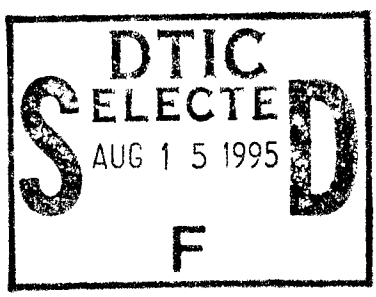
3-1995

Formal Foundations for the Specification of Software Architecture

Mark J. Gerken

AFIT/DSG/ENG/95M-01



FORMAL FOUNDATIONS FOR THE SPECIFICATION OF SOFTWARE ARCHITECTURE

DISSERTATION

Mark James Gerken
Captain, USAF

AFIT/DSG/ENG/95M-01

Approved for public release; distribution unlimited

AFIT/DSG/ENG/95M-01

FORMAL FOUNDATIONS FOR THE SPECIFICATION OF SOFTWARE ARCHITECTURE

DISSERTATION

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Mark James Gerken, B.S., M.S.

Captain, USAF

March, 1995

AFIT/DSG/ENG/95M-01

FORMAL FOUNDATIONS FOR THE SPECIFICATION OF SOFTWARE ARCHITECTURE

Mark James Gerken, B.S., M.S.

Captain, USAF

Approved:

Paul D. Bailor, Chairman — 10 MAR 95

Thomas C. Hartrum — 10 Mar 95

Eugene Santos, Jr — 10 MAR95

Mark E. Oxley — 10 Mar 95

Gary B. Lamont — 10 MAR 95

Robert A. Calico, Jr

Dean, Graduate School of Engineering

Acknowledgements

I would like to thank my advisor, Dr. Paul Bailor, for his insight, guidance, and encouragement — it was instrumental to the successful completion of this research. I would like to thank my committee, Dr. Eugene Santos, Dr. Thomas Hartrum, and Dr. Mark Oxley, for their advice given during the course of the investigation and for their suggestions for improving this document. I am indebted to several other researchers, especially Dr. Douglas Smith, Dr. Richard Jüllig, and Dr. Yellamraju Srinivas of the Kestrel Institute for answering my multiple questions concerning category theory and specification construction techniques. I would also like to thank a fellow graduate student, Captain Frank Young, for the many hours we spent discussing specification techniques.

I especially wish to thank my wife, Debora, for her patience and understanding.

Mark James Gerken

Table of Contents

List of Figures

List of Tables

List of Symbols

List of Abbreviations

List of Definitions

AFIT/DSG/ENG/95M-01

Abstract

This investigation establishes a formal foundation for software architecture that allows for the specification of large, non-trivial software systems using well founded, consistency preserving construction techniques. Two fundamental problems were addressed: how to define and express architectures formally using the concept of theories, and how architecture theories can be practically applied in specification construction. The initial stages of this investigation sought to establish a formal, mathematical relationship between functional specifications of behavior and specifications defining system structure. Experimental results lead to the conclusion that architectures defining the structure of functional operations can be defined using functional logic, but more complex architectures require a separate process logic. A process logic based on Hoare's Communicating Sequential Processes (CSP) was selected for representing and reasoning about system structure and was used in the definition of a process-based specification development system. Specifically, CSP was used to define a category of process-based specifications and specification morphisms. This allowed well-founded specification construction techniques such as specification morphisms, colimits, and interpretations to be applied to the construction of consistent software architecture. Architecture theories expressed in terms of functional and process-based specifications were defined, and translations between these architecture theories were investigated. A feasibility analysis on an image processing application demonstrated that architecture theories can be used to develop specifications for large, non-trivial applications.

