkably simple, even if its API's do not make the development of Pin components or assemblies a matter of just a few lines of code.

This essential simplicity is demonstrated by the straightforward mapping of PinCharts to Nubs; custom code essentially runs as a callback of an event dispatching loop provided for the component developer by the standard Pin component container.

The component developer has no visibility to the environment (to other components or to the runtime) beyond that provided by its container, so external code dependencies are sharply reduced, which of course leads to simpler programs.[5] Pin also enforces a model of "pure assembly," wherein components are integrated by declaratively composing larger systems from components by connecting their pins ('$\rightsquigarrow$>').

Overall, Pin satisfies this design objective.

Objective 3: Provide multiple ways of enforcing design constraints.

The Pin architecture and component model provide several locations for enforcing constraints:

- Containers can enforce runtime constraints (see λ-SS descriptions in Chapters 8 and 9).
- Assemblies can enforce runtime constraints as well, though these need to be programmed into a code generator.
- The Pin Interface can be extended in various ways; a notable example is the introduction of UML change events and time events as peers to Pin events.
- The Pin RTOS can also be extended, for example to include a different scheduling discipline such as "earliest–deadline first," though these need to be programmed as kernel extensions to RTOS.

Overall, Pin satisfies this design objective.

Objective 4: Be adaptable to new target environments.

Pin is adaptable *in principle* because it is relatively small (objective 1), simple (objective 2) and provides several loci for constraint enforcement (objective 3).

However, *in practice* Pin is not quite so easy to work with. The choice of C rather than C++ as an implementation language for Pin means that the library interfaces and their implementations must "mimic" useful object–oriented programming concepts, which leads to a certain level of awkwardness. In retrospect, C++ might have been a better choice, even though C++ is regarded somewhat skeptically by developers of hard real–time systems.

[5]Of course, component developers may use PCL verbatim code to violate this isolation if they choose.)

It has already been observed that extending RTOS requires “kernel mods” and therefore deep familiarity with the RTOS implementation.

Overall, Pin fell slightly (but not decisively) short on this design objective.

Objective 5: Be freely distributable and installable on conventional PC desktops.

Pin is available in the PSK, and can be used to build interesting hard, firm and soft real–time applications on conventional desktop computers. Pin satisfies this design objective.

Chapter 8

Reasoning Frameworks

Reasoning frameworks are the semantic extension points of a PECT. They permit automated analysis of designs, and automated prediction of component and assembly runtime behavior. Designers who use PECT will decide which reasoning frameworks to use, and therefore which design constraints to satisfy, depending on the kinds of behavior they wish to make predictable by construction. The $\lambda*$ reasoning framework can be used to reason about (hence predict) latency, schedulability and other "real–time" system properties; the ComFoRT reasoning framework can be used to reason about (hence predict, and in some cases provably verify) patterns of behavior over time.

This chapter introduces the overall structure of a reasoning framework in §8.1, and then provides an overview of the $\lambda*$ reasoning framework in §8.2, and ComFoRT reasoning framework in §8.6.

8.1 The Structure of Reasoning Frameworks

The top–level, logical structure of a reasoning framework is shown in Figure 8.1. Its three major internal components are an interpretation, model representation, and decision procedure:

- The *interpretation* checks that a design is well–formed to the reasoning framework, and if so generates the corresponding *behavioral model* of the component.
- The (semi) *decision procedure* uses the generated model to answer questions posed on the design, for example to predict its worst–case latency.

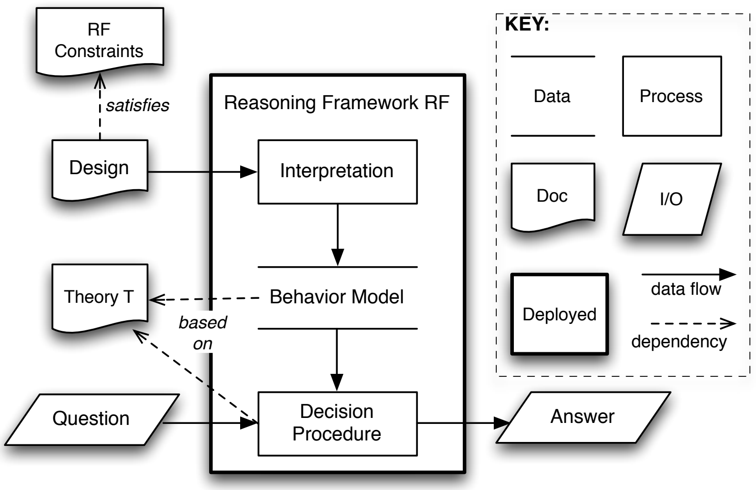


Figure 8.1: Reasoning Framework Structure

The $\lambda *$ and ComFoRT reasoning frameworks are described in a way that strongly reflects their logical structure:

- Theory: What are the key terms, formulas, relationships, etc., that define the behavioral theory?
- Constraints: What restrictions are imposed to ensure that theory assumptions are satisfied?
- Decision Procedure: How is analysis automated?

Two points are worth noting. First, although a reasoning framework is an independently–deployable unit of semantic extension in a PECT, and as such it is a kind of component in its own right, this chapter is not concerned with reasoning framework interfaces, or what might be regarded as the component model *of* the PECT (in contrast to the component model *supported by* the PECT, i.e., Pin). Such details, while important in a practical sense, pose no special challenges.

Second, many of the detail presented in this chapter have been culled from existing reports and papers. For $\lambda *$ these include: *Overview of the Lamba-Star Reasoning Framework* [60] and *Performance Property Theories for Predictable Assembly from Certifiable Components* [67]. For ComFoRT these include: *The ComFoRT Reasoning Framework* [34], *Overview of ComFoRT: A Model Checking Reasoning Framework* [81], and *Certified Binaries for Software Components* [32].

8.2 $\lambda *$ Reasoning Framework

When the correctness of the system requires not only producing the right result but producing it at the right time, the system is called a real-time system. Klein and colleagues present a framework to describe and reason about realtime systems [89].

$\lambda *$ is both a reasoning framework (in that it is packaged as a PECT component as shown in Figure 8.1) as well as a *suite* of reasoning frameworks (in that it contains several distinct theories, interpretations and decision procedures). This suite of reasoning frameworks evolved to meet the needs of industrial case studies described in Chapter 9, and under the operation of a reasoning framework design process called "co–refinement" described in Chapter 10.

8.2.1 $\lambda *$ Preliminaries

The timing requirements in real-time systems are expressed relative to an *event*. An event is some sort of stimulus to which the system has to respond. An event can be environmental, such as the push of a button or data arriving from the network, or it can be timed, that is, generated at specific times or after a given amount of time elapses. Events can also be classified according to their arrival pattern. In this dimension, events can be *periodic* if the time between arrivals is constant or *aperiodic* when it is not. The computation that must be performed upon the arrival of an event is called the *response.* The amount of time it takes to complete the response to an event since the arrival of that event is called *response time* or *latency.*

Timing requirements, then, can be expressed as requirements on the response time. Furthermore, when a requirement imposes an upper bound on a response time, the upper bound is referred to as a *deadline.*

Timing requirements are usually classified as:

Hard: Deadlines must be met at all times because failing to do so has severe consequences. For instance, reacting to a critical overcurrent condition to prevent damage on an electric motor has a hard deadline.

Firm: Deadlines have to be met most times but occasionally missing a deadline does not have severe consequences. In addition, once the response misses the firm deadline, there is no value in completing it. For example, in live video streaming, dropping a video frame once in a while is not a big problem.

Soft: The value of responding to an event gradually decreases past the deadline, which is usually referred to as a soft deadline. For example, refreshing the display of some instrument in a panel may have a soft deadline of 30ms; however, should the refresh occasionally take longer, it will not cause a failure.

There are three main contributors to the latency of a response that have to be accounted for to predict it:

1. *Execution* is the amount of time that the response takes to perform its computation without any interference from other tasks in the system.
2. *Preemption* is the amount of time that the response is not able to execute because the processor is being used by a higher priority task.
3. *Blocking* is the amount of time that the response is waiting—and consequently, not executing—for a shared resource to become available.

8.2.2 $\lambda*$ Common Theory

The performance model used in $\lambda*$ is based on an analysis technique developed by Gonzalez Harbour and colleagues [62]. This technique, which we refer to as “HKL,” works for systems that comprise a set of tasks that execute concurrently. The work carried out by each task is represented by a sequence of *subtasks* that execute serially. Each subtask represents a portion of the computation that executes at a fixed priority level, does not voluntarily yield the processor, and does not access resources for which it could block. In this way, the subtask does not introduce the opportunity of a scheduling point—a point in time at which the scheduler makes a scheduling decision—in the middle of its execution. Changing priority level, acquiring and releasing shared resources, or entering and leaving critical sections is done at the boundary between subtasks.

The main attributes of a subtask are its execution time and priority level.

8.2.3 λ∗ Common Behavioral Model

A *metamodel* for λ∗ is shown in Figure 8.2. The three classes at the top (i.e., *PerformanceModel*, *Task*, and *Subtask*) directly correspond to HKL: a performance model defines a collection of tasks, and each task in turn has a collection of subtasks. The metamodel is more general than HKL, for example to model tasks that are not periodic and that have non-constant execution times.

The characterization of event interarrivals is done in two different ways depending on whether the task is a periodic task (*PeriodicTask*) or an aperiodic task (*AperiodicTask*). In the former, the period attribute in the derived class PeriodicTask represents the period of the task or the event that triggers the task. For the latter, the event interarrival distribution is modeled with an instance of a Distribution, an abstract class representing different kinds of statistical distributions.

The two most important attributes for the subtasks are the priority—a proper attribute in the class—and the execution time distribution, represented with an instance of the *Distribution* class as well. The *SSTask* represents an aperiodic task that is scheduled by a sporadic server [154].

Not all the concepts in the metamodel are used by each of the reasoning frameworks in λ∗. For example, unbounded statistical distributions cannot be used in worst-case latency prediction (for reasons described later). Therefore, the metamodel is more general than it needs to be for any particular reasoning framework, but it can be readily adapted to future needs. Moreno has also developed a more elaborate intermediate representation for performance models to simplify integration with various performance analysis tools [126, 127]. For present purposes, however, the metamodel shown here is sufficient.

8.2.4 λ∗ Common Constraints

The following are basic constraints of the λ∗ performance reasoning frameworks:

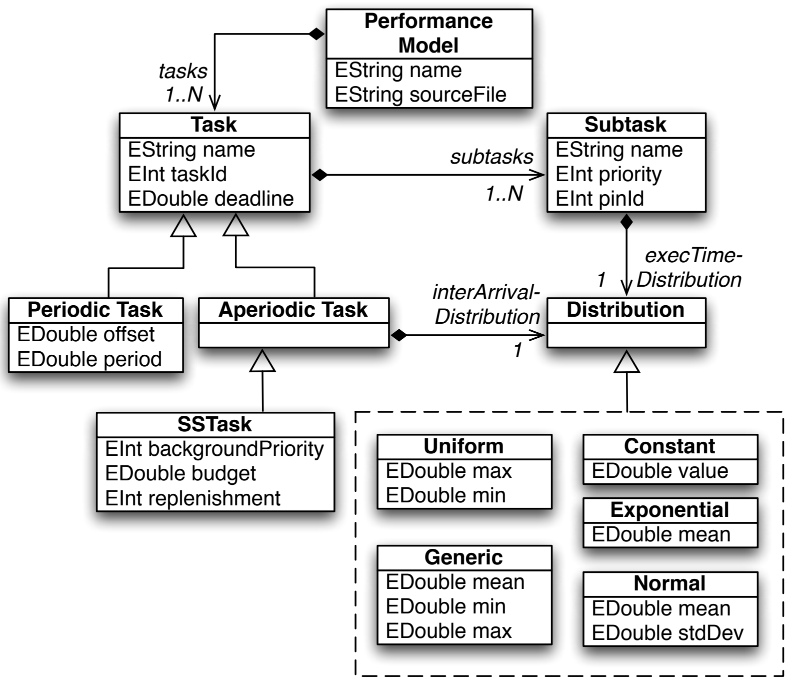


Figure 8.2: $\lambda*$ Metamodel

1. The assembly executes in a single processing unit.
2. Tasks are scheduled by preemptive fixed-priority scheduling.
3. Components complete their work first and then interact with other components.
4. Each sink pin event produces interactions on all source pins in its reaction.
5. There are no loops in the connection graph of components.
6. Components do not suspend themselves during their execution. That means that they do not yield the CPU by sleeping or invoking operations that could block, such as I/O.
7. Priority of reactions must conform to the highest locker protocol (a.k.a. priority ceiling emulation).
8. If the computations corresponding to two sink pins within the same response can be ready to execute at the same time, they must have different priorities.

8.3 λ-WBA Reasoning Framework

λ-WBA stands for "latency prediction, for worst case, with blocking and asynchronous interactions permitted."

8.3.1 λ-WBA Questions and Answers

λ-WBA predicts the worst-case latency for the response to an event, and can be used to predict whether the response to an event will complete before its deadline. The computed value is an upper-bound for the latency because the worst-case component execution times, blocking, and preemption effects are assumed to occur simultaneously.

8.3.2 λ-WBA Theory

The underlying theory GRMA, and more specifically a technique for analyzing the schedulability of a set of tasks with varying priorities [62].

According to this theory, each task or response is composed of a sequence of subtasks that have an associated execution time and priority level. This makes it possible to analyze situations in which the response to an event is composed of several computations executing at different priorities, which is the kind of response found in a component-based system, where each component carries out a portion of the response and can execute at its own priority level if it has its own thread of execution. In this case, the assignment of priorities can be based on deadlines or the semantic importance of the component [62]. In addition, the theory can also account for the effect of the synchronization between responses when using a priority-based synchronization protocol.

The most complex aspect of this theory involves computing the preemption effect. In regular rate monotonic analysis, each task executes at a fixed priority, so the set of tasks that can preempt the task being analyzed is constant, and they preempt every time. With the varying priorities method, priorities can vary throughout the execution of both the task being analyzed and the other tasks in the system. Therefore, the set of tasks that can preempt the task being analyzed is not constant. The algorithm for computing the worst-case latency for tasks with varying priorities classifies the other tasks in the system based on their ability to preempt each of the subtasks in the task being analyzed.

First, the task being analyzed is transformed to canonical form, a special form of the task wherein the priority of consecutive subtasks does not decrease and that for worst-case analysis is equivalent to the original task. If P is the priority of the subtask being analyzed, the rest of the tasks are classified in the following sets:

H: The set of tasks whose lowest priority is higher than or equal to P. These tasks preempt every time (when they execute at a priority equal to P, they are assumed to preempt, the worst effect).

HL: The set of tasks that start at a priority higher than or equal to P and then drop below P. These tasks preempt only once because when they arrive they are higher priority, but once they drop to low priority they cannot complete until the task being analyzed completes.

LH: The set of tasks that start at a priority lower than P and eventually rise over P. Only one of the tasks in this set can preempt since a task from this set can only preempt if it is already executing its high-priority segment when the subtask being analyzed starts; only one of them could be executing its high-priority segment at that time.

L: The set of tasks whose priority is always lower than P, and these never preempt. The algorithm then uses these sets in the process of computing the worst-case response time of the subtask being analyzed.

8.3.3 λ-WBA Constraints

1. Only lower bounded interarrival time distributions are allowed.
2. Only upper bounded execution time distributions are allowed.

Only these bounded distributions are supported because for worst-case analysis, the worst interarrival and execution times are used. If they were described by unbounded distributions, then the analysis would assume events arrive with infinite frequency and components have infinite execution time, which of course results in the impossibility to schedule the tasks.

8.3.4 λ-WBA Decision Procedure

λ-WBA uses MAST [61], a worst-case analysis tool that implements the procedure described above. The worst-case latency is computed by constructing the worst possible alignment of preemption and blocking effects for each task.

8.4 λ-ABA Reasoning Framework

λ-ABA stands for "latency prediction, for average case, with blocking and asynchronous interactions permitted."

8.4.1 λ-ABA Questions and Answers

During the execution of a system each job of a response (i.e., each instance of a response) can be affected differently by other tasks and thus exhibit different latencies. λ-ABA predicts the average latency for the response to an event by taking into account how different jobs are affected by other tasks. Instead of creating an alignment of tasks that causes the worst case for a response, as in λ-WBA, λ-ABA uses the alignment that naturally occurs from the arrival patterns and execution times of the different tasks.

8.4.2 λ-ABA Theory

λ-ABA is, essentially, a discrete event simulation of a collection of fixed–priority scheduled tasks. Nevertheless, λ-ABA shares many concepts with λ-WBA (from which it was, in fact, derived), which it uses to improve the performance of the λ-ABA discrete event simulation. In particular:

- The highest locker protocol is used to do priority-based task synchronization. In this way, the simulation does not need to handle synchronization specifically because it is handled by virtue of its simulation of fixed-priority preemptive scheduling.
- When all the tasks are periodic, it is possible to find a hyper-period, defined as the least common multiple (LCM) of the periods of all the tasks. Hyper-period analysis can be used only if the execution times of the components are constant or have a negligible variance; in other cases, looking at a single hyper-period would not allow for the varying execution times of a component to be sampled.

8.4.3 λ-ABA Constraints

λ-ABA does not introduce additional constraints beyond the λ∗ common constraints. However, the different evaluation procedures supported by λ-ABA introduce constraints due to tool limitations:

SIM–MAST	Only constant, uniform and exponential interarrival distributions allowed.
	Only constant, uniform and generic execution time distributions allowed. Sporadic servers are not allowed.
Extend	Only constant execution time allowed.
	Only constant, uniform, normal and exponential interarrival distributions allowed.
	Explicit deadline annotations not allowed.

8.4.4 λ-ABA Decision Procedure

λ-ABA uses a discrete–event simulation to make latency predictions by simulating the execution of the system, generating random event interarrival and execution times following the distributions specified in the model. While running the simulation, they keep track of best, average, and worst latency. Three different discrete-event simulation procedures are supported: SIM–MAST [113], QSIM [158], and Extend [91].

Rather than simulate the execution of a system, λ-ABA simulates the execution of the performance model. The advantage of doing this is that the simulator does not need to handle blocking (other than the resulting from fixed-priority scheduling), nor does it need to maintain a call stack. The latter is due to the fact that the interpretation has transformed all the calls—both synchronous and asynchronous—into plain sequences of subtasks. That is, the scheduling within a task has already been done by the interpretation, leaving less work for the simulation to do.

8.5 λ-SS Reasoning Framework

λ-SS stands for "latency prediction, sporadic server."

The sporadic server algorithm provides a good quality of service to high–priority aperiodic tasks while at the same time bounding the invasiveness of aperiodic tasks on hard real-time periodic tasks. When analyzing the hard real-time periodic part of the system, a component managed by a sporadic server container aperiodic can be regarded as a periodic task with execution time equal to the sporadic server budget, and period equal to sporadic server replenishment period. Assemblies containing components managed by sporadic servers may therefore be analyzed with λ-WBA and λ-ABA. A summary of the sporadic server algorithm can be found in §8.5.2.

8.5.1 λ-SS Questions and Answers

λ-SS predicts the average latency for the response to an event when the response is carried out by a component managed by a sproradic server container.

8.5.2 λ-SS Preliminaries

The sporadic server (SS) scheduling algorithm [154] was invented to solve the problem of protecting periodic events with hard deadlines from bursts of high priority stochastic events, while being able to accord high priority to processing stochastic events. The hallmark of a sporadic server is that it provides a periodic "virtual processor" within which aperiodic events can be processed and analyzed.

Implementations of the SS algorithm are based on the general premise that a server (a process within an operating system, or a thread of control within a process) that handles high priority stochastic events will execute at either one of two priorities: foreground (i.e., high) or background (i.e., low).

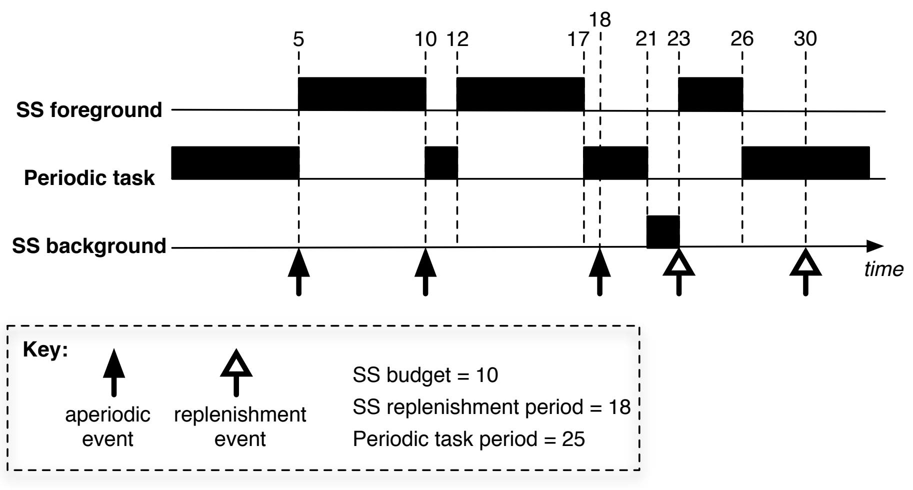


Figure 8.3: Example Sporadic Server Task Timeline

An aperiodic task will execute at foreground priority if the sporadic server has not exhausted its execution budget. If the SS has no remaining execution budget, then the aperiodic task is restricted to background priority. A SS that has been restricted to background priority is not restored to foreground priority until its execution budget is replenished.

Implementations of the SS algorithm can reside in the kernel (e.g., the thread scheduler) or in application space, which vary slightly in detail and effect. We chose to use an application–level server because it is substantially easier to implement, and because Pin's support of component containers makes this the natural choice. Figure 8.3 provides a task timeline to illustrate the basic ideas of the application–level algorithm, adapted from [63].

In this example, each aperiodic event takes 5 units of time to be serviced. The first two aperiodic requests arrive at $t = 5$ and $t = 12$ and are serviced immediately. This is because at $t = 5$, the execution budget of the SS is decreased by 5 units of time (as each event takes 5) still leaving a remaining execution budget of 5 units which permits the SS to execute at foreground priority. Also at $t = 5$, a replenishment event is scheduled for $t = 23$ (i.e., for the event occurring at 5 + the replenishment period 18). At $t = 12$, the execution budget is again reduced by 5 units of time and replenishment is scheduled for $t = 30$, and the SS can still execute at foreground priority. After $t = 12$, the execution budget is exhausted and when the next aperiodic event arrives at $t = 18$, the SS is restricted to execute at background priority. The additional execution budget for 5 units of time is replenished at the scheduled times of $t = 23$ and $t = 30$, respectively, for the first two requests thereby restoring the execution budget of the SS.

8.5.3 λ-SS Theory

λ-SS uses queueing theory to predict the latency of the response to a stochastic event. The expected or average latency $E[W]$ can be computed as the sum of the mean queueing time $E[Q]$ and the mean service time $E[Sa]$, as in Eq. 8.1:

$$E[W] = E[Q] + E[S_a] \tag{8.1}$$

Assuming exponentially distributed interarrival times, the mean wait time can be determined using the Pollacek-Khinchin formula [90], shown in Eq. 8.2:

$$E[Q] = (\frac{\rho}{1-\rho})(\frac{E[S_a^2]}{2E[S_a]}) \tag{8.2}$$

where $\rho = E[S_a]/E[T]$, in where T is the mean interarrival time of the aperiodic events.

Now, to compute the mean wait time, the mean service time $E[S_a]$ is needed. However, the service time of the aperiodic task in the sporadic server depends on the amount of high priority execution budget available during its execution. An important result presented by Hissam and colleagues [67] allows us to determine the mean service time from the point of view of the queue in a special case called *continuous background.*

While a sporadic server has execution budget, its aperiodic task can execute at high *foreground* priority. However, once the sporadic server budget is exhausted, its aperiodic task can execute only when the periodic tasks are not executing, at then only at low *background* priority. For example, if there is one periodic task with execution time 8ms and period 10ms, background execution time will be available for 2ms every 10ms. If the period of the periodic is reduced while keeping the same utilization, for instance execution time 0.2ms and period 1ms, background is available in smaller chunks but more often. If this is taken to the extreme of having an infinitesimal period for the periodic task, background becomes available for infinitesimal periods of time infinitely often, hence the name continuous background.

In continuous background, it looks as if the aperiodic task were executing in a slower processor, with a "degrade" service time that can be computed as Eq. 8.3:

$$\hat{S}_a = \frac{S_a}{1-U_p} \tag{8.3}$$

where U_p is the utilization of the periodic tasks.

What is equally important is that from the point of view of the events waiting to be serviced in the queue, the apparent service time of the aperiodic

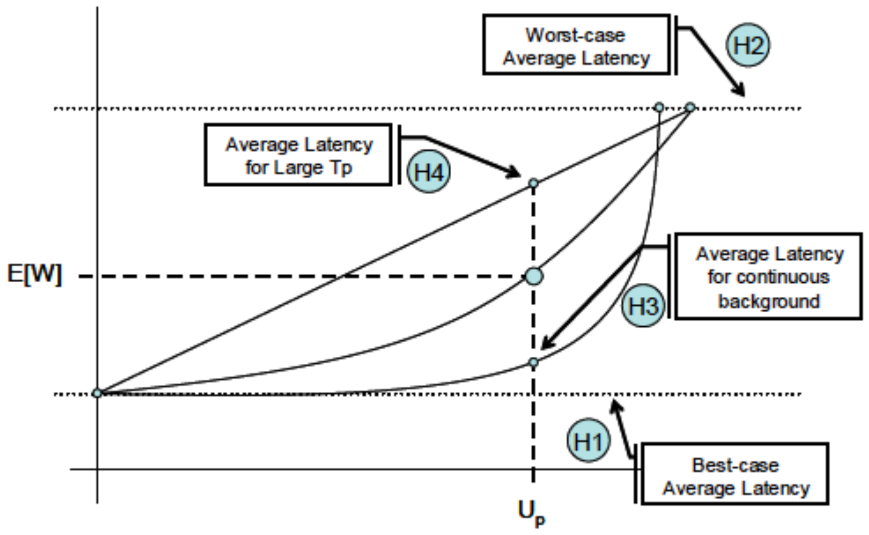


Figure 8.4: λ-SS Performance Envelope

task is always the one given by Eq. 8.3, *regardless* of whether the task is executed completely in the sporadic server at high priority, completely in background, or some hybrid of both. This is because even if it executes at high priority, the task waiting in the queue still has to wait for the backlog of periodic work to be worked off before it can be executed. (See [67] for the proof.)

With the first term $E[Q]$ of Eq. 8.1 computed, the rest of the theory is concerned with computing the second term, the mean service time $E[S_a]$. This requires computing the distribution of sporadic server, background, and hybrid arrivals, and also the distribution of high-priority execution in the latter. This is done drawing from results of queueing and renewal theory, the detailed derivations of which may be found in [67].

However, the essence of these theory is expressed by four heuristic equations that define a performance "envelope" for sporadic server tasks, depicted in Figure 8.4.

H1: The "no periodics" case. For a given aperiodic service time (S_a) and inter–arrival interval (T_a), the best-case average latency occurs when there are no periodics ($U_p = 0$). The latency for this case is predictable by Eq. 8.1.

H2: The "no background" case. For a given aperiodic service time and inter–arrival interval, the worst case average latency occurs when the periodic utilization is large enough so that aperiodics execute only within the sporadic server. The latency for this case is predictable by Eq. 8.1, where $S_a = T_{ss}$.

H3: The "continuous background" case, applies when $0 < U_p < 1 - S_{ss}/T_{ss}$.

- Given U_p, $E[Q]$ can be predicted very accurately by using Eq. 8.2 with $S_q = \hat{S}_a$.
- $E[S_s]$ can be approximated by a weighted average of S_a and $\hat{S}_a$, and therefore lies between those two extremes. As U_p gets larger, $\hat{S}_a$ approaches T_{ss} with diminishing room for background processing. Even though $E[Q]$ increases, $E[S_s]$ approaches S_a.

H4: The "large periodic" case, applies when $0 < U_p < 1 - \rho$. For very large periodic periods, average latency as a function of U_p approximates the convex combination of the no–periodics (NP) and no–background (NB) cases:

$$E[W] = (\frac{E[W_{NB}] - E[W_{NP}]}{1 - \rho})U_p + E[W_{NP}] \tag{8.4}$$

where

$$E[W_{NP}] = (\frac{\rho}{1-\rho})(\frac{[ES_a^2]}{2E[S_a]}) + E[S_a] \tag{8.5}$$

and where

$$E[W_{NB}] = (\frac{\hat{\rho}}{1-\hat{\rho}})\frac{E[\hat{S}_a^2]}{2E[\hat{S}_a]} + E[S_a] \tag{8.6}$$

If more precision than the bounds provided by this closed-formula evaluation procedure is required, a simulation-based evaluation procedure can be used.

8.5.4 λ-SS Constraints

λ-SS introduces several constraints beyond the common $\lambda*$ constraints:

- An assembly may have exactly one sporadic server task (the rest must be periodic).
- The interarrival distribution of the sporadic server task is exponential.
- The task managed by the sporadic server must have constant execution time.
- The sporadic server budget must be equal to its execution time.
- The background priority of the sporadic server is lower than that of any periodic.

8.5.5 λ-SS Decision Procedure

In addition to heuristic equations H1–H4, the aperiodic task in the sporadic server, along with the complete set of periodic tasks in the application (instead of being represented by a single utilization parameter), can be simulated in Extend. Simulation is used when a more precise estimate of $E[W]$ is desired between the H3–H4 bounds.

8.6 ComFoRT Reasoning Framework

In formal verification, a system is modeled mathematically, and its specification (also called a *claim* in model checking) is described in a formal language. When the behavior in a system model does not violate the behavior specified in a claim, the model *satisfies* the specification.

8.6.1 ComFoRT: Preliminaries

Model checking [42] is a fully automated form of formal verification that uses algorithms to check whether a system satisfies a desired claim through an exhaustive search of all possible executions of the system. The exhaustive nature of model checking renders the typical testing question of adequate coverage unnecessary. One advantage of restricting ourselves to finite-state systems is that verification can be performed automatically. Given sufficient resources, model checking always terminates with a yes or no answer.

The "Achilles Heal" of model checking is "state space explosion," where the size of finite models can grows too quickly, and becomes too large, for any reasonable definition of sufficient resource (time or memory). And unlike hardware systems, which exhibit genuinely finite behavior (and which are now routinely verified by model checking) software typically does not exhibit finite behavior. For software model checking, additional techniques are required to construct finite approximations of infinite-space behavior.

Edmund Clarke is fond of remarking that there are only two approaches to solving state space explosion: *abstraction* and *composition*:

- *Abstraction*: A smaller abstract system is constructed such that *if* a claim is satisfied by the abstract system it is also satisfied for the original system.
- *Composition*: The verification is partitioned into checks of individual modules while the global correctness of the composed system is established by constructing a proof outline that exploits the modular structure of the system.

An enormous body of literature has been developed for both of these approaches, and contemporary software model checkers can be characterized largely in terms of which specific abstraction and composition approaches used.

Abstraction

Abstraction is one of the principal complexity reduction techniques ([15, 39, 105] are just a few of many). Abstraction techniques reduce the state space by mapping the concrete set of states of the actual system to an abstract set of states that preserve the actual system's behavior. Abstractions are usually performed in an informal, manual manner and require considerable expertise. (The model checking case study in §9.3.3, pp. 197 is a good illustration of manual abstraction.)

Predicate abstraction [57] is one of the most popular and widely applied methods of automated abstraction. It maps concrete data types to abstract data types through predicates over the concrete data. ComFoRT combines predicate abstraction with "CEGAR," an automated technique of iterative abstraction refinement. Predicate abstraction and CEGAR are described in §8.6.3.

Composition

The main approach to compositional model checking use some form of "assume–guarantee" reasoning ([1, 40, 115] are just a few of many). An assume–guarantee scheme to demonstrate that a system composed of modules M_1 and M_2 satisfies the claim Φ proceeds by demonstrating (1) $M_1 \parallel \Phi_2 \models \Phi_1$ and (2) $M_2 \parallel \Phi_1 \models \Phi_2$ and from this concludes (3) $M1 \parallel M2 \models \Phi$. This approach uses the local claims Φ_1, Φ_2 as the constraining environments (assumptions) with regard to the behavior of M_2, M_1, taken in isolation from M_1, M_2, respectively. Assume–guarantee reasoning has been successful in verifying large hardware systems, but there are some major difficulties in its application to software systems, most notably in (1) decomposing the system (in this case, component boundaries may impose constraints) and (2) identifying suitable environment assumptions. As with manual abstraction, manual assume–guarantee reasoning requires considerable expertise.

Bobaru and colleagues have reported promising preliminary results in obtaining automated assume–guarantee reasoning in a predicate abstraction and CEGAR framework [21], but these results have not yet been incorporated into ComFoRT. Earlier experiments in automated assume–guarantee in ComFoRT achieved some success [36, 29, 35] but are currently not prominent features in the deployed reasoning framework, and are not further discussed.[1] Compositional reasoning techniques used within ComFoRT's CEGAR are briefly described in §8.6.3.

[1]These features are however available in the model checking engine used by ComFoRT, which is available as a standalone tool.

The ComFoRT interpretation maps PCL specifications to a combination of ANSI–C and the FSP [111] process algebra used by the model checking engine. Details on the interpretation can be found elsewhere [83, 81].

Note on Terminology. To maintain readability, the term "component" is used imprecisely in the following description. Strictly speaking, the ComFoRT model checker composes and verifies *processes*, in the process–algebraic sense. The ComFoRT interpretation takes care of the details of mapping component *reactions* to *processes*. This task is task made complex by the fact that components in PCL may define several reactions, and that reactions may be *threaded*, in which case they correspond quite naturally to processes, and *unthreaded*, in which case modeling them as processes produces an over–approximation of real concurrency which, though sound, carries with it the prospect for spurious counterexamples. Thus the ComFoRT interpretation uses information from PCL assembly specifications to construct a process–theoretic representation that is more faithful to real concurrency. However, distinguishing between assembly, component, reaction and process quickly becomes tedious without adding any clarity. Hence, for the remainder of this chapter I will adopt the user's perspective and regard all of the theoretical aspects of model checking in terms of components and assemblies. In this it might be useful to regard components as specifying exactly one threaded reaction each, although this is not a constraint imposed by ComFoRT.

8.6.2 ComFoRT: Questions and Answers

Some system behaviors are best expressed in terms of sequences of actions that occur over time. For example, we may wish to verify that "resources are never accessed *until* locked," and that "all locks are *eventually* released." Terms such as "until" and "eventually" introduce notions of temporal orderings of events, and indicate a need to verify behaviors that involve orderings of events in logical (as distinct from real) time.

A *temporal logic* can be used to express such behaviors [139], and is obtained by extending propositional logic with *modal* operators that captures the notion of "until," "eventually," and other temporal concepts. Model checking is a technique for automatically verifying that the system exhibits behaviors that are specified in temporal logic.

There are two broad classes of behavior that can be expressed in a temporal logic: *safety* (informally, a specified "bad" condition will never happen) and *liveness* (informally, a specified "good" condition will eventually happen). Alpern and Schneider demonstrate that a wide variety of system properties can be specified as a conjunction of a safety and liveness property [10].

8.6.3 ComFoRT: Theory

The basic theory components of ComFoRT are summarized in the following sections.

Kripke Structures and Search

In classical model checking [42], systems are modeled mathematically as state transition systems and claims are specified using temporal logic [139].

The model checking problem is succinctly expressed as a *search* problem [41] and is typically formulated in terms of Kripke structures:

Definition 8.1 (Software Model Checking Problem) *Given a temporal logic formula f, and a Kripke structure $M =< S, I, R, L >$, where S is a finite set of states, $M \subseteq S$ is a set of initial states, I is a transition relation $I \subseteq S \times S$ such that $\forall s \in S, \exists s' \in S \bullet (s, s') \in I$, and $L : S \longrightarrow 2^P$ is a labeling function (or semantic interpretation) for atomic propositions P; find all states $\bar{s} \in S$ such that $M, \bar{s} \models f$.*

In this definition, programs (components, assemblies) are represented as *finite* (a key term!) transition systems (i.e., the S, I, R components of M), and L represents propositions about each state in the transition system. Because there are a finite number of states, model checking reduces to a search problem, and $\bar{s}$ can be computed in finite time and space.

Temporal Logics

Temporal logic is used to define formulas that describe system behavior over time, where the propositions of the logic are behaviors of interest involving state information (current state or values of variables) or events. Temporal logic formulas combine such propositions with temporal operators to describe interesting patterns of propositions over time, for example: "a file is never written without having first been locked, and all locks on files are eventually released."

There are two main classes of temporal logic, computation tree logic (CTL) and linear temporal logic (LTL). Both temporal logics use $\bar{s}$ to assign different, and incomparable, definitions of semantic entailment $M \models f$, i.e., to determine whether the program modeled by the finite transition system in M satisfies the temporal logic claim f. Which is the better for software model checking is a longstanding debate. Ultimately, though, both LTL and CTL are too restrictive when verifying component-based systems; useful claims often involve patterns of communication among components that are dependent on the state of the participants.

For example, the Bluetooth L2CAP specification[2] asserts:

[2]Haartsen, J. Bluetooth Baseband Specification, Version 1.0. Published in 2003.

> "when an 'L2CAP_ConnectRsp' event is received in a 'W4_L2CAP_CONNECT_RSP' state, within one time unit, an 'L2CAP' process may send out an 'L2CAP_ConnectInd' event, disable the 'RTX' timer, and move to state 'CONFIG'."

As this example shows, both states (W4_L2CAP_CONNECT_RSP and CONFIG) and events (L2CAP_ConnectRsp and L2CAP_ConnectInd) are required to properly capture the desired L2CAP behavior.

To increase the usability of model checking for verification of software designs, ComFoRT introduced a new formalization of the model checking problem in which state-based and event-based claims could be verified in a variant of LTL called SE–LTL [30]. The formalization extends the usual definition of Kripke structure shown in Def. 8.1 with a *labeled* Kripke structure in which transitions (not just states) are labeled with *actions*, and where an *event* is a particular kind of action.

SE–LTL is formally no more expressive than LTL. However, the claims involving combinations of state and event behavior are much more conveniently expressed in SE–LTL than conventional LTL. Experiments have shown that standard, efficient LTL model checking algorithms can be used with SE–LTL, at no extra cost in space or time [30].

SE–LTL formulas are constructed from the usual operators of propositional logic augmented with three *temporal* operators:

- **G**: "Always." G p means that p is always true from the current state forward.
- **F**: "Eventually." F p means that p is either true in the current state or will become true at some point in the future.
- **U**: "Strong until." p U q means that p is true until q becomes true *and* q must eventually become true.
- **X**: "Next." X p means that p must be true in the next state.

Propositions may made about program state (i.e., a program variable) or about events (i.e., a pin event). So for example G ^s => F [x ==0] means that if a "begin p" event occurs the program variable x will always eventually become 0.

Predicate Abstraction

Predicate abstraction automatically constructs an abstract model that describes the behaviors of the original component in terms of these predicates. For example, let x, s, and t be integer variables of a component C, let P and Q be predicates defined as $P \leftarrow x < 5$ and $Q \leftarrow s + t = 3$.