FORMAL FOUNDATIONS FOR THE SPECIFICATION OF SOFTWARE ARCHITECTURE

I. Introduction

"Software Development will not become software engineering until the traditional methods of engineering are incorporated." Richard D'Ippolito.(30:256)

1.1 Purpose and Motivation

Traditional engineering makes extensive use of models and libraries of reusable entities; "without reusable technology, engineering could not support the level of productivity and product success that it now enjoys."(30:256) The lessons of engineering have not been lost on the software profession; software researchers and developers have been concerned about re-usability for some time. In 1967, McIlroy (71) "proposed the idea of a software components catalog from which software parts could be assembled, much as is done with mechanical or electronic parts;"(83:99) McIlroy envisioned "interchangeable source code parts."(85) Although research continues in the area of reusing source code, over time the emphasis of re-usability has changed; there is now an increased emphasis on reusing *knowledge*, such as domain theories and specialization information, rather than reusing the *implementation.*

Knowledge can be reused in several ways. One of these is to reuse knowledge about *architecture.* Before proceeding, the use of the term "architecture" needs to be clarified. Different authors have different definitions of architecture:

- A unifying or coherent form or structure; a method or style of building.(68)
- A selection from a set of models and rules of composition that defines the structure, performance, and use of a system relative to a set of engineering goals.(61)

- The high level packaging structure of functions and data, their interfaces and controls, to support the implementation of applications in a domain.(58)

An architecture defines the entities of a system and defines composition rules for these entities. Pipelines are an example of an architecture. The entities in a pipeline architecture are pipes and stages, and the composition rules state that stages can be connected to other stages only through a pipe such that the resulting structure is acyclic and connected.

Some developmental systems such as REACTO (124) directly incorporate architectural notions. Specifications developed in the REACTO environment are based on an implicit architecture for hierarchical state machines. Other systems, such as the Central Archive for Reusable Software (CARDS),(123, 122) use knowledge structures to explicitly represent the architecture of a collection of related applications. In CARDS, new systems are created by instantiating the architectural models defined in these knowledge structures. CARDS explicitly represents potential architectural solutions for related applications, but the definition of architecture in CARDS is informal. One of the purposes of this investigation is to make explicit and formal the definition and use of software architecture in the development of software specifications.

1.2 Investigation Overview

The basic premise of this investigation is that software architecture can be formally defined and used to develop specifications for complex applications where such specifications are constructed using well defined, consistency preserving techniques. Algebraic specifications are one such technique, and they form the basis of this investigation.

In general, algebraic specifications consist of three parts:(108:107)

1. Signatures: Each specification has a collection of sort names and operation names. A signature identifies the basic [elements] of the domain being described, and thus forms a vocabulary for the domain.
2. Axioms: These are formulas generated using the vocabulary provided by the signature. The axioms define the behavior of the operations in terms of properties of values in the sorts.

3. Models: Models are usually algebras which form the denotation of the specification. An algebra corresponding to a signature provides a set for each sort and a function or relation for each operation. The behavior of these sets, functions, and relations are such that they satisfy the axioms.

Axioms are expressed in a logic appropriate for the domain. Functional (stateless) specifications can be expressed in first order predicate calculus. Other logics, such as modal logic, can also be used to define specifications. See, for example, (96, 62) or (77) for a treatment of reactive system specification. The interested reader can find any number of articles and texts describing algebraic specification, such as (108, 91, 119, 16, 24, 54, 34, 35, 111, 127, 128, 23, 42, 43, 40, 56, 91, 45, 41) and (49).

Algebraic specifications are used in this investigation to define architecture. Conceptual foundations for the definition of architecture were established through an investigation of the relationship between algebraic specification and architectural structure. After establishing the conceptual relationship between architecture and algebraic specification, a formal definition of architecture is developed and several architectures are defined within a specification framework. Specifically, three types of architecture are formally defined:

1. A functional (stateless) architecture expressed within a higher order logic;
2. A process-based architecture expressed within a process logic; and
3. A component-based architecture, where specifications are expressed using both higher-order logic and process logic.

Several process-based architecture specifications are formally defined, including pipeline, layered, repository, batch-sequential, and client-server, and relationships between various process-based architectures are investigated.

The feasibility of using architecture specifications in the development of large, non-trivial software specifications is demonstrated through the development of a specification for a portion of an image recognition application. Architecture specifications provide a mechanism through which algebraic specification can be scaled up to the specification of large, complex problems.