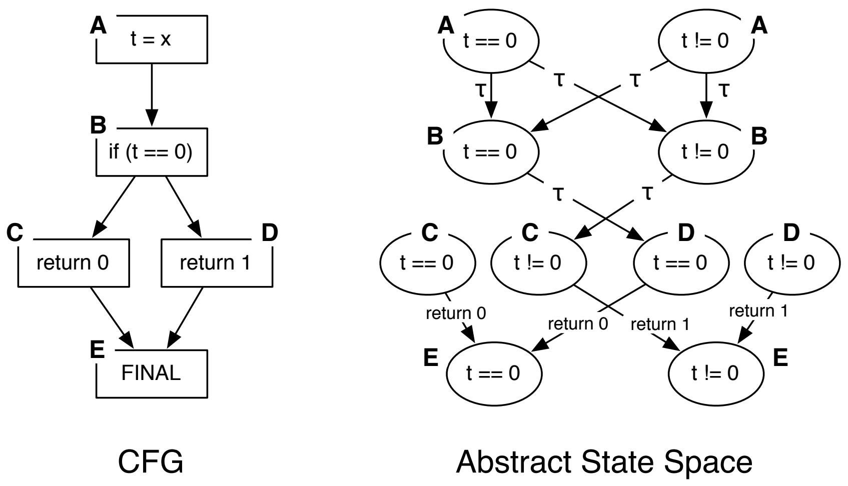


Figure 8.5: Predicate Abstraction

Once a finite set of predicates is chosen, the states of the corresponding abstract model are simply valuations of the predicates. Each abstract state A symbolically represents the *set* of states of the original component that agree with A on the valuations of the predicates. For example, the abstract state $(P = True, Q = False)$ corresponds to all component states where variable x is less than 5, and the sum of s and t is *not* equal to 3.

Figure 8.5 illustrates these ideas. The left side shows the control flow graph (CFG) of a simple component with two integer variables x and t. If we define a single predicate $P \leftarrow t = 0$, two abstract states correspond to each control location in CFG, i.e., where P is either True or False. The right-hand side of shows the abstract model that we obtain via predicate abstraction. Transitions are labeled with actions that can represent synchronization events (absent here), the return values of procedure calls, and internal actions (τ). Note that certain abstract states are unreachable (e.g., the state corresponding to location C and valuation True for P). Intuitively, this is true because the component can never take the else branch of the if statement in location B when t is equal to zero.

The initial set of predicates can be obtained in many ways. The most common way is to collect formulas appearing in conditional expressions as well as in the claim to be checked. The user can also specify predicates of interest, perhaps based on some deeper understanding of the system. New predicates are generated, if needed, in the model refinement phase, which is described next.

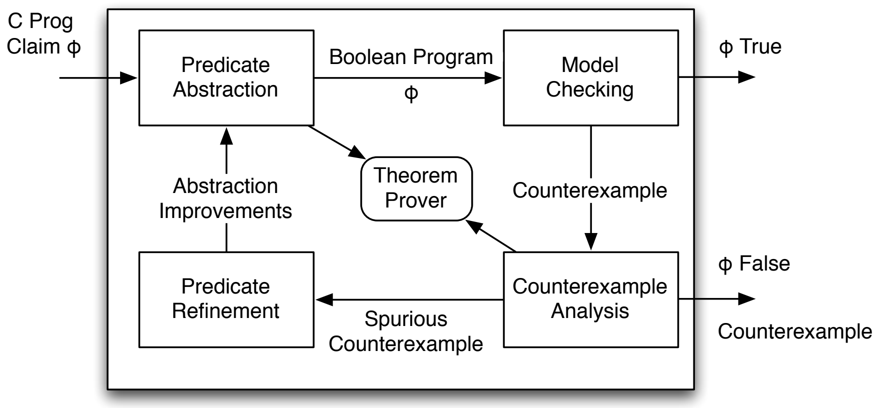


Figure 8.6: CEGAR Loop

Counterexample–Guided Abstraction Refinement

The model constructed by predicate abstraction is guaranteed to be a conservative abstraction of the original system, meaning that each behavior in the original system is represented by some behavior in the model, although the model may contain more behaviors. As a result, if the model satisfies the claim, so does the original system [39].

However, a counterexample obtained by verifying the model may be *spurious*; in Figure 8.5 any counterexample that involved a state corresponding to location C and valuation True for P would be spurious because that state could never be reached in the component program. Using a theorem–prover, the model checking engine analyzes the counterexample and, if it is spurious, uses this information to derive additional predicates to construct a new, finer–grained abstraction of the system. The verification is then repeated, with the refined model. The process continues until either the claim is shown to be satisfied, the claim is refuted and a counterexample is produced, or the model checker runs out of time or memory.

This iterative refinement process, depicted in Figure 8.6, is known as *counterexample guided abstraction refinement* (CEGAR) [95], and, besides its use in ComFoRT, has been successfully used to verify industrial hardware [38] and, when combined with predicate abstraction as in ComFoRT, industrial software [15][66].

Compositional Reasoning in CEGAR

In addition to automated abstraction procedures, the model checking engine applies compositional reasoning within the CEGAR framework to further reduce verification complexity. Assume that an assembly A consists of components $C_1 \ldots C_n$ executing concurrently. The algorithms that check whether a claim Φ holds for A use the following three-step iterative process.

1. *Abstract.* Create an abstract model $M = M_1 \parallel \ldots M_n$. Note that the construction of the M_i's can be done one component at a time without constructing the full state space of A. Further, it can be shown that if A has an error, so does M.

2. *Verify.* Check if a claim $M \models \Phi$. If it does, report success and exit. Otherwise, let CE be a counterexample that indicates where Φ fails in M.

3. *Refine.* Check whether CE is a valid counterexample with respect to A. Once again, this is done one component at a time. If CE corresponds to a real behavior, the algorithm reports a failure and a fragment of each M_i that shows why $\neg(A \models \Phi)$. If CE is spurious, refine M using CE to obtain a more precise abstract model and repeat from Step 1.

Note that only the verification stage (Step 2) requires the explicit composition of components, though this composition always involves only the abstract models. All other stages can be performed compositionally (i.e., one component at a time).

Deadlock Detection in CEGAR

Verifying the absence of deadlock in a composed system is a common requirement, especially for safety-critical systems. As always, if deadlock is detected, it is highly desirable to be able to provide system designers and developers with feedback showing what caused the deadlock.

However, despite significant efforts, validating the absence of deadlock in systems of realistic complexity remains a major challenge. The problem is especially acute for concurrent processes that communicate via mechanisms with blocking semantics (e.g., synchronous message-passing and semaphores). Abstraction and compositional reasoning are less useful in detecting deadlock because deadlock is inherently non–compositional, and its absence is not preserved by standard abstractions.

ComFoRT extends CEGAR with *abstract refusals* to either detect a deadlock or to prove that no deadlock exists. These extensions are grounded in standard CSP [75]. The resulting CEGAR approach for deadlock detection is completely automated and provides a counterexample whenever a deadlock is detected.

Proof Certificates

The outcome of the CEGAR loop shown in Figure 8.5 exhibits an odd asymmetry. While verification failures are accompanied by a *witness*—a machine–checkable counterexample, verification successes must be taken on faith, as their is no corresponding *witness* to success. The ComFoRT reasoning framework can, in some circumstances, generate a witness to success, and as such it may be regarded as a *certifying* model checker [129, 94].

Chaki shows that CEGAR can be used to extract a witness Ω, and defines a procedure for generating a verification condition (VC) from Ω [28]. He also demonstrates that a component C will satisfy a policy Φ if and only if VC is valid (i.e., True under all variable assignments in C). The witness is constructed and shipped by the code producer along with C and the proof P(VC). The code consumer uses Ω to *reconstruct* VC, and then checks P(VC), which is efficient because proof checking is generally a linear–complexity operation. Therefore, the witness Ω and the proof P may be regarded as a certificate that C respects Φ.

8.6.4 ComFoRT Constraints

In the case of $\lambda*$ where each reasoning framework constraint is intended to satisfy one or more analytic assumptions, constraints in ComFoRT are motivated by the need to reduce the size of the model state space.

The following constraints are imposed by ComFoRT:

- PCL types **string**, **float** and array variables are not permitted.
- PCL **proc** and **extern proc** are not permitted.
- verbatim code is not permitted, but this constraint can be overridden.
- certifying code generation applies to components only (not assemblies).

8.6.5 ComFoRT Decision Procedure

Strictly speaking, the decision procedure for ComFoRT is CEGAR and its extension to accommodate deadlock detection.

CEGAR is a semi–decision procedure: it produces correct results if it terminates, but there is no guarantee of termination. Nevertheless, it works reasonably well in practice. For example, it was used to find a bug in Micro-C OS version 2.00, a real-time operating system for embedded software consisting of about 3,000 lines ANSI C. It has also been used to verify an extensive set of claims against the OpenSSL implementation, an open source implementation of the Secure Socket Layer (SSL).

Figure 8.7 illustrates the overall structure of ComFoRT. All of the technical details of model checking described above take place within the “model checking” box. The remaining elements of ComFoRT provide the interpretations and reverse–interpretations needed to apply the model checker to components and assemblies specified in PCL. One reverse interpretation is required to present counterexamples as PinChart sequence charts; another is

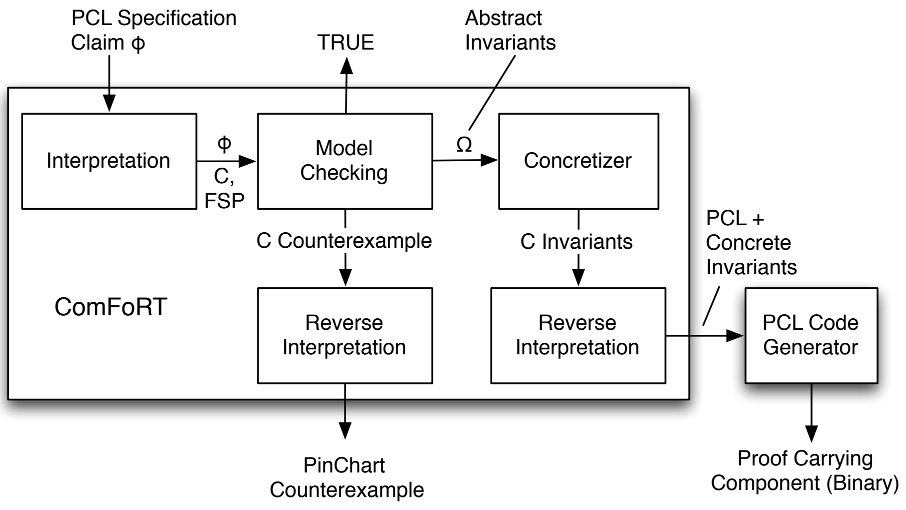


Figure 8.7: ComFoRT Workflow

required to embed the abstract proof certificate Ω into a modified PCL specification. This, in turn, is used by the PCL code generator to produce proof–carrying component binaries [33]. This approach follows the general scheme for "proof carrying code" introduced by Necula and Lee [131], with ComFoRT playing the role of a fully–automated verification condition generator ("VCG" in [131]).

Part III

Experiences

Chapter 9

Industrial Cases

Industrial case studies were crucial to the research described here, in revealing the structure of the Seam and the pragmatics of achieving predictability by construction. Case studies also guided the development of the PECT that is available today in the PSK.

The remainder of this chapter describes the key results of case studies in electric grid substation automation systems and industrial robot control systems. §9.1 introduces a few preliminaries pertaining to the structure of model problems and the statistical techniques used to demonstrate formal predictability in solutions to these problems. §9.2 and §9.3 provide details on case studies in substation automation and industrial robot control, respectively. Finally, §9.4 summarizes the key results of these efforts.

9.1 Preliminaries

The terms "model problem," "model solution," and "statistical label" appear *passim* throughout the case study descriptions that follow. A brief introduction to these terms is provided in §9.1.1 *Model Problems* and §9.1.2 *Statistical Labels.*

9.1.1 Model Problems

The structure of model problems in these case studies is depicted in Figure 9.1, and is analogous in its essential aspects to model problems as described by Wallnau, Hissam, and Seacord [170].

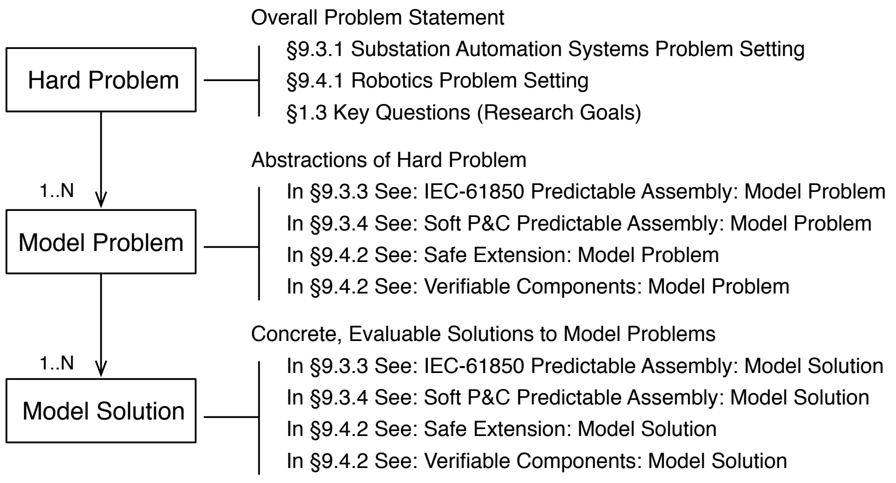

Overall Problem Statement
§9.3.1 Substation Automation Systems Problem Setting
§9.4.1 Robotics Problem Setting
§1.3 Key Questions (Research Goals)
Hard Problem
1..N
Abstractions of Hard Problem
In §9.3.3 See: IEC-61850 Predictable Assembly: Model Problem
In §9.3.4 See: Soft P&C Predictable Assembly: Model Problem
In §9.4.2 See: Safe Extension: Model Problem
In §9.4.2 See: Verifiable Components: Model Problem
Model Problem
1..N
Concrete, Evaluable Solutions to Model Problems
In §9.3.3 See: IEC-61850 Predictable Assembly: Model Solution
In §9.3.4 See: Soft P&C Predictable Assembly: Model Solution
In §9.4.2 See: Safe Extension: Model Solution
In §9.4.2 See: Verifiable Components: Model Solution
Model Solution

Figure 9.1: Structure of a Model Problem

Essentially, a model problem is a reduction of a design issue to its simplest form from which one or more model solutions can be investigated. A model solution is a demonstration of a design, implementation, or example that addresses the issue posed by the model problem.

The "art" in defining a model problem is to ensure that the intended audience—usually a technology adopter—agrees that a successful model solution says something meaningful about the real problems that are abstracted by the model problem.

9.1.2 Statistical Labels

Statistical models figure prominently in the case studies. One use is descriptive, for example to express the estimated value of component properties, and the quality of these estimates. A second use is inferential, for example to express the quality of reasoning framework predictions for some components and assemblies not yet designed.

We refer to statistical models that describe component and reasoning framework properties "labels" by analogy with food labels. See [124] for a brief tutorial discussion on statistical labels for software components and reasoning frameworks. See Larsson's dissertation [98] for a detailed description of the validation techniques used in the research summarized here.

Component Labels

In general, a property of interest (the "measurand") is a function of N values: Y = f($\chi 1$, $\chi 2$, . . . , χN) [165]. For $\lambda *$, these values included, in different combinations depending upon component type, execution time, blocking time, and period. We would like to know the true value of Y, component latency. Of course, the true value of Y or any $\chi 1$, $\chi 2$, etc., is not obtainable, as the following definition makes clear:

Definition 9.1 (True Value) *True Value: the mean (μ) that would result from an infinite number of measurements of the same measurand carried out under repeatability conditions, assuming no systematic error.*

Because we can not, even in principle, know the true value of μ, we must use an estimator for it, produced by statistical methods. For example, we take a sample of observations of χ, and use its average as the estimator of m, a population parameter. The uncertainty associated with this estimator is expressed as the standard deviation s such that the true—and unknown—value of μ will fall within some interval with some specified confidence. The factor k is known as coverage factor. When k=1, yields a 68% confidence interval. 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.

Confidence Interval λ-ABA

Proportion (ρ)................	80%
Upper Bound (MRE).....	< 1%
Confidence (γ).............	> 99%

Figure 9.2: λ-ABA Confidence Interval

Prediction Labels

We use γ “confidence” and ”tolerance” intervals to characterize how effective a property theory is likely to be for future predictions, where γ is typically chosen to be either 95% or 99%, but is sometimes itself calculated.

γ **Confidence Interval** We use confidence intervals if we want to compute the proportion p of a population of a population that will satisfy stated interval. The interval is specified in measurement units appropriate for a property, and the probability ρ (called the “population parameter”, not to be confused with “confidence”) is calculated an object in the population will satisfy that interval.

γ **Tolerance Interval** We use tolerance intervals if we want to compute the interval that contains some proportion p of a population. The population parameter ρ is specified, and the interval is calculated is calculated.

For prediction labels we are interested in the describing the magnitude of relative error (MRE) between the assembly latency predicted by a theory, and the latency observed in the environment:

Definition 9.2 (Magnitude of Relative Error)

$$MRE = \frac{|a.\lambda - a.\lambda'|}{a.\lambda'} \tag{9.1}$$

where $a.\lambda$ and $a.\lambda'$ are the predicted and measured latency of assembly a, respectively.

Figure 9.2 shows the statistical label assigned to the λ-ABA reasoning framework. The label may be interpreted as saying “80% of all assemblies well–formed to λ-ABA (i.e., predictable by construction with respect to λ-ABA) will exhibit less than 1% MRE between actual and predicted performance, and we have more than 99% confidence in this upper bound.”

9.2 Substation Automation Systems: Soft P&C

The electric power grid (where no confusion will arise, simply "the grid") is responsible for generating electrical power, transmitting electricity over short and long distances, and, ultimately, for distributing power to consumers. The electric grid is a quintessential example of "critical infrastructure" that requires the application of disciplined engineering of the highest order.

Substations are nodes in the grid that (among other things) monitor, protect and control transmission lines external to the substation, as well as a variety of equipment managed by each substation, including transformers, switches, circuit breakers, and meters. Substation operations are performed by substation automation systems (SAS), which may involve human operators or be wholly autonomous.

The technical and business drivers that shaped the research are described in §9.2.1. Background on the industrial standard IEC-61850, which had a strong influence on the work, is provided in §9.2.2. The research was carried out over several years, which for convenient exposition are described as corresponding to two stages of work. The first stage (§9.2.3) concentrated on developing the technologies needed to demonstrate the *technical* feasibility of the approach, and the methods used to develop those technologies. The second stage (§9.2.4) concentrated on demonstrating the *practical* feasibility of the approach by having a collaborator from ABB use the infrastructure developed in the first stage to tackle a more complex design problem.

9.2.1 SAS Problem Setting

As is most industry sectors, the electricity industry is increasingly reliant on digital technology as a way of reducing the cost of products while also improving their quality and introducing new and competitive product features. The most prominent business drivers that influenced the direction of this research (with the first item adding urgency to the remaining two) were:

- New technical and business strategies are needed to meet the explosive growth in demand for SAS in developing economies, notably China and India.
- SAS must be susceptible to third–party integration using from products supplied by different, and possibly competing vendors.
- SAS functions that currently execute on dedicated real–time computers must be consolidated to execute on shared, commodity computers.

One strategy for meeting these business drivers is *Soft Protection and Control* (Soft P&C), which, for the purpose of this research, is defined as follows :

Definition 9.3 (Soft Protection and Control) *A complete substation automation system that is implemented on a centralized (more or less standard) industrial personal computer, with no proprietary hardware.*

The transition from substation automation systems constructed from proprietary, specialized (and, historically, analog) equipment to Soft P&C represents a significant change for electricity industry. However, when viewed from the vantage of real–time computing, the technical challenge of implementing major SAS functions on commodity computers and operating systems are not particularly daunting. Instead, the real challenge is in persuading a relatively conservative industry and customer base that the cost and schedule benefits of Soft P&C do not come at the expense of reliability [99]—the PECT emphasis on producing justifiable confidence in predicted behavior was a good fit to meet this challenge.

9.2.2 Preliminaries on IEC–61850

The IEC–61850 *Communication Networks and Systems in Substations*[78] standard played an important role in the case study.[1] While there are many potential benefits promised by the adoption of IEC–61850, one claim frequently encountered ([26, 92] are just two of many cases) is that IEC 61850 allows, with some paraphrasing, "free allocation of functions to devices", meaning that devices previously dedicated to one or some other small but fixed number of functions could be used for an open–ended number of functions, subject to performance or other resource constraints, and "free" can be taken to mean "third party."

Preiss and Wegmann [140] further speculated ("claimed" would be too strong) that, *if* IEC 61850–defined functions can be systematically mapped to software components, *then* "free allocation" can be extended to include "free composition" as well: more complex functions might be composed from a library of (possibly third–party) SAS components; the component parts of these functions might be allocated to different devices or to a single device, and each device might have allocated to it multiple functions or component parts of functions.

The kind of flexibility envisaged by Preiss and Wegmann would benefit from predictability by construction. We might go further to suggest that predictability by construction is a prerequisite to this vision if, in addition to compositional flexibility, we also want to a) drastically compress the cost and schedule to engineer and commission substations, and b) engender justifiable confidence in the quality of the resultant SASs in the minds of customers, regulators and industry partners.

[1]See `http://www.abb.com/industries/us/9AAC30200310.aspx?country=US`, last accessed on 12 August 2010, as an indicator of ABB's business interests in IEC 61850.

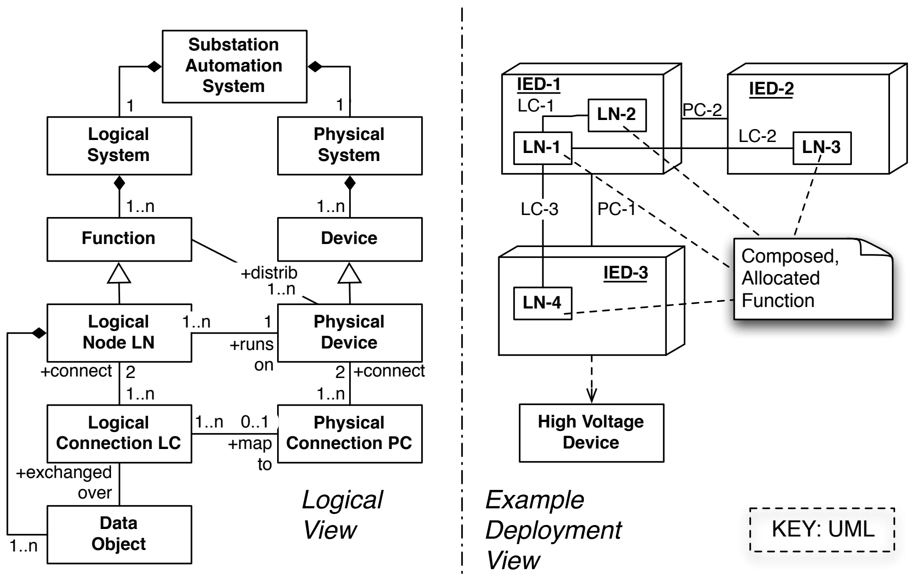


Figure 9.3: Relevant IEC 61850 Concepts

With this in mind, IEC–61850 took on a prominent role in the case study, serving as an authoritative source of SAS function definitions, extra–functional requirements, and model problems in predictability by construction in Soft P&C. Figure 9.3 (adapted from [140]) summarizes a number of important terms used by IEC-61850 used in the Soft P&C case study:

- The left side of Figure 9.3 shows the logical system—one or more SAS *functions* (sometimes called *logical functions* or *logical nodes*), *connections* among functions, and *data objects* transmitted on these connections. This view constitutes "engineering data" maintained in by tools in the SAS engineering environment. IEC–61850 defines standards (for example, it defines a configuration language) for the representation or this data. The logical system is then allocated to the physical system.

- The right side of Figure 9.3 is a simple illustration of a logical SAS mapped to a physical SAS. The term "intelligent electronic device" (IED) in this research always refers to a computer of some kind. In this example, four logical functions are composed into an unnamed composite function; each is also deployed on one of three IEDs. The composed function manages a high voltage device attached to one of the IEDs, although the figure is not detailed enough for the reader to deduce this.

9.2.3 Stage–1: Developing the Basic PECT

Very little technical infrastructure was available at the outset of this case study, and it therefore a two–stage approach made sense. In the first stage (described in this section) we would produce the required tooling (component technology, reasoning frameworks, specification notations, semantic interpretations to reasoning frameworks, measurement and validation infrastructure, etc.) and use this to establish the overall technical and practical feasibility of the Seam and using PECT to automate the Seam. In the second stage (described in §9.2.4) we would apply the results of Stage–1 to a more demanding model problem, the details of which would depend on the outcome of Stage–1.

IEC–61850 Predictable Assembly: Model Problem

Working with Preiss and others at ABB, we defined three classes of Soft P&C assembly from which model problems in predictability by construction could be defined:

1. Operator assemblies. These are the operator interface subsystems of an SAS, and are composed substantially from human–machine interface (HMI) components.

2. Control assemblies. These are the subsystems of an SAS that provide protection and control of grid equipment, and are composed from IEC-61850 components.

3. Combined operator and controller assemblies. These are subsystems of an SAS that allow operators to interact with grid equipment, and are composed from operator assemblies and controller assemblies.

These classes are depicted in Figure 9.4, along with the nominal confidence intervals we required for predicted latency for each class of assembly.

Figure 9.4 may be interpreted as describing a hierarchy of PECTs, one for composing controllers from components, one for composing operator stations from components, and one for composing substation automation systems from operator station assemblies and controller assemblies. We initially thought this might be a possible result, similar (in some ways) to Szyperski's *tiered* component systems [160]. As it turned out, however, one PECT was sufficient for all three classes of assembly.

Stage–1 focused on building the technical infrastructure (the PECT) and defining the design methods and workflows for developing a PECT, and in particular for designing and validating reasoning frameworks. The prediction requirements for these model problems did not pose a severe test of GRMT, but posed enough of a test to make exercise meaningful, that predictability by construction can be achieved for a well–defined class of design problems,

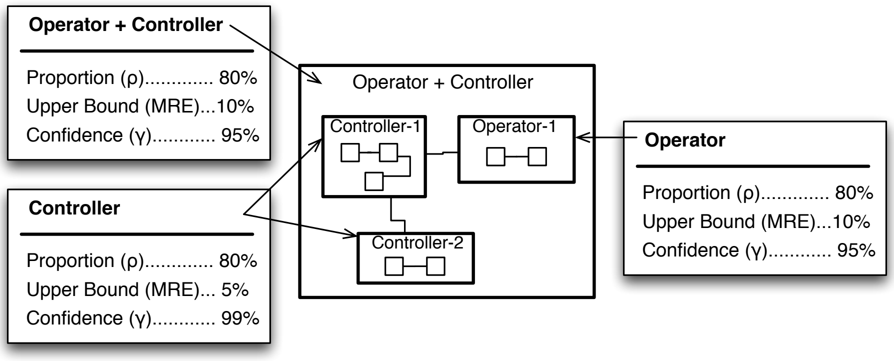

Operator + Controller
Proportion (ρ)............. 80%
Upper Bound (MRE)...10%
Confidence (γ)............ 95%
Controller
Proportion (ρ)............. 80%
Upper Bound (MRE)... 5%
Confidence (γ)............ 99%
Operator + Controller
Controller-1
Operator-1
Controller-2
Operator
Proportion (ρ)............. 80%
Upper Bound (MRE)...10%
Confidence (γ)............ 95%

Figure 9.4: Initial SAS Model Problems

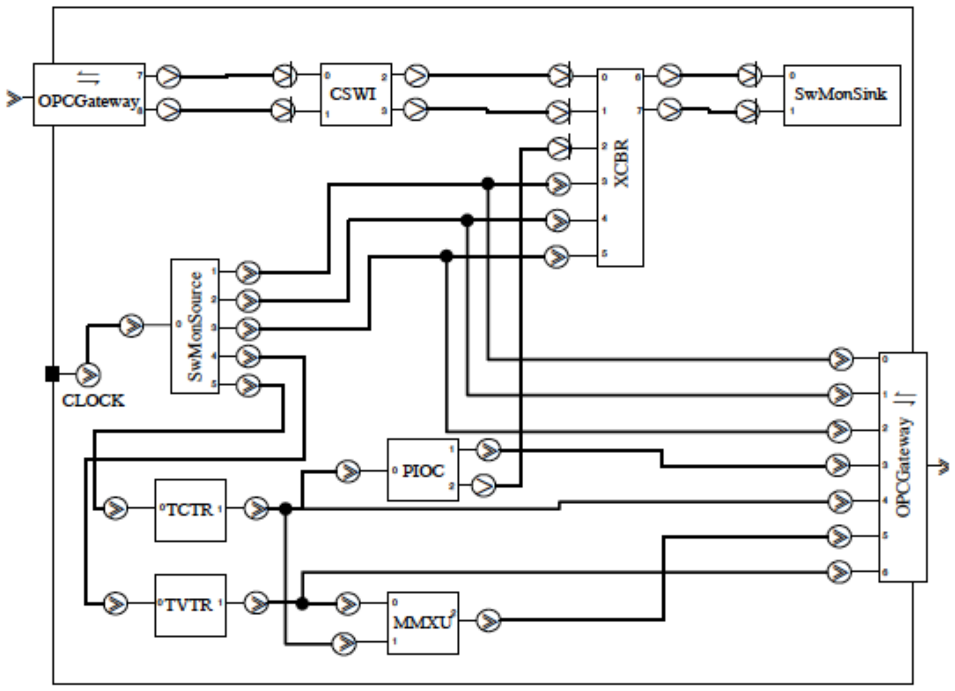


Figure 9.5: Concrete 61850-Based Controller Assembly

and to establish that a PECT and its reasoning frameworks can be tuned to different required strength of evidence for predictions, and different degrees of intrusiveness on designer and programmer prerogatives.

Details of what we learned about the PECT design methods and workflows are presented later in the discussion of *Co–Refinement*, in §10.2 on pp. 213. Much of that discussion is concerned with the stepwise development of the $\lambda*$ performance reasoning framework—its theories, interpretations, and validation. Those details will not be repeated here.

IEC–61850 Predictable Assembly: Model Solutions

While it may be possible to develop and validate a reasoning framework using only synthetic components and assemblies, a small number of concrete model problems, with "real" components and assemblies, is needed as a proxy for reality—as a kind of plot device to get the PECT designers "into the shoes" of the end–user, and conversely to help our end–user collaborators to relate to what is, after all, an integration of several non–trivial technoogies. Thus, we specified assembly instances for each of the classes of assembly illustrated in Figure 9.4. We used these concrete instances to "spot check" the PECT at various stages of its co–refinement.

For the controller class, we used the controller assembly depicted in Figure 9.5 (the other exemplars can be found in [68]). This assembly controls a switching device (not shown in the figure), custom fabricated for use in Stage–1. The device consists of a software and hardware switch arranged in series, with the idea that we want the software switch to "trip" to demonstrate soft protection, but have the hardware switch as backup and to report failures of soft protection. The OPCGateway *services* permit interactions among operator and controller assemblies. The XCBR, CSWI, TCTR, TCVR, PIOC, and MMXU component instances each corresponds to a logical node defined by IEC-61850. The functional roles of these components, and of other components either fully specified in PCL or wrapped from legacy software, are summarized in Table 9.1.

The switch device had a rheostat that an operator/experimenter could use to vary line current (which could also be done programmatically, of course), and a light that would indicate situations where the hardware switch was tripped due to missed software deadlines. While the combined controller and device was little more than a "toy," it served its above–stated purposes admirably.

Model Checking Analysis

For the most part, all of the work in Soft P&C (both Stage–1 and Stage–2) concentrated on formal predictability and predictability by construction of real time behavior. However, we also used the opportunity to initiate work

Name	**Role**	**Description**	**Key**[a]
CILO	Station or Bay Interlocking	Interlocking may be totally centralized or totally decentralized.	2/g
CSWI	Switch Controller	Controls all switching conditions above process level.	1,2/g
IHMI	Operator Interface	Front–panel interface at bay level, local operator interface at station level.	2/g
TCTR	Current Transformer	Delivers current as sampled values.	1
TCVR	Voltage Transformer	Delivers voltage as sampled values.	1
PDIF	Differential Protection	Protect against percentage, phase angle or other quantitative difference of two currents or other quantities.	2/g
PIOC	Instantaneous Overcurrent Protection	Protect against an excessive value of current or an excessive rate of current rise.	1
PTOC	AC Time Overcurrent Protection	Act as a relay when the AC input current exceeds a predetermined value.	2/w
PTOV	DC Overvoltage Protection	Act as a relay when its input voltage is more than a predetermined value.	2/w
PTRC	Protection Trip Conditioning	Connects the "operate" signal of PIOC to trip signal of XCBR. If condition of tripping the XCBR is met then the circuit breaker is to trip.	2/g
MDIF	Differential Measurements	Acquire differential values from current and voltage transformers.	2/g
MMXU	Measurement	Acquire values from current and voltage transformers.	1,2/w
XCBR	Circuit Breaker	Models switches with short circuit breaking capability.	1,2/g
XSWI	Switching Devices Unable to Switch Short Circuits	Line switch is a switch used as a disconnecting, load-interrupter, or isolating switch on an AC or DC power circuit.	2/g

[a]Stage–1 components ('1') are hand written; Stage–2 components ('2') are generated from PCL ('/g') or are generated wrappers ('/w') for legacy SAS code.

Table 9.1: IEC 61850 Components Implemented in Pin

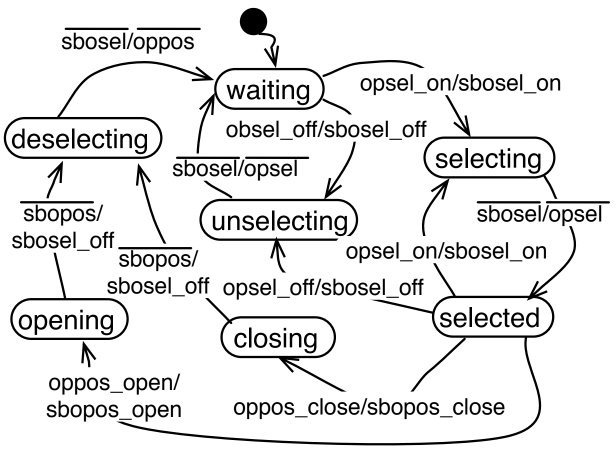
on the development of a reasoning framework for verifying component and assembly behaviors specified in some form of temporal logic, or possibly in some other automata–theoretic representation. We were inspired in particular by the Kramer and McGee's use of the FSP process algebra to specify and check the behavior of concurrent Java programs [110] and software architectures specified in Darwin [112, 55], although there were other examples as well.

The choice of action language for PCL had yet to be decided (and was not implemented until Stage–2). We had contemplated using FSP, or possibly CSP [75], to specify component behavior. However, we ultimately judged that such notations are too arcane for practitioners. The Java/FSP approach was ruled out because the FSP model checker, the Labeled Transition System Analyser (LTSA), did not scale beyond trivial models, and because C and C++ rather than Java are the programming languages of choice for real–time and embedded software. In lieu of deciding on an action language for PCL, we used CSP as a starting point because it is more expressive than FSP, and because we had already begun to formalize the semantics of the Pin component model that was being also being developed.

As a first step we specified the behavior of CSWI in CSP. CSWI was chosen because it implemented a relatively simple (though conventional) protocol for opening and closing switches, and for selecting and deselecting switches prior to changing their open/close position. As a further simplification, we restricted the model to externally visible behavior on Pins, and left to later modeling program variables and other such code–level details (which at this point we were not certain should be modeled at all). We knew of course that whatever model checker we used for initial experiments would be susceptible to state space explosion problems, and we preferred to deal with those problems separately. The interface specification for CSWI is:

```
typedef enum {on, off} TSelect;
typedef enum {open, close} TPos;
component CSWI()
{
  sink synch opsel (consume TSelect val);
  sink synch oppos (consume TPos val);
  source synch sbosel(produce TSelect val);
  source synch sbopos(produce TPos val);
}
```

Using CSP at least had the advantage consistency with the work that was occurring simultaneously to define the compositional semantics of Pin [82]. While we could have used FDR directly on the CSP model, we also anticipated the need to use a range of model checkers, some already being used in industry, and others that were only just emerging from research labs. These considerations suggested the need to define an interpretation, from



waiting
deselecting
selecting
unselecting
opening
closing
selected
$\overline{\text{sbosel}}/\overline{\text{oppos}}$
opsel_on/sbosel_on
obsel_off/sbosel_off
$\overline{\text{sbosel}}/\overline{\text{opsel}}$
$\overline{\text{sbopos}}$/
sbosel_off
$\overline{\text{sbopos}}$/
sbosel_off
$\overline{\text{sbosel}}/\overline{\text{opsel}}$
opsel_on/sbosel_on
opsel_off/sbosel_off
oppos_open/
sbopos_open
oppos_close/sbopos_close

Figure 9.6: Model Checking Interpretation of CSWI

whatever action language we ultimately would choose, to a neutral semantic representation that could be further translated to notations used by specific model checkers.

The CSP model of CSWI is not shown here, but its interpretation to a neutral semantic representation (which was formally defined but done using "paper and pencil"), for which purpose we chose UML Statecharts, is shown in Figure 9.6. The Statechart follows the conventions introduced in Chapter 6 to denote the CSP interpretation of pin p as a pair of CSP events p and $\bar{p}$, that denote begin/end interaction events, respectively, on p. Also, parameters are encoded directly in the events themselves as "_value" suffixes on event names. For example, the rightmost transition from *waiting* to *selecting* is triggered by the arrival of a request on the 'opsel' sink pin with the parameter value 'on', and this results in CSWI generating a request on its 'sbosel' source pin, also with the parameter value 'on'.

A further interpretation was defined from the Statechart representation to the input language of the NuSMV model checker [37], and then various behavioral claims were specified for the component and verified by the model checker, for example:

$$\mathrm{AG}((\mathrm{state} = \mathrm{waiting}) \wedge (\mathrm{input} = \mathrm{opsel\_on}) \rightarrow \mathrm{AX}(\mathrm{output} = \mathrm{sbosel\_on}))$$

which specifies the "liveness" claim (informally: something good *eventually* happens) that for all paths globally (AX), whenever CSWI is waiting (state = waiting) and the operator select arrives (input = opsel_on), the sbo select signal (output = sbosel_on) will eventually arrive (AX). And,

$$\neg\mathrm{E}[\neg(\mathrm{output} = \mathrm{sbosel\_on})\ \mathcal{U}\ (\mathrm{output} = \mathrm{sbopos\_open})]$$

which specifies a “safety” claim (informally: something bad *never* happens) that it is always impossible for the switch top be opened before it has been selected.

The CSWI model and its two–step interpretation to NuSMV served one analogous purpose to the “toy” switch and assembly by facilitating dialog between different stakeholders in the PECT development activity, in this case between experts in temporal logic model checking and industry experts in substation automation. From these dialogs, we made several observations and reached an important (if tentative) conclusion about the design of a model checking reasoning framework.

A model checking reasoning framework would impose on its users a steep learning curve. Temporal logics and temporal logic model checking were (and still are) almost completely unknown to software engineering practitioners, even those who had some familiarity with “formal methods” for specifying program behavior. The steepness of the learning curve was (and still is) exacerbated by the relative immaturity of model checking technology (compared with GRMA), especially regarding the treatment of state space explosion. According to Giannakopoulou [55], this in turn requires the users of model checkers to pick and choose from among various arcane heuristics to ameliorate the effects of state space explosion. And this, in turn, required familiarity with the algorithms used by the model checker, and with a substantial amount of model checking theory.

Model checkers tend to adopt a state–based approach (such as NuSMV), which models behavior as sequences of changes to program state variables, or a transition–based approach (such as LTSA), which models behavior as sequences of transition events. Both approaches model equivalent behaviors, but of course the syntactic form these claims take, and the mental models they impose, are quite different. However, component systems such as Pin are naturally thought of as combining state–based behavior (changing state in component instances) and event–based behavior (sequences of interactions among component instances). Forcing users to an exclusively state– or event–based style of specification was unnatural, and introduced additional and wholly artificial complexity to the use of model checking technology.

For a PECT–style model checking reasoning framework to succeed, it would need to hide the substantial complexity of model checking tools from its users, and we were uncertain about the extent to which this could be achieved. However, two important design decisions were made for the model checking reasoning framework:

1. We required a verification model (and associated temporal logic) that combined event–based and state–based behavior; this ultimately led us to develop the state–event approach [30] that was later adopted by ComFoRT.

2. We decided to adopt UML Statecharts as a basis for specifying the

behavior of Pin components, because it offered a notation that combined states and transitions, and was familiar to software engineering practitioners.

One additional decision was made as well: to suspend further co–refinement of the model checking reasoning framework while we worked out the details of how to do 1 and 2, above. We would not return to the subject of model checking Soft P&C components or assemblies. That particular thread of investigation was picked up in the case study described in §9.3.

9.2.4 Stage–2: Distributed Soft P&C

Attention shifted in Stage–2 to testing the PECT instance on a more demanding range of concerns. We wished to determine:

- Would the λ-ABA validation work on "real" rather than synthetic assemblies?
- Were the design rules imposed by the PECT overly restrictive for practitioners?
- Did the overall PECT apparatus support a complementary architect/programmer view of the Soft P&C design problem?

Soft P&C Assembly Model Problems

Thomas Werner from ABB Corporate Research, Switzerland joined the research effort at this point to play the role of SAS engineering practitioner, as well as the role of lead designer using the prototype PECT to develop a prototype Soft P&C–based SAS. As before, the first order of business was developing an appropriate model problem with which to answer our questions.

IEC-61850 defines several classes of SAS. For the model problem we chose to design a prototype in the D2 "Medium Distribution Substation" class, the most common class of SAS. This class is defined as having more than five but fewer than twenty elements (feeders, transformers, etc.) and a station–wide communication network. Figure 9.7 depicts the line diagram of the system on the left, and the allocation of Soft P&C functions to two IEDs (SoftPC–A and SoftPC–B) on the right. The substation elements labeled Qx (x = 0, 1, etc.) denote various kinds of switch; BBx denote busbars, and T1 denotes a transformer. Seven concrete demonstration scenarios were defined:

1. SoftPC–A and SoftPC–B operate correctly in steady–state.
2. Manual operation of switches by substation operator.
3. Manual setting of equipment parameters by substation operator.

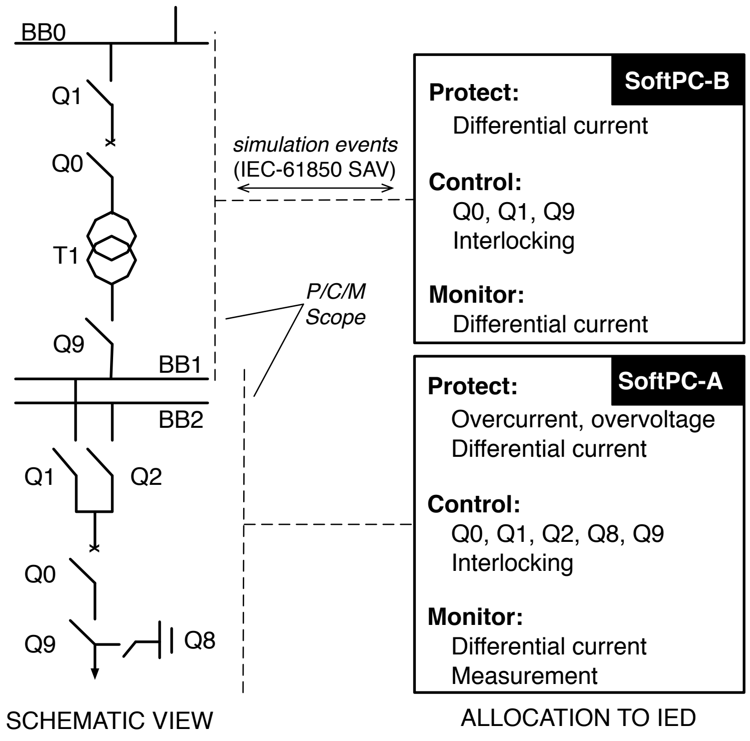


Figure 9.7: Soft P&C Top–Level Logical View

4. Function tripping (overcurrent and differential protection.)
5. Reporting of protection events.
6. Synchronization of SoftPC–A and SoftPC–B.
7. Predictability by Construction.

Scenarios 1–3 show the system under routine operation. Scenarios 4–6 show the system in various states of protection: responding to a protection event (scenario 4), producing timely reports of a protection event (scenario 5) and bringing the distributed system back to steady state (scenario 6). Scenario 6 demonstrated the capability of a substation designer to change the design/implementation by replacing, modifying or reallocating components; predicting the timing behavior of the modified system; and then comparing actual and predicted timing behavior under the scenarios 1–6.

Soft P&C Assembly Model Solutions

We used a suite of IEC-61850 interoperability testing tools used by Seimens, ABB and Areva, in conjunction with device simulators, to approximate a physical implementation of the substation. The interoperability testing tools generate synthetic *sampled values* (IEC–61850 SAV messages), which are then packetized and fed to equipment simulators. The details of these tools are not essential to understand the results of the experiments, and are not further discussed.

The experimental setup is depicted in Figure 9.8. Each of the boxes denotes a commodity laptop running a conventional version of Microsoft Windows. The *SoftPC-A* and *SoftPC-B* laptops each execute an assembly of IEC-61850 components that implement their allocated protection, control and monitoring functions. *Packet Gen* posts the synthetic sampled data to the network. *Time Tracer* performed event capture and after–run timing analysis. An experiment controller laptop (upper rightmost in the figure) provided a console view of the substation, hosted the device simulators, and hosted the PECT interactive development environment to permit interactive experimentation with different substation configurations. IEC-61850 Generic Object–Oriented Substation Event (GOOSE) messages are produced and consumed by SoftPC's A and B to communicate commands, report status or other events of interest, and are also generated by the experimenter from the console to simulate commands issued by a substation operator.
All scenarios were successfully demonstrated.

9.2.5 Summary of SAS Case Study Results

On completing Stage–2, we had a working instance of a PECT that included all of the elements depicted in Figure 5.1, pp. 76, excluding the Com-

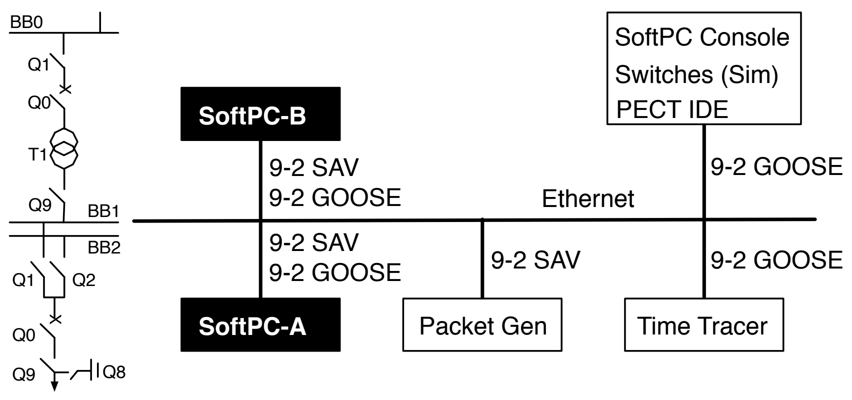


Figure 9.8: Soft P&C Experimental Setup

FoRT model checking framework, and the λ-SS component of λ∗ (which is the subject of the case study in §9.3). A substantial technology infrastructure had been developed. However, several questions were also posed on entry to Stage–2. The case study is concluded with brief answers to each.

Would the λ-ABA validation work on "real" rather than synthetic assemblies?

Of course it is important to caveat "real": we are referring to a prototype simulated class D2 substation rather than a physical mock–up. Nonetheless, I claim that the essentials of the Seam part of the approach, demonstrated in the "method" used to develop the prototype, and in the form of component–based buildtime and runtime used in the prototype solution, would carry over in their entirety to a reasonable next evaluation step involving a physical system.

It is also worth observing that the IEC–61850 components and SoftPC-A/B assemblies developed wholly within the PECT were not trivial. The PCL source specification of SoftPC-A is provided in Appendix B §B.1, pp. 262, and requires 16 IEC–61850 component types, 38 component instances, 128 pin connections, and 100 annotations, the great majority of which define the λ-ABAanalytic interface of components and their various denotable substructures. An analogous degree of reality can be seen in the source listing of the PTRC component found in Appendix B §B.2.

On balance, the case study adequately demonstrates the viability of the approach for problems of realistic complexity.

Were the design rules imposed by the PECT overly restrictive for practitioners?

Here it is important to distinguish restrictions due to limitations in tooling (including all aspects of automation and language design) from those that were intentionally imposed on the PECT users and genuinely impinge on a necessary degree of the user's design freedom. My concern here is with the latter.

At no point did the basic apparatus provided by PCL, Pin, or λ-ABA impose any meaningful restriction of design freedom, and that all restrictions that were *were deliberately imposed* by PCL or λ-ABA were more than offset by corresponding gains in predictability. After some sufficient number of experiences developing assemblies that seldom differ in their predicted and actual timing behavior by more than 1%, we came ultimately to expect this as a norm.

It should also be noted that several of the IEC–61850 component types used in the prototype substation were *not* developed in PCL, but were industrial–strength library components provided by ABB and *wrapped* by PCL , and therefore made available as Pin components. PCL wrapper templates were developed that, with the code generator, make this trivial to do, assuming the legacy code is compiled, native code (ideally, as libraries, as was true of this case study). The ability to wrap external code, or to drop through the trap door of "verbatim" code in PCL, makes it easy to incorporate existing functions into Pin.

Conversely makes it easy for designers to sidestep any restrictions that might encounter—with consequences on predictability, of course.

Did the overall PECT apparatus support a complementary architect/programmer view of the Soft P&C design problem?

I claim that the case study demonstrated this complementarity in a compelling way: units of composition, units of deployment, units of analysis all, from the architect and programmer perspective, coincided with the notion of Pin component.

Scenario 7 also demonstrated one of the stated objectives of the Seam—to support a rational design process whereby design decisions can be objectively justified as making progress towards a possibly distant design goal. In the *generate and test* metaphor of design search, the case study demonstrates an analytic approach to "test" and an analytically–informed constructive approach to "generate."

9.3 Industrial Robotic Control Systems

ABB is a leading provider of industrial robots. One ABB product line architecture for industrial robot control, called “S4” in this case study, had been a substantial commercial success and had been in deployment for several years.

Various enhancements to S4 were being considered, one of which was called the “Open Robot Controller” (ORC). In the business model supported by S4, ABB licensed S4 to other ABB business units to create new industrial robots, or more commonly to perform whatever adaptation would be necessary to apply an existing S4–controlled robot to a new industrial automation problem. The ORC, if pursued, would permit external licensing of the controller as a platform, to independent (non–ABB) vendors.

An industrial robot controller is a large, complex, real–time system. The S4 system architect reported that over 1000 person–years of effort had been invested in the development and implementation of S4, and despite being a product line, each instantiation of S4 required substantial quality assurance efforts to ensure the end–product would perform as expected, and be *safe* when used in industrial applications. “Opening” the platform for third–party extension offered attractive business opportunities, but introduced several serious technical challenges to ensure that all third–party extensions would continue to exhibit safe and predictable behavior.

9.3.1 Open Robot Controller Problem Setting

A critical design feature of this new platform is the ability to customize the controller with user-added extensions. These extensions, made by ABB or other companies, are augmentations to the core controller platform. However, it is not difficult to foresee the potential poor performance or instability introduced through a user-added extension. By analogy, common off-the-shelf operating systems (OSs) permit third-party device drivers that, when flawed, cause unexpected or unwanted behavior potentially impacting quality attributes of the system as a whole.

ABB Robotics wanted a way to ensure that the impact of third-party extensions to the ORC could be predicted. Obviously, it was important to demonstrate that third–party extensions could do nothing that would interfere with the robot’s ability to meet critical control deadlines. At the same time, ABB believed that third–party users of the ORC would also want guarantees *from the ORC*, for example that controller extensions would execute in some predictable way. In short, the ability to ensure safe controller operation while also providing third–parties with concrete performance guarantees was regarded by ABB as a potentially important business discriminator.

Two investigations were conducted in the suitability of PECT to safe and predictable third–party extensions to the ORC. The first, described in

§9.3.2, regarded the design problem as one that was amenable to conventional GRMA, for which the prototype PECT developed for SAS would be a good point of departure. The second, described in §9.3.3, regarded the design problem as ultimately requiring some form of automated verification along the lines of the static driver verification tools that were then being developed by Microsoft [14], and have since been introduced in practice for driver certification.[2]

9.3.2 Safe Extension of Open Controllers

The technical challenge is to define an ORC mechanism that would permit third–party component extensions that would simultaneously:

1. Provide suitable guarantees of hard controller deadlines to ABB.
2. Provide suitable performance guarantees to third–party component developers and integrators.

To achieve 1 we determine if the extension causes any controller task to miss its deadline. To achieve 2 we determine the average latency of plug–in components.

Safe Extension: Model Problem

The typical hardware platform for the ORC consists of a single Intel Celeron processor running VxWorks; this is referred to as the main computer. The main computer communicates with one or more axis computers, each of which controls a degree of motion on the robot itself.

The model problem focuses on the interaction between tasks in the main computer, which is responsible for running robot movement control programs (written in a high-level robot programming language) that generate work orders. Those orders are decomposed into sub–work orders that are further processed, ultimately into movement commands expressed in microcoordinates that are communicated to an axis computer that controls robot movement.

The main computer executes many periodic and aperiodic tasks; however, only a small subset of these must meet hard real time deadlines. Priority–based scheduling is used to ensure these deadlines are met. Communication among tasks makes use of FIFO queues. Figure 9.9 summarizes the task structure of the most critical real–time tasks within the ORC:

- Task A_i receives work orders and create movement control plans expressed as a sequence of sub–work orders that are asynchronously sent to B_i.

[2]See `http://research.microsoft.com/en-us/projects/slam/`, last accessed 21 August 2010.

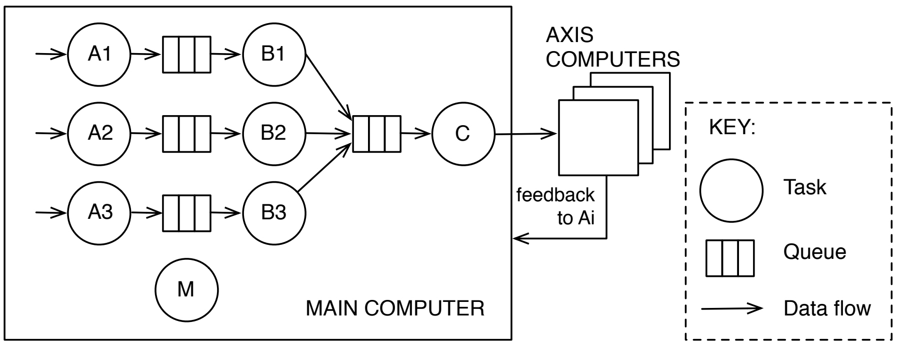


Figure 9.9: Simplified S4 Task Structure

Task	Priority	Arrivals	Exec. Time
A_i	Low	Exponential ($\mu = 75$ms)	Exponential ($\mu = 9$ms)
B_i	High	Constant (24 ms)	Uniform (1–2 ms)
C	Very High	Constant (4 ms)	Uniform (0.5–1 ms)
M	Medium	Exponential ($\mu = 100$ms)	Uniform (15–25 ms)

Table 9.2: ORC Task Details

- Task B_i generates, for each sub-work order, six movement commands expressed in micro-coordinates, and sends these to C.
- Task C receives movement commands from one or more B_i, and converts these to control commands sent to one or more axis computers that control robot movement.
- Task M represents the third–party extension.

Communication channels between these tasks use FIFO queues. Ai will “block” if the Ai–B_i queue becomes full, and likewise Bi will block if the B_i–C queues become. However, the robot is considered to have entered an unsafe state if the B_i–C queue becomes empty—the controller will under these conditions have no appropriate orders to send the axis computers, and the robot will shut down. Consequently, it is also highly undesirable (though not in itself fatal) for the A_i–B_i queues to become empty.

Table 9.2 summarizes the performance parameters of the ORC tasks; the details for tasks A_i, B_i and C are faithful to the S4, while those for task M were conjectured. Note that A_i arrivals and execution time are exponentially distributed; the planning function on occasion takes a long time to complete (very rarely causing queue underflows). Note also that the priority of M is greater than that of any A_i; we must ensure that it does not starved any A_i to the extent that it causes the A_i–B_i queues to empty and cascade to the unsafe state where B_i–C becomes empty.

The assembly depicted in Figure 5.6, pp. 88 shows one possible realization of the ORC task structure for just one instance of A_i (the TrajectoryPlanner component) and B_i (the MovementPlanner component), and C split into two components of equal (and highest) priority. The problem at hand is to permit additional M components to be introduced that will not cause queues to empty, but which will experience predictable average latency.

Safe Extension: Model Solution

The key observation (made by Mark Klein) was that the problem of queue underflow that results from the variable behavior of A_i, and the problem of ensuring that M extensions never interfere with any Ai, can *both* be resolved by using the well–known *sporadic server* (SS) solution [154]. The SS scheduling algorithm protects periodic events with hard deadlines from bursts of high priority stochastic events, while being able to accord high priority to processing stochastic events. The hallmark of a sporadic server is that it provides a periodic “virtual processor” within which aperiodic events can be processed and analyzed, and bounds the invasiveness of aperiod events to the period of the virtual processor. The sporadic server algorithm can be implemented in the kernel space (i.e., in the scheduler) or in application space. We chose to implement it in application space because Pin’s support of component containers made this the natural decision. Details about SS algorithm can be found in §8.5.2, pp. 156.

Returning to the case study, the first objective to “provide suitable guarantees of hard controller deadlines” can be addressed. A_i can be converted to use the sporadic server algorithm simply by deploying it in a sporadic server container, and can be given just enough execution time to ensure that there is always at least one sub–work order in the A_i–B_i queues. The possibility of queue underflow is eliminated. Further,A_i can now be treated like periodic tasks, meaning that the collection of A_i, B_i and C are amenable to conventional GRMA schedulability analysis supported by the then available $\lambda*$ reasoning framework. Similarly, third–party component extensions can likewise be deployed into sporadic servers, and their invasiveness on A_i, B_i and C similarly bounded.

Addressing the second objective to “provide suitable performance guarantees to third–party component developers and integrators” was not so easily dispatched, because it required the development of an analytic theory for predicting average latency for periodic tasks managed by a sporadic server, something that had not previously been done, the results of which are documented in [67].

One main result of λ-SS is the so–called “Banana Curves” described by

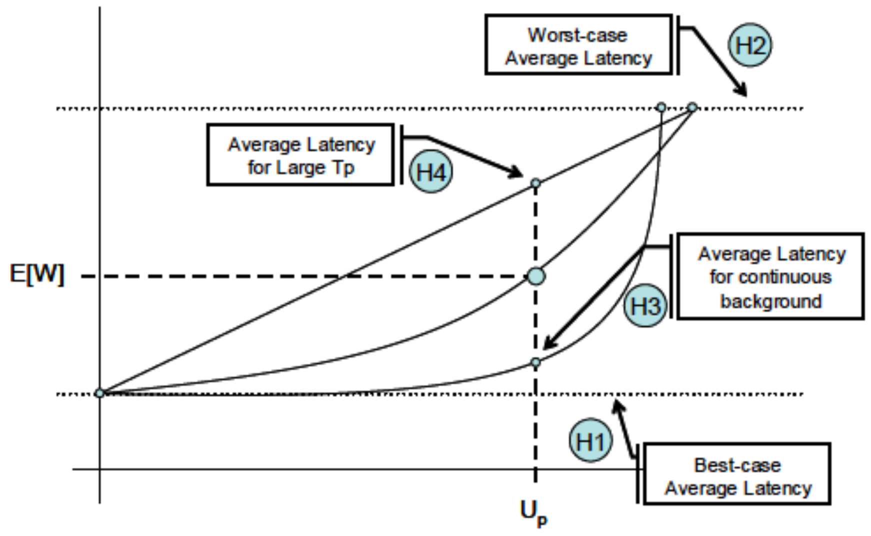


Figure 9.10: Sporadic Server "Banana Curves"

four heuristic equations, labeled H1 through H4 in Figure 9.10 (repeated from Chapter 8 for convenience). The curves describe the case of an assembly with two tasks, one periodic, and one aperiodic in a sporadic server. H1–H4 define a *performance envelope* for the average latency of the sporadic server task. These curves are discussed in 8.5, near pp. 159.

The assumption of only one periodic task is not as limited as it first appears, though. As long as the system is work-conserving (i.e., it continues to do work without idling as long as there is work to do) the priority structure and subtask structure of the periodics does not influence the average latency of the aperiodics. (See [67] for a demonstration of this fact). Therefore, for analysis purposes a set P of periodic tasks can be regarded as a single periodic with a period equal to the longest period in the task in the set, and service time equal to the sum of the service times of the tasks in the set. Because the model problem represents an extreme case where $U_p = 1 - S_{ss}/T_{ss}$, (i.e., the no background case), H2 is used:

$$\begin{aligned} E[W] &= (\frac{\rho_q}{1-\rho_q})(\frac{E[S_q^2]}{2E[S_q]}) + E[S] \\ &= (\frac{.24}{.76})(\frac{24^2}{2(24)}) + 14 \\ &= 17.78947 \text{ ms} \end{aligned} \tag{9.2}$$

The solution to H2 in Eq. 9.2 was compared to the average latency observed in many simulations of the model problem, and the predicted and simulated values for task latency fell within two standard errors:

Samples		$n = 1035$
Measured S_a	=	17.79473
Standard deviation σ	=	0.12574
Predicted $E[W]$	=	17.78947

In this case, the heuristic was quite sufficient to make an accurate prediction. If greater precision is required, however, the λ-SS reasoning framework provides a simulator that can generate arbitrarily many points between H1 and H4.

Although we never subjected λ-SS to the formal rigors of empirical validation applied to λ-ABA, the result achieved in this case is representative of our general experiences. Perhaps this should not be surprising since the theory constituents of λ-SS (λ-WB and λ-ABA) were extensively validated, and no significant new assumptions were introduced beyond those that were enforced by the λ-SS sporadic server container.

9.3.3 Model Checking Industrial Robot Code

λ-SS provided explicit, statistically–bounded assurance about certain temporal properties of components and controller assemblies. However, there were sure to be many ways that a third–party component could compromise robot safety. As mentioned in §9.3.1, the analogy between third–party extension of ORC and third–party Windows device drivers suggested the use of model checking as a useful and possibly necessary tool.

We had had mixed experiences with model checking in Stage–1 of the SAS case study, but in the interim between the conclusion of that line of investigation and the initiation of the ORC case study two conditions had changed that warranted revisiting the possibility of using model checkers to ensure safe ORC extension:

1. PCL had evolved to the point that it now supported PinCharts and code generators, which removed one level of indirection between the design specification and the model checker: we had expectations now of allowing temporal logic claims to be specified directly on PinCharts.

2. Xie, Brown and Levin [173] had developed a *bona fide* model checking reasoning framework called *ObjectBench*, which Sharygina, Kurshan and Brown had previously used to verify the behavior of a controller for a deep–space robot [148].

At this point, Natasha Sharygina (the principal investigator of the work reported in [148]) joined the case study effort, and arranged to have Object-Bench made available for our use.

Verified IPC Library: Model Problem

ObjectBench was referred to, above, as a *bona fide* reasoning framework because it composed all of the essential ingredients of a reasoning framework, as described in Chapter 8:

- An interpretation from a design notation, in this case and conveniently, xUML [118], sometimes referred to as "executable state machines."
- A behavioral theory and decision procedure, in this case the COSPAN model checker [64].
- An interpretation from the design language to the input language of the decision procedure, in this case "S/R," the input language of COSPAN.

Not surprisingly, S4 was not documented in xUML, and therefore to make use of ObjectBench we would need to create models "from scratch." Rather than repeat the experience of creating a simple synthetic component to verify (as was done for the IEC–61850 CSWI component (Figure 9.6, 185), we decided to manually extract an xUML model from "real" industrial robot code as a proxy for the kind of complexity that ORC plug–in extensions would exhibit. Not having a sample plug–in at hand, we chose instead to use a portion of the inter–thread/inter–process communication library used by S4 controllers.

The library provides operations for message-based communication among threads, which supports a variety of synchronous and asynchronous forms of communication and includes such realistic but complicating features as:

- timeouts on both sender and receiver operations
- shared memory-based message queues implementing a message-based communication style
- three different types of synchronization primitives to coordinate operations invoked by different threads

We focused on the most complicated portion of the library—those parts that deal with synchronous message exchange, with the typical use summarized in the sequence diagram in Figure 9.11. A sending thread initiates the interaction by sending a message and waiting for the answer (by calling ipc_sendwait). A receiving thread requests its next message (ipc_receive) and eventually sends a response back to the sending thread (ipc_answer).

We then worked with ABB engineers to document a range of behavior claims to verify. One lesson we took from our initial experience with model checking was to write temporal logic claims first in natural language. Several claims were defined:

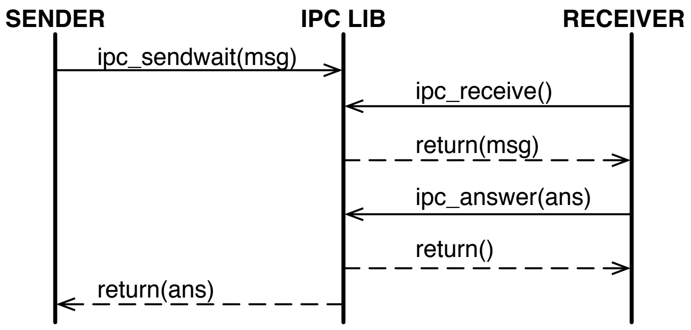


Figure 9.11: A Simple Coordination Protocol

Claim 1 Whenever a message is sent to X, X eventually receives that message (barring timeouts).

Claim 2 Whenever a message is sent to X, $Y \neq X$ never receives that message.

Claim 3 Whenever a sender receives an answer, it is the answer to the most recently sent message.

Claim 4 A sender is never blocked while trying to write to a message queue that is not full.

Claim 5 Messages (or answers) are never written to a slot that has disconnected.

Claim 6 There are never more than the maximum number of messages in a message queue.

Claim 7 LeaveCriticalSection is never called by a thread that is not the current owner of the critical section.

With the exception of Claim 7, and the reference to "slot" in Claim 5, the claims could be formulated from the pattern shown in Figure 9.11 and did not rely on the availability of source code.

Verified Components: Model Solution

A detailed account of our experiences are provided by Ivers [80], from which two main lessons can be culled.

The first lesson was an emphatic confirmation of Giannakopoulou observation that a model checker should be regarded by its users as a kind of "toolbox" for which considerable expertise in the theory of model checking and with the tool at hand are required [55].

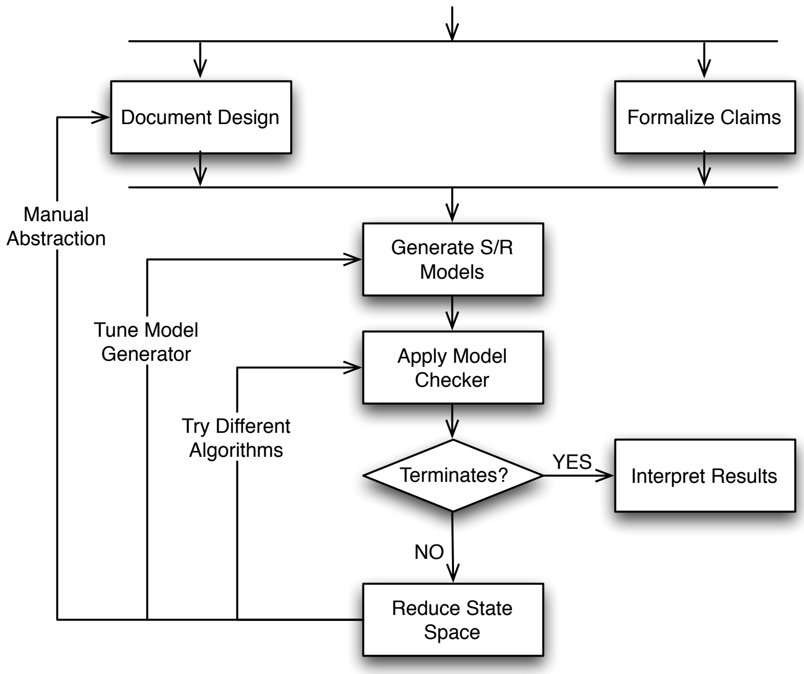


Figure 9.12: Workflow for Using ObjectCheck Reasoning Framework

Figure 9.12 shows the process of using ObjectCheck from the architect or programmer perspective. Most of the effort required to construct the model solution (i.e., to successfully demonstrate or refute the defined behavior claims) was spent in reducing state space. The first xUML model was estimated by COSPAN to have 2×10^{1932} states (!), well beyond the reach of COSPAN. Several iterations were required before the first claims could be analyzed.

The second lesson was an equally emphatic demonstration of the capability of model checkers to find deep, subtle design flaws. Claim 3 specifies an important property of a robot controller, essentially saying that different components in the controller are logically synchronized. Interestingly, ObjectCheck *falsified* the claim, and produced the *counterexample* that is shown (in abstracted form) in the sequence chart in Figure 9.13.

The first line of the counterexample shows that at step 21 in an execution trace, the first request is made by Sender (m == 1). As a result of various message timeouts (steps 49, 113, 149), Sender and Receiver lose synchronization, resulting in Sender receiving an answer to the wrong request at step 179 (reads a == 2). ABB engineers confirmed that the validity of the counterexample, and noted that this particular design flaw had been only recently discovered after being in deployment for over five years, and that it had been the source of pernicious, though rare, failure in the control system.

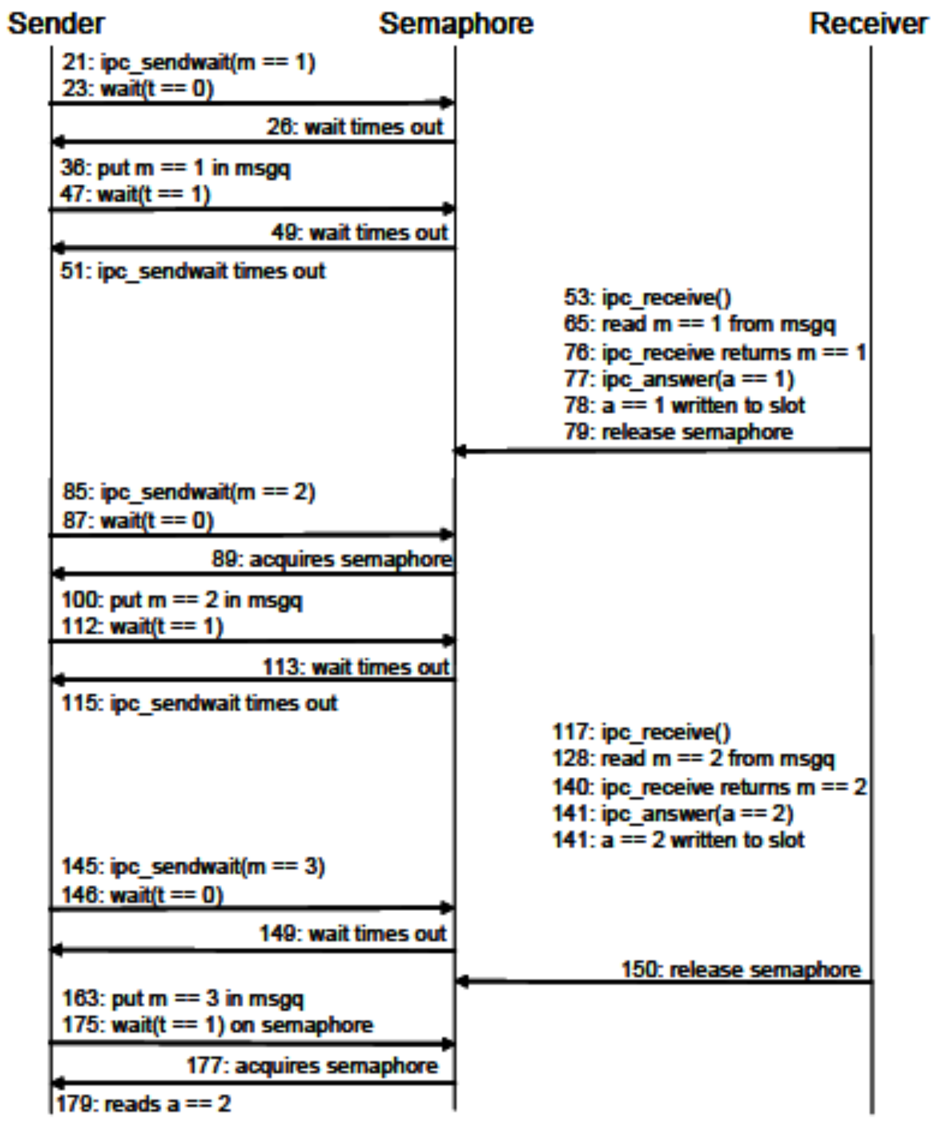


Figure 9.13: Verifiable Counterexample

The nondeterministic effects of timeouts produces too many interleavings of actions between Sender and Receiver to be tested using conventional means. While COSPAN made no guarantee that counterexample presented is the shortest one, programmers familiar with the challenges of writing "correct" multi–threaded software will recognize in this counterexample as paradigmatic of those challenges.

9.3.4 Summary of Robotics Case Study Results

The "safe extension" case study (§9.3.1) demonstrated several important results for the Seam and for PECT automation of the Seam:

- The concepts and technologies developed in a PECT for SAS were preserved intact when used for industrial robot control. This is not surprising—both share many characteristics with may kinds of embedded (and mostly periodic) real–time systems.
- With the exception of the theory for predicting average latency of tasks managed in sporadic servers, the $\lambda *$ reasoning framework was developed almost entirely from well–established prior work in real–time system analysis, as can be seen from the dates of the relevant cited publications for the $\lambda *$ theories.
- The λ-SS container demonstrated another kind of synergy between component technology and predictability by construction, by providing a natural locus for enforcing reasoning framework constraints and thereby acting as a kind of "analytic sandbox." In this case the sandbox provided reciprocal guarantees to components executing within the container, and to assemblies composed from these "analytically sandboxed" components.
- λ-SS can be regarded as an extension rather than as a refinement of λ-WB and λ-ABA. This suggests that λ-WB and λ-ABA acted, in some sense, as a toolkit for building new performance theories. This is an area for potential further investigation.

The "Model Checking" case study (§9.3.3) also demonstrated several important results:

- Model checking could be used to find subtle design flaws in industrial software, and in some cases can find flaws that can not be found by even the most extensive testing regimes.
- Practitioners can state temporal logic claims quite easily in natural language, which suggested that with modest training they could also be taught to use a more concise and formal notation for expressing claims.

- Requiring practitioners to perform manual abstraction of software to document designs in xUML inhibits (if not destroys) the transparency between design specification and analysis–specific semantics and decision procedures. This led us to model checking technology that performs automated abstraction of models from source code, and led ultimately to the ComFoRT reasoning framework available in the PSK.

9.4 Summary of Key Results

Overall, I claim that the case studies demonstrated the technical and practical feasibility of the Seam, and the use of PECT to automate the Seam and thereby achieve predictability by construction. This chapter described several applications of PECT to address non–trivial industrial engineering challenges. The next chapter steps back from the use of PECT and reflects on our experiences in the design of a PECT, and in particular in the design of the reasoning frameworks that distinguish PECT from conventional component technology and conventional model–based software engineering.

Chapter 10

Theories and Co-Refinement

In 1993 C.A.R. Hoare published an essay "Algebra and Models" in which he described what constitutes a theory (or model) of computational behavior[1], why new theories are developed, and how they are intended to be used [74]. His essay had three main aims:

1. To justify a traditional separation of concerns in computing between science and engineering, with computer scientists *developing* theories, and software engineers *using* these theories to achieve practical ends.
2. To define the main features of any theory of computational behavior, and to emphasize the ways that theories of computational phenomena differ from theories of natural phenomena.
3. To describe the *modus operandi* of the scientist–qua–theory developer, and thereby also establish normative guidelines on the direction and conduct of computer science.

In advancing the first of these aims, Hoare justified the development of a proliferation of theories "as numerous as the seeds scattered by the winds" of which "only very few will...take root." Moreover, he encouraged the development of theories that might be useful for *future* rather than *existing* (and therefore "real") problems for which he regarded new theories as necessarily arriving far too late.

However, the aims of the Seam will not be met in the scattering of innumerable theories, but in the targeted development and cultivation of those theories that serve immediate (and quite possibly limited) ends. Accordingly, our interest therefore is in Hoare's latter two aims, and in the application of his ideas to the design of reasoning frameworks.

This chapter has three sections. §10.1 introduces fundamental ideas of the design of theories of computing behavior, based on ideas described by Hoare [74]. §10.2 shows these basic ideas applied to the design of reasoning frameworks by means of *co-refinement*, using the co-refinement of the $\lambda*$ reasoning framework to illustrate the main ideas. Finally, §10.3 draws some conclusions about the co-refinement process.

10.1 Seam as Theory Design

Hoare's key ideas, and how they relate to the research described in this dissertation, are discussed in the remainder of this section. §10.1.1 discusses pragmatic concerns, touching on issues such as the scope of theories, their analytic and constructive compositionality, and their economics. §10.1.2 describes the basis of theories in observations, and emphasizes the role placed

[1]The meaning of "theory of computational behavior," which denotes any theory of any kind of observable computational behavior, is distinct from that of "theory of computation," which denotes a theory of computable functions.

in this research on the role of decision procedure in such theories. §10.1.3 describes the duality of prediction and specification championed by Hoare, and relates this to the formally weaker but more practical notion of predictability by construction. §10.1.4 relates Hoare's characterization of indirect observations and experiments to annotations and environments discussed in Chapter 6 *Pin Component Language*. §10.1.5 then relates the Hoare's notions of *satisfaction* and *specification strength* to the definition of *candidate designs* and *preference structures* described in Chapter 3 *Rational Design*. §10.1.6 relates Hoare's notion of *implementable sets* to the definition of *formally predictable behavior* given in Chapter 3. Finally, §10.1.7 highlights a missing element in Hoare's characterization with which theories can be formally related to their problem scopes, and thus made amenable to incremental development. This sets the stage for the discussion of *Co-Refinement* in §10.2.