This investigation includes a description of a software development formalism based on the concept of software architecture. This research, like the CARDS system, places an emphasis on reusing domain knowledge. However, it is on a more formal basis than that of CARDS. Instead of composing code fragments together to define an application, our formalism can be used to create an application from algebraic specifications. The software development formalism incorporates category theory, architecture specifications, and advances in domain modeling. It allows the system developer to create application specifications, specialize algorithm specifications, explore communications issues, and translate specifications from one architecture to another.

Related work is described in the following section. The assumptions and contributions of this investigation are described in Section 1.4 and Section 1.5, respectively. Section 1.6 outlines the sequence of presentation.

1.3 Related Work

This section describes existing work that is either directly or indirectly related to this investigation. This section begins by taking a look at some existing software development systems such as the Domain Specific Software Architecture (DSSA) program.

1.3.1 Software Development Systems.

1.3.1.1 DSSA. There are six projects within the DSSA program; four of the projects are in military specific domains such as avionics and navigation, while two of the projects, hybrid control and prototyping technology, address underlying support technology.(69) Between these six projects, there are three distinct approaches to software architecture:(72)

1. The avionics project managed by Loral Federated Systems (14) and the command and control project managed by GTE (84) are based on the domain modeling approaches of Prieto-Diaz (e.g., (85, 83, 82)). That is, an architecture is drawn from a domain model of the problem class, where the architecture describes a family of solutions. The Loral project, called DSSA-ADAGE, has as a goal "to provide system devel-

opers with the necessary environment to locate, adapt, compose, generate (write), integrate, and evaluate avionics applications ... by analyzing a problem domain and *creating/refining a set of standardized solutions within it*"(114:22) (emphasis added). The emphasis is on instantiation and verification of pre-existing architectural solutions using code modules.

The Loral project uses a formal language called LILEANNA (115) to describe architectures. LILEANNA is used to specify class hierarchies (in the object oriented sense) and compose them into Ada packages.(25) The GTE approach also defines an architectural model that is instantiated with reusable code modules based on the requirements of the application. Although formal languages are used, verification is limited to type checking and simulation.(14)

This is a fundamentally different approach than the theory-based approach described in Chapter II. Instead of using a formal language to declare an architecture and populate it with reusable code fragments, our approach uses a specialization process with inference rules and soundness axioms to *compose* and *specialize* specifications using architecture theories.

2. The distributed intelligent control project managed by Teknowledge Federated Systems is based on a particular "architectural style" similar to the object-connection-update (OCU) model proposed by the Software Engineering Institute (SEI).(61) This project includes a "domain controller" which generates plans of action. The Teknowledge approach is based on work in robotic control,(3) wherein an architecture "... proposes a multi-level hierarchy of controllers."(72:4) Two levels of controller are used. "A 'Domain Controller' has responsibility for determining plans of action without regard to time constraints, ... [and a] 'meta-controller' directs the execution of the planned actions with the goal of maximizing the use of scarce resources, particularly time constraints."(72:4)

3. The intelligent guidance project managed by Honeywell (1) and the hybrid control project managed by the ORA Corporation (78) are based on formal models of the application domain where the software architecture is derived from these formal models.

(a) The Honeywell system uses a formal language called MetaH to describe architectures. The MetaH language "...allows users to specify how source modules are combined into higher level entities such as processes and modes of operation."(121:1) Honeywell, like Loral, uses a formal language to encapsulate source code written in a traditional programming language. As noted in the MetaH language reference manual, "...an application developer must produce two things: a collection of Ada source modules that implement the functions of the application system; and an application specification that describes how these source modules communicate, share resources, and are to be scheduled in the application system."(121:5) Application specifications are written in MetaH. "A MetaH specification identifies and groups Ada source code modules into entities to be included in the application, describes interfaces for entities, [defines] resource sharing and connections between entities, and [defines] attributes of entities. Applications are specified as one or more modes of operation, where a mode of operation is a collection of processes together with a pattern of communication and resource sharing between processes. Processes are specified as groups of monitors, packages, and subprograms written in the Ada programming language."(121:6) The Honeywell approach, like the Loral and the GTE approaches, is fundamentally different than our approach. MetaH is used to describe relationships between existing source code modules where these relationships define the architecture for an application family.