10.1.1 Pragmatic Concerns of Theories

Hoare was unapologetic in his defense of the need to develop many speculative theories of computational behavior, and the corresponding need to accept the requisite failure of most of these to obtain any practical use. He was, however, committed to the idea that the goal of developing *any* new theory of computational behavior is, ultimately, relevance to practical computing problems. In his introductory remarks, Hoare described several areas of pragmatic concern to be addressed by the scientist. Italics are added to highlight key terms related to the Seam:

Theories Address a Defined Problem Class. A theory must address problems "which may be solved by application of some computing device" and moreover must be expressed "in the terminology in which they[2] are described."

The Ultimate Goal is Engineering Predictability. Hoare regarded the ultimate purpose of a theory to be a foundation that serves the engineer's goal "to design and implement a product which can be *predicted by the theory* to exhibit the *specified properties*."

Theories Exploit Divide and Conquer. A theory must reflect the way that "solutions to complex problems can be found by decomposition," and the way that more complex systems "can be constructed by *connecting subassemblies and components*" in some technology.

Theories are Evaluated in Practice. The value of a theory depends not only on its absolute merits, but also in "comparative cost and efficiency of alternative methods" to solving that class of problem.

[2]"the problems being addressed"

Each of these stipulations is reflected in the practice of *Co–Refinement* described in §10.2, which builds on Hoare's original ideas by forging an explicit relationship between problem class and theory (which was only tacit in [74]), and by developing practical techniques for incremental design and evaluation of theories.

10.1.2 Theories, Observations and Decision Procedures

A theory defines what can be observed, and what can be controlled. In Chapter 3, the notion of observation was ultimately reduced to that which could be modeled on an ideal computing device (see Defs. 3.9 and 3.10, pp. 56). Hoare is not as restrictive in this regard, requiring only that what is defined is "observable, controllable, or otherwise relevant" to the phenomena being described.

Hoare takes a different approach as well in his description of a theory as a collection of predicates (equations, inequalities, etc.), where free variables are observations, and where the behaviors modeled by the theory are those variable bindings that make the predicates "true." In this case there is a substantive difference with the definition of theory (Def. 3.12, pp. 58). Specifically, Hoare does not include a (semi–)decision procedure as a criterion of a theory. Predictability is defined by the Seam in terms of an ideal computing device both as a way of ensuring soundness *and* automation.

Hoare was not indifferent to the need for automated analysis, but likely regarded that as an inessential detail in a discussion of how theories are constructed. He refers to the desirability of decision procedures (e.g., term rewriting) for finite systems, but eschews the possibility of decision procedures for all but highly constrained recursive systems. This is perhaps the only area of Hoare's essay that has been invalidated (at least in part), in this case by advances in model–checking technology, including (with some irony) the Failure Divergence Refinement (FDR) model checker for Hoare's CSP [142].

10.1.3 Preconditions, Predictions and Specifications

Hoare distinguishes a class of predicates he calls *preconditions*, which refer to observations that are under the direct control of the end–users of a system, or the environment in which the system executes. Preconditions in this sense correspond to the formal invariants of predictable phenomena in the Seam (Def. 3.11, pp. 57). Both Hoare and the Seam are a bit liberal in the interpretation of precondition and formal invariant, however. Hoare equivocates, stating that preconditions "*usually* mention" observations, and they are "*generally*" in control of the environment, etc., while the Seam regards formal invariants as syntactic rules of well–formedness in an interpretation, or anything else that might be "*appropriately* demonstrated" (see discussion of Def. 3.10, pp. 57).

Hoare's formulation of observation and theory provide elegant and symmetric definitions of *prediction* and *specification*: A set of predicates that has all free variables bound, and which (as a set) is satisfiable[3] can be regarded simultaneously as a *prediction* of the behavior of a system described by those predicates, and as a *specification* of the behavior of a system that has yet to be produced. Hoare has in mind that a plausible goal of an engineering design is to establish the following theorem:

$$D \Rightarrow (P \Rightarrow \neg FAIL \wedge S) \tag{10.1}$$

Here, D denotes some design (what Hoare calls a "delivered product"), P denotes preconditions of a theory and S denotes a specification of required behavior. Hoare was adamant that a theory needs to describe not only the behaviors to be achieved, but behaviors to be avoided as well, which he aggregated into a singe predicate $FAIL$. Hoare gives as his interpretation of Eq.10.1:

> "*. . . if the precondition P is satisfied, then every observation of the behavior of the delivered product D will be a non–failing observation, and will also satisfy the specification S.*"

It is worth pointing out that Eq. 10.1 is an abuse of notation that confuses sets (of observations) with the predicates defined by a theory that define those sets. With this slight abuse in mind, Eq. 10.1 imposes a proof obligation on the designer, who must demonstrate that preconditions are satisfied and that the implication holds.

Note that Eq. 10.1 has the effect of demonstrating that a design is a *candidate solution*, as discussed in Chapter 3 *Rational Design)*. Predictability by construction in the Seam has a more modest aim than Hoare, which however can be expressed in analogous terms:

$$D_{\text{imp}} \Rightarrow (P_T \Rightarrow \exists Interp(D_{\text{desc}}) \in T) \tag{10.2}$$

with the interpretation that if the preconditions of a theory T are satisfied, then the observable behavior of the product implementation D_{imp} are observable in T under a semantic interpretation $Interp$ of that product's formal description D_{desc}.[4]

Eq. 10.2 does not establish any facts about D_{imp} other than that it's behavior is predictable in T. This is another way of stating that predictability

[3]There is *at least one* possible observation for some specified variable/value bindings

[4]*Description* is used instead of the more natural *specification* to avoid confusion with Hoare's use of that term.

describes a semantic relation between designs and theories, but it need not (though of course it could) establish any theorems about D_{imp} by way of $\mathit{Interp}(D_{\text{desc}})$. The aim of predictability by construction is to ensure that only analyzable designs are produced, i.e., are *testable* in Simon's *generate and test* operationalization of "design as search" as described in Chapter 3 *Rational Design*. The evaluation of those designs is a different matter.

10.1.4 Direct and Indirect Observations

Hoare notes that while "end–user specifications" (a term he does not elaborate but which is more or less clear) are generally stated in terms of directly–observable behavior, the value of theories lies in their ability to make *indirect* observations; in the performance reasoning framework in Chapter 8, execution time and preemption are direct and indirect observations, respectively. He goes on to make two points about indirect observations that are pertinent to the later discussion of co–refinement.

First, he notes that confirming or refuting a theory requires a sometimes complex experimental apparatus, the behavior of which (as understood by the experimenter) might also depend on the theory being developed. There are many practical consequences of this dependency. One, which is not further elaborated but is worth mentioning, is that Seam theories, the static tools and runtime environments that discharge their assumptions, and the apparatus used to validate theories all undergo simultaneous development. It is in the nature of things that all must therefore be simultaneously "debugged." Although debugging is itself something of a black art, our practical experience suggests that having a reasonably well thought out theory of behavior is itself immensely useful in finding errors in tools and environments. In most cases, discrepancies between observed behavior and behavior predicted by a theory was due to errors in the Pin runtime environment rather than a problem with the theory.

Hoare also notes that what are regarded as direct observations at one level of system organization can be indirect observations at another level, possibly (but not necessarily) of another, more basic theory. Using performance again, average-case latency can be indirect at one level (combining execution time and preemption effects over the hyperperiod of a set of periodic tasks) and direct at another (a measure of an assembly). Or, analogously, a temporal logic claim on a component implies a host of indirect observations in the underlying model checking theory, but once a claim has been established it can be regarded as a directly–observable behavior of a component.

10.1.5 Correctness, Preference and Tactics

In Chapter 3, design was expressed as a search problem, with functional correctness a necessary condition of a candidate design solution, and a preference structure to rank competing candidate solutions (candidate designs and preference structures are discussed in §3.2 on or near pp.49). The preference structure is a proxy for subjective design judgement, but it is nonetheless based in theory observations such as those discussed by Hoare.

However, Hoare uses the apparatus of set theory to move beyond "correctness." In Hoare's set–theoretic interpretation of theories, both S and D in Eq. 10.1 describe sets of observations, and discharging the proof obligation of Eq. 10.1 involves, in essence, demonstrating that $D \subseteq S$, i.e., that only those observations that are described by S are observable in D. However, specifications can be ordered by subset inclusion:

$$D \subseteq S_n \subseteq S_{n-1} \subseteq S_{n-2} \cdots \subseteq S_0 \tag{10.3}$$

where each S_k can be considered as "stronger" than all $S_{j<k}$ by virtue of its permitting fewer observable behaviors. Should any particular S_m be a candidate solution, then each $S_{k>m}$ can be regarded as "preferable" in some sense to S_m (more deterministic, faster, etc.). Thus, the same theory that establishes "correctness" can also serve as the preference relation $\preceq$ that operationalizes "design as search." Hoare is quite careful to make particular note of circumstances where tradeoffs among behaviors are required, which he regards as necessarily "left to the good judgement of the engineer. No amount of mathematical theorising can ever replace that!"

Hoare asserts that "one of the main objectives of a mathematical theory is to provide a comprehensive collection of such correctness–preserving, but efficiency–increasing transformations." This research regards such efficiency–increasing transformations, elsewhere called "tactics" ([13, 145]) as a complementary agenda, one that requires predictability by construction as a precursor. The nature of the complement is hinted at in Step #3 in the PECT use scenario sketched in Table 4.2 on or near pp.72.

10.1.6 Implementable Sets

In Hoare's characterization, theory T defines a universe of possible observations; a specification $S \subseteq T$ defines the acceptable observations of any satisfying product design; and the engineer's goal is to implement a product $D \subseteq S$ that will satisfy S along the lines of Eq. 10.1. But how much liberty does an engineer have in choosing D? Indeed, how does the theory designer establish that for all *interesting* S there exists at least one implementable D, where "interesting" can be taken to mean of practical significance to the class of problems that T purports to address?

To address this issue Hoare introduces the idea of *implementable sets*, which he calls "PROC." Each PROC defines a family of specifications that are "implementable in a particular envisaged language or technology," i.e., a language or technology that is already at hand or that is perhaps conjectured by T.

> "*The conditions defining membership of PROC... is the first and most serious difficulty in the construction of realistic models; what is worse, their sufficiency and validity can be established only at the very last step in the evaluation of the model by practical use. That is why Dana Scott once characterized formalisation*[5] *as an experimental science.*"

Implementable sets describe essentially the same idea as *predictable* in (Figure 1.2, pp.22) and *predictive range* (Figure 4.1, pp.71), and also captures the same idea of *formally–predictable behavior* (Def.3.13, pp.58).

10.1.7 Incremental Theory Refinement

As Hoare describes matters, the criteria that define implementable sets arise from intuition and inspired guesswork. There is no formal connection made between these criteria and the class of design problems (§10.1.1) addressed by the theory. For example, the problem class can itself be represented as a set of observations $PROB$ that defines the problem set, which is interpreted as all possible observations that are interesting to the class of problems at hand. This, in turn, can serve as a specification for the theory, with the objective:

$$PROC \subseteq PROB \subseteq T$$

for some targeted collection of implementable sets. Without such a specification there can be no basis on which theories can be systematically designed, but rather they can only invented and, at the last step, confirmed.

A clear specification of $PROB$, however, is not itself sufficient to resolve the difficulty observed by Hoare and Scott: here the criteria that defines $PROB$ rather than PROC can only be tested at the last instant.

What is needed for the design of theories is an analogous notion to that of *refinement* introduced by Hoare in his CSP process algebra: each successive design preserves correctness (refines its specification) but also reduces non-determinism, increases efficiency, or makes some other incremental improvement that is a step towards some ultimately satificing design. In this way the algebra captures the essence of incremental design and verification. The sequence:

$$\begin{aligned} &PROC_f \subseteq T_f \\ &\qquad \dots \\ &PROC_1 \subseteq T_1 \\ &PROC_0 \subseteq T_0 \end{aligned} \tag{10.4}$$

[5]"theory development"

captures the intuition that, starting from some simple base theory T_0 some ultimately acceptable final theory T_f can be arrived at through a succession of intermediate theories. This formulation is no more abstract than Hoare's original characterization of $PROC$; it simply adds incremental theory development.

Of course, the main caveat is that for this abstract scheme to be practicable, there must be substantial benefits to designing and validating a sequence of theories instead of heading for the final one straightaway. This may explain in part why Hoare did not regard incremental theory development as a matter requiring discussion.

However, if the Seam is to enable a sustainable engineering practice, its major components—reasoning frameworks—must be practically susceptible to incremental design and verification. For this purpose we require:

- an understanding of the nature of the relation between each $(T_k, T_k{+}1)$, $(PROC_k, PROC_k + 1)$ and similarly for each $(PROC_k, T_k)$;
- effective heuristics with which a theory designer can construct T_{k+1} from T_k, and each$PROC_{k+1}$ from $PROC_k$; and,
- a preference relation $RF_k \preceq RF_{k+1}$, $RF =< PROC_k, T_k >$ with which to initiate a search for a suitable T_f whose features may not be well understood (as discussed in Chapter 3 *Rational Design*).

A method of *Co-Refinement* described next offers clues about each of these.

10.2 Seam as Language Design: Co-Refinement

The invention of a new theory of computational behavior is no doubt challenging, and perhaps Hoare is correct that this is an area of intellectual activity for which there is no clearly reducible, systematic practice; there may only be that fortuitous mixture of knowledge, skill, luck and intuition that characterizes the successes of leading computer scientists of our age.

However, the Seam is an engineered construct, and as such is meant to operate within the practical bounds of a range of well–defined and recurring design problems. In this context, designing a small number of fundamental theories of behavior is of less concern than designing a collection of highly specialized but otherwise "good enough" theories.

With this in mind we take inspiration for the design of *reasoning frameworks* from Wirth's "stepwise refinement" for the *design of computer programs* [172]. Wirth's prescription inspires because it provides a practical technique for solving programming design problems of realistic complexity, and also because it provides a pedagogy for teaching that design process to future practitioners. This is not to say that stepwise refinement is sufficient for all design problems, but it is without question an essential design skill for software engineers.

We also take inspiration from modern applications of type theory in programming language design and program analysis. Robert Harper expressed the key idea quite succinctly when he observed, first, that "restrictions entail stronger invariants," and, second, that "flexibility arises from controlled relaxation of strictures, not from their absence".[6] These two points combine quite nicely. The first suggests a way of finding what we have elsewhere called "smart constraints" [73] where "smart" refers to those constraints that, when enforced, lead to predictable system behavior. As Harper also remarked, "well–typed means well–behaved".[7] The second suggests a direction for refinement, with each refinement relaxing stricture.

We coined the term "co-refinement" to describe the stepwise refinement process used to develop the initial version of the $\lambda*$ reasoning framework [68]. The "co" in co-refinement expresses the idea that each of $PROC$, T, and $Interp$ is simultaneously refined, in the direction of relaxing constraints, and in the context of the refinement of the others. As with stepwise refinement, co-refinement is a heuristic rather than a mathematical notion (in contrast to *refinement* in Hoare's CSP [75] or to, say, *co-induction* [85]).

The remainder of this section describes co-refinement in more detail. §10.2.1 provides background on the initial conditions of co-refinement. §10.2.2 through §10.2.7 summarize the co-refinement steps leading, ultimately, to the $\lambda*$ reasoning framework. Much of this material was documented in a contemporary account of the work [68], but is substantially extended here to reflect later experiences.

10.2.1 Background on Co-Refinement of $\lambda*$

At the beginning of work in the substation automation "protection and control" case study in 2002, the broad outlines of predictability by construction were understood:

- The IEC61850 specification for substation automation systems [78], combined with IEC1131-3 function block notation [101] in the industrial tool chain, seemed ideally suited to a component technology such as Pin: IEC61850 described a "domain model" of substation automation functionality in terms of logical components, while IEC1131 syntactically mirrored the Pin component model.

[6]"The Practice of Type Theory in Programming Languages," presented at Dagstuhl 10th Anniversary Symposium [171]. Presentation available online `www.cs.cmu.edu/~rwh/talks/Dagstuhl` (last accessed August 3, 2010).

[7]For consistency with Harper this catch–phrase is equivalent to "smart restrictions" or "smart strictures."

Step 5: λ-ABA(**Average case, Blocking, Asynchrony)**		
	• synch/asynch pins	• pin re-writing • spot check
Step 4: λ-AB(**Average case, Blocking)**		
• execution jitter • monte carlo		• full validation • assembly labels
Step 3: λ-WB(**Worst case, Blocking)**		
• varying task priority	• priority ceiling • (un-)/threaded • (non-)/reentrant	• spot check • component labels
Step 2: λ-A(**Average case)**		
• non-zero phasing • hyperperiods • simulation		• measurement
Step 1: λ-W(**Worst case)**		
• periodic • deadline < period • non–blocking	• unthreaded only • reentrant only • (synch pins only)	
Step 0: **Starting Points**		
• GRMA[89]	• Basic Pin	• No toolchain

Table 10.1: Five Stepwise Iterations to λ-ABA

- Preiss and Wegmenn used IEC61850 to define a number of model problems for predictability by construction [140], including protocols that might be verified with temporal logic model checking, and performance requirements that might be analyzed with rate monotonic analysis.

While we suspected that predictability by construction would prove to be *technically* feasible, but there were many unknowns about its practical feasibility:

- Did the proximity of Pin, IEC61850 and IEC1131 extend beyond surface syntax? That is, could Pin be a key Seam abstraction for substation automation systems, or would Pin components prove to be too constraining (or too "heavyweight") for this purpose?

- Were the measures on extra-functional behaviors, and measures of confidence in these described by Preiss and Wegmenn achievable? Were these normative measures or could they be traded against other design qualities such as ease of programming?

- Could implementation constraints be imposed on industrial substation automation system designers and developers? Could "increased confidence" be traded for a "less convenient programming" for example?

Matters were complicated still more because of dependencies among these (and similar) questions. We did not know how to assign business value to predictive confidence, for example whether there was any difference in value between a 95% and 99% confidence interval on average–case latency, and therefore whether the added cost and complexity of modeling low–level platform details such as context–switching time, or introducing real–time scheduling to a base Windows platform made "business sense." The detailed design objectives and tradeoffs for a performance reasoning framework would need to be discovered—and addressed—while exploring a largely uncharted design space for reasoning frameworks in general, and for $\lambda *$ in particular.

Table 10.1 summarizes the co-refinement sequence from an initial understanding of the design problem to an automated, validated performance reasoning framework. Each step in the sequence (other than *Step 0*) corresponds to a discrete theory in $\lambda *$. The table is partitioned into three vertical columns which highlight for each iteration its effect on (from left to right) the theory, on design rules, and on validation, respectively.

We defined several criteria for choosing each next step in the design process:

Valuable. Each step must deliver an analytic capability that is useful to end–users.

Predictable. Each step must define an interpretation from the component model to the analytic theory; and the predictions made the theory, and each independent variable used by that theory, must be *in principle* observable by measurement or by analysis.

Progress. Each step (except the first) must extend the prior step in one or more ways, by generalizing the analytic theory, or relaxing design constraints, or improving predictive quality.

"Component model" in the criteria refers to the Pin component model *and in addition* any theory–imposed design rules that constrain the use of Pin at some co-refinement step. Where no confusion will arise, "Pin" will denote the Pin component model, and "Pin$_k$" will denote the component model at *Step* k. Analogously, λ_k will denote the performance theory at *Step* k.

10.2.2 Step 0: Starting Points

At the outset, the Pin component model had only just been implemented. In its early form, Pin supported a variety of pin types and basic hierarchical assembly (see [82]), but did not support real–time scheduling. There were no code generators or measurement infrastructure.

Substation automation would require worst–case performance analysis to guarantee that critical *protection and control* task deadlines are met, and the periodic nature of many of the tasks strongly suggested generalized rate monotonic scheduling theory [89] (“GRMA” in the table) as a theory foundation.

However, control loops involving the operator, along with a stated business objective to explore “soft” protection and control (i.e., use of Windows or other non–real time platforms with fast processors as sufficient to ensure deadlines) also suggested that some form of average–case latency would be important. It was unclear whether GRMA could be directly applied or be generalized for this purpose, or whether an alternative theory such as Lehoczky’s real–time queueing theory [49] (RTQT), or some hybrid of GRMA and RTQT would be required.

Although we did not know in detail what sort of reasoning framework would emerge, we did define three elementary “model problems” that any candidate solution must solve. Each model problem is thus an exemplar of a class of industrially significant problems, for substation automation in particular but for other real–time systems domains too. Each model problem uses real (though sometimes simplified) IEC61850–defined functionality, and IEC61850–defined quality measures for predictions. See §9.1.2 pp. 175 for a discussion of statistical techniques used in this research, and Eq. 9.1, pp. 176 for the definition of magnitude of relative error (MRE).

The three model problems and the required predictive quality of their solutions are:

1. Controller Assemblies: Predict timing behavior of one or more protection and control functions, implemented in one Pin assembly, executing on a dedicated, single computing device. The required confidence interval is 0.99 confidence that predictions will exhibit, with 0.80 probability, a MRE less than 0.05.

2. Operator Station Assemblies: Predict timing behavior of an assembly of human/machine interface (HMI) components, at least some of which manage controller assemblies. The required confidence interval is 0.95 confidence that predictions will exhibit, with 0.80 probability, a MRE less than 0.10.

3. Substation Assemblies: Predict timing behavior of assemblies of controller and operator station assemblies that communicate using a local area network. The required confidence interval is 0.95 confidence that predictions will exhibit, with 0.80 probability, a MRE less than 0.10.

10.2.3 Step 1: Establish λ_1 Worst Case Non-Blocking Latency

The first step resulted in a working base case that could be further refined. Two considerations were dominant:

1. We knew that Pin was going to need to be re-hosted from native Windows to some other platform if it were to support predictability in a real–time setting. Consequently, focusing the early iterations on only the most basic real–time predictions seemed both prudent and useful as a way of testing rehosted Pin versions.

2. We also knew that worst–case latency and schedulability analyses were going to be critical elements of performance analysis for control systems, so these were likely candidates for early implementation.

For Pin$_1$ we restricted components and assemblies to the bare minimum. Only *unthreaded* reactions were allowed; each reaction would ultimately execute on a single thread provided by the environment *clock* service. Restricting components to unthreaded reactions also implied restricting components to *synchronous* pins. We had also yet to decide whether to use *priority inheritance* [147] (which would require support of the Pin runtime) or *priority ceiling* [56] (which could be enforced by the interpretation). For this reason we deferred the need to make a decision by further restricting components to "non-blocking" sink pins.

For the highly restrictive Pin$_1$, λ_1 required only the most basic elements of GRMA for worst–case schedulability analysis. Eq. 10.5 summarizes the theory used to predict the worst case latency of the i*th* task by finding its fixed point (i.e., $L_{n+1} = L_n$:

$$L_{n+1} = \sum_{j=0}^{i-1} \left\lceil \frac{L_n}{T_j} \right\rceil C_j + C_i \tag{10.5}$$

where C_i is the execution time of the i*th* task, T_i is its period, and C_j has higher scheduling priority than C_k if and only if $j < k$.

Formal predictability was demonstrated with a combination of pencil–and–paper exercises and spot–checking of predictions. A pencil–and–paper interpretation was defined without difficulty: the execution time of task C_k was defined to be the sum of Pin component execution times of the *hierarchy* of Pin components rooted at a pin component C'_k, with the priority of C_k retrieved from an annotation of a C'_k sink pin. The period of C_k would be obtained from the annotated source pin of an environment–provided clock service (essentially, an event generator). We also defined a protocol for measuring component and assembly execution times, and used this to spot check predictions.

Because the theory was so basic we were certain that any variance between predicted and actual latency would arise due to a programming error in the Pin infrastructure. Needless to say, such errors did exist, and strong confidence in theory predictions were quite useful for tracking down problems.

10.2.4 Step 2: Generalize λ_1 to Average Case Latency

Step 2 was almost entirely concerned with generalizing λ_1 to predict average–case latency, and therefore had no consequential impact on the Pin_1 (beyond "debugging" the implementation).

Conceptually, generalizing to the average case was straightforward once it was observed that the pattern of preemption for task C_k repeats every $NP = LCM(T_0, T_1, \ldots, T_{k-1}, T_k)/T_k$ periods, where NP is the number of periods in the *hyperperiod* of that task, and LCM is "least common multiple" of the task periods of C_k and all higher–priority tasks than C_k. The average latency of C_k then is the average of each instance of C_k (each "job") in its hyperperiod, within which only one task will experience the "critical instant" where all higher priority jobs become ready at exactly the same time.

As with *Step* 1, predictability was demonstrated with a combination of pencil–and–paper exercises and spot–checking of predictions. However, when spot–validating λ_2 we were surprised that predicted latency did not match the observed latency.

What we discovered (without too much investigation) is that in Pin the first job of each task (for all but the highest priority task) did not begin instantly at time 0 as was modeled in λ_2, but rather at a later time—at some *task offset*. At application startup the highest priority task C_0 is "launched" by setting the clock timer for that task. The clock will not generate its first event until T_0 time has expired, and thus C_0 will not begin until at least one period has expired, and so forth for the other tasks.

The problem, such as it was once diagnosed, was resolved by allowing the reasoning framework to communicate to the measurement infrastructure the number of periods to skip for each task before recording time measurements. In this way the preemption patterns predicted by λ_2 would be exactly aligned with the patterns experienced in the Pin_2 runtime. In retrospect this was a simple problem, but at the time we were relieved to so easily diagnose the problem; as mentioned earlier, this was possible because we had a performance theory that we trusted and that provided us with useful "hints" on where to look.

10.2.5 Step 3: Generalize λ_2 for Blocking

The generalization of λ_2 to include blocking allowed us to relax Pin_2 design rules that prohibited the use of threaded components, as well as the requirement that all sink pins (more accurately: all *reactions*) had to be reentrant. Recalling that these design rules were introduced exclusively to rule out potential blocking behavior, then *Step* 3 can be understood as primarily motivated by the need to provide software developers with a more expressive Pin_3 component model.

The main outlines of the λ_3 theory were in fact recognizable even as

Step_1 was underway. Mark Klein, one of my collaborators in this research and an inventor of GRMA, recognized the similarity of Pin assemblies with the "concurrent pipeline" architectural style that was used as a canonical example in earlier work on attribute–based architectural styles [88] (ABAS). In the concurrent pipeline ABAS Klein *et al* demonstrated that analysis of worst–case latency can be obtained from a prior GRMA result that modeled tasks that have time–varying priority [62].

Thus, in concurrent pipelines the worst–case latency of the i*th* task is again computed by finding its fixed point:

$$L_{n+1} = \sum_{j \in H} \left\lceil \frac{L_n}{T_j} \right\rceil C_j + C_i + \sum_{j \in HL} C_j^H + max_{j \in LH}(C_j^H) \qquad (10.6)$$

where, given $C_{i,low}$ is the lowest priority task in C_i, H, HL, and LH partition the pipelines into sets:

- H: "High" pipelines for which all tasks have a higher priority than $C_{i,low}$
- HL: "High then Low" pipelines which begin at a higher priority than $C_{i,low}$ but become lower than $C_{i,low}$
- LH: "Low then High" pipelines which begin at a priority lower than $C_{i,low}$ but become higher than $C_{i,low}$

Eq. 10.6 simplifies matters a bit because it omits details of a preliminary step that is needed to translate an arbitrary set of pipelines into semantically–equivalent (for latency calculations) canonical H, HL and LH assemblies. It also does not describe details of the iteration on each pipeline. However, comparing λ_1's theory in Eq. 10.5 with the ABAS theory in Eq. 10.6 shows the combined simplifying effects of restricting Pin components to unthreaded reactions and non–blocking sink pins: only the first two terms of Eq. 10.6 are required by λ_1. It would have been possible to define λ_3 using Eq. 10.6, and relax Pin_2 to allow threaded reactions but maintain the restriction to non–blocking sink–pins. However, we were sufficiently confident from our earlier progress to relax Pin_2 further to permit blocking sink pins. Thus, λ_3 needed to accommodate pipelines that might be preempted by higher–priority tasks, and might also *block* on shared resources.

An interpretation had also become necessary for λ_3 precisely the relaxed design rules in Pin_3 significantly enlarged the constructive and predictive range of the reasoning framework (what Hoare called, in aggregate, "implementable sets"). Complex acyclic assembly topologies could now be constructed and analyzed, although their timing behavior sometimes defied intuition; at this point questions about how the theory would be validated began taking on extra prominence.

One key decision made at this point was to use priority ceiling rather than priority inheritance as a design rule in Pin_3, in large part because this could be enforced at the application–level through interpretation rather than requiring Pin runtime support and hence introducing a significant platform dependency. The interpretation constraint was simple to implement: if two Pin components pc1 and pc2 synchronize on the same *non–reentrant* Pin component pc3 (i.e., the assembly contains both pc1:r$\rightsquigarrow$>pc3.s and pc2:r$\rightsquigarrow$>pc3.s), then the interpretation ensures that the priority of pc3 is higher than both pc1 and pc2. Priority ceiling emulation is slightly more intrusive on the designers, but otherwise its advantages outweighed its disadvantages at this point in the development of the Pin.

10.2.6 Step 4: Generalize λ_3 for Average Case

Although we did not realize it at the time, *Step* 4 was to be the last major hurdle in developing a performance reasoning framework suitable for soft protection and control of substation automation systems. λ_3 and Pin$_3$ covered a significant range of possible designs; spot checks of predictions for complex assembly topologies seemed to agree with observed assembly behavior, although the statistical correlations had not been established. And hand–checked assemblies also agreed with our expectations of what the theory ought to predict for certain “edge” cases.

When computing worst–case latency, the computations sketched in Eq. 10.6 needed to be concerned only with finding so–called “critical instants,” when all tasks of higher priority than a given task are simultaneously ready to execute. For average case latency, many more interleaved executions needed to be examined. For this reason, a purely analytic approach to the λ_4 decision procedure became unwieldy, and discrete event simulation was used to implement the reasoning framework decision procedure. At this point of the co–refinement process, the analytic theory, its simulator, a measurement infrastructure, component technology and substation automation applications were undergoing simultaneous development.

Significant attention was also now being paid to establishing the predictive quality of λ_3, which we expected to be already close if not already sufficiently accurate, at least with respect to the relatively generous target confidence intervals defined in the model problems.

During this step, Magnus Larsson developed a validation approach that combined judgment sampling (using substation automation experts to define key assembly characteristics) and random assembly generation [98], and we began extensive validation of λ_4. Not surprisingly, we were easily able to satisfy the statistical requirements for predictions, achieving a MRE somewhere between 0.05 and 0.075, that is, between 5–7%, for the required confidence and population parameters specified by the model problems.

It is worth highlighting the role of the empirical validation infrastructure developed by Larsson (a summary of which is depicted in Figure 10.1). As observed by Hoare, an obvious challenge is to develop an experimental

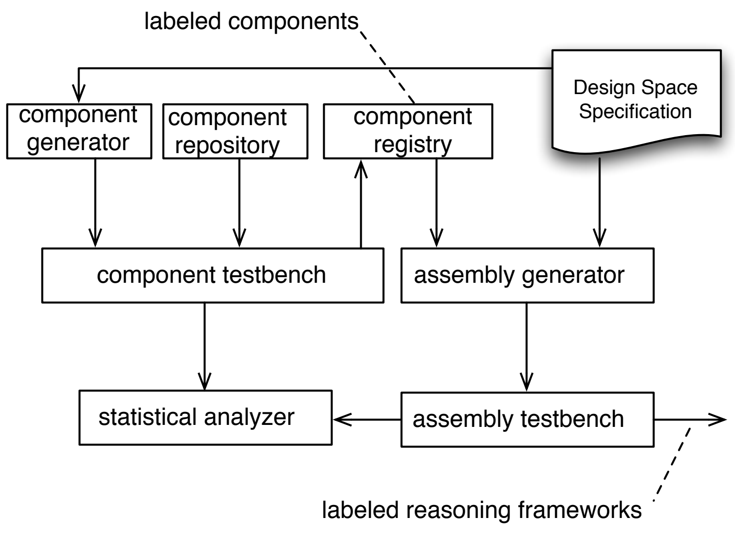


Figure 10.1: Measurement Infrastructure and Workflow

apparatus that depends to large extent on the nature of the theory being validated (see 10.1.4). However, our experience suggests that the discipline of creating such an infrastructure forced us to be more precise about tacit theory assumptions, such as identifying which among the various types of Pin events (sink or source events, begin or end interaction events) to use as measurement anchors.

Another observation is that the techniques used by Larsson to define the sample space of components and assemblies can be regarded as a systematic way of defining Hoare's *implementable sets* (see 10.1.6) of a theory. Just as the development of the experimental apparatus imposed discipline that reinforced theory development, so, too, the design space specification required our industry collaborators to be more concrete about the scope of a theory—what kinds of topologies should be expected, and what range of behaviors (in this case, defined by the $\lambda*$ theories being developed) to expect from components and assemblies.

Returning to the spot validations of λ_4, although we had satisfied the statistical requirements for prediction quality, we were far from satisfied with the results. In fact, we were eager to understand why the predictions were not at least one order of magnitude more accurate-. What aspect of the Pin runtime environment had we not modeled? Examining the validation cases also showed that while some assemblies had an exceedingly small MRE, there were outliers that exhibited significant error, i.e., an MRE of 0.20 or more.

From this, and from measurement logs, Gabriel Moreno was able to isolate the problem to assemblies where low priority task would complete execution near the release time of a higher priority task. Variance in execution

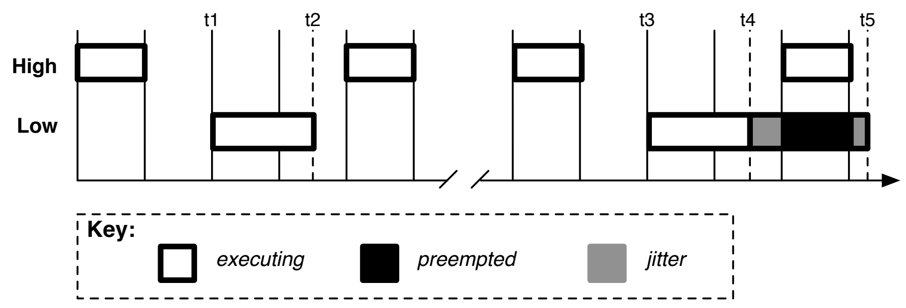


Figure 10.2: Effects of Execution Jitter on Latency

time might result in a low priority task failing to complete before being preempted by the high priority task. The problem is represented schematically in the bottom pair of task shown in the scheduling timeline fragment in Figure 10.2, where as a consequence of jitter the first (jitter–free) latency of the low priority job is t2 − t1, while later the effects of jitter result in job completing past its intended release point, and possibly beyond its deadline, with latency $t5 - t3 \gg t2 - t1$.

There were several approaches to remedy the problem. Perhaps the most principled approach would have been to ferret out the source of interference that caused execution jitter—perhaps it was caused by some unidentified high priority Windows process that was interfering with the scheduler? or possibly cache effects were being experienced? Instead, the λ_4 simulator was extended with Monte Carlo simulation using the execution bounds (for coverage factor k=2 or 95% confidence) *and* execution distribution for component execution times. The result was highly satisfactory, with λ_4 substantially surpassing the quality requirements established in *Step* 0 to achieve 0.99 confidence (original requirement: both 0.99 and 0.95), with a probability of 0.80 (unchanged), that predictions will exhibit a MRE less than 0.01 (original requirement: between 0.05 and 0.10, with an average MRE of 0.005.

10.2.7 Step 5: Generalize λ_4 to Asynchronous Pins

Pin$_4$ was sufficiently expressive for substation automation, and was a close fit to how control functions were written and composed in at least one IEC1131–based tool chain being used by our industrial collaborator. Still, we were somewhat dissatisfied at losing asynchronous pins, and begin looking into how to extend λ_4 to handle asynchronous as well as synchronous pins.

At this point Gabriel Moreno had a remarkable insight: any Pin assembly that contained asynchronous pins and interactions could be replaced by a semantically–equivalent (using λ_4 as the semantic theory) assembly that contained only synchronous pins and interactions.[8] This was formalized as a Pin rewrite grammar (essentially, a syntax–directed tree–transformation grammar) and included in the interpretation. Details of the rewrite grammar are provided in [68], and are not repeated here. As a consequence, Pin_5 was relaxed from Pin_4 to allow asynchronous pins and connections, and λ_5 was unchanged from λ_4, but was appropriately re–christened as λ-ABA.

10.3 Learning from Co-Refinement

The challenge of incrementally designing reasoning frameworks is evidently far larger in its social scale, and messier in its technological detail, than the design problems described by Hoare in *Algebra and Models* or Wirth in *Program Development by Stepwise Refinement.* Two broad lessons stand out from our experiences with co–refinement of both the $\lambda*$ and ComFoRT reasoning frameworks. The first, discussed in §10.3.1, is in the nature of the design forces that influence the direction of the search through the reasoning framework design space. The second, discussed in §10.3.2, is in the engineering roles involved in co–refinement, and how different roles interact with one another, and how they influence, and are influenced by, the co–refinement design forces.

10.3.1 Co-Refinement Design Forces

Christopher Alexander observed that, in nature, a tree can be regarded as a diagram of the forces that have acted on it [6]. The design of certain classes of software system will be strongly influenced by characteristic design forces—cost, schedule, safety, etc.—though these will differ from domain to domain.

Three forces appear to most strongly influence the design of reasoning frameworks: generality, predictability, and practicability:

Generality. This force applies to the constructive and analytic constituents of a reasoning framework. Increasing generality of the constructive constituents means increasing the extent of the reasoning framework's implementable set, and is achieved by relaxing stricture. Increasing generality of the analytic constituents means increasing the extent of a theories observations, and is achieved by relaxing assumptions.

Predictability. This force applies to the interpretation and validation constituents of a reasoning framework. Increasing predictability means increasing the coverage of interpretation, and is usually achieved by reducing the extent of the set of assemblies that are wrongly rejected as "ill–formed" to a theory.

[8]This required that no two tasks having the same priority could never be ready to execute simultaneously, a condition that was not difficult to enforce.

Practicability. This force applies to how the extent of implementable sets is defined (i.e., the "application design space" supported by the reasoning framework), how much transparency or opacity must be achieved between the reasoning framework theory and the users of the framework, and how normative goals for the quality of reasoning framework predictions are defined.

These forces interact in the characteristic ways shown in Figure 10.3. Predictability and practicability are mutually reinforcing—increasing (decreasing) one generally results in increasing (decreasing) the other. Generality, however, tends to work against predictability and practicability.

To illustrate, consider GRMA and timed automata, which are comparable in the sense that both may be used to predict real–time behaviors. Without question, GRMA is a "weaker" theory than timed automata in that it makes many more assumptions about the environment (for example, the apparatus for priority–based scheduling) than timed automata, and makes far fewer observations (for example, largely confined to scheduling points). On the other hand, timed automata exhibit quite poor practicability, due in large part to the intractable computational complexity of its possible decision theories.

An analogous effect can be seen in the different outcomes in the co–refinement of $\lambda *$ and ComFoRT. In the case of $\lambda *$, co–refinement could be initiated from a highly restrictive starting point, in terms of the constructive freedom allowed designers, and in the observations that made by the theory. In contrast, the starting point for ComFoRT exhibited (to a large extent) the debilitating effects of generality on practicability—the reasoning framework can hardly be claimed to have ameliorated the effects of state–space explosion.

In retrospect, it is difficult to escape the conclusion that a key heuristic to co–refinement is to start from the most specialized theories possible and then follow Harper's prescription of steady "controlled relaxation." In contrast, co–refinement of ComFoRT can be regarded as having been initiated from an overly–generalized starting point, from which various specializations have been developed (buffer overflow detection is one example) that tend to be mutually incomparable, if not incompatible.

10.3.2 Engineering Roles in Co–Refinement

Developing a PECT is a non–trivial undertaking, requiring collaboration among different specialty skills. I have chosen to distinguish these roles by adding various prefixes to "engineer." Where Hoare would consider theory development to be the purview of the computer scientist (and it should be), the emphasis in the Seam is on the application, and tuning, of existing theories to meet specific objectives at hand, for which "engineer" seems more

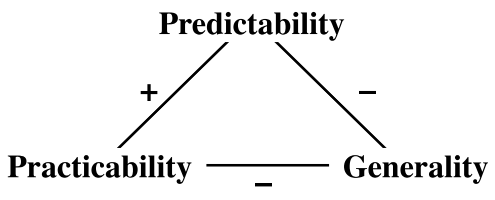


Figure 10.3: Reasoning Framework Design Forces

attuned.

Role	Concerns
PECT Engineer	Is expert in component technologies, the ideas of *The Seam* and *Predictability By Construction.* Is "Chief Architect" of the PECT, or PECT Reasoning Framework.
Application Engineer	Is expert in the application domain that is the target of the PECT, the standards and practices for developing applications of requisite quality, and the business objectives that will be satisfied by predictability by construction.
Theory Engineer	Is expert in a behavior theory, its observations and assumptions, how it can be specialized or generalized, and how its decision procedures can be automated.
Language Engineer	Is expert in the design and implementation of programming languages, including the specification of type systems and language semantics and the development of code generators.
System Engineer	Is expert in the design and implementation of operating systems, system instrumentation, measurement, and testing.

The PECT engineer can be regarded as the logical unitary design authority who is responsible for orchestrating the interactions among the other roles. Although two co–refinement data points are hardly sufficient basis for sweeping generalizations about the social aspects of co–refinement and the design of a PECT, a few minor observations are in order.

First, and perhaps surprisingly given the apparent theoretical complexity of some of the technologies integrated in a PECT, the entire enterprise hinges on the strength of the application engineer and the existence of a genuine and reasonably well–defined business objective. The application engineer gives voice to "practicability" concerns, which must be based in a concrete problem at hand, without which the entire design process can become unhinged and wander off into interesting but not always useful regions of the PECT design space.

Second, the design activity requires a significant level of communication among all of the roles, throughout the entire PECT or reasoning framework design process. There are as yet no clear interfaces that can separate the design of the constituents of a PECT—reasoning frameworks, component technologies, specification languages, validation infrastructures, code generators. The mutual dependencies in the design tradeoffs among these means they can not be designed in isolation of one another.

Last, a PECT is a complex product that requires a significant investment to develop. As with any complex product, its architectural design must be adequately documented if it is to be sustained, and if it is to adapt to new engineering practices—and perhaps be the impetus for incremental improvement in engineering practice. In this regard we might consider the apparatus of programming language design to be well–suited to documenting a PECT, as "analytic theory as language semantics" is certainly a central tenant of the overall PECT philosophy.

Part IV

Conclusions

Chapter 11

Summary of Results

In Chapter 1 *Introduction* I argued that contemporary software engineering theory and practice has, since its conception, been predicated on a false dichotomy between design and implementation, and therefore between architect and programmer, and this in turn has adverse consequences on both technical and social aspects of software engineering practice. I then introduced the major Theses of this research:

1. A region of complimentary design concerns for software architecture and computer programs, called *The Seam*, provides common ground on which to reconstruct software engineering practice to more effectively integrate architecture and programming practice.
2. A new software engineering capability, called *Predictability By Construction*, can be obtained by focusing the Seam on the invention and use of design rules that yield systems that exhibit analytically–predicted runtime behavior, with an explicit basis for justifiable confidence in these predictions.
3. A new kind of software component technology, called *Prediction–Enabled Component Technology* (PECT) is particularly well–suited to provide substantial automation of the Seam, and provides a range of shared design abstractions that have dual meaning to architects and programmers.

In this chapter I review in §11.1 how the results presented in this dissertation support the theses, and in §11.2 answer the key research questions posed by the Theses. Taken together, these two sections strongly justify the conclusion that the thesis presented by this research has been sustained. I close in §11.3 with the limitations of the approach reported here, and identify areas of possible future work.

11.1 Results in Support of the Theses

As noted early in this dissertation, establishing the viability of the Seam required applying it to non–trivial engineering design challenges. It is for this purpose that industrial case studies took on a prominent formative and normative role in the research approach—formative in giving shape to our understanding of the Seam, PBC, and PECT; and normative by establishing objective criteria with which to judge the effectiveness of triad.

Several industrial case studies were undertaken, with results described in Chapters 9–9. Each of these efforts confirmed one or more of the main theses presented in this research. A brief summary of key confirming results from the case studies is provided in Table 11.1.

Table 11.1: Seam Results From Case Studies and Prototypes

Concerns	• Distributed Soft P&C prototypes have predictable worst/average–case latency (§9.2.3, §9.2.4). • Hard real–time robot controller can be safely extended by third-party components (§9.3.2).
Theories	• λ-ABA predictions satisfy confidence interval $\gamma = 0.99$, $\rho = 0.99$, UB < 0.01 (§9.2.3, §9.2.4). • ObjectBench generates counterexample that confirms presence of deep latent "bug" in previously released industrial robotics communication library (§9.3.3).
Rules	• $\lambda*$ assumptions are enforced by a combination of Pin runtime mechanism, application constraints enforced by PCL and by the various $\lambda*$ interpretations (§10.2). • λ-SS assumptions are enforced by a combination of λ-ABA interpretation and by a specialized container that implements an application-level sporadic server (§9.3.2).
Explanations	• λ-ABA allows substation engineers to assess performance impact of different allocations of logical nodes to IEDs, up to an explicit confidence bound (§9.2.4) • λ-SS allows the architect to assess impact of different sporadic server refresh rates on overall system performance, and on service times of third-party components, up to an explicit confidence bound (§9.3.2).
Abstractions	• Pin components used to generate new and wrap legacy substation controller software, consistent with IEC-61850 logical nodes used by substation designers (§9.2.3, §9.2.4). • Constructive (Pin) interfaces and Analytic ($\lambda*$) interfaces provide dual functional and extra-functional view of code. (§9.2.4, §9.3.1). • λ-SS container provides bounded performance guarantees to Open Controller provider, Open Controller user, and third-party component developer (§9.3.1).

11.2 Answers to Key Research Questions

Question 1: What makes a runtime behavior "predictable" and what constitutes "sufficiently predictable" behavior?

This research provides a definition of *formally predictable behavior* (Def. 3.13, pp. 58) that combines theories of computation (Def. 3.9, pp. 56), and objective evidence (Def. 3.11, pp. 57), each constituent of which is an essential part of a PECT reasoning framework.

The question of "sufficiency" was addressed in the method of co–refinement (§10.2), which combines Hoare's ideas of theory development with Wirth's ideas of stepwise refinement of program design; the use of co–refinement on for the continuing evolution of $\lambda*$ through several years of industrial trial demonstrate the viability of the approach.

Question 2: How are theories program behavior packaged as "non–traditional semantics" of programs?

This research demonstrates that *reasoning frameworks* can be developed and independently deployed as "prediction-enabling" components of a PECT. ComFoRT (§8.6) and $\lambda*$ (§8.2) are existence proofs of at least two such non–standard semantics, although it would be more accurate to view $\lambda*$ as comprising three distinct non–standard semantics, and a similar case can be made for ComFoRT , where its certifying– and non–certifying decision procedures impose different constraints and have subtle differences in their interpretations mark these as defining distinct non–standard semantics.

Question 3: How are "design rules" that lead to predictable behavior identified and enforced?

This research demonstrates that *co-refinement* can be used to develop reasoning frameworks that impose variable degrees of stricture on designers, and to systematically uncover theory assumptions that must be satisfied to obtain specific theory observations and those that might be satisfied by static checking or runtime enforcement.

This research also demonstrated that software component technology provides various loci for enforcing constraints:

- The component runtime provides real–time services for $\lambda*$, and support for UML–style time– and change–triggered events, for ComFoRT and for PCL program generators.
- The component model provides containers and connectors to implement interaction policies used by λ-SS to manage component execution budgets and replenishment periods (§8.5.2) and ComFoRT to construct faithful (not over–) approximations of real application concurrency.

Question 4: How is justifiable confidence in program behavior established, and how is it used?

This research demonstrates that empirical and formal evidence of component and assembly behavior can be obtained:

- Empirical evidence can be constructed using conventional techniques of statistical inference, as shown by Larsson in his contribution to PECT [98], and as used to satisfy the objectives of the industrial case studies.
- Formal evidence that confirms or refutes a specified behavior can be constructed from model checkers and in the former case the resulting "proof certificates" obtained from component specifications and ultimately embedded in the deployable component binaries (in §8.6, Figure 8.7, pp. 170, reported in [33]).

The relatively "tight" λ-ABA confidence interval also allowed us to demonstrate the use of justifiable confidence to conduct "what if" experiments with alternative Soft P&C designs to assess their respective impact on performance (§9.2.4). Thus, reasoning frameworks support an efficient form of "test" in Simon's "generate and test" model of the design process (§3.1). An analogous capability was provided by the containerized plug–in approach demonstrated for the Open Robot Controller (§9.3.2).

Question 5: How is software component technology used to provide substantial automation of the seam?

This research demonstrates that familiar component-based abstractions and implementation techniques can be used to implement the seam. The PECT developed as a consequence of the research reported here is publicly available[1], and the development of non–trivial Soft P&C prototypes with an earlier and far less capable version of the public release PECT are persuasive demonstrations of the viability of component technology to automate the Seam.

11.3 Limitations and Future Work

In response to a question about a problematic "corner case" in the semantics of the (then newly minted) Ada programming language, its inventor, Jean Ichbiah, replied that "if you go looking for trouble, you are sure to find it." For those interested in building on the results reported here, I offer the following trouble spots, and suggest possible avenues of investigation to address them.

[1]See `http://www.sei.cmu.edu/predictability/tools/starterkit/index.cfm`.

New Dependencies and New Sources of Complexity

The PECT prototype described in this dissertation has a reasonably well–defined component structure, and in principle should be adaptable to different specification languages, target languages, component technologies, and reasoning frameworks. However, reasoning frameworks impose idiosyncratic (to the reasoning framework) constraints, and there is no reason to believe that the constraints imposed by an arbitrary collection of reasoning frameworks would be consistent or satisfiable.

Ultimately, we will need better abstractions for talking about constraints, and for composing and validating sets of constraints, if we ever hope to compose reasoning frameworks. This is closely related to the question of *modular language semantics*, which is known to be notoriously difficult to achieve.

Gentle Slope Adoption

Despite the fact that all of the constituent parts of the prototype PECT described here were largely pre–existing (GRMA, C, UML Statecharts, Model Checkers), the cost of developing the prototype was substantial. Indeed, the cost of developing each reasoning framework was substantial, and as observed in §10.3, co–refinement requires substantial interaction among people who possess specialized (and likely costly) expertise.

Building blocks for reasoning frameworks are necessary (possibly along the lines of what Dwyer and colleagues attempted for model checkers [50]) if they are to be developed under practical time and resource constraints, and if "co–refinement" is to move from research lab to engineering practice.

The Value of Confidence

What is the business value of a ($\gamma = .99, \rho = .80, \mathrm{UB} < 0.01$) confidence interval, and how much additional value is obtained by improving ρ, and what preference should we express for incomparable alternative intervals, for example one that improves ρ at the expense of UB versus one that improves γ at the expense of ρ? What is it worth to a component consumer to have a machine–checkable proof that the component satisfies some explicit security policy?

The axis of predictability by construction turns on the assumption that there is economic and business value in having an objective basis for confidence in analytic predictions, but this premise remains ungrounded in current practice. The value proposition for PECT, and for PBC, can not be established without first obtaining a better appreciation of how different qualities of evidence translate to social or business value.

Reconsidering PCL

As the designer of PCL I can safely say that I don't like its current syntax, and its major constituents (Pin component model, UML Statecharts, C action language) are not cleanly modularized in the language syntax, semantics, or implementation. This is a significant inhibitor to further development of the PECT prototype, and hinders its adoption by researchers and practitioners who might otherwise have an interest in extending the prototype.

There are several interesting approaches that might be investigated:

- Replace the PCL frontend with a commercial CASE[2] tool can provide a UML Statechart frontend.
- Replace the PCL frontend with an industrially–viable architecture description language such as AADL.
- Identify the minimal extensions needed to ANSI–C to allow C programmers to define Pin–like component abstractions, and replace PCL with a frontend for this minimally extended dialect as a kind of component–based analogy to "cfront," the first implementations of C++.

A final idea about extending PCL which can be undertaken without abandoning its current design and implementation would be to make pins in PCL denotable (can be named), expressible (can be used in expressions), and storable (can be saved and referred to by indirect means). In the current design, pins are only denotable. With these extensions (not conceptually difficult) it would be possible to implement something closely analogous to channel passing in Occam-π[3] by allowing pins to be passed by reference as pin parameters. More directly—this would permit a form of runtime evolution of component topology that also has an available behavioral theory in the π–Calculus.

[2]Computer Aided Software Engineering

[3]See `http://www.cs.kent.ac.uk/projects/ofa/kroc/`, last accessed 30 Aug 2010

Part V

Appendixes

Appendix A

PCL Semantics

PCL defines an *interaction* and a *reaction* semantics. The interaction semantics is given in §A.1; it defines the meaning of the '$\leadsto$>' operator that wires two (or more) components together. The reaction semantics is given in §A.2; it defines how components interact with the environment, and defines the PinChart execution model for component reactions.

A.1 Interaction Semantics

A.1.1 Preliminaries

A denotational style of definition is given in which well–formed phrases in the syntax domain (PCL) are mapped to one or more mathematical objects in the semantic domain (CSP [75]). PCL phrases are enclosed in $[[doublebrackets]]$.

$\mathcal{P}[[E]] = P$

describes a semantic function $\mathcal{P}$ that maps the PCL phrase E to CSP object P. A denotational definition is convenient because the syntactic structure of PCL aligns easily with CSP. Using CSP for the semantic domain is also quite natural and it (and other process algebras) have been used to describe composition semantics [8, 108, 106]. The following CSP operators are used in this summary:

Channels In CSP *processes* describe computational behavior, and they communicate by synchronizing on *channels*. For example, $s?x!y$ represents a communication channel s that receives data x and sends data y.

'$\rightarrow$' Step $\Phi \rightarrow r$ specifies a computational step from Φ to r, where r is a channel name offered to the environment. $\Phi \xrightarrow{\alpha} r$ specifies two computational steps, from Φ to r, with the action α performed before r is offered.

'=' Process $P = s!x \rightarrow r?y \rightarrow P$ specifies a process P that performs s (and sends x) and then becomes a process that performs r (and receives y) and then becomes once again the process P.

'$|||$' Interleave $s \rightarrow P \; ||| \; r \rightarrow Q$ (*interleaving*) denotes independent processes P and Q that can perform s or r (or any events in a longer prefix) in any order.

'$\|$' Parallel $s \rightarrow P \parallel s \rightarrow Q$ (*parallel*) which denotes two processes $s \rightarrow P$ and $s \rightarrow Q$ that must simultaneously perform s (synchronize on s) and then become P and Q, respectively.

'$\Box$' External Choice $P = x \rightarrow P \,\Box\, y \rightarrow P$ denotes a process P that can perform x or y, depending on which the environment chooses to present.

Component types and instances are generally denoted by C and c, respectively. Sink pins and source pins are generally denoted by s and r, respectively. Subscripts, superscripts and other markings are used as necessary.

A.1.2 A Simple Example

Let us assume two PCL component types **component** C1 and **component** C2 with the following specifications:

Example A.1: Motivating PCL Assembly.

```
component C1 ()
{
  sink synch s1();
  source sink r1();
  threaded react R1 (s1, r1)
  {
    start   -> listen {}
    listen  -> act {trigger ^s1; action ^r1();}
    act     -> listen {trigger $r1; action $s1();}
  }
}

component C2 ()
{
  sink synch s2();
  source sink r2();
  threaded react R1 (s1, r1)
  {
    start     -> listen {}
    listen    -> listen {trigger ^s2; action $s2();}
  }
}

assembly A () (E) {
  assume {}

  C1 c1(), c3();
  C2 c2();

  c1:r  ~>  c2:s;
  c3:r  ~>  c2:s;

  expose {}
}
```

We wish to model the behavior of c1:r ~>c2:s, setting aside as a minor detail the fact that we ought to define this behavior on an instance of **assembly** A rather than its type. Let:

$$\mathcal{C}[[\text{C1 c1()}]] \trianglerighteq P_1 = s_1 \to r_1 \to \bar{r}_1 \to \bar{s}_1 \to P_1$$
$$\mathcal{C}[[\text{C2 c2()}]] \trianglerighteq P_2 = s_2 \to \bar{s}_2 \to P_2$$

be the CSP process descriptions of the two component instances c1 and c2, where '$\trianglerighteq$' signifies that the CSP processes are the ultimate outcome of the action of semantic functions which have not yet been defined. A naive interaction semantics

$\mathcal{X}$ is then given by:

$\mathcal{X}$[[c1:r ⤳>c2:2]] $\overset{\triangleright}{=} \mathcal{C}[[c1]]_{[r_1 \leftarrow r_{1,2}]} \parallel \mathcal{C}[[c2]]_{[r_2 \leftarrow r_{1,2}]}$

where $P_{[a \leftarrow b]}$ denotes a new process P' in which *all occurrences* of channel name a in P are replaced by b.

This interpretation is quite reasonable, but for our purposes is inadequate because it fails to account for two aspects of interaction behavior that we wish to model:

1. It does not express the queuing of events on C2:s that will result from c3:r⤳>c2:s. In such circumstances, one of c1 or c3 may find itself queued beyond interactions initiated by the other. In once sense blocking behavior is just blocking behavior, but in another there are different causes of blocking, and waiting in a queue to be served is different than waiting while being served. Further, there is no way to express the queuing policies that c2:s will use to service incoming requests.

2. It does not distinguish blocking and non–blocking behavior on C1:r. In this example only synchronous interactions are involved, and so this deficiency is not immediately apparent. However, had the pins in Example A.1 been *asynchronous* rather than *synchronous*, the behavior would have incorrectly modeled c1 as blocking on c1:r ⤳>c2:s until the reaction denoted by c2:s completes.

We account for 1 and 2 by using two different kinds of "glue" processes to model these behaviors—*source* glue processes, denoted P^r, and *sink* glue processes, denoted P^s.

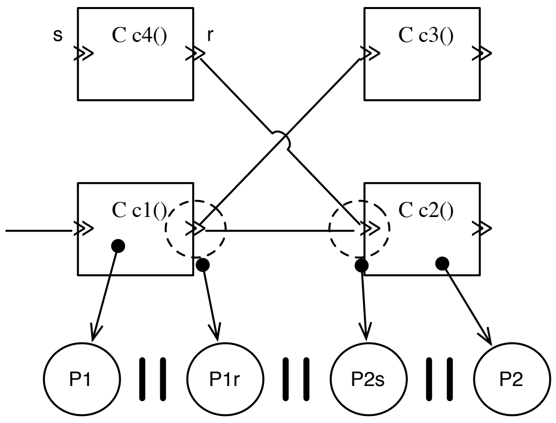


Figure A.1: Interaction Semantics: Schema (Redux)

Figure A.1 illustrates the overall scheme used to define the semantics of interaction. The source glue process P1r defines where blocking occurs in the initiating reaction, and the order in which events are queued to c2:s and c3:s. The definition of "glue" processes depends on details of connection topology, and in this example P1r is constructed from, and it's *alphabet* is defined by, C1:r⤳>C2:s and C1:r⤳>C3:s, and similarly P2s is constructed from, and alphabet defined by, C1:r~>C2:s and C4:r~>C2:s.

The CSP process defined by P1r || P2s observes asynchronous interactions between component instances c1() and c2() on their source and sink pins c1:r and c2:s, respectively. It can be regarded as a "connector" process, but if it instantiates a connector type then the type must be parameterized by all processes that interact on C1:r and C2:s. An analogous semantic interpretation for synchronous interactions likewise observes the behavior of "synchronous connectors."

A.1.3 Top–Level Process

Later it will be shown how to construct for any PCL component instance (with any number of reactions and sink and source pins) a single CSP process that denotes its behavior. Therefore, without loss of generality, we describe interaction semantics in terms of simple cases:

- Components will have exactly one reaction.
- Components (and reactions) will be stateless (state is addressed in §A.2 Reaction Semantics).
- Components will have at most one sink and one source pin.
- The top–level assembly is constructed from component and services instances only (i.e., no sub–assemblies).

The following abstract syntax suffices for this simplified language:

Definition A.1 (Initial Syntax Domain)

$$\gamma = \mathit{Asm} ::= \mathit{Inst}^{*} \ \ \mathit{Wire}^{*}$$
$$\mathit{Inst} ::= \mathit{Id}_C \ \mathit{Id}_s^{+} \mathit{Id}_r^{+}$$
$$\mathit{Wire} ::= \mathit{Id}_C.\mathit{Id}_P \rightsquigarrow \mathit{Id}_C.\mathit{Id}_P$$

where Inst defines the name of a component instance (Id_C) and a set of sink ($\mathrm{Id}_\mathrm{s}^{+}$) and source pin ($\mathrm{Id}_\mathrm{r}^{+}$) names, and where Wire connects two component instances on their respective source and sink pins. Id_C and Id_P are identifiers that denote components and pins, respectively. The semantic domains of interest are:

Definition A.2 (Initial Semantic Domain)

$\mathit{CSP} =$ the domain of CSP process descriptions

$\rho : \mathit{Env} = \mathit{Id} \to \mathit{CSP}$

$\mathcal{A} : \mathit{Asm} \to \mathit{Env} \to \mathit{CSP}$

$\mathcal{D} : \mathit{Inst} \to \mathit{Env} \to \mathit{Env}$

$\mathcal{C} : \mathit{Inst} \to \mathit{CSP}$

$\mathcal{X} : \mathit{Wire} \to \mathit{Env} \to \mathit{Env}$

Env is a function that when presented with an identifier Id returns a CSP process associated with that Id. It is customary for the denotation of an identifier to be the identifier, i.e., $[[Id]] = Id$. Asm, Env, Inst, and Wire correspond to syntactic phrase groups in the abstract syntax in Def. A.1. $\mathcal{D}$ is the name of a (higher order) function that when presented with an Inst and an Env will produce another Env; $\mathcal{C}$ takes an Inst and produces a CSP process, etc.

Definition A.3 (Initial Semantic Equations)

$\mathcal{D}[[Id_C(Id_P^+)]]\rho = \rho[\mathcal{C}[[Id_C(Id_P^+)]]/Id_C]$
$\mathcal{D}[[\gamma_1; \gamma_2]] = \mathcal{D}[[\gamma_2]] \circ \mathcal{D}[[\gamma_1]]$
$\mathcal{C}[[Id_C(Id_P^+)]] =$ *See §A.1.4.*
$\mathcal{X}[[Id_C.Id_P \rightsquigarrow Id_C.Id_P]]\rho =$ *See §A.1.5.*
$\mathcal{X}[[\gamma_1\gamma_2]] = \mathcal{X}[[\gamma_2]] \circ \mathcal{X}[[\gamma_1]]$

Def. A.3 defines the skeletal structure of the interaction semantics. The first equation shows how instantiations are added to the environment ρ, where the notation $\rho[x/y]$ means the *new* function ρ' such that if $a = y$ then $\rho'(a) = x$; otherwise $a \neq y$ and $\rho'(a) = \rho(a)$. The second equation (and the last) show the paradigmatic way that sequential composition in the syntactic domain is handled in the semantic domain—as function composition.

A.1.4 Component Instance Processes

We construct for each PCL component instance a single CSP process that specifies its behavior, regardless of how many (threaded and unthreaded) reactions the PCL component types specify. Each such CSP process must be constructed to in such a way that it has a unique channel alphabet. In CSP, processes synchronize on shared channel names; and processes with unique alphabets will never synchronize. Then the semantic function $\mathcal{X}$ will, among other things, selectively rename channels so that process synchronization is possible, and hence component interaction. Component instance processes are constructed in the following way:

- Each pin [[p]] corresponds to a pair of CSP channels, $\hat{p} = (p, \bar{p})$, where p may have input data for each consume parameter, i.e., $p?x?y$ if p has two consume parameters, and $\bar{p}$ may analogously have output data for each produce parameter, i.e., $\bar{p}!a!b$.

- Each reaction [[w]] of component instance [[c]] corresponds to a CSP process $\hat{w}$ whose channels are the set $\hat{P}$ of CSP channels that correspond to the pin parameters of [[w]], and where each $\hat{p} \in \hat{P}$ is suitably renamed to ensure their uniqueness across all processes.

- Each component instance [[c]] corresponds to an interleaved CSP process $\hat{c} = |||_{\hat{w}^+}$, each $\hat{w}$ corresponding to a reaction of [[c]]'s component type. Interleaving represents nondeterministic (for the purpose of constructive composition semantics) scheduling of reaction threads.

Example A.2: Unique Instance Processes.

```
component C ()
{
  sink synch s1();
  sink asynch s2();
  source sink r();
  unthreaded threaded react R1 (s1, r)
  {
    int x = 0;
    start   -> listen {}
    listen  -> act {trigger ^s1; action x++; ^r();}
    act     -> listen {trigger $r; action $s1();}
  }
  threaded react R2 (s2, r)
  {
    start   -> listen {}
    listen  -> act {trigger ^s2; action ^r();}
    act     -> listen {trigger $r; action $s2();}
  }

}
```

There are many possible ways to construct globally unique names. However, to make matters concrete, consider the definition of **component** C in Example A.2. We wish to construct a single process that represents the behavior of instances of **component** C.

$\mathcal{C}[[\mathrm{C\ c()}]]\rho = \mathcal{R}[[c : R1]]\rho \ ||| \ \mathcal{R}[[c : R2]]\rho$

where

$\mathcal{R}[[c : R1]] \overset{\trianglerighteq}{=} \mathrm{cR1} = cs1 \xrightarrow{\mathrm{x{+}{+}}} cR1r \to \overline{cR1r} \to \overline{cs1} \to \mathrm{cR1}$

and

$\mathcal{R}[[c : R2]] \overset{\trianglerighteq}{=} \mathrm{cR2} = cs2 \to cR2r \to \overline{cR2r} \to \overline{cs2} \to \mathrm{cR2}$

Pins define the alphabet of the CSP processes, and are mapped to CSP channel pairs, as described earlier. Actions (i.e., PCL *statements*, see §6.2.5, pp. 104) that do not generate channel events are regarded as internal (τ) transitions, and appear as transition labels.

The technique used to achieve unique process alphabets relies on the static scoping rules enforced by PCL that requires unique denotable names for certain syntactic constructs, such as declarations, within scopes such as an assembly specification; therefore, the name of a component instance within an assembly provides a unique prefix. Because source pins may be shared by different reactions, the concatenation of instance name and reaction name provides a unique prefix.[1]

So a unique alphabet can be constructed from two rules:

[1]PCL requires that each sink pin be allocated to exactly one reaction, so no such disambiguation of sink pin names is required.

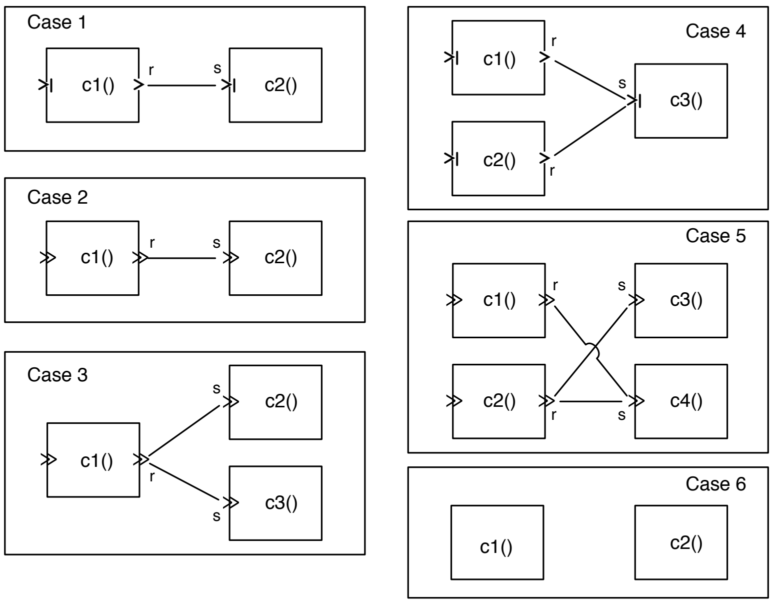


Figure A.2: Basic Interaction Patterns

1. for sink pin c:s, $\mathcal{Q}[[\text{c:s}]] = \text{cs}$
2. for source pin c:r, $\mathcal{Q}[[\text{c:r}]] = \{\text{c}\gamma\text{r} \bullet \gamma \text{ is a reaction name of c()}\}$

Details of $\mathcal{R}$ and $\mathcal{C}$ will not be further elaborated in this semantics. In fact, we can regard the process outlined above for creating unique processes something that can be carried out entirely within the syntactic domain as a "pre–processing" step. This is, in fact, a reasonable interpretation of the abstract syntax give in Def. A.1, which simply declares component instances and their pins. We lose no generality by assuming that all component instance names, and all pin names, are unique for all component instances.

A.1.5 Interacting Processes

Here we describe the rules for composing a top–level CSP process to denote by a top–level PCL assembly from pairwise syntactically composed PCL components.

Figure A.2 depicts six cases that are discussed in the following sections; these cases lead to the final semantic specification provided in §A.1.6. Cases 1 and 2 illustrate the different approaches needed to accommodate synchronous and asynchronous interaction. Cases 3 and 4 illustrate the composite effects of interactions of N components on one pin, with c2() and c3() on c1:r in Case 3, and c1() and c2() on c3:s in Case 4. Case 5 combines Cases 3 and 4. Finally, Case 6 handles the case where two component instances do not interact.

To make the discussions concrete where they need to be, the cases are built from component types: CStimγ and CRespγ, where $\gamma \in$ {Asych, Synch} denote component types that use asynchronous and synchronous pins, respectively. The

CSP process descriptions of *CStimγ* and *CRespγ* instances are defined as:

$$\overrightarrow{P} = \mathcal{C}[[\text{CStim}\gamma\ \text{c1()}]] \overset{\triangleright}{=} P = s \rightarrow r \rightarrow \bar{r} \rightarrow \bar{s} \rightarrow P$$

$$\overleftarrow{P} = \mathcal{C}[[\text{CResp}\gamma\ \text{c2()}]] \overset{\triangleright}{=} P = s \rightarrow \bar{s} \rightarrow P$$

and instances of each of the components in the cases are replaced by their corresponding process descriptions.

Case 1: One–To–One Synchronous Interaction

We assume a simple *synchronous* interaction among two component instances c1 and c2 with process behaviors specified $\mathcal{C}[[\text{c1()}]] \overset{\triangleright}{=} \overrightarrow{P}_1$ and $\mathcal{C}[[\text{c2()}]] \overset{\triangleright}{=} \overleftarrow{P}_2$. Then the semantics of basic synchronous interaction is given by:

$$\mathcal{X}[[\text{c1:r}{\sim}{>}\text{c2:s}]] \overset{\triangleright}{=} \overrightarrow{P}_1 \| \mathcal{G}^r[[\text{c1:r}{\sim}{>}\text{c2:s}]] \parallel \mathcal{G}^s[[\text{c1:r}{\sim}{>}\text{c2:s}]] \parallel \overleftarrow{P}_2$$

where $\mathcal{G}^r$ and $\mathcal{G}^s$ are source and sink glue constructors, respectively, with:

$$\mathcal{G}^r[[\text{c1:r}{\sim}{>}\text{c2:s}]] \overset{\triangleright}{=} G^r = c1r \rightarrow c1rc2s \rightarrow \overline{c1rc2s} \rightarrow \overline{c1r} \rightarrow G^r$$

and

$$\begin{aligned}\mathcal{G}^s[[\text{c1:r}{\sim}{>}\text{c2:s}]] \overset{\triangleright}{=} G^s &= c1rc2s \rightarrow c2s \rightarrow \text{Reacting}\\ \text{Reacting} &= \overline{c2s} \rightarrow \overline{c1rc2s} \rightarrow G^s\end{aligned}$$

where $c1r$ and $c2s$ are globally unique channel names obtained by means described earlier, and where $c1rc2s$ is a globally unique channel name constructed by $\mathcal{G}^r$ and $\mathcal{G}^s$ to describe glue processes.

$\mathcal{G}^r[[\text{c1:r}{\sim}{>}\text{c2:s}]]$ (the source glue) models the expected blocking behavior. In G^r, the transition $c1r \rightarrow c1rc2s$ denotes the synchronous event being queued, while the transition $c1rc2s \rightarrow \overline{c1rc2s}$ is the acknowledgement that the event has been queued. $\overrightarrow{P}_1$ remains "blocked," however, until it synchronizes on the matching $\overline{c2s}$ generated by the sink glue.

$\mathcal{G}^s[[\text{c1:r}{\sim}{>}\text{c2:s}]]$ (the sink glue) does not model queueing behavior because there are no component instances other than c1() to require message queueing on c2:s. In G^s, the transition $c1rc2s \rightarrow c2s$ denotes the receipt of a request from some arbitrary component, and the forwarding of this request to c2:s, at which point G_2^s becomes a Reacting sink glue process. The transition $\overline{c2s} \rightarrow \overline{c1rc2s}$ denotes the completion of the reaction initiated on c2:s.

To simplify notation, unique pin/channel names will not be constructed from component instance and pin names, and the above source and sink glue processes are equivalent to:

$$\mathcal{G}^r[[\text{c1:r}{\sim}{>}\text{c2:s}]] \overset{\triangleright}{=} G^r = r \rightarrow u_{r,s} \rightarrow \bar{u}_{r,s} \rightarrow \bar{r} \rightarrow G^r$$

and

$$\begin{aligned}\mathcal{G}^s[[\text{c1:r}{\sim}{>}\text{c2:s}]] \overset{\triangleright}{=} G^s &= u_{r,s} \rightarrow s \rightarrow \text{Reacting}\\ \text{Reacting} &= \bar{s} \rightarrow \bar{u}_{r,s} \rightarrow G^s\end{aligned}$$

Case 2: One–To–One Asynchronous Interaction

We assume a simple *asynchronous* interaction among two component instances c1 and c2 with process behaviors specified $\mathcal{C}[[\text{c1()}]] \overset{\triangleright}{=} \overrightarrow{P}_1$ and $\mathcal{C}[[\text{c2()}]] \overset{\triangleright}{=} \overleftarrow{P}_2$. Then, as with Case 1, the semantics of basic asynchronous interaction is given by:

$$\mathcal{X}[[\text{c1:r}{\sim}{>}\text{c2:s}]] \overset{\triangleright}{=} \overrightarrow{P}_1 \parallel \mathcal{G}^r[[\text{c1:r}{\sim}{>}\text{c2:s}]] \parallel \mathcal{G}^s[[\text{c1:r}{\sim}{>}\text{c2:s}]] \parallel \overleftarrow{P}_2$$

However, in this case we need a different source glue process than that specified for Case 1. In that case, the source glue forced the initiating process $\overrightarrow{P}_1$ to wait until the reaction on $\overleftarrow{P}_2$ completed. In this case we want $\overrightarrow{P}_1$ to block only as long as required to queue the event:

$$\overset{\gg}{\mathcal{G}^r}\,[[\text{c1:r}{\sim}{>}\text{c2:s}]] \overset{\triangleright}{=} G^r = r \to u_{r,s} \to \bar{u}_{r,s} \to G^r$$

A different sink glue is required as well because it is now possible for c1:r to initiate a sequence of interactions on c2:s, perhaps more quickly than can be handed by c2(), in which case these events must be queued. As it turns out, however, the management of queueing policies on sink glues is insensitive to whether the pins involved are synchronous or asynchronous, and therefore we can define a generalized form of sink glue that works in all circumstances for any $\overleftarrow{P}_j$:

Definition A.4 (Generalized Sink Glue)

$$\mathcal{G}^s[[c{:}r{\sim}{>}cj{:}s]] = G^s = \Box_{i=1}^{n}(u_{r,s} \to s_j \to \mathit{Reacting}_j)$$
$$\mathit{Reacting}_j = \Box_{i=1}^{n}(u_{r,s} \xrightarrow{push(Q,u_{r,s})} \mathit{Reacting}_j)$$
$$\Box \bar{s}_j \xrightarrow{u_{x,s}=pop(Q)} \bar{u}_{x,y} \xrightarrow{empty(Q)?} [G^s, \mathit{Reacting}_j]$$

where $\xrightarrow{p} [X, Y]$ is interpreted as X if p is True and Y otherwise, and where '$\Box_m^n$' is indexed external choice.

The transition $u_{r,s} \to s_j$ initiates an interaction on cj:s, and then with $s_j \to$ Reacting$_j$ the sink glue waits for either the arrival of another $u_{r,s}$ event or the completion of a pending s_j event. The first line of Reacting$_j$ handles the first case by queuing the new request, while the second line handles the second case by popping the queue with $u_{x,s} =$ pop(Q) and generating the matching $\bar{u}_{x,s}$ event, in FIFO order.

Case 3: One–To–Many Asynchronous Interaction

We assume a simple *asynchronous* interaction among N component instances c1:r~>{c2:s,c3:s ,..., cn:s}, where $c1$ is an instance of **component** C1 and c1, c2, c3, etc. are instances of **component** C2. Case 3 is a straightforward generalization of Case 2. Beginning somewhat imprecisely, with S ≡ c1:r~>{c2:s,c3:s ,..., cn:s}:

$$\mathcal{X}[[S]] \overset{\triangleright}{=} P_1 \parallel G_1^r \parallel (G_2^s \parallel P_2) \parallel (G_3^s \parallel P_3) \parallel \ldots \parallel (G_n^s \parallel P_n)$$

where $P_i = \mathcal{C}[[\text{C ci()}]]$, and where parenthesis have been added to highlight the structure of the expression but have no effect on its meaning.

A more concise formulation is given using the "indexed parallel" operator ($\|_m^n$):

$$\mathcal{X}[[S]] \overset{\triangleright}{=} P_1 \parallel G_1^r \parallel \Big(\|_{i=2}^n \, (G_i^s \parallel P_i)\Big)$$

Case 3 adds one new wrinkle, though: the definition of G_1^r depends on both pairs of interaction c1:r⤳>c2:s and c1:r⤳>c3:s. Among other things, G_1^r must specify an order of interaction—should events be transmitted to c2:s and then c3:s or the other way around? In fact, the choice made by PCL is to define the order of interaction as nondeterministic, which leads to the following generalized form for asynchronous source glues:

Definition A.5 (Generalized Asynchronous Source Glue)

$$\overset{\gg}{G_j^r} = r_j \to \Big(\|_{i=1}^n \, (u_{r,s} \to \bar{r}_j \to STOP.)\Big); \overset{\gg}{G_j^r}$$

where '$\|_m^n$' is indexed parallel, and where ';' is the CSP sequential composition operator.

The asynchronous source glue creates a set of short–lived processes, one for each x in cj:r ⤳>cx:s, and which terminates immediately after acknowledging that the asynchronous event has been placed on the appropriate sink queue.

Note that synchronous source glues are quite simple in comparison since one–to–many synchronous interactions are not permitted by PCL.[2] Thus it is possible to state the generalized form for synchronous source glues:

Definition A.6 (Generalized Synchronous Source Glue)

$$\overset{>}{G_j^r} = r_j \to u_{r,s} \to \bar{u}_{r,s} \to \bar{r}_j \to \overset{>}{G_j^r}$$

Case 4: Many–To–One Synchronous Interaction

This case is a straightforward combination of Case 1 with the generalized sink glue in Eq. A.6 used to accept events from two sources.

Case 5: Many–To–Many Asynchronous Interaction

This case is a straightforward combination of Case 3 with the generalized asynchronous source glue in Eq. A.5 and generalized sink glue in Eq. A.6.

Case 6: Non–Interaction

This is the base case in assemblies that consist of several non–interacting subassemblies, for example a sequence of non–synchronizing pipelines. The semantic interpretation is simply:

[2]If there were **consume** parameters on the synchronous source pin, from which interaction should it obtain results? There are many possible answers, some of them interesting, but this interaction pattern was not considered useful and could in fact have counterintuitive behavior.

$\mathcal{A}[[c1();c2();]] \overset{\triangleright}{=} \mathcal{C}[[c1()]] \parallel \mathcal{C}[[c1()]]$

where we use parallel ($\|$) rather than interleaved ($|||$) because $\mathcal{C}$ always denotes processes that have unique alphabets.

A.1.6 PCL Interaction Semantics (Final)

We can now consolidate the previous discussion. The syntactic domain specified in Def. A.7 is generalized from Def. A.1. It defines two languages: γ, which includes environments and services (which reduce to a placeholder 'ε' production), and γ' which includes only assemblies and components. Because services and components are identical under this semantics, γ' will be used. Subassemblies are not included in γ', although this is a minor omission since what is defined can be quite easily extended to accommodate hierarchical assembly by simply adding a **expose** phrase to the abstract syntax and defining a direct semantic interpretation from that phrase to CSP restriction. Pins reduce to unary prefix operators applied to pin identifiers; these operators combine the direction of the pin ($\leftarrow, \rightarrow$ for sink, source pins, respectively), and protocol ($\gg, >$ for asynchronous, synchronous, respectively). No abstract syntax is provided for reactions (React ::= ε) for reasons described later.

Definition A.7 (Syntax Domain)

$$\begin{array}{cclcl}
\gamma & = & \mathit{Main} & ::= & \mathit{Serv}\ \mathit{Asmb} \\
 & & \mathit{Serv} & ::= & \mathit{Serv} \diamond \mathit{Serv}' \mid \varepsilon \\
\gamma' & = & \mathit{Asmb} & ::= & \mathit{Decl}\ \mathit{Inst}\ \mathit{Wire} \\
 & & \mathit{Decl} & ::= & \mathit{Decl} \diamond \mathit{Decl}' \mid C(\mathit{Id}_c\ \mathit{React}\ \mathit{Pin}) \\
 & & \mathit{React} & ::= & \varepsilon \\
 & & \mathit{Pin} & ::= & \mathit{Pin} \diamond \mathit{Pin}' \mid \overset{\gg}{\leftarrow}\mathit{Id}_s \mid \overset{>}{\leftarrow}\mathit{Id}_s \mid \overset{\gg}{\rightarrow}\mathit{Id}_r \mid \overset{>}{\rightarrow}\mathit{Id}_r \\
 & & \mathit{Inst} & ::= & \mathit{Id}_c\ \mathit{Id}_i \\
 & & \mathit{Wire} & ::= & \rightsquigarrow(\mathit{Id}_i\ \mathit{Id}_r\ \mathit{Id}_i'\ \mathit{Id}_s)
\end{array}$$

The semantic domains are given in Def. A.8. CSP is the principal semantic domain, from which P is defined for processes as the "sum" (+) domain, meaning that values of P are either processes (CSP) or undefined ($\perp$).

From P other process domains are defined as G, suitably decorated to denote the three varieties of glue constructed (two kind is of source glue, one kind of sink glue). *Glue* is defined as a "sum" (+) domain of three kinds of glues.

If D is a sum domain $D = D_1 + D_2 + \ldots + D_k$, and if $v_1 \in D_1, v_2 \in D_2, \ldots, v_k \in D_k$, then $e = [v_1, v_2, \ldots, v_k]$ is an index function that uses $\phi \in D$ as an index. In the example, if $\phi \in D_2$, then $e\,\phi = v_2$.

Cproc (for "component processes") is defined as a "product" ($\times$) domain, meaning that values of *Cproc* are pairs (P_c, B), where B is the 3–point domain $B = \perp + F + \top$ that is often used to model the domain of Boolean truth values, with F modeling "False," $\top$ modeling "True," and $\perp$ modeling "Undefined." Values of a product domain are produced using the tuple–forming $< \ldots >$ operator.

If D is a product domain $D = D_1 \times D_2 \times \ldots \times D_k >$, then D_j where $0 \geq j \leq k$ is the projection function defined for D such that if $d \in D$ then $d_1 \in D_1, d_2 \in D_2$, etc., and $d_x = \perp$, and $d_j = \perp$ if $j \geq k \vee j \leq 0$.

CType is defined as a product domain of values of *Cproc* and *Glue*. *Cenv* and *Env* are defined as function (‘$\rightarrow$’) domains, mapping from identifiers to *Glue* and *CType* values, respectively.

Definition A.8 (Semantic Domain)

$$\begin{array}{rcll}
CSP & = & \textit{well–formed CSP formulae} & \\
P & = & CSP + \bot & \text{(process)} \\
\overset{\gg r}{G} & = & P & \text{(asynch source glue, Def. A.5)} \\
\overset{>r}{G} & = & P & \text{(synch source glue, Def. A.6)} \\
\overset{s}{G} & = & P & \text{(sink glue, Def. A.4)} \\
Glue & = & \overset{\gg r}{G} + \overset{>r}{G} + \overset{s}{G} & \\
Cproc & = & P_c \times B & (\mathrm{B} = \bot + F + \top) \\
\pi : Cenv & = & Id \rightarrow Glue & \\
CType & = & Cproc \times Cenv & \\
\rho : Env & = & Id \rightarrow CType &
\end{array}$$

The semantic function types are specified in Def. A.9. They define (in Def. A.9) a homomorphic relation between the abstract syntax γ' and the CSP processes that denote component instances and glue processes that denote connectors. For example, $\mathcal{A}$: Asmb $\rightarrow$ Env $\rightarrow$ $\mathrm{P_A}$ specifies a higher–order semantic function $\mathcal{A}$ that, when given a syntactic phrase belonging to Asmb (see Def. A.7), yields a function Env $\rightarrow$ $\mathrm{P_A}$ from an environment Env to functions a CSP process P_A.

Definition A.9 (Semantic Functions)

$$\begin{array}{lcl}
\mathcal{A} & : & Asmb \rightarrow Env \rightarrow P_A \\
\mathcal{D} & : & Decl \rightarrow Env \rightarrow Env \\
\mathcal{P} & : & Pin \rightarrow Cenv \rightarrow Cenv \\
\mathcal{I} & : & Inst \rightarrow Env \rightarrow Env \\
\mathcal{X} & : & Wire \rightarrow Env \rightarrow Env
\end{array}$$

Several auxiliary functions are introduced in Def. A.10 that simplify the presentation of the semantics. These functions are not formally defined here because, with the exception of Ψ, these details are routine and do not contribute to the exposition; Ψ is not formally defined because that aspect of PCL is addressed by the reaction semantics defined in §A.2 *Reaction Semantics.* However, more will be said about Ψ and reaction semantics in the introduction to §A.2.

Definition A.10 (Auxiliary Semantic Functions)

$$\begin{array}{lcll}
\Gamma & : & CType \rightarrow Id \rightarrow CType & \text{(extends glue with new interaction)} \\
\Lambda & : & CType \rightarrow Id \rightarrow CType & \text{(new process with Id as prefix in } \Sigma) \\
\Omega & : & Env \rightarrow Id \rightarrow CSP & \text{(all process in } \rho, \pi \text{ in parallel)} \\
\Psi & : & React \rightarrow P_c & \text{(CSP interpretation of PinChart)}
\end{array}$$

An informal description of the auxiliary functions introduced in Def. A.10 is:

Γ This function is polymorphic on glue process types, and simply extends whichever process is denoted with an interaction on channel Id.

Λ Given a value from $p \in CType$, (possibly but not necessarily with $(p_1)_2 = T$), this function produces p' such that all channel names used by any process in p are prefixed in p' by Id.

Ω This function performs an indexed parallel composition $\|_x$ where x ranges over all component *instance* processes (the first member of $CType$) and all glue processes for that instance.

Ψ This function constructs a CSP process P_r denoted by a PinChart specification. For interaction semantics we regard Ψ as constructing a process that reacts to each sink pin event by interacting on each of its source pins.

Definition A.11 (Interaction Semantics)

$$\begin{array}{lcl}
\overset{s}{G_0}, \overset{\gg r}{G_0}, \overset{>r}{G_0} & = & \text{(initial glues, see Defs. A.4, A.5, A.6, resp.)} \\
\pi_0 & = & \forall Id \bullet \pi\ Id = \perp \qquad \text{(initial environment)} \\
\rho_0 & = & \forall Id \bullet \rho\ Id = \perp \qquad \text{(initial environment)}
\end{array}$$

$$\begin{array}{lcl}
\mathcal{A}[[Decl\ Inst\ Wire]]\rho & = & \Omega\ (\mathcal{X}[[Wire]] \circ \mathcal{I}[[Inst]] \circ \mathcal{D}[[Decl]])\rho \\
\\
\mathcal{D}[[C(Id_c\ React\ Pin)]]\rho & = & \rho[<< \Psi[[React]], \top >, \mathcal{P}[[Pin]]\pi_0 > /Id_c] \\
\mathcal{D}[[Decl \diamond Decl']]\rho & = & \mathcal{D}[[Decl']] \circ \mathcal{D}[[Decl]]\rho \\
\\
\mathcal{P}[[\phi Id_p]]\pi & = & \pi[[\overset{\gg r}{G_0}, \overset{>r}{G_0}, \overset{s}{G_0}, \overset{s}{G_0}]\phi/Id_p] \\
\mathcal{P}[[Pin \diamond Pin']]\pi & = & \mathcal{P}[[Pin']] \circ \mathcal{P}[[Pin]]\pi \\
\\
\mathcal{I}[[Inst \diamond Inst']]\rho & = & \mathcal{I}[[Inst']] \circ \mathcal{I}[[Inst]]\rho \\
\mathcal{I}[[Id_c\ Id_i]]\rho & = & \rho[\Lambda\ (\rho\ Id_c)\ Id_i/Id_i] \\
\\
\mathcal{X}[[\rightsquigarrow (Id_i\ Id_r\ Id_i{}'\ Id_s)]]\rho & = & \rho[\Gamma\ (\rho\ Id_i)\ Id_s/Id_i, \Gamma\ (\rho\ Id_i')\ Id_r/Id_i'] \\
\mathcal{X}[[Wire \diamond Wire']]\rho & = & \mathcal{X}[[Wire']] \circ \mathcal{X}[[Wire]]\rho
\end{array}$$

Def. A.11 describes how the semantics maps each syntactic phrase in γ' to a corresponding domain in CSP. At the top level, the $\mathcal{A}$ "assembly" function composes component declarations, component instantiations, and wiring of component instances; the result of which is an environment ρ' that has one $CType$ value for each declared component type and one for each instantiated component type. *Gamma* constructs a parallel process to denote the assembly by parallel composing all component instance processes, along with all glue processes associated with those component instance processes.

The $\mathcal{D}$ "declaration" function directly constructs the declared component processes and glues; had there been additional syntactic phrases that could be declared (here there is only component types) there would have been a semantic domain for each alternative (for example, $\mathcal{C}$ for components), and $\mathcal{C}$ would have been used to construct the component value. In any case, the value is constructed using $< \ldots >$

tuple–forming operator, nested in this case because the first element of *CType* is itself a product domain, where we use $\top$ to denote a “type” process.

The $\mathcal{P}$ “pin declaration” function produces an environment *Cenv* that maps pin names (channel names) to their initial glue processes. The expression $\pi[[\overset{\gg r}{G_0}, \overset{>r}{G_0}, \overset{s}{G_0}, \overset{s}{G_0}]\phi/\mathrm{Id}_p]$ uses ϕ as an index into a vector of glue processes; note that both asynchronous and synchronous sink pins will map to the same initial $\overset{s}{G_0}$ sink glue.

The $\mathcal{I}$ “instantiation” function recovers the component type declaration from the environment and uses the auxiliary Λ function to produce a clone of the component type with each channel (in the component instance process and all glue processes of that instance) are prefixed by the instance name.

The $\mathcal{X}$ “wiring” function recovers from the environment the component processes associated with the instantiated components ($\rho\, Id_i$ and $\rho\, Id_i'$, respectively); source pin Id_r is used to extend its source glue with an interaction on Id_s, and analogously sink pin Id_s is used to extend its sink glue with an interaction on Id_r.

A.2 Reaction Semantics

A.2.1 Preliminaries

Pin components are composed at runtime from two constituents (see §7.3, pp. 132):

1. A Pin container.
2. Custom code (“Nub”) managed by the container.

The Pin container manages event queues for each component reaction (one queue per reaction per instance). When an event arrives on a reaction’s inbound event queue the container invokes a callback on the reaction called its *Reaction Handler*. The PCL reaction semantics describes the behavior of this reaction handler, assuming that the reaction is specified by PCL PinCharts.

One constraint enforced by the PCL frontend is that a PinChart is completely partitioned by two disjoint sets of *accepting* states and *reacting* states. Each implicitly or explicitly defined state in a PCL reaction belongs to exactly one of these sets. Informally:

- All outbound transitions from accepting states must be triggered by a ^sinkpin begin sink event, timed “**after**” event, or “**when**” change event.
- All outbound transitions from reacting states must be completions (defines no trigger) or be triggered by a $sourcepin end source event.

This partition is exploited by division of responsibility between the reaction handler and container. Essentially, reaction handlers return control to the container when they reach an accepting state; and while a reaction interacts with other components via its source pins, it remains in a reacting state.

The reaction handler specified (in pseudo–code) in §A.2.2 abstracts the actual execution of behavior encoded as transition actions, state actions, trigger definitions (to evaluate time and change conditions) and transition guards. In reaction handlers generated by PCL, the PinChart reaction is encoded directly in the body of the handler as an inline “C switch” statement, with each state modeled as a “case”

Reaction	Pin	Augmented for semantics
currState()	AUG	Current AST State.
env()	AUG	Function from names to locations
sto()	AUG	Function from locations to values
installTimeTriggers	Pin	Supported by Pin RTOS
cancelTimeTriggers	Pin	Supported by Pin RTOS
installChangeTriggers	Pin	Supported by Pin RTOS
cancelChangeTriggers	Pin	Supported by Pin RTOS
State	AST	Operations on states
Transition	AST	Operations on transitions
triggeredBy(e)	AST	True if transition is e–triggered
TransitionSet	AST	Set of transitions
triggeredBy(e)	AST	Set of e–triggered transitions
eval	AUG	Interpreter for PCL action language.

Table A.1: Pseudo–Classes for Reaction Semantics

in the switch statement. In this semantics, we refer (directly or indirectly) to an evaluation function (Def. A.12):

Definition A.12 (Abstracted Execution Environment)

$Loc = machine\ locations$

$DV = Loc + Int + Float + String + \ldots + Component + Assembly + \ldots$

$SV = Int + Float + String + Bool + \ldots$

$Env = Id \to DV$

$Store = Loc \to SV$

$void\ eval(Env\ \rho, Store\ \sigma, Actions\ \alpha)$

The *eval* function takes three arguments—an environment, a store, and a program. The value of a variable named "MyVar" is retrieved by two function applications: "$\sigma\ \rho$ MyVar" and, following notational conventions established earlier, updating the value of "MyVar = 0" is achieved by updating the store "$\sigma[0/\rho$ MyVar]."

An informal pseudo–code notation is used to define the behavior of the Nub reaction handler. Class–like abstractions are used by the pseudo–code to e.g. obtain the set of transitions defined on a PinChart state, to interact with the component runtime, etc. These abstractions (briefly summarized in Table A.1) do not correspond precisely to Pin Interfaces, but are convenient for exposition. For example, the Pin definition of Reaction does not include member functions for retrieving the current PinChart state, but this information is logically associated with reactions by PCL. Each interface is labeled as "Pin" if it is provided by the Pin component model, "AST" if it is provided by the abstract syntax tree produced by PCL, and "AUG" if it is an augmentation for exposition.

The pseudo–code for the reaction handler is specified in §A.2.2 and described in §A.2.3.

A.2.2 Reaction Handler

```
int handleEvent(Reaction r, Event e)
{
  TransitionSet triggered, enabled, completions;
  Transition firing;

  // accepting states "wait" for events from container
  assert(r.acceptingState(r.currState));

  // discard events that don't match triggers
  triggered = r.currState.triggeredBy(e);
  if ( triggered.empty() ) return;

  // evaluate guards on transitions triggered by e
  enabled = triggered.evalGuards(r.sto, r.env);

  // discard events that have no satisfied guards
  if (enabled.empty()) return;

  // cancel previous timers and watches
  r.cancelTimeTriggers();
  r.cancelChangeTriggers();

  while(1)
  {
    // if > 1 guard satisfied, make non-deterministic choice
    firing = enabled.nonDeterministicChoice();

    // execute transition actions
    eval(r.sto, r.env, firing.actions());

    // make the transition target the new state
    r.currState = firing.targetState();

    // install timers for time triggered transitions
    // start time events relative to state entry
    if (r.acceptingStates(r.currState))
    {
      r.installTimeTriggers(r.CurrState);
    }

    // execute state's entry (and only) actions
    eval(r.sto, r.env, r.currState.actions());

    // install watches for change triggered transitions
    if (r.acceptingState(r.currState))
    {
      // returns all watches that evaluated as true
      enabled = r.installChangeTriggers(r.CurrState).
49          evalGuards(r.sto, r.env);
50      }
51      else // r.reactingState(r.currState)
52      {
53        // $source triggers are implicit
54        enabled = r.currState.
55          completions().evalGuards(r.sto, r.env);
56      }
57
58      // return control to container or be STUCK
59      if (enabled.empty())
60      {
61        if ( r.isAcceptingState(r.currstate) )
62        {
63          // wait for next event
64          return;
65        }
66        else // r.reactingStates(r.currState)
67        {
68          STUCK();
69        }
70      }
71    }
72  }
```

A.2.3 Reaction Handler Description

7 Components are in an accepting state when waiting on events, and reacting states when handling events. A reaction is always in either an accepting state or a reacting state; these are disjoint sets that partition the set of reaction states.

10–11 Events are discarded that do not have corresponding triggers on outbound transitions from the current state.

13–17 Events are discarded if no guards on event–triggered transitions are satisfied (i.e., evaluate to True). All transitions that match the event and whose guards are satisfied are considered to be *enabled*.

20–21 The TimeTrigger and ChangeTrigger interfaces (which are implicit in the method names used here) are provided by the Pin to support UML time and change events. Prior to "leaving" a state all timers (corresponding to PCL '**after**' triggers) and all watches (corresponding to PCL '**when**' triggers) are cancelled. All timer events are purged from the event queue; change events are not purged.

23 The Nub remains in control until it reaches an accepting state and returns control to the container.

26 If more than one transition is enabled, then one is chosen non-deterministically to be *fired*.

29 The first step in firing a transition is to execute the transition actions. r.env() is a function from names to locations, and r.sto() is a function from locations to values; eval() is an interpreter for PCL statements. In the PSK implementation, eval(), sto(), and env() are replaced by generated inline C code.

32 The next step in firing a transition is to change the current state to the target state of the transition.

36–39 PCL enforces the rule that only accepting states have time–triggered transitions. Timers are started relative to the state's entry point (i.e., when r.currState is updated).

42 The next step in firing a transition is to execute the state's actions.

45–56 If the current state is an accepting state, change triggers are installed for change–triggered transitions (if any); otherwise the state is a reacting state which are either *triggerless* (are "completions" in UML terminology) or are $sourcePin triggered. The latter case are handed implicitly by the reaction's use of SendOutSourcePin and SendOutSourcePinWait methods provided by Pin's *ContainerInterfaces* interface (see Table 7.2, pp. 139). At the end of this block, the transition has fired and is regarded as "completed."

48 installChangeTriggers installs the evaluates the (side–effect free) condition specified by the PCL **when** condition clause; the set of transitions that satisfy their conditions are returned, and their guards evaluated, with those transitions satisfying the guard considered to be *enabled*.

61–65 If the new state is *accepting* state, then either a) at least one change event is enabled and must be fired (i.e., continue with the next iteration of the **while** loop initiated on line 23), or there are no enabled transitions in which case control must be returned to the reaction's (component instance's) container.

66–69 If the new state is a *reacting* state, then either a) at least one completion (including implicit $sourcePin triggered transitions) is enabled and must be fired (continue with the next loop iteration), or no transition is enabled, in which case the reaction is deadlocked. This latter condition can only arise in PinCharts when all guard conditions on outbound transitions of reacting states evaluate to False.

Appendix B

Examples from Soft P&C Case Study

B.1 PCL Assembly Specification for SoftPC–A

Example B.1: PCL Assembly Specification of Soft P&C SPC–A.

```

///--- include global declarations
#include "Common/GlobalDeclarations.ccl"
#include "Common/LN0IncludeTemplate.ccl"

//--- include procedures and functions
#include "Common/Includes/CommandIsEquals.ccl"
#include "Common/Includes/GetBreakerPositionAsString.ccl"
#include "Common/Includes/GetBreakerPositionAsType.ccl"
#include "Common/Includes/IsSelf.ccl"
#include "Common/Includes/IsBroadcast.ccl"
#include "Common/Includes/ComposeProxyCommand.ccl"
#include "Common/Includes/ComposeAcknowledgeCommand.ccl"


//--- include boundary components, shield boundary services
#include "HelperComponents/PinDebugOutput.ccl"
#include "HelperComponents/CSWIGenerator.ccl"

//--- include logical nodes
#include "LogicalNodes/C/CILO.ccl"
#include "LogicalNodes/C/CSWI.ccl"
#include "LogicalNodes/X/XSWI.ccl"
#include "LogicalNodes/X/XCBR.ccl"
#include "LogicalNodes/P/PTOC.ccl"
#include "LogicalNodes/P/PTOV.ccl"
#include "LogicalNodes/P/PDIF.ccl"
#include "LogicalNodes/P/PTRC.ccl"
#include "LogicalNodes/M/MDIF.ccl"
#include "LogicalNodes/M/MMXU.ccl"

//--- include proxy components
#include "ServiceComponents/GOOSEListener.ccl"
#include "ServiceComponents/GOOSESender.ccl"
#include "ServiceComponents/SAVSender.ccl"

//--- include LN proxy components
#include "ProxyComponents/T/TCTRProxy.ccl"
#include "ProxyComponents/T/TVTRProxy.ccl"
#include "ProxyComponents/T/TXTRProxy.ccl"
#include "ProxyComponents/X/XCBRProxy.ccl"
#include "ProxyComponents/X/XSWIProxy.ccl"
#include "ProxyComponents/I/IHMIProxy.ccl"

//--- include other service components
#include "ServiceComponents/SAVSynchronizer.ccl"
```

```

environment RTX() {

//—— include boundary services
#include "ServiceComponents/SAVListenerProxy.ccl"
#include "ServiceComponents/SAVSenderProxy.ccl"
#include "ServiceComponents/GOOSEListenerProxy.ccl"
#include "ServiceComponents/GOOSESenderProxy.ccl"
}

assembly SPCAAssembly() (RTX) {

  assume {
    RTX:SAVListenerProxy   sav_in();
    RTX:SAVSenderProxy     sav_out();
    RTX:GOOSEListenerProxy  goose_in();
    RTX:GOOSESenderProxy   goose_out();
  }

// Proxy Components

  const TUniqueDeviceID uniqueDeviceIDgl =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCA};
  GOOSEListener gooseListener(
    uniqueDeviceIDgl, C_MacAddress_SPCA);
  goose_in:CommandWithMac ~> gooseListener:CommandWithMac;

  annotate goose_in:Main
    {"Pin", const int priority = 121}
  annotate gooseListener:External
    {"Pin", const int priority = 60}

  const TUniqueDeviceID uniqueDeviceIDgs =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCA};
  GOOSESender gooseSender(
    uniqueDeviceIDgs, C_MacAddress_SPCA);
  gooseSender:CommandWithMac ~> goose_out:CommandWithMac;

  annotate goose_out:External
    {"Pin", const int priority = 124}
  annotate goose_out:External
    {"Pin", const int queueLength = 100}
  annotate gooseSender:External
    {"Pin", const int priority = 63}

  SAVSender savSender(
    C_MacAddress_SPCB, C_MacAddress_SPCA,
    C_MergingUnitName_SPCA, C_PulsePerSecond);
  savSender:IU3NOut ~> sav_out:IU3N;

  annotate sav_out:Main
    {"Pin", const int priority = 123}
  annotate sav_out:Main
    {"Pin", const int queueLength = 100}
  annotate savSender:Main
    {"Pin", const int priority = 119}

// TCTR

  TCTRProxy tctrProxy1(
    C_MergingUnitName_MU1, C_SystemParameters);
  sav_in:SAV ~> tctrProxy1:SAV;
  tctrProxy1:I3N ~> savSender:I3N;

  annotate sav_in:Main
    {"Pin", const int priority = 122}
  annotate tctrProxy1:Main
    {"Pin", const int priority = 116}

// TVTR

  TVTRProxy tvtrProxy1(
    C_MergingUnitName_MU1, C_SystemParameters);
  sav_in:SAV ~> tvtrProxy1:SAV;

  annotate tvtrProxy1:Main
    {"Pin", const int priority = 117}

// TXTR --> for MMXU --> a bundled Current/Voltage stream

  TXTRProxy txtrProxy1(
    C_MergingUnitName_MU1, C_SystemParameters);
  sav_in:SAV ~> txtrProxy1:SAV;

  annotate txtrProxy1:Main
    {"Pin", const int priority = 30}

// Input from SPCB as input to Synchronizer -> MDIF -> PDIF

  TCTRProxy tctrProxy2(
    C_MergingUnitName_MU2, C_SystemParameters);
  sav_in:SAV ~> tctrProxy2:SAV;

  annotate tctrProxy2:Main
    {"Pin", const int priority = 115}

// PTOC

```

```
  const TUniqueLNID uniqueLNIDptoc1 =
    {C_PhysicalDeviceName,
    C_LogicalDeviceName_SPCA, "PTOC", "PTOC1"};
 PTOC ptoc1(
    uniqueLNIDptoc1, C_SystemParameters,
    Phase1, 20000, C_ResetManual);
  tctrProxy1:I3N ~> ptoc1:I3N;
  gooseListener:CmdOutConfig ~> ptoc1:CommandIn;
  ptoc1:CommandOut ~> gooseSender:CommandIn;

  annotate ptoc1:Main  {"Pin", const int priority = 95}

  const TUniqueLNID uniqueLNIDptoc2 =
    {C_PhysicalDeviceName,
    C_LogicalDeviceName_SPCA, "PTOC", "PTOC2"};
 PTOC ptoc2(
    uniqueLNIDptoc2, C_SystemParameters,
    Phase2, 20000, C_ResetManual);
  tctrProxy1:I3N ~> ptoc2:I3N;
  gooseListener:CmdOutConfig ~> ptoc2:CommandIn;
  ptoc2:CommandOut ~> gooseSender:CommandIn;

  annotate ptoc2:Main  {"Pin", const int priority = 96}

  const TUniqueLNID uniqueLNIDptoc3 =
    {C_PhysicalDeviceName,
    C_LogicalDeviceName_SPCA, "PTOC", "PTOC3"};
 PTOC ptoc3(
    uniqueLNIDptoc3, C_SystemParameters,
    Phase3, 20000, C_ResetManual);
  tctrProxy1:I3N ~> ptoc3:I3N;
  gooseListener:CmdOutConfig ~> ptoc3:CommandIn;
  ptoc3:CommandOut ~> gooseSender:CommandIn;

  annotate ptoc3:Main  {"Pin", const int priority = 97}

  const TUniqueLNID uniqueLNIDptoc0 =
    {C_PhysicalDeviceName,
    C_LogicalDeviceName_SPCA, "PTOC", "PTOC0"};
 PTOC ptoc0(
    uniqueLNIDptoc0, C_SystemParameters,
    PhaseN, 20000, C_ResetManual);
  tctrProxy1:I3N ~> ptoc0:I3N;
  gooseListener:CmdOutConfig ~> ptoc0:CommandIn;
  ptoc0:CommandOut ~> gooseSender:CommandIn;

  annotate ptoc0:Main  {"Pin", const int priority = 98}

// PTRC from PTOC
```

```

  const TUniqueLNID uniqueLNIDptrc1 =
    {C_PhysicalDeviceName,
    C_LogicalDeviceName_SPCA, "PTRC", "PTRC1"};
  PTRC ptrc1(
    uniqueLNIDptrc1, 1, 4,
    C_SystemParameters, true, true, true, true);
  gooseListener:CmdOutConfig ~> ptrc1:CommandIn;
  ptrc1:CommandOut ~> gooseSender:CommandIn;
  ptoc1:Start ~> ptrc1:Start1;
  ptoc2:Start ~> ptrc1:Start2;
  ptoc3:Start ~> ptrc1:Start3;
  ptoc0:Start ~> ptrc1:Start4;
  ptoc1:Trip ~> ptrc1:Trip1;
  ptoc2:Trip ~> ptrc1:Trip2;
  ptoc3:Trip ~> ptrc1:Trip3;
  ptoc0:Trip ~> ptrc1:Trip4;

  annotate ptrc1:Main  {"Pin", const int priority = 86}
  annotate ptrc1:Main  {"Pin", const int queueLength = 100}

// PTOV

  const TUniqueLNID uniqueLNIDptov1 =
    {C_PhysicalDeviceName,
    C_LogicalDeviceName_SPCA, "PTOV", "PTOV1"};
  PTOV ptov1(
    uniqueLNIDptov1, C_SystemParameters,
    Phase1, 50000, C_ResetManual, 20);
     // need 20 voltage samples before trip positive
  tvtrProxy1:U3N ~> ptov1:U3N;
  gooseListener:CmdOutConfig ~> ptov1:CommandIn;
  ptov1:CommandOut ~> gooseSender:CommandIn;

  annotate ptov1:Main  {"Pin", const int priority = 100}

  const TUniqueLNID uniqueLNIDptov2 =
    {C_PhysicalDeviceName,
    C_LogicalDeviceName_SPCA, "PTOV", "PTOV2"};
  PTOV ptov2(
    uniqueLNIDptov2, C_SystemParameters,
    Phase2, 50000, C_ResetManual, 20);
    // need to see 20 voltage samples before trip positive
  tvtrProxy1:U3N ~> ptov2:U3N;
  gooseListener:CmdOutConfig ~> ptov2:CommandIn;
  ptov2:CommandOut ~> gooseSender:CommandIn;

  annotate ptov2:Main  {"Pin", const int priority = 101}

  const TUniqueLNID uniqueLNIDptov3 =
    {C_PhysicalDeviceName,
    C_LogicalDeviceName_SPCA, "PTOV", "PTOV3"};
  PTOV ptov3(
    uniqueLNIDptov3, C_SystemParameters,
    Phase3, 50000, C_ResetManual, 20);
    // need to see 20 voltage samples before trip positive
  tvtrProxy1:U3N ~> ptov3:U3N;
  gooseListener:CmdOutConfig ~> ptov3:CommandIn;
  ptov3:CommandOut ~> gooseSender:CommandIn;

  annotate ptov3:Main   {"Pin", const int priority = 102}

  const TUniqueLNID uniqueLNIDptov0 =
    {C_PhysicalDeviceName,
    C_LogicalDeviceName_SPCA, "PTOV", "PTOV0"};
  PTOV ptov0(
    uniqueLNIDptov0, C_SystemParameters,
    PhaseN, 50000, C_ResetManual, 20);
    // need to see 20 voltage samples before trip positive
  tvtrProxy1:U3N ~> ptov0:U3N;
  gooseListener:CmdOutConfig ~> ptov0:CommandIn;
  ptov0:CommandOut ~> gooseSender:CommandIn;

  annotate ptov0:Main   {"Pin", const int priority = 103}

// PTRC from PTOV

  const TUniqueLNID uniqueLNIDptrc2 =
    {C_PhysicalDeviceName,
    C_LogicalDeviceName_SPCA, "PTRC", "PTRC2"};
  PTRC ptrc2(
    uniqueLNIDptrc2, 1, 4,
    C_SystemParameters, true, true, true, true);
  gooseListener:CmdOutConfig ~> ptrc2:CommandIn;
  ptrc2:CommandOut ~> gooseSender:CommandIn;
  ptov1:Start ~> ptrc2:Start1;
  ptov2:Start ~> ptrc2:Start2;
  ptov3:Start ~> ptrc2:Start3;
  ptov0:Start ~> ptrc2:Start4;
  ptov1:Trip ~> ptrc2:Trip1;
  ptov2:Trip ~> ptrc2:Trip2;
  ptov3:Trip ~> ptrc2:Trip3;
  ptov0:Trip ~> ptrc2:Trip4;

  annotate ptrc2:Main   {"Pin", const int priority = 87}
  annotate ptrc2:Main   {"Pin", const int queueLength = 100}

// Synchronizer
```

```
  const TUniqueLNID uniqueLNIDsync1 =
    {C_PhysicalDeviceName,
    C_LogicalDeviceName_SPCA, "SYNC", "SYNC1"};
  SAVSynchronizer savSynchronizer(
    uniqueLNIDsync1, C_SystemParameters);
  tctrProxy1:I3N ~> savSynchronizer:stream1;
  tctrProxy2:I3N ~> savSynchronizer:stream2;
  gooseListener:CmdOutConfig ~> savSynchronizer:CommandIn;
  savSynchronizer:CommandOut ~> gooseSender:CommandIn;

  annotate savSynchronizer:Main
    {"Pin", const int priority = 94}

// MDIF
  const TUniqueLNID uniqueLNIDmdif1 =
    {C_PhysicalDeviceName,
    C_LogicalDeviceName_SPCA, "MDIF", "MDIF1"};
  MDIF mdif1(
    uniqueLNIDmdif1, C_SystemParameters, C_ResetManual);
  gooseListener:CmdOutConfig ~> mdif1:CommandIn;
  mdif1:CommandOut ~> gooseSender:CommandIn;
  savSynchronizer:synchronizedStream ~> mdif1:IU3NSync;

  annotate mdif1:Main   {"Pin", const int priority = 93}

// PDIF
  const TUniqueLNID uniqueLNIDpdif1 =
    {C_PhysicalDeviceName,
    C_LogicalDeviceName_SPCA, "PDIF", "PDIF1"};
  PDIF pdif1(
    uniqueLNIDpdif1, C_SystemParameters,
    500, C_ResetManual);
  gooseListener:CmdOutConfig ~> pdif1:CommandIn;
  pdif1:CommandOut ~> gooseSender:CommandIn;
  mdif1:Diff1 ~> pdif1:Diff;

  annotate pdif1:Main   {"Pin", const int priority = 90}

  const TUniqueLNID uniqueLNIDpdif2 =
    {C_PhysicalDeviceName,
    C_LogicalDeviceName_SPCA, "PDIF", "PDIF2"};
  PDIF pdif2(
    uniqueLNIDpdif2, C_SystemParameters,
    500, C_ResetManual);
  gooseListener:CmdOutConfig ~> pdif2:CommandIn;
  pdif2:CommandOut ~> gooseSender:CommandIn;
  mdif1:Diff2 ~> pdif2:Diff;

  annotate pdif2:Main   {"Pin", const int priority = 91}

  const TUniqueLNID uniqueLNIDpdif3 =
    {C_PhysicalDeviceName,
    C_LogicalDeviceName_SPCA, "PDIF", "PDIF3"};
  PDIF pdif3(
    uniqueLNIDpdif3, C_SystemParameters,
    500, C_ResetManual);
  gooseListener:CmdOutConfig ~> pdif3:CommandIn;
  pdif3:CommandOut ~> gooseSender:CommandIn;
  mdif1:Diff3 ~> pdif3:Diff;

  annotate pdif3:Main  {"Pin", const int priority = 92}

// PTRC from PDIF

  const TUniqueLNID uniqueLNIDptrc3 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCA,
    "PTRC", "PTRC3"};
  PTRC ptrc3(
    uniqueLNIDptrc3, 1, 3,
    C_SystemParameters, true, true, true, false);
  gooseListener:CmdOutConfig ~> ptrc3:CommandIn;
  ptrc3:CommandOut ~> gooseSender:CommandIn;
  pdif1:Start ~> ptrc3:Start1;
  pdif2:Start ~> ptrc3:Start2;
  pdif3:Start ~> ptrc3:Start3;
  pdif1:Trip ~> ptrc3:Trip1;
  pdif2:Trip ~> ptrc3:Trip2;
  pdif3:Trip ~> ptrc3:Trip3;

  annotate ptrc3:Main  {"Pin", const int priority = 85}
  annotate ptrc3:Main  {"Pin", const int queueLength = 100}

// PTRC from PTRC1, PTRC2, PTRC3

  const TUniqueLNID uniqueLNIDptrc4 =
    {C_PhysicalDeviceName,
    C_LogicalDeviceName_SPCA, "PTRC", "PTRC4"};
  PTRC ptrc4(
    uniqueLNIDptrc4, 1, 1,
    C_SystemParameters, true, true, true, false);
  gooseListener:CmdOutConfig ~> ptrc4:CommandIn;
  ptrc4:CommandOut ~> gooseSender:CommandIn;
  ptrc1:Start ~> ptrc4:Start1;
  ptrc2:Start ~> ptrc4:Start2;
  ptrc3:Start ~> ptrc4:Start3;
  ptrc1:Trip ~> ptrc4:Trip1;
  ptrc2:Trip ~> ptrc4:Trip2;
  ptrc3:Trip ~> ptrc4:Trip3;

  annotate ptrc4:Main   {"Pin", const int priority = 125}
  annotate ptrc4:Main   {"Pin", const int queueLength = 100}

// Q0 - XCBR intance resides in another assembly

  const TUniqueLNID uniqueLNIDxcbrproxy1 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCA,
    "XCBRProxy", "XCBR0"};
  const TUniqueLNID uniqueLNIDxcbr1 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCXA,
    "XCBR", "XCBR0"};
  XCBRProxy xcbrProxy1(
    uniqueLNIDxcbrproxy1, uniqueLNIDxcbr1, "Q0",
    C_MacAddress_SPCA, C_MacAddress_SPCX);
  xcbrProxy1:CommandWithMacOut ~> goose_out:CommandWithMac;

  annotate xcbrProxy1:External
    {"Pin", const int priority = 127}

// Q1 - XSWI
  const TUniqueLNID uniqueLNIDxswiproxy1 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCA,
    "XSWI", "XSWI1"};
  const TUniqueLNID uniqueLNIDxswi1 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCXA,
    "XSWI", "XSWI1"};
  XSWIProxy xswiProxy1(
    uniqueLNIDxswiproxy1, uniqueLNIDxswi1,
    "Q1", C_MacAddress_SPCA, C_MacAddress_SPCX);
  xswiProxy1:CommandWithMacOut ~> goose_out:CommandWithMac;

  annotate xswiProxy1:External
    {"Pin", const int priority = 55}

// Q2 - XSWI
  const TUniqueLNID uniqueLNIDxswiproxy2 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCA,
    "XSWI", "XSWI2"};
  const TUniqueLNID uniqueLNIDxswi2 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCXA,
    "XSWI", "XSWI2"};
  XSWIProxy xswiProxy2(
    uniqueLNIDxswiproxy2, uniqueLNIDxswi2, "Q2",
    C_MacAddress_SPCA, C_MacAddress_SPCX);
  xswiProxy2:CommandWithMacOut ~> goose_out:CommandWithMac;

  annotate xswiProxy2:External
    {"Pin", const int priority = 56}
```

```

// Q8 - earth switch - XSWI
  const TUniqueLNID uniqueLNIDxswiproxy3 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCA,
    "XSWI", "XSWI3"};
  const TUniqueLNID uniqueLNIDxswi3 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCXA,
    "XSWI", "XSWI3"};
  XSWIProxy xswiProxy3(
    uniqueLNIDxswiproxy3, uniqueLNIDxswi3, "Q8",
    C_MacAddress_SPCA, C_MacAddress_SPCX);
  xswiProxy3:CommandWithMacOut ~> goose_out:CommandWithMac;

  annotate xswiProxy3:External
    {"Pin", const int priority = 57}

// Q9 - XSWI
  const TUniqueLNID uniqueLNIDxswiproxy4 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCA,
    "XSWI", "XSWI4"};
  const TUniqueLNID uniqueLNIDxswi4 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCXA,
    "XSWI", "XSWI4"};
  XSWIProxy xswiProxy4(
    uniqueLNIDxswiproxy4, uniqueLNIDxswi4, "Q9",
    C_MacAddress_SPCA, C_MacAddress_SPCX);
  xswiProxy4:CommandWithMacOut ~> goose_out:CommandWithMac;

  annotate xswiProxy4:External
    {"Pin", const int priority = 58}


// wire trip signals

  ptrc4:Trip ~> xcbrProxy1:Trip;


/*
every component checks if a command is targeted for itself
(IsSelf) or is a broadcast command (IsBroadcast) and then
processes the command.
*/
  const TUniqueLNID uniqueLNIDihmiProxy1 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCA,
    "IHMIProxy", "IHMI1Proxy"};
  const TUniqueLNID uniqueLNIDihmi1ReportTo =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCIHMI,
    "IHMI", "IHMI1"};
  IHMIProxy ihmiproxy1(
    uniqueLNIDihmiProxy1, uniqueLNIDihmi1ReportTo,
    C_MacAddress_SPCA, C_MacAddress_SPCIHMI);
  ihmiproxy1:CommandWithMacOut ~> goose_out:CommandWithMac;

  annotate ihmiproxy1:External
    {"Pin", const int priority = 20}

//------------------ C***


// CSWI0
  const TUniqueLNID uniqueLNIDcswi0 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCA,
    "CSWI", "CSWI0"};
  CSWI cswi0(uniqueLNIDcswi0);
  cswi0:OperateOut ~> xcbrProxy1:Operate;
  gooseListener:CmdOutConfig ~> cswi0:CommandIn;
  cswi0:CommandOut ~> gooseSender:CommandIn;

  annotate cswi0:Main   {"Pin", const int priority = 50}

// CSWI1
  const TUniqueLNID uniqueLNIDcswi1 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCA,
    "CSWI", "CSWI1"};
  CSWI cswi1(uniqueLNIDcswi1);
  cswi1:OperateOut ~> xswiProxy1:Operate;
  gooseListener:CmdOutConfig ~> cswi1:CommandIn;
  cswi1:CommandOut ~> gooseSender:CommandIn;

  annotate cswi1:Main   {"Pin", const int priority = 51}

// CSWI2
  const TUniqueLNID uniqueLNIDcswi2 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCA,
    "CSWI", "CSWI2"};
  CSWI cswi2(uniqueLNIDcswi2);
  cswi2:OperateOut ~> xswiProxy2:Operate;
  gooseListener:CmdOutCommands ~> cswi2:CommandIn;
  cswi2:CommandOut ~> gooseSender:CommandIn;

  annotate cswi2:Main   {"Pin", const int priority = 52}

// CSWI3
  const TUniqueLNID uniqueLNIDcswi3 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCA,
    "CSWI", "CSWI3"};
  CSWI cswi3(uniqueLNIDcswi3);
  cswi3:OperateOut ~> xswiProxy3:Operate;
  gooseListener:CmdOutCommands ⇝ cswi3:CommandIn;
  cswi3:CommandOut ⇝ gooseSender:CommandIn;

  annotate cswi3:Main  {"Pin", const int priority = 53}

// CSWI4
  const TUniqueLNID uniqueLNIDcswi4 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCA,
    "CSWI", "CSWI4"};
  CSWI cswi4(uniqueLNIDcswi4);
  cswi4:OperateOut ⇝ xswiProxy4:Operate;
  gooseListener:CmdOutCommands ⇝ cswi4:CommandIn;
  cswi4:CommandOut ⇝ gooseSender:CommandIn;

  annotate cswi4:Main  {"Pin", const int priority = 54}


// cilo subscribers
  const TUniqueLNIDArray ciloSubArray = {
    uniqueLNIDxcbr1[0], uniqueLNIDxcbr1[1],
    uniqueLNIDxcbr1[2], uniqueLNIDxcbr1[3],
    uniqueLNIDxswi1[0], uniqueLNIDxswi1[1],
    uniqueLNIDxswi1[2], uniqueLNIDxswi1[3],
    uniqueLNIDxswi2[0], uniqueLNIDxswi2[1],
    uniqueLNIDxswi2[2], uniqueLNIDxswi2[3],
    uniqueLNIDxswi3[0], uniqueLNIDxswi3[1],
     uniqueLNIDxswi3[2], uniqueLNIDxswi3[3],
    uniqueLNIDxswi4[0], uniqueLNIDxswi4[1],
    uniqueLNIDxswi4[2], uniqueLNIDxswi4[3],
    "", "", "", "",
    "", "", "", "",
    "", "", "", "",
    "", "", "", "",
    "", "", "", ""
  };

  const TUniqueLNID uniqueLNIDcilo0 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCA,
    "CILO", "CILO0"};
  CILO cilo0(uniqueLNIDcilo0, 5, ciloSubArray);
  gooseListener:CmdOutCommands ⇝ cilo0:CommandIn;
  gooseListener:CmdOutConfig ⇝ cilo0:CommandIn;
  gooseListener:CmdOutHardware ⇝ cilo0:CommandIn;
  cilo0:CommandOut ⇝ gooseSender:CommandIn;

  cswi0:CanExecute ⇝ cilo0:CanExecute;
  cswi1:CanExecute ⇝ cilo0:CanExecute;
  cswi2:CanExecute ⇝ cilo0:CanExecute;
  cswi3:CanExecute ⇝ cilo0:CanExecute;
  cswi4:CanExecute ~> cilo0:CanExecute;

  annotate cilo0:Main   {"Pin", const int priority = 126}

// connect position change commands to CILO
  gooseListener:CmdOutHardware ~> cilo0:CommandIn;
  xcbrProxy1:CommandOut ~> cilo0:CommandIn;
  xswiProxy1:CommandOut ~> cilo0:CommandIn;
  xswiProxy2:CommandOut ~> cilo0:CommandIn;
  xswiProxy3:CommandOut ~> cilo0:CommandIn;
  xswiProxy4:CommandOut ~> cilo0:CommandIn;


// ------------------- M****

// MMXU
  const TUniqueLNID uniqueLNIDmmxu1 =
    {C_PhysicalDeviceName, C_LogicalDeviceName_SPCA,
    "MMXU", "MMXU1"};
  MMXU mmxu1(
    uniqueLNIDmmxu1, C_SystemParameters, 10, false);
  gooseListener:CmdOutConfig ~> mmxu1:CommandIn;
  mmxu1:CommandOut ~> gooseSender:CommandIn;
  txtrProxy1:IU3N ~> mmxu1:IU3N;
  mmxu1:CommandOutReport ~> ihmiproxy1:CommandInReport;

  annotate mmxu1:Main   {"Pin", const int priority = 25}

  expose { }
}

// annotate external applications needed to start up
typedef string TString2[2];
annotate SPCAAssembly {
  "Platform",
  const TString2 driver = { // strings truncated for listing
    "%CCL_GENERATED_CODE_DIR%\..",
    "%CCL_GENERATED_CODE_DIR%\.."
  }
}


RTX env() {
  RTX:SAVListenerProxy    sav_in_env();
  RTX:SAVSenderProxy      sav_out_env();
  RTX:GOOSEListenerProxy  goose_in_env();
  RTX:GOOSESenderProxy    goose_out_env();
};
```

```
SPCAAssembly SPCAApp3() {
  SPCAAssembly:sav_in = env:sav_in_env;
  SPCAAssembly:sav_out = env:sav_out_env;
  SPCAAssembly:goose_in = env:goose_in_env;
  SPCAAssembly:goose_out = env:goose_out_env;
};

//
// lambda* performance reasoning framework annotations
//
annotate SPCAAssembly:sav_in
  {"period", const int period = 1000 }
annotate SPCAAssembly:goose_in
  {"period", const int period = 1000 }

annotate SPCAAssembly:mmxu1:IU3N
  {"lambda*", const int downsamplingFactor = 1 }

//
// include file for benchmarks performed on
pcaof.sei.cmu.edu
//
#include "Common/lambdaStar/pcaof.sei.cmu.edu.ccl"

annotate env {
  "lambda*",
  const float connectionOverhead = 7.168 + 4.55 }

const int SCN_IGNORE = 0;
const int SCN_SAV = 1;
const int SCN_GOOSE = 2;
const int SCN_MONITOR = 4; // I can add scenarios
const int SCN_TRIP = 8;
const int SCN_UNKNOWN = 16;
typedef int TwoScenarios[2];
typedef int ThreeScenarios[3];

// annotate all the SAV pins (blue SCN_SAV)
annotate SPCAAssembly:sav_in:SAV
  { "scenario", const int scenario = SCN_SAV }
annotate TVTRProxy:U3N
  { "scenario", const int scenario = SCN_SAV }
annotate TCTRProxy:I3N
  { "scenario", const int scenario = SCN_SAV }
annotate PTOV:Start
  { "scenario", const int scenario = SCN_SAV }
annotate PTOV:Trip
  { "scenario", const int scenario = SCN_SAV }
annotate PTOC:Start
```

```
  { "scenario", const int scenario = SCN_SAV }
annotate PTOC:Trip
  { "scenario", const int scenario = SCN_SAV }
annotate PDIF:Start
  { "scenario", const int scenario = SCN_SAV }
annotate PDIF:Trip
  { "scenario", const int scenario = SCN_SAV }
annotate SAVSynchronizer:synchronizedStream
  { "scenario", const int scenario = SCN_IGNORE }
annotate MDIF:Diff3NSAV
  { "scenario", const int scenario = SCN_SAV }

// annotate the TRIP pins (black dashed SCN_TRIP)
annotate PTRC:Start
  { "scenario", const int scenario = SCN_TRIP }
annotate PTRC:Trip
  { "scenario", const int scenario = SCN_TRIP }
annotate XCBRProxy:CommandWithMacOut
  { "scenario", const int scenario = SCN_TRIP }

// annotate the MONITOR pins (purple SCN_MONITOR)
annotate TXTRProxy:IU3N
  { "scenario", const int scenario = SCN_MONITOR }
annotate TXTRProxy:SAV
  { "scenario", const int scenario = SCN_MONITOR }
annotate MMXU:CommandOutReport
  { "scenario", const int scenario = SCN_MONITOR }
annotate IHMIProxy:CommandWithMacOut
  { "scenario", const int scenario = SCN_MONITOR }

// annotate all GOOSE pins (green SCN_GOOSE)
annotate SPCAAssembly:goose_in:CommandWithMac
  { "scenario", const int scenario = SCN_GOOSE }
annotate GOOSEListener:CmdOutCommands
  { "scenario", const int scenario = SCN_GOOSE }
annotate GOOSEListener:CmdOutHardware
  { "scenario", const int scenario = SCN_GOOSE }

// annotate the UNKNOWNS pins (orange SCN_UNKNOWN)
annotate CSWI:OperateOut
  { "scenario", const int scenario = SCN_UNKNOWN }
annotate CSWI:CanExecute
  { "scenario", const int scenario = SCN_UNKNOWN }
annotate XCBRProxy:CommandOut
  { "scenario", const int scenario = SCN_UNKNOWN }
annotate XSWIProxy:CommandWithMacOut
  { "scenario", const int scenario = SCN_UNKNOWN }
annotate XSWIProxy:CommandOut
  { "scenario", const int scenario = SCN_UNKNOWN }
```

```
annotate SAVSender:IU3NOut
  { "scenario", const int scenario = SCN_UNKNOWN }

// all the sources to be ignored by the interpretation
annotate PTOV:CommandOut
  { "scenario", const int scenario = SCN_IGNORE }
annotate PTOC:CommandOut
  { "scenario", const int scenario = SCN_IGNORE }
annotate PTRC:CommandOut
  { "scenario", const int scenario = SCN_IGNORE }
annotate MDIF:CommandOut
  { "scenario", const int scenario = SCN_IGNORE }
annotate PDIF:CommandOut
  { "scenario", const int scenario = SCN_IGNORE }
annotate CSWI:CommandOut
  { "scenario", const int scenario = SCN_IGNORE }
annotate GOOSESender:CommandWithMac
  { "scenario", const int scenario = SCN_IGNORE }

annotate TCTRProxy:CommandOut
  { "scenario", const int scenario = SCN_IGNORE }
annotate TVTRProxy:CommandOut
  { "scenario", const int scenario = SCN_IGNORE }
annotate TXTRProxy:CommandOut
  { "scenario", const int scenario = SCN_IGNORE }
annotate CILO:CommandOut
  { "scenario", const int scenario = SCN_IGNORE }
annotate GOOSEListener:CommandOut
  { "scenario", const int scenario = SCN_IGNORE }
annotate GOOSEListener:CmdOutConfig
  { "scenario", const int scenario = SCN_IGNORE }
annotate SAVSender:CommandOut
  { "scenario", const int scenario = SCN_IGNORE }
annotate IHMIProxy:CommandIn
  { "scenario", const int scenario = SCN_IGNORE }
```

B.2 PCL Component Specification for PTRC

Example B.2: PCL Component Specification of Soft P&C PTRC.

```



// Logical Node Group: P (protection functions)
//
// Assumptions
// SAV samples are received on a per-synch-basis
//
// (PIN) Common
// Direction   Name     Description
// <-    Start    indicates that the function entered the
"supervision" mode
// <- Trip     a trip is that a fault has occured and needs
to be clared
// ->    Command    Command input
// <-    Acknowledge Command acknowledge output
// ->    Start(1..4) Start inputs
// ->    Trip(1..4)   Trip inputs
//
// (Commands) Common
// Type    Name     ParameterName    Description
// CMD     SET, GET  OPERATIONMODE    get's or set's the
operation mode
// ACK     SET, GET  OPERATIONMODE    acknowledge for set/get
operation mode request
// CMD     SET, GET  PARAMETER      get or set a function
parameter
// ACK     SET, GET  PARAMETER      acknowledge for set/get
function parameter

#include "Common//GlobalDeclarations.ccl"
#include "Common//Includes//ComposeAcknowledgeCommand.ccl"

typedef boolean TBoolArray[100];

component PTRC(
  TUniqueLNID uid,
  int delay,
  int xOutOf4,
  TSystemParameters systemParameters,
  boolean channelActive1,
  boolean channelActive2,
  boolean channelActive3,
  boolean channelActive4)
{

  // sinks
  // incoming from SAVListener
  sink asynch Start1(
    consume TUniqueLNID xSource,
    consume boolean xInitiateWatch,
    consume int xSampleCount);
  sink asynch Start2(
    consume TUniqueLNID xSource,
    consume boolean xInitiateWatch,
    consume int xSampleCount);
  sink asynch Start3(
    consume TUniqueLNID xSource,
    consume boolean xInitiateWatch,
    consume int xSampleCount);
  sink asynch Start4(
    consume TUniqueLNID xSource,
    consume boolean xInitiateWatch,
    consume int xSampleCount);
  sink asynch Trip1(
    consume TUniqueLNID xSource,
    consume boolean xTrip,
    consume int xSampleCount,
    consume int xFaultValue);
  sink asynch Trip2(
    consume TUniqueLNID xSource,
    consume boolean xTrip,
    consume int xSampleCount,
    consume int xFaultValue);
  sink asynch Trip3(
    consume TUniqueLNID xSource,
    consume boolean xTrip,
    consume int xSampleCount,
    consume int xFaultValue);
  sink asynch Trip4(
    consume TUniqueLNID xSource,
    consume boolean xTrip,
    consume int xSampleCount,
    consume int xFaultValue);
  sink asynch CommandIn(
    consume TUniqueLNID xSource,
    consume TUniqueLNID xDestination,
    consume TCommand xCommand,
    consume int xSampleCount,
    consume string xStringVal,
    consume int xIntVal,
    consume boolean xBoolVal,
    consume float xFloatVal);

  // sources
  source unicast Start(
    produce TUniqueLNID xSource,
    produce boolean xInitiateWatch,
    produce int xSampleCount);
  source unicast Trip(
    produce TUniqueLNID xSource,
    produce boolean xTrip,
    produce int xSampleCount,
    produce int xFaultValue);

  source unicast CommandOut(
    produce TUniqueLNID xSource,
    produce TUniqueLNID xDestination,
    produce TCommand xCommand,
    produce int xSampleCount,
    produce string xStringVal,
    produce int xIntVal,
    produce boolean xBoolVal,
    produce float xFloatVal);

  threaded react Main(
    Start1, Start2, Start3, Start4,
    Trip1, Trip2, Trip3, Trip4, Trip,
    Start, CommandIn, CommandOut)
  {

    TUniqueLNID      theUID, tempUID, tempSrcUID,
    tempDestUID;
    TCommand      tempCmd;
    TOperationMode  theOperationMode = C_Unblock;
    TOperationMode  tempOperationMode = C_Unblock;
    int theDelay = 10;
    int theXOutOf4 = 3;
    boolean theIncludeStart = false;
    boolean tempDoTrip = false;
    boolean tempDoStart = false;
    boolean tempDoExecuteCommand = false;
    boolean eval = false;
    int theSampleCount = 0;

    // Arrays that maintain the trip window
    // The size of a window corresponds to the specified
    delay
    TBoolArray trip1Window;
    TBoolArray trip2Window;
    TBoolArray trip3Window;
    TBoolArray trip4Window;

    // Variables that are true if a channel is tripping
    given the delay
    boolean theTrip1 = true;
    boolean theTrip2 = true;
    boolean theTrip3 = true;
    boolean theTrip4 = true;
    int winPosCounter = 0;

    // Variables that indicate that the first signal is
    arriving on a channel
    // Needed for error handling reasons
    boolean firstTimeOn1;
    boolean firstTimeOn2;
    boolean firstTimeOn3;
    boolean firstTimeOn4;

    // Variable for keeping track of the number of lost
    samples
    int noOfMissedSamples = 0;
    // Loop counter for the missed samples logic
    int i=0;

    // State variables that keep track of which channels
    are connected
    boolean theChannelActive1 = false;
    boolean theChannelActive2 = false;
    boolean theChannelActive3 = false;
    boolean theChannelActive4 = false;

    // State variables that keep track of the latest
    observed trip value for each channels
    boolean lastObservedTrip1 = false;
    boolean lastObservedTrip2 = false;
    boolean lastObservedTrip3 = false;
    boolean lastObservedTrip4 = false;

    // State variables that keep track of the latest
    observed time stamp for each channels
    int lastObservedSampleCount1 = 0;
    int lastObservedSampleCount2 = 0;
    int lastObservedSampleCount3 = 0;
    int lastObservedSampleCount4 = 0;

    // Variables that keeps track of the sample count range
    TSystemParameters theSystemParameters;
    int theMaxSampleCount = 0;
    int theMinSampleCount = 0;
    // Varible used for calculating a normilized sample
    count in the form 0 - N
    int theNormalizedSampleCountOffset = 0;


    int theValue = 0;
    boolean XOutOf4Tripped=false;

    int numberOfActiveChannels = 0;
    boolean error = false;

    int tempSampleCount = 0;
    int tempIntValue;
    string tempStrValue;
    boolean tempBoolValue;
    float tempFloatValue;


#include "Common//Includes//CommandIsEquals.ccl"
#include "Common//Includes//IsSelf.ccl"
#include "Common//Includes//IsBroadcast.ccl"

proc boolean ResetTripWindows()
{
  int i=0;

  for (i=0; i<theDelay; i++)
  {
    trip1Window[i] = false;
    trip2Window[i] = false;
    trip3Window[i] = false;
    trip4Window[i] = false;
  }
  return true;
}

proc boolean CheckForTrip() {

  boolean samplesTimeSynched = false;
  int tripCount = 0;

  // Check if "theXOutOf4" trip channels are true.
  // If a trip channel is left unconnected in an assembly
  it is false by default.
  // First check that all active (connected) channels are
  in synch

  // If all channels are active...
   if (theChannelActive1 && theChannelActive2 &&
   theChannelActive3 && theChannelActive4)
    if ((lastObservedSampleCount1 ==
    lastObservedSampleCount2) &&
       (lastObservedSampleCount2 ==
       lastObservedSampleCount3) &&
       (lastObservedSampleCount3 ==
       lastObservedSampleCount4))
       samplesTimeSynched = true;

  // If all but one channal are active...
  if (theChannelActive1 && theChannelActive2 &&
  theChannelActive3 && !theChannelActive4)
    if ((lastObservedSampleCount1 ==
    lastObservedSampleCount2) &&
        (lastObservedSampleCount2 ==
        lastObservedSampleCount3))
        samplesTimeSynched = true;
    if (theChannelActive1 && theChannelActive2 &&
    !theChannelActive3 && theChannelActive4)
      if ((lastObservedSampleCount1 ==
      lastObservedSampleCount2) &&
        (lastObservedSampleCount2 ==
        lastObservedSampleCount4))
        samplesTimeSynched = true;
    if (theChannelActive1 && !theChannelActive2 &&
    theChannelActive3 && theChannelActive4)
      if ((lastObservedSampleCount1 ==
      lastObservedSampleCount3) &&
        (lastObservedSampleCount3 ==
        lastObservedSampleCount4))
        samplesTimeSynched = true;
    if (!theChannelActive1 && theChannelActive2 &&
    theChannelActive3 && theChannelActive4)
      if ((lastObservedSampleCount2 ==
      lastObservedSampleCount3) &&
        (lastObservedSampleCount3 ==
        lastObservedSampleCount4))
        samplesTimeSynched = true;

    // If two channels are active...
    if (theChannelActive1 && theChannelActive2 &&
    !theChannelActive3 && !theChannelActive4)
      if (lastObservedSampleCount1 ==
      lastObservedSampleCount2)
        samplesTimeSynched = true;
    if (theChannelActive1 && !theChannelActive2 &&
    theChannelActive3 && !theChannelActive4)
      if (lastObservedSampleCount1 ==
      lastObservedSampleCount3)
        samplesTimeSynched = true;
    if (theChannelActive1 && !theChannelActive2 &&
    !theChannelActive3 && theChannelActive4)
      if (lastObservedSampleCount1 ==
      lastObservedSampleCount4)
        samplesTimeSynched = true;
    if (!theChannelActive1 && theChannelActive2 &&
    theChannelActive3 && !theChannelActive4)
      if (lastObservedSampleCount2 ==
      lastObservedSampleCount3)
        samplesTimeSynched = true;
    if (!theChannelActive1 && theChannelActive2 &&
    !theChannelActive3 && theChannelActive4)
    if (lastObservedSampleCount2 ==
    lastObservedSampleCount4)
      samplesTimeSynched = true;
  if (!theChannelActive1 && !theChannelActive2 &&
  theChannelActive3 && theChannelActive4)
    if (lastObservedSampleCount3 ==
    lastObservedSampleCount4)
      samplesTimeSynched = true;

  // If only one channel are active...
  if (theChannelActive1 && !theChannelActive2 &&
  !theChannelActive3 && !theChannelActive4)
      samplesTimeSynched = true;
  if (!theChannelActive1 && theChannelActive2 &&
  !theChannelActive3 && !theChannelActive4)
      samplesTimeSynched = true;
  if (!theChannelActive1 && !theChannelActive2 &&
  theChannelActive3 && !theChannelActive4)
      samplesTimeSynched = true;
  if (!theChannelActive1 && !theChannelActive2 &&
  !theChannelActive3 && theChannelActive4)
      samplesTimeSynched = true;

  // If all active channels are synchronized, check if
  sufficiently many of them are tripping
  if (samplesTimeSynched)
  {
    if (theTrip1)
      tripCount++;
    if (theTrip2)
      tripCount++;
    if (theTrip3)
      tripCount++;
    if (theTrip4)
      tripCount++;
    if (tripCount >= theXOutOf4)
    {
      return true;
    }
  }
  return false;
}

    start -> initializing {
      action {
        theUID = uid;
        theUID[2] = "PTRC";
        theDelay = delay;
        theXOutOf4 = xOutOf4;
        theChannelActive1 = channelActive1;
        theChannelActive2 = channelActive2;
        theChannelActive3 = channelActive3;
        theChannelActive4 = channelActive4;
        theSystemParameters = systemParameters;
        theMaxSampleCount = theSystemParameters[C_PPS_UpperBoundIndex];
        theMinSampleCount = theSystemParameters[C_PPS_LowerBoundIndex];
        firstTimeOn1 = true;
        firstTimeOn2 = true;
        firstTimeOn3 = true;
        firstTimeOn4 = true;

        theTrip1 = false;
        theTrip2 = false;
        theTrip3 = false;
        theTrip4 = false;

        // Initilize the trip window for each channel
        for (i=0; i < theDelay; i++)
        {
          trip1Window[i] = false;
          trip2Window[i] = false;
          trip3Window[i] = false;
          trip4Window[i] = false;
        }
        // Calculate offset for normalizing sample count into the form 0 to N
        if ((theMinSampleCount < 0) || (theMinSampleCount > 0))
          theNormalizedSampleCountOffset = -theMinSampleCount;
        else if (theMinSampleCount == 0)
          theNormalizedSampleCountOffset = 0;

        // Check that the "theXOutOf4" doesn't exceed number of active channels
        numberOfActiveChannels = 0;
        if (theChannelActive1 == true)
          numberOfActiveChannels++;
        if (theChannelActive2 == true)
          numberOfActiveChannels++;
        if (theChannelActive3 == true)
          numberOfActiveChannels++;
        if (theChannelActive4 == true)
          numberOfActiveChannels++;
        if (numberOfActiveChannels < theXOutOf4)
          error = true;
```

```
      }
    }

    initializing -> listening {
    }


//------------- handling the start pins
    listening -> receivingStart1 {
      trigger ^Start1;
    }

    receivingStart1 -> executingStart1 {
    }

    executingStart1 -> listening {
      action {
        $Start1();
      }
    }


    listening -> receivingStart2 {
      trigger ^Start2;
    }

    receivingStart2 -> executingStart2 {
    }

    executingStart2 -> listening {
      action {
        $Start2();
      }
    }


    listening -> receivingStart3 {
      trigger ^Start3;
    }

    receivingStart3 -> executingStart3 {
    }

    executingStart3 -> listening {
      action {
        $Start3();
      }
    }


    listening -> receivingStart4 {
      trigger ^Start4;
    }

    receivingStart4 -> executingStart4 {
    }

    executingStart4 -> listening {
      action {
        $Start4();
      }
    }



//———————— handling trip inputs


    //TripIn1
    listening -> gotTripIn1 {
      trigger ^Trip1;
      action {
        if (theXOutOf4 == 1)
        {
          lastObservedSampleCount1= Trip1.xSampleCount;
          theValue = Trip1.xFaultValue;
          XOutOf4Tripped = true;
        }
        else
        {
          if (firstTimeOn1)
          {
            lastObservedTrip1 = Trip1.xTrip;
            firstTimeOn1 = false;
          }
          else if
          (
            (theNormalizedSampleCountOffset
            +
            Trip1.xSampleCount
            -
            theNormalizedSampleCountOffset
            +
            lastObservedSampleCount1 == 1)
            ||
            (theNormalizedSampleCountOffset
            +
            theMaxSampleCount
      -
      theNormalizedSampleCountOffset
      +
      lastObservedSampleCount1
      +
      theNormalizedSampleCountOffset
      +
      Trip1.xSampleCount) == 0)
    )
    {
      // The samples are in correct order, go ahead
      and calculate the trip window
      lastObservedTrip1 = Trip1.xTrip;
    }
    else
    {
      // One or more samples missed!
      // Assume latest observed value for all missed
      samples

      // Calculate number of misses in the normalized
      form (0 to N)
      noOfMissedSamples =
        theNormalizedSampleCountOffset
        +
        Trip1.xSampleCount
        -
        theNormalizedSampleCountOffset
        +
        lastObservedSampleCount1;

      // Check if sequence number wrap around...
      if (noOfMissedSamples < 0)
      {
        noOfMissedSamples =
          theNormalizedSampleCountOffset
          +
          theMaxSampleCount
          -
          theNormalizedSampleCountOffset
          +
          lastObservedSampleCount1
          +
          theNormalizedSampleCountOffset
          +
          Trip1.xSampleCount;
      }
      for (i=1; i<=noOfMissedSamples; i++)
      {
```

```
                trip1Window[
                 (theNormalizedSampleCountOffset
                 +
                 lastObservedSampleCount1
                 + i) % theDelay] = lastObservedTrip1;
              }
              lastObservedTrip1 = Trip1.xTrip;
            }
            lastObservedSampleCount1 = Trip1.xSampleCount;

            // Put the latest sample into the trip window
            ring-buffer
            trip1Window[
              (theNormalizedSampleCountOffset
              +
              lastObservedSampleCount1) % theDelay] =
              Trip1.xTrip;

            // Check if "theDelay" number of consecutive
            trips have been observed
            theTrip1 = true;
            for (winPosCounter=0; winPosCounter<theDelay;
            winPosCounter++)
            {
              if (trip1Window[winPosCounter] == false)
                theTrip1 = false;
            }

            //logic to be used in all 4 states
            theValue = Trip1.xFaultValue;
            XOutOf4Tripped = CheckForTrip();
          }
        }
      }

      gotTripIn1 -> listening {
        guard !XOutOf4Tripped;
        action $Trip1();
      }

      gotTripIn1 -> execute1 {
        guard XOutOf4Tripped;
        action {
          XOutOf4Tripped = false;
          // Should we also reset the PTRC here?
          ^Trip(theUID, true, lastObservedSampleCount1,
          theValue);
        }
      }
```

```

    execute1 -> listening {
      trigger $Trip;
      action {
        $Trip1();
      }
    }

    //TripIn2
    listening -> gotTripIn2 {
      trigger ^Trip2;
      action {
        if (theXOutOf4 == 1)
        {
          lastObservedSampleCount2= Trip2.xSampleCount;
          theValue = Trip2.xFaultValue;
          XOutOf4Tripped = true;
        }
        else
        {
          if (firstTimeOn2)
          {
            lastObservedTrip2 = Trip2.xTrip;
            firstTimeOn2 = false;
          }
          else if
          (
            (theNormalizedSampleCountOffset
            +
            Trip2.xSampleCount
            -
            theNormalizedSampleCountOffset
            +
            lastObservedSampleCount2 == 1)
          ||
          (
            theNormalizedSampleCountOffset
            +
            theMaxSampleCount
            -
            theNormalizedSampleCountOffset
            +
            lastObservedSampleCount2
            +
            theNormalizedSampleCountOffset
            +
            Trip2.xSampleCount == 0)
          )
          {
```

```
          // The samples are in correct order,
          //go ahead and calculate the trip window
          lastObservedTrip2 = Trip2.xTrip;
        }
        else
        {
          // One or more samples missed!
          // Assume latest observed value for all missed
          samples

          // Calculate number of misses in the normalized
          form (0 to N)
          noOfMissedSamples =
            theNormalizedSampleCountOffset
            +
            Trip2.xSampleCount
            -
            theNormalizedSampleCountOffset
            +
            lastObservedSampleCount2;

          // Check if sequence number wrap around...
          if (noOfMissedSamples < 0)
          {
            noOfMissedSamples =
              theNormalizedSampleCountOffset
              +
              theMaxSampleCount
              -
              theNormalizedSampleCountOffset
              +
              lastObservedSampleCount2
              +
              theNormalizedSampleCountOffset
              +
              Trip2.xSampleCount;
          }

          for (i=1; i<=noOfMissedSamples; i++)
          {
             trip2Window[(theNormalizedSampleCountOffset
             + lastObservedSampleCount2 + i) % theDelay]
             = lastObservedTrip2;
          }
          lastObservedTrip2 = Trip2.xTrip;
        }
        lastObservedSampleCount2 = Trip2.xSampleCount;

```

```
          // Put the latest sample into the trip window
          ring-buffer
          trip2Window[(theNormalizedSampleCountOffset +
          lastObservedSampleCount2) % theDelay] =
          Trip2.xTrip;

          // Check if "theDelay" number of consecutive
          trips have been observed
          theTrip2 = true;
          for (winPosCounter=0; winPosCounter<theDelay;
          winPosCounter++)
          {
            if (trip2Window[winPosCounter] == false)
              theTrip2 = false;
          }

          //logic to be used in all 4 states
          theValue = Trip2.xFaultValue;
          XOutOf4Tripped = CheckForTrip();
        }
      }
    }

    gotTripIn2 -> listening {
      guard !XOutOf4Tripped;
      action $Trip2();
    }

    gotTripIn2 -> execute2 {
      guard XOutOf4Tripped;
      action {
        // Should we also reset the PTRC here?
        XOutOf4Tripped = false;
        ^Trip(theUID, true, lastObservedSampleCount2,
        theValue);
      }
    }

    execute2 -> listening {
      trigger $Trip;
      action {
        $Trip2();
      }
    }

    //TripIn3
    listening -> gotTripIn3 {
      trigger ^Trip3;
      action
      {
        if (theXOutOf4 == 1)
        {
          lastObservedSampleCount3 = Trip3.xSampleCount;
          theValue = Trip3.xFaultValue;
          XOutOf4Tripped = true;
        }
        else
        {
          if (firstTimeOn3)
          {
            lastObservedTrip3 = Trip3.xTrip;
            firstTimeOn3 = false;
          }
          else if
          (
            (
            theNormalizedSampleCountOffset
            +
            Trip3.xSampleCount
            -
            theNormalizedSampleCountOffset
            +
            lastObservedSampleCount3 == 1)
          ||
            (
            theNormalizedSampleCountOffset
            +
            theMaxSampleCount
            -
            theNormalizedSampleCountOffset
            +
            lastObservedSampleCount3
            +
            theNormalizedSampleCountOffset
            +
            Trip3.xSampleCount == 0)
          )
          {
            // The samples are in correct order, go ahead
            and calculate the trip window
            lastObservedTrip3 = Trip3.xTrip;
          }
          else
          {
            // One or more samples missed!
            // Assume latest observed value for all missed
            samples

          // Calculate number of misses in the normalized
          form (0 to N)
          noOfMissedSamples =
            theNormalizedSampleCountOffset
            +
            Trip3.xSampleCount
            -
            theNormalizedSampleCountOffset
            +
            lastObservedSampleCount3;

          // Check if sequence number wrap around...
          if (noOfMissedSamples < 0)
          {
            noOfMissedSamples =
              theNormalizedSampleCountOffset
              + theMaxSampleCount
              - theNormalizedSampleCountOffset
              + lastObservedSampleCount3
              + theNormalizedSampleCountOffset
              + Trip3.xSampleCount;
          }

          for (i=1; i<=noOfMissedSamples; i++)
          {
             trip3Window[
              (theNormalizedSampleCountOffset
              + lastObservedSampleCount3 + i)
              % theDelay] = lastObservedTrip3;
          }
          lastObservedTrip3 = Trip3.xTrip;
        }
        lastObservedSampleCount3 = Trip3.xSampleCount;

        // Put the latest sample into the trip window
        ring-buffer
        trip3Window[(theNormalizedSampleCountOffset +
        lastObservedSampleCount3) % theDelay] =
        Trip3.xTrip;

        // Check if "theDelay" number of consecutive
        trips have been observed
        theTrip3 = true;
        for (winPosCounter=0; winPosCounter<theDelay;
        winPosCounter++)
        {
          if (trip3Window[winPosCounter] == false)
            theTrip3 = false;
        }
```

```

          //logic to be used in all 4 states
          theValue = Trip3.xFaultValue;
          XOutOf4Tripped = CheckForTrip();
        }
      }
    }

    gotTripIn3 -> listening {
      guard !XOutOf4Tripped;
      action $Trip3();
    }

    gotTripIn3 -> execute3 {
      guard XOutOf4Tripped;
      action {
        // Should we also reset the PTRC here?
        XOutOf4Tripped = false;
        ^Trip(theUID, true, lastObservedSampleCount3,
        theValue);
      }
    }

    execute3 -> listening {
      trigger $Trip;
      action {
        $Trip3();
      }
    }

    //TripIn4
    listening -> gotTripIn4 {
      trigger ^Trip4;
      action
      {
        if (theXOutOf4 == 1)
        {
          lastObservedSampleCount4 = Trip4.xSampleCount;
          theValue = Trip4.xFaultValue;
          XOutOf4Tripped = true;
        }
        else
        {
          if (firstTimeOn4)
          {
            firstTimeOn4 = false;
            lastObservedTrip4 = Trip4.xTrip;
          }
          else if
```

```
(
  (theNormalizedSampleCountOffset
  +Trip4.xSampleCount
  - theNormalizedSampleCountOffset
  + lastObservedSampleCount4 == 1)
||
  (theNormalizedSampleCountOffset
  + theMaxSampleCount
  - theNormalizedSampleCountOffset
  + lastObservedSampleCount4
  + theNormalizedSampleCountOffset
  +Trip4.xSampleCount == 0)
)
{
  // The samples are in correct order, go ahead
  and calculate the trip window
  lastObservedTrip4 = Trip4.xTrip;
}
else
{
  // One or more samples missed!
  // Assume latest observed value for all missed
  samples

  // Calculate number of misses in the normalized
  form (0 to N)
  noOfMissedSamples =
    theNormalizedSampleCountOffset
    +Trip4.xSampleCount
    - theNormalizedSampleCountOffset
    + lastObservedSampleCount4;

  // Check if sequence number wrap around...
  if (noOfMissedSamples < 0)
  {
    noOfMissedSamples =
    theNormalizedSampleCountOffset
    + theMaxSampleCount
    - theNormalizedSampleCountOffset
    + lastObservedSampleCount4
    + theNormalizedSampleCountOffset
    + Trip4.xSampleCount;
  }

  for (i=1; i<=noOfMissedSamples; i++)
  {
     trip4Window[
      (theNormalizedSampleCountOffset
      + lastObservedSampleCount4 + i)
```

```
                % theDelay] = lastObservedTrip4;
            }
            lastObservedTrip4 = Trip4.xTrip;
          }
          lastObservedSampleCount4 = Trip4.xSampleCount;

          // Put the latest sample into the trip window
          ring-buffer
          trip4Window[
            (theNormalizedSampleCountOffset
            + lastObservedSampleCount4)
            % theDelay] = Trip4.xTrip;

          // Check if "theDelay" number of consecutive
          trips have been observed
          theTrip4 = true;
          for (winPosCounter=0; winPosCounter<theDelay;
          winPosCounter++)
          {
            if (trip4Window[winPosCounter] == false)
              theTrip4 = false;
          }

          //logic to be used in all 4 states
          theValue = Trip4.xFaultValue;
          XOutOf4Tripped = CheckForTrip();
        }
      }
    }

    gotTripIn4 -> listening {
      guard !XOutOf4Tripped;
      action $Trip4();
    }

    gotTripIn4 -> execute4 {
      guard XOutOf4Tripped;
      action {
        // Should we also reset the PTRC here?
        XOutOf4Tripped = false;
        ^Trip(theUID, true, lastObservedSampleCount4,
        theValue);
      }
    }

    execute4 -> listening {
      trigger $Trip;
      action {
        $Trip4();
      }
    }


//————————————— command and acknowledge

    listening -> executingCommand {
      trigger ^CommandIn;
      action {
        tempDoExecuteCommand = false;
        tempCmd = CommandIn.xCommand;
        tempSrcUID = CommandIn.xSource;
        tempDestUID = CommandIn.xDestination;
        tempStrValue = "";
        tempIntValue = 0;
        tempBoolValue = false;
        tempFloatValue = 0.0;
        if ( IsSelf(theUID, tempDestUID)  ||
        (IsBroadcast(tempDestUID) )
        { // @@@ self = broadcast
          // only accept commands if xDestination == self
          %{
            PRINT_DEBUG((
              "CSWI␣Command␣received␣from␣=␣␣%s\n",
              $ccl$tempSrcUID[3]));
          %}
          eval = CommandIsEquals(
              tempCmd, "CMD", "SET", "OPERATIONMODE", "");
          if (eval == true) {
            %{
              PRINT_DEBUG(("Command␣evaluated␣=␣
              SetOperationmode\n"));
            %}

            if (CommandIn.xStringVal == "BLOCK") {
                tempOperationMode = C_Block;
            }
            if (CommandIn.xStringVal == "UNBLOCK") {
              tempOperationMode = C_Unblock;
            }
            if (tempOperationMode != theOperationMode) {
              tempCmd[0] = "ACK";
              tempDoExecuteCommand = true;
              theOperationMode = tempOperationMode;
              tempDestUID = tempSrcUID;
              tempStrValue = CommandIn.xStringVal;
            }
          }
          eval = CommandIsEquals(
            tempCmd,
            "CMD", "GET", "OPERATIONMODE", "");
          if (eval == true) {
            tempDoExecuteCommand = true;
            tempCmd[0] = "ACK";
            tempDestUID = tempSrcUID;
            if (theOperationMode == C_Block) {
              tempStrValue = "BLOCK";
            }
            else {
              tempStrValue = "UNBLOCK";
            }
          }
          eval = CommandIsEquals(
            CommandIn.xCommand,
            "CMD", "GET", "PARAMETER", "XOutOf4");
          if (eval == true) {
            tempDoExecuteCommand = true;
            tempCmd[0] = "ACK";
            tempDestUID = tempSrcUID;
            tempIntValue = theXOutOf4;
          }
          eval = CommandIsEquals(
            CommandIn.xCommand,
            "CMD", "SET", "PARAMETER", "Delay");
          if (eval == true) {
            tempDoExecuteCommand = true;
            tempCmd[0] = "ACK";
            tempDestUID = tempSrcUID;
            theDelay = CommandIn.xIntVal;
            tempIntValue = theDelay;  // and send it out
            again!
          }
          eval = CommandIsEquals(
            CommandIn.xCommand,
            "CMD", "GET", "PARAMETER", "Delay");
          if (eval == true) {
            tempDoExecuteCommand = true;
            tempCmd[0] = "ACK";
            tempDestUID = tempSrcUID;
            tempIntValue = theDelay;
          }
          eval = CommandIsEquals(
            tempCmd, "CMD", "RESET", "", "");
          if (eval == true) {
            tempDoExecuteCommand = true;
            ComposeAcknowledgeCommand("RESET", "", "",
            tempCmd);
            tempDestUID = tempSrcUID;
```

```
                // Reset the trip window
                ResetTripWindows();
                // Reset the trip indication for each channel
                theTrip1 = false;
                theTrip2 = false;
                theTrip3 = false;
                theTrip4 = false;
              }
            }
          }
        }

        executingCommand -> acknowledging {
          guard tempDoExecuteCommand;
          action {
            tempDoExecuteCommand = false;
            ^CommandOut(
              theUID,
              tempDestUID,
              tempCmd,
              tempSampleCount,
              tempStrValue,
              tempIntValue,
              tempBoolValue,
              tempFloatValue);
          }
        }

        acknowledging -> listening {
          trigger $CommandOut;
          action {
            $CommandIn();
          }
        }

        executingCommand -> listening {
          guard !tempDoExecuteCommand;
          action {
            $CommandIn();
          }
        }
      }
    }
```

Appendix C

Acronyms

Table C.1: Acroynms

$\lambda*$	Performance Reasoning Framework
λ-ABA	Performance Reasoning Framework, Average Case, with Blocking and Asynchronous Interaction
λ-WBA	Performance Reasoning Framework, Worst Case, with Blocking and Asynchronous Interaction
λ-SS	Performance Reasoning Framework, Sporadic Server
AADL	Architecture Analysis and Design Language
ABAS	Attribute–Based Architecture Style
ADL	Architecture Description Language
API	Application Programming Interface
CBS	COTS–Based Systems
CBSE	Component Based Software Engineering
CEGAR	Counterexample Guided Abstraction Refinement
ComFoRT	Component Formal Reasoning Technology
COTS	Commercial Off–The–Shelf
CSP	Communicating Sequential Processes (a process algebra)
CTL	Computation Tree Logic
DLL	Dynamically Linked Library
FDR	Failure Divergence Refinement (model checker)
FIFO	First In First Out
FSP	Finite State Processes (a process algebra)
HKL	Concurrent pipeline pattern named for the authors of [62]
IED	Intelligent Electronic Device (from IEC–1850)
GOOSE	Generic Object–Oriented Substation Event (from IEC61850)
GRMA	Generalized Rate Monotonic Analysis
GRMT	Generalized Rate Monotonic Scheduling Theory
LCM	Least Common Multiple (from IEC61850)
LN	Logical Node (from IEC61850)
LTL	Linear Temporal Logic
LTSA	Labeled Transition System Analyzer (model checker)
MBE	Model–Based Engineering
MRE	Magnitude of Relative Error
NATO	North Atlantic Treaty Organization
ORC	Open Robot Controller
PACC	Predictable Assembly from Certifiable Components
PCC	Proof–Carrying Code
PCL	Pin Component Language
PECT	Prediction–Enabled Component Technology
PMM	Performance Metamodel
PSK	PACC Starter Kit
RMS	Rate Monotonic Analysis
RTOS	Real Time Operating System
RTQT	Real Time Queueing Theory
SAS	Substation Automation Systems
SAV	Sampled Values (from IEC61850)

(continued next page)

Table C.1: *(continued)*

SEI	Software Engineering Institute
SE–LTL	State–Event Linear Temporal Logic
Soft P&C	Soft Protection and Control
SS	Sporadic Server
TCB	Trusted Computing Base
UML	Unified Modeling Language

Bibliography

[1] Abadi, M., and Lamport, L. Conjoining specifications. *ACM Transactions on Programming Languages and Systems 3*, 17 (1995), 507–531.

[2] Abowd, G., Allen, R., and Garlan, D. Using style to understand descriptions of software architecture. In *SIGSOFT '93: Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering* (New York, NY, USA, 1993), ACM Press, pp. 9–20.

[3] Abowd, G. D., Allen, R., and Garlan, D. Formalizing style to understand descriptions of software architecture. *ACM Trans. Softw. Eng. Methodol. 4*, 4 (1995), 319–364.

[4] Achermann, F., and Nierstrasz, O. Applications = Components + Scripts — A Tour of Piccola. In *Software Architectures and Component Technology*, M. Aksit, Ed. Kluwer, 2001, pp. 261–292.

[5] Aldrich, J. Using types to enforce architectural structure. In *Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)* (Los Alamitos, CA, USA, 2008), IEEE Computer Society.

[6] Alexander, C. *Notes on the Synthesis of Form.* Harvard University Press, Cambridge, MA, Jan 1964.

[7] Alexander, C. *A Pattern Language.* Oxford University Press, 1977.

[8] Allen, R., and Garlan, D. Formalizing architectural connection. In *ICSE '94: Proceedings of the 16th international conference on Software engineering* (Los Alamitos, CA, USA, 1994), IEEE Computer Society Press, pp. 71–80.

[9] Allen, R. J. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon University, Mar 1997. CMU–CS–97–144.

[10] Alpern, B., and Schneider, F. B. Recognizing safety and liveness. *Distributed Computing 2* (1987), 117–126.

[11] Anderson, P. More is different. *Science 177* (1972), 393–396.

[12] Babar, M. A., and Lago, P. Design decisions and design rationale in software architecture. *Journal of Systems and Software 82*, 8 (2009), 1195 – 1197. SI: Architectural Decisions and Rationale.

[13] Bachmann, F., Bass, L., and Klein, M. Deriving architectural tactics: A step toward methodical architectural design. Technical Report CMU/SEI-2003-TR-004, Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Avenue, Pittsburgh, PA, 15026, 2003.

[14] Ball, T., Cook, B., Levin, V., and Rajamaii, S. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated Formal Methods* (2004).

[15] Ball, T., Majumar, R., Millstein, T., and Rajamani, S. Automatic predicate abstraction of c programs. In *2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY., June 2001), Association for Computing Machinery (ACM), pp. 203–213.

[16] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*, second ed. Pearson Education, 2003.

[17] Basu, A., Bozga, M., and Sifakis, J. Modeling heterogeneous real-time components in bip. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 3–12.

[18] Becker, S., Koziolek, H., and Reussner, R. Model-based performance prediction with the palladio component model. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance* (New York, NY, USA, 2007), ACM, pp. 54–65.

[19] Berry, G., and Boudol, G. The chemical abstract machine. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1990), ACM, pp. 81–94.

[20] Bilchev, G., and Parmee, I. C. The ant colony metaphor for searching continuous design spaces. In *Selected Papers from AISB Workshop on Evolutionary Computing* (London, UK, 1995), Springer-Verlag, pp. 25–39.

[21] Bobaru, M. G., Pasareanu, C., and Giannakopoulou, D. Automated assume–guarantee reasoning by abstraction refinement. In *International Conference on Computer–Aided Verification (CAV 08)* (2008), vol. 5123 of *Lecture Notes in Computer Science.*

[22] Both, A., and Zimmermann, W. Automatic protocol conformance checking of recursive and parallel component-based systems. In *CBSE '08: Proceedings of the 11th International Symposium on Component-Based Software Engineering* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 163–179.

[23] Brooks, Jr., F. P. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer 20*, 4 (1987), 10–19.

[24] Burge, J. e. Design rationale: Researching under uncertainty. *Artif. Intell. Eng. Des. Anal. Manuf. 22*, 4 (2008), 311–324.

[25] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern–Oriented Software Architecture, Volume 1, A System of Patterns.* Wiley, 1996.

[26] Caetano, C., and Pernes, M. Introducing iec61850 in distribution substations. In *Powergrid Europe Transmission and Distribution* (Madrid, Spain, June 26–28 2007).

[27] Carpenter, B., Roman, M., Vasilatos, N., and Zimmerman, M. The rtx real–time subsystem for windows nt. In *USENIX Windows NT Workshop* (August 1997).

[28] Chaki, S. Sat-based software certification. Technical Report CMU/SEI-2006-TN-004, Software Engineering Institute,Carnegie Mellon University, Pittsburgh, PA, USA, February 2006.

[29] Chaki, S., Clarke, E. M., Sinha, N., and Thati, P. Automated assume–guarantee reasoning for simulation conformance. In *International Conference on Computer–Aided Verification (CAV 05)* (2005), vol. 3576 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 534–547.

[30] Chaki, S., Edmund, Clarke, E. M., Sharygina, N., and Sinha, N. State/event-based software model checking. In *Integrated Formal Methods* (Berlin - Heidelberg - New York, 2004), Springer-Verlag, pp. 128–147.

[31] Chaki, S., Gurfinkel, A., Wallnau, K., and Weinstock, C. Assurance cases for proofs as evidence. In *Workshop on Proof-Carrying Code and Software Certification (PCC'09)* (August 2009), E. Denney and T. Jensen, Eds., Affiliated with Twenty-Fourth IEEE Symposium on Logic in Computer Science (LICS'09).

[32] Chaki, S., Ivers, J., Lee, P., Wallnau, K., and Zeilberger, N. Certified binaries for software components. Technical Report CMU/SEI-2007-TR-001, Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Avenue, Pittsburgh, PA, 15026, 2007.

[33] Chaki, S., Ivers, J., Lee, P., Wallnau, K., and Zeilberger, N. Model-driven construction of certified binaries. In *Model Driven Engineering Languages and Systems* (Berlin - Heidelberg - New York, 2007), Lecture Notes in Computer Science, Springer-Verlag, pp. 668–681.

[34] Chaki, S., Ivers, J., Sharygina, N., and Wallnau, K. The comfort reasoning framework. In *17th International Conference on Computer Aided Verification* (July 2005), vol. 3576 of *Lecture Notes in Computer Science*.

[35] Chaki, S., Oaknine, J., Yoray, K., and Clark, E. Automated compositional abstraction refinement for concurrent c programs: A two–level approach. *ENTCS 89*, 3 (2003).

[36] Chaki, S., and Strichman, O. Optimized l*–based assume–guarantee reasoning. In *TACAS 07* (2007), vol. 4424 of *Lecture Notes in Computer Science*, pp. 276–291.

[37] Cimatti, A., Clark, E., Giunchiglia, F., and Roveri, M. Nusmv: A new symbolic model verifier. *International Journal on Software Tools for Technology Transition 2*, 4 (2000), 410–425.

[38] Clark, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement. In *12th International Conference on Computer–Aided Verification (CAV 2000)* (2000), vol. 1855 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 154–169.

[39] Clark, E., Grumberg, O., and Long, D. Model checking and abstraction. In *19th Annual SIGPLAN–SIGACT Symposium on the Principles of Programming Languages* (January 1992), Association for Computing Machinery (ACM), pp. 343–354.

[40] Clark, E., Long, D., and McMillan, K. Compositional model checking. In *4th International Symposium on Logic in Computer Science* (1989), pp. 353–362.

[41] Clarke, E. M. The birth of model checking. In *25 Years of Model Checking: History, Achievements, Perspectives*, O. Grumberg and H. Veith, Eds. Springer-Verlag, Berlin, Heidelberg, 2008, pp. 1–26.

[42] Clarke, E. M., and Emerson, A. A. The design and synthesis of synchronization the design and synthesis of synchronization skeletons using temporal logic. In *Workshop Workshop on Logics of Programs,* (Berlin - Heidelberg - New York, 1981), no. 131 in Lecture Notes in Computer Science, Springer-Verlag, pp. 52–71.

[43] Clarke, E. M., Grumberg, O., Jham, S., Lu, Y., and Veith, H. Counterexample-guided abstraction reinement for symbolic counterexample-guided abstraction re nement for symbolic counterexample–guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM) 50*, 5 (September 2003), 752–794.

[44] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. *Documenting Software Architectures: Views and Beyond.* Pearson Education, 2002.

[45] Coplien, J. O., and Schmidt, D. C. *Pattern Languages of Program Design.* Addison-Wesley, 1995.

[46] Crnkovic, I., Sentilles, S., Vulgarakis, A., and Chaudron, M. R. V. A Classification Framework for Component Models. *IEEE Transactions on Software Engineering* (submitted).

[47] Cusumano, M. A. The Software Factory: A Historical Interpretation. *IEEE Software 6*, 1 (March 1989), 23–30.

[48] Dijkstra, E. W. On the cruelty of really teaching computing science. circulated privately, Dec. 1988.

[49] Doytchinov, B., Lehoczky, J., and Shreve, S. Real-time queues in heavy traffic with earliest- deadline-first queue discipline. *Annals of Applied Probability 11*, 2 (2001), 332–378.

[50] Dwyer, M. B., Hatcliff, J., Hoosier, M., and Robby. Building your own software model checker using the bogor extensible model checking framework. In *Computer Aided Verification*, K. Etessami and S. K. Rajamani, Eds., vol. 3576 of *Lecture Notes in Computer Science.* Springer-Verlag, 2005, pp. 148–152.

[51] Fielding, R. T. *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, Irvine, 2000. Chair-Richard N. Taylor.

[52] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Design.* Addison-Wesley, 1995.

[53] Garlan, D., Allen, R., and Ockerbloom, J. Exploiting style in architectural design environments. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering* (New York, NY, USA, 1994), ACM Press, pp. 175–188.

[54] Garlan, D., Monroe, R., and Wile, D. Acme: an architecture description interchange language. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research* (1997), IBM Press, pp. 169–183.

[55] Giannakopoulou, D. *Model Checking for Concurrent Software Architectures.* PhD thesis, Imperial College of Science Technology and Medicine, University of London, March 1999.

[56] Goodenough, J., and Sha, L. The priority ceiling protocol: A method for minimizing the blocking of high-priority ada tasks. Special Report CMU/SEI-88-SR-004, Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Avenue, Pittsburgh, PA, 15026, 1988.

[57] Graf, S., and Saïdi, H. Construction of abstract state graphs with pvs. In *9th International Conference on Computer Aided Verification (CAV '97)* (June 22–25 1997), vol. 1254 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 72–83.

[58] Gui, S., Luo, L., Liu, Q., Guo, F., and Lu, S. Ucas: A schedulability analysis tool for aadl models. *Embedded and Ubiquitous Computing, IEEE/IFIP International Conference on 2* (2008), 449–454.

[59] Hanninen, K., J., M.-T., Nolin, M., Lindberg, M., Lundback, J., and Lundback, K. The rubus component the rubus component model for resource constrained real-time systems. In *International Symposium on Industrial Embedded Systems SIES 2008* (2008), pp. 177–183.

[60] Hansen, J., and Moreno, G. A. Overview of the lambda-star performance reasoning frameworks. Technical Report CMU/SEI-2008-TR-020, Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Avenue, Pittsburgh, PA, 15213–3890, February 2008.

[61] Harbour, G., Garcia, G., Gutierrez, P., and Moyano, D. Mast: Modeling and anaysis suite for real–time applications. In *Euromicro Conference on Real–Time Systems (ECRTS)* (June 2001), IEEE Computer Society.

[62] Harbour, G., Klein, M. H., and Lehoczky, J. Fixed priority scheduling of periodic tasks with varying execution priority. In *Proceedings of IEEE Real-Time Systems Symposium* (San Antonio, Texas, December, December 1991), IEEE Computer Society Press, pp. 116–128.

[63] Harbour, M. G., and Sha, L. An application-level implementation of the sporadic server. Technical Report CMU/SEI-91-TR-026, Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Avenue, Pittsburgh, PA, 15026, 1991.

[64] Hardin, R., Z. Har'El, Z., and Kurshan, R. Cospan. In *8th International Conference Computer Aided Verification (CAV 1996)* (Berlin - Heidelberg - New York, 1996), vol. 1102 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 423–427.

[65] Heineman, G. T., and Councill, W. T. *Component Based Software Engineering: Putting the Pieces Together*, first ed. Addison-Wesley, 2001.

[66] Henzinger, T., Jhala, R., Majumar, R., and Sutre, G. Lazy abstraction. In *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 02)* (New York, NY, USA, January 2002), vol. 37 of *SIGPLAN Notices*, ACM Press, pp. 58–70.

[67] Hissam, S., , Klein, M., Lehoczky, J., Merson, P., and Wallnau, G. M. K. Performance property theories for predictable assembly from certifiable components (pacc). Technical Report CMU/SEI-2004-TR-017, Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Avenue, Pittsburgh, PA, 15026, 2004.

[68] Hissam, S., Hudak, J., Ivers, J., Klein, M., (ABB), M. L., Moreno, G., Northrop, L., Plakosh, D., Stafford, J., Wallnau, K., and Wood, W. Predictable assembly of substation automation systems: An experiment report, 2nd edition. Technical Report CMU/SEI-2002-TR-031, Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Avenue, Pittsburgh, PA, 15026, 2003.

[69] Hissam, S., Ivers, J., Plakosh, D., and Wallnau, K. Pin component technology (v1.0) and its c interface. Technical Note CMU/SEI-2005-TN-001, Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Avenue, Pittsburgh, PA, 15026, 2005.

[70] Hissam, S., Moreno, G., Stafford, J., and Wallnau, K. Packaging predictable assembly with prediction-enabled component technology. Technical Report CMU/SEI-2001-TR-024, Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Avenue, Pittsburgh, PA, 15026, 2001.

[71] Hissam, S., Moreno, G., Stafford, J., and Wallnau, K. Packaging predictable assembly. In *Component Deployment*, J. Bishop, Ed., vol. 2370 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2002, pp. 108–124.

[72] Hissam, S., Moreno, G., and Wallnau, K. Predictability by Construction (Tutorial). In *30th International Conference on Software Engineering* (New York, NY, USA, 2008), ACM.

[73] Hissam, S. A., Moreno, G. A., and Wallnau, K. C. Using containers to enforce smart constraints for performance in industrial systems. Technical Note CMU/SEI-2005-TN-040, Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Avenue, Pittsburgh, PA, 15026, 2005.

[74] Hoare, C. Algebra and Models. *SIGSOFT Software Engineering Notes 18*, 5 (December 1993), 1–8.

[75] Hoare, C. A. R., and Hoare, C. A. R. Communicating sequential processes. *Communications of the ACM 21* (1985), 666–677.

[76] Hunt, J. M. A practical state machine project. In *ACM-SE 47: Proceedings of the 47th Annual Southeast Regional Conference* (New York, NY, USA, 2009), ACM, pp. 1–6.

[77] Hunt, J. M., and McGregor, J. D. Building software that is predictable by construction. *J. Comput. Small Coll. 25*, 2 (2009), 203–204.

[78] IEC-TC57-WG10/11/12. Iec61850: Communications networks and systems in substations. International Electrotechnical Commission Stanmdard, 1999.

[79] Inverardi, P., and Wolf, A. L. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Trans. Softw. Eng. 21*, 4 (1995), 373–386.

[80] Ivers, J. Lessons learned model checking an industrial communications library. Technical Note CMU/SEI-2005-TN-039, Software Engineering Institute, Carnegie Mellon University, September 2006.

[81] Ivers, J., and Sharygina, N. Overview of comfort: A model checkin reasoning framework. Technical Note CMU/SEI-2004-TN-018, Carnegie Mellon University, Software Engineering Institute, April 2004.

[82] Ivers, J., Sinha, N., and Wallnau, K. A basis for composition language cl. Technical Report CMU/SEI-2002-TN-026, Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Avenue, Pittsburgh, PA, 15026, 2002.

[83] Ivers, J., and Wallnau, K. C. Preserving Real Concurrency. In *2003 ECOOP Workshop on Correctness of Model-Based Software Composition (CMC)* (July 2003), vol. Technical Report 2003-13, Universitat Karlsruhe.

[84] Jackson, D., Martyn, T., and Millett, L. I. *Software for Dependable Systems: Sufficient Evidence?* National Academy Press, Washington, DC, USA, 2007.

[85] Jacobs, B., and Rutten, J. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin 62* (1997), 62–222.

[86] kerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P., and Tivoli, M. The save approach to component-based development of vehicular systems. *J. Syst. Softw. 80*, 5 (2007), 655–667.

[87] Klein, M., and Kazman, R. Attribute-Based Architectural Styles. Technical Report CMU/SEI–99–TR–022, Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Avenue, Pittsburgh, PA, 15213–3890, Oct 1999.

[88] Klein, M. H., Kazman, R., Bass, L. J., Carrière, S. J., Barbacci, M., and Lipson, H. F. Attribute-based architecture styles. In *WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)* (Deventer, The Netherlands, The Netherlands, 1999), Kluwer, B.V., pp. 225–244.

[89] Klein, M. H., Ralya, T., Pollak, B., Obenza, R., and Harbour, M. G. *A practitioner's handbook for real-time analysis.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[90] Kleinrock, L. *Queueing Systems Volume1: Theory.* Wiley-Interscience, 1975.

[91] Krahl, D. Extend: the extend simulation environment. In *34th Winter Simulation Conference WSC–02* (Arlington, VA, December 2001), IEEE Computer Society.

[92] Krishnan, R., and Palki, B. First experiences with design and engineering of iec 61850 based substation automation systems in india. In *Conference on Electric Power Supply Industry (CEPSI) 2006* (Mumbai, India, 2006).

[93] Kruchten, P. The 4+1 view model of architecture. *IEEE Softw. 12*, 6 (1995), 42–50.

[94] Kupferman, O., and Vardi, M. From complementation to certification. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 04)* (2004), vol. 2988 of *Lecture Notes in Computer Science*, pp. 591–606.

[95] Kurshan, R. *Computer–Aided Verification of Coordinating Process: the Automata–Theoretic Apprlach.* Princeton University Press, Princeton, NJ, 1994.

[96] Lakoff, G., and Johnson, M. *Metaphors We Live By.* University of Chicago Press, Chicago, IL, USA, 1980.

[97] Lakoff, G., and nez, R. N. *Where Mathematics Comes From.* Basic Books, a member of Perseus Books Group, New York, NY, USA, 1990.

[98] Larson, M. *Predicting Quality Attributes in Component-based Software Systems.* PhD thesis, Malardalen University, 2004. Doctoral Dissertation No.8.

[99] Larson, M., Wall, A., and Wallnau, K. Predictable assembly: The crystal ball to software! *ABB Review*, 2 (2005).

[100] Lenz, G., and Wienands, C. *Practical Software Factories in .NET.* Apress, 2008.

[101] Lewis, R. *Programming industrial control systems using IEC 1131-3*, revised ed. No. 50 in IEE Control Engineering Series. The Institution of Electrical Engineers (IEE), IEE, Michael Faraday House, Six Hills Way, Stevenage, Herts. SG1 2AY, United Kingdom., 1999.

[102] Li, B., Jeon, W., Kalter, W., Nahrstedt, K., and hyuk Seo, J. Adaptive middleware architecture for a distributed omni-directional visual tracking system. In *Proceedings of SPIE/ACM MMCN 2000* (2000), pp. 101–112.

[103] Liskov, B., and Zilles, S. Programming with abstract data types. *SIGPLAN Not. 9*, 4 (1974), 50–59.

[104] Liskov, B. H., and Wing, J. M. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16*, 6 (1994), 1811–1841.

[105] Loiseaux, C., Graf, S., Sifakis, J., Bouajiani, A., and Bensalem, S. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design 6*, 1 (1995), 11–44.

[106] Lumpe, M. *A π–Calculus Based Approach for Software Composition.* PhD thesis, Institute für Informatiik und angewandte Mathematik, 1999.

[107] Madsen, K. H. A guide to metaphorical design. *Communications of the ACM 37*, 12 (1994), 57–62.

[108] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference* (London, UK, 1995), Springer-Verlag, pp. 137–153.

[109] Magee, J., and Kramer, J. Dynamic structure in software architectures. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering* (New York, NY, USA, 1996), ACM Press, pp. 3–14.

[110] Magee, J., and Kramer, J. *Concurrency: State Models & Java Programs.* John Wiley & Sons, April 1999.

[111] Magee, J., and Kramer, J. *Concurrency: State Models & Java Concurrency: State Models and Java Programs.* Wiley, West Sussex, England, 2001.

[112] Magee, J., Kramer, J., and Giannakopoulou, D. Behaviour analysis of software architectures. In *WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)* (Deventer, The Netherlands, The Netherlands, 1999), Kluwer, B.V., pp. 35–50.

[113] Martinez, P. L., Medina, J. L., and Drake, J. M. Sim–mast: Simulador de sistemas distribuidos de tiempo real. In *XII Jornadas de Concurrencia y Sistemas Distribuidos* (2004).

[114] Mcilroy, M. D. 'mass produced' software components. In *Software Engineering: A Report On a Conference Sponsored by the NATO Science Committee* (October 1968), P. Naur and B. Randell, Eds., pp. 138–155.

[115] McMillan, K. A composition rule for hardware design refinement. In *9th International Conference on Computer Aided Verification (CAV '97)* (1997), vol. 1254 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 24–35.

[116] Medvidovic, N., Oreizy, P., Robbins, J. E., and Taylor, R. N. Using object-oriented typing to support architectural design in the c2 style. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering* (New York, NY, USA, 1996), ACM Press, pp. 24–32.

[117] Mehta, N. R., Medvidovic, N., and Phadke, S. Towards a taxonomy of software connectors. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering* (New York, NY, USA, 2000), ACM Press, pp. 178–187.

[118] Mellor, S., and Balcer, M. *Executable UML: A Foundation for Model-Driven Architecture.* Addison-Wesley, Boston, MA, 2002.

[119] Meyer, B. Applying "design by contract". *Computer 25*, 10 (1992), 40–51.

[120] Meyer, B., Mingins, C., and Schmidt, H. Providing trusted components to the industry. *Computer 31*, 5 (1998), 104–105.

[121] Milner, R. *A Calculus of Communicating Systems.* No. 92 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[122] MILNER, R. *Communicating and Mobile Systems: The π–Calculus.* Cambridge University Press, 1999.

[123] MILNER, R., TOFTE, M., AND HARPER, R. *The definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1990.

[124] MORENO, G., HISSAM, S., AND WALLNAU, K. Statistical models for empirical component properties and assembly-level property predictions. In *Fifth ICSE Workshop on Component-Based Software Engineering* (New York, NY, USA, 2002), ACM.

[125] MORENO, G. A. Creating custom containers with generative techniques. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering* (New york, NY, USA, 2006), ACM, pp. 29–38.

[126] MORENO, G. A., AND MERSON, P. Model–driven performance analysis. In *4th International Conference on the Quality of Software Architectures* (2008).

[127] MORENO, G. A., SMITH, C. U., AND WILLIAMS, L. G. Performance analysis of real-time component architectures: a model interchange approach. In *WOSP '08: Proceedings of the 7th international workshop on Software and performance* (New york, NY, USA, 2008), ACM, pp. 115–126.

[128] MOSSES, P. D. Denotational semantics. In *Handbook of theoretical computer science (vol. B): formal models and semantics.* Elsevier Science, 1990, ch. 11, pp. 575–631.

[129] NAMJOSHI, K. S. Certifying model checkers. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification* (London, UK, 2001), Springer-Verlag, pp. 2–13.

[130] NECULA, G. Proof Carrying Code. In *Proceedings of the 24th Symposium on Principles of Programming Langauges (POPL í97)* (Paris, France, January 1997), SIGPLAN-SIGACT, Association for Computing Machinery, pp. 106–119.

[131] NECULA, G., AND P., L. Safe kernel extensions without runtime checking. In *2nd USENIX Symposium on Operating System Design and Implementation (OSDI 96)* (1996), ACM Press, pp. 229–243.

[132] PALESI, M., AND GIVARGIS, T. Multi-objective design space exploration using genetic algorithms. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign* (New York, NY, USA, 2002), ACM, pp. 67–72.

[133] PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15*, 12 (1972), 1053–1058.

[134] PARNAS, D. L., AND CLEMENTS, P. C. A rational design process: How and why to fake it. *IEEE Trans. Softw. Eng. 12*, 2 (1986), 251–257.

[135] PERRY, D. E., AND WOLF, A. L. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes 17*, 4 (1992), 40–52.

[136] Pierce, B. C., and Turner, D. N. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner* (2000), G. Plotkin, C. Stirling, and M. Tofte, Eds., MIT Press.

[137] Plakosh, D., Smith, D., and Wallnau, K. Builder's Guide for Waterbeans Components. Technical Report CMU/SEI-99-TR-024, Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Avenue, Pittsburgh, PA, 15026, December 1999.

[138] Plotkin, G. D. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming 60-61* (December 2004), 17–139.

[139] Pnueli, A. The temporal logic of programs. In *18th IEEE Symposium on Foundations of Computer Science* (1977), IEEE Computer Society, pp. 46–57.

[140] Preiss, O., and Wegmann, A. Towards a composition model problem based on iec61850. *Journal of Systems and Software 65* (2003).

[141] Räihä, O., Koskimies, K., and Mäkinen, E. Genetic synthesis of software architecture. In *SEAL '08: Proceedings of the 7th International Conference on Simulated Evolution and Learning* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 565–574.

[142] Roscoe, A. Model–checking csp. In *A Classical Mind, Essays in Honour of C.A.R. Hoare*. Prentice–Hall, 1994.

[143] Roy, B. The outranking approach and the foundations of electre methods. *Theory and Decision 31* (1991), 49–73.

[144] Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F. *Pattern–Oriented Software Architecture, Volume 1, Patterns for Concurrent and Networked Objects*. Wiley, 1996.

[145] Scott, J., and Kazman, R. Realizing and refining architectural tactics: Availability. Technical Report CMU/SEI-2009-TR-006, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, August 2009.

[146] Sentilles, S., Vulgarakis, A., Bureš, T., Carlson, J., and Crnković, I. A component model for control-intensive distributed embedded systems. In *CBSE '08: Proceedings of the 11th International Symposium on Component-Based Software Engineering* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 310–317.

[147] Sha, L., Rajkumar, R., and Lehoczky, J. Priority inheritance protocols: An approach to real–time synchronization. *IEEE Transactions on Computers 39*, 9 (1990), 1175–1185.

[148] Sharygina, N., Kurshan, R., and Brown, J. A formal object–oriented analysis for software reliability. In *4th International Conference on Formal Aspects of Software Engineering (FACE 2001)* (Genova, Italy, April 2001), vol. 2029 of *Lecture Notes in Computer Science*.

[149] Shaw, M. Truth vs knowledge: The difference between what a component does and what we know it does. In *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design* (Washington, DC, USA, 1996), IEEE Computer Society, p. 181.

[150] Shaw, M., and Garlan, D. Characteristics of Higher-level Languages for Software Architecture. Tech. Rep. CMU-CS-94-210, Carnegie Mellon University, School of Computer Science, December 1994.

[151] Simon, H. A. *Models of man: social and rational: mathematical essays on rational human behavior in a social setting.* Wiley, 1957.

[152] Simon, H. A. *Sciences of the Artificial*, third ed. MIT Press, 1996.

[153] Smith, L. M. C., and Samadzadeh, M. H. An annotated bibliography of literate programming. *SIGPLAN Not. 26*, 1 (1991), 14–20.

[154] Sprunt, B., Sha, L., and Lehoczky, J. Scheduling sporadic and aperiodic events in a hard real-time system. Technical Report CMU/SEI-89-TR-011, Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Avenue, Pittsburgh, PA, 15026, 1989.

[155] Standard. *IEEE Standard for a Software Quality Metrics Methodology.* No. IEEE Std 1061-1998. IEEE Computer Society, 1998.

[156] Standard. *Software engineering – Product quality – Part 3: Internal metrics.* No. ISO/IEC TR 9126-3:2003. International Organization for Standardization (ISO), 2003.

[157] Stevens, W., Meyers, G., and Constantine, L. Structured design. *IBM Systems 3*, 2 (1974), 115–139.

[158] Sullivan, K. M., and Grivell, I. Qsim: A queueing theory model with various probability distribution functions. Tech. Rep. NUWC-NPT Technical Document 11,418, Naval Undersea Warfare Center, Newport, Rhode Island, March 2003.

[159] Sutherland, D. F. *The Code of Many Colors: Semi-automated Reasoning about Multi-Thread Policy for Java.* PhD thesis, School of Computer Science Carnegie Mellon University, May 2008.

[160] Szyperski, C., Gruntz, D., and Murer, S. *Component software: beyond object-oriented programming*, second ed. Component Software Series. Addison-Wesley, 2002.

[161] Tang, A., Babar, M. A., Gorton, I., and Han, J. A survey of architecture design rationale. *J. Syst. Softw. 79*, 12 (2006), 1792–1804.

[162] Taylor, F. W. *Principles of Scientific Management.* Gutenberg Project, http://www.gutenberg.org/etext/6465, 1911.

[163] Taylor, F. W. *Shop Management.* Gutenberg Project, http://www.gutenberg.org/etext/6464, 1911.

[164] Taylor, F. W. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, 2004.

[165] Tayor, B., and Kuyatt, C. Guidelines for evaluating and expressing the uncertainty of nist measurement results. Tech. Rep. NIST Technical Note 1297, US National Institute of Science and Technology (NIST), Gathersburg, MD, 1994.

[166] van Ommering, R. Building product populations with software components. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering* (New York, NY, USA, 2002), ACM Press, pp. 255–265.

[167] Van Wyk, C. J. Literate programming. *Commun. ACM 30*, 12 (December 1987), 1000–1010.

[168] Vassilev, V. K., Fogarty, T. C., and Miller, J. F. Smoothness, ruggedness and neutrality of fitness landscapes: from theory to application. 3–44.

[169] Wallnau, K. Volume III: A technology for predictable assembly from certifiable components. Technical Report CMU/SEI-2003-TR-009, Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Avenue, Pittsburgh, PA, 15026, 2003.

[170] Wallnau, K. C., Hissam, S. A., and Seacord, R. C. *Building Systems from Commercial Components.* Software Engineering. Addison-Wesley Longman, Lowell, MA, Jun 2002.

[171] Wilhelm, R. *Informatics - 10 Years Back. 10 Years Ahead*, vol. 2001 of *Lecture Notes in Computer Science.* Springer, 2001.

[172] Wirth, N. Program development by stepwise refinement. *Communications of the ACM 14*, 4 (April 1971), 221–227.

[173] Xie, F., Brown, J., and Levin, V. Objectcheck: Model checking tool for model checking executable object-oriented software designs. In *Fundamental Aspects of Software Engineering (FACE-2002)* (Berlin - Heidelberg - New York, April 2002), vol. 489 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 64–79.