(b) The ORA Corporation is "investigating an environment for the design, implementation, and evaluation of hierarchical, distributed, intelligent, hybrid control."(78:73) The term 'hybrid control' refers to "an integrated approach to continuous physical devices (mechanical, electrical, hydraulic, etc.) being controlled by discrete computational units (digital CPUs)," where "one attempts to study the problem without ...reducing the discrete to the continuous or the continuous to the discrete."(78:74) These researchers are developing both a theoretical basis for hybrid control as well as design tools whose emphasis is on analyzing and simulating dynamic systems. They are also investigating syn-

thesis of non-linear controllers as a means of "automatically generating control algorithms and software for non-linear hybrid control."(78:75)

The DSSA projects described above contain "reusable engineering experience" (knowledge) in the form of source code modules arranged to define an architecture for a family of related systems. Each of these architectures serves as a model or blueprint for applications developed in their respective domains.

These DSSA projects each define a specific architecture for a problem domain and instantiate the architecture with existing or custom developed modules based on application requirements. In none of the projects is an architecture for an application developed as a specialization of more general architecture theories.

1.3.1.2 CARDS. The Comprehensive Approach for Reusable Defense Software (CARDS) is another code-based reuse system.(123, 122) Like the DSSA-ADAGE project, CARDS exploits architecture-based reuse of code modules. CARDS explicitly represents solution space objects using a frame-based knowledge structure, where the knowledge structure describes relationships between solution space objects. The solution space objects in CARDS are source code and object code modules. CARDS uses two processes during the creation of an application:

1. An elicitor that obtains the requirements and instantiates portions of the ontology for the application.
2. A harvester that collects instantiated objects and complies and links them into an executable form.

CARDS uses the frame-based KL-ONE language (22) to represent relationships between solution-space objects.

1.3.1.3 J-MASS. Another model-based reuse program is the Joint Modeling and Simulation System (J-MASS) Program.(47) The J-MASS system concept document (SCD) describes a software development environment based on templates which are instantiated with code fragments. The J-MASS SCD describes two libraries:

1. A Software/Data Structures Library containing "software structural templates used for development of software components"(47:7) as well as "data structure templates."
2. A Software Components Library containing pre-compiled "parts" that are ready to be assembled into simulation models.

1.3.1.4 LEAP. The Lockheed Environment for Automatic Programming (LEAP) is a software development environment built by the Lockheed Software Technology Center. LEAP solicits application requirements through a series of templates. Based on the requirements, a suitable architecture for the application is created. This architecture is an instantiation/specialization of an architecture description stored in the LEAP knowledge base. Constraint propagation is used to facilitate requirements acquisition and thereby facilitate instantiation of the architectural template. LEAP contains at least two features related to the research described here:

1. Object-based interface. LEAP allows a user to define architectural aspects of an application via a graphical user interface (GUI). Inputs to the GUI are automatically reflected in the underlying application.
2. A parameterized, executable theory-based language called Common Intermediate Design Language (CIDL).(80) Applications developed using LEAP are represented internally as a collection of CIDL components. The CIDL representation of an application can be compiled and executed, or it can be translated into Ada or LISP for execution. Although CIDL is theory based, the axioms of the language are not yet used,(79) nor is it used to define architecture.

1.3.1.5 Kestrel Interactive Development System (KIDS). KIDS is a formal and mathematically well-founded software specification and synthesis system.(100) In KIDS, a problem is specified using sorts and operations defined in an underlying domain theory.

KIDS includes a collection of algorithm theories which can be specialized for a given problem. Once an algorithm has been specialized, it is transformed from its specification

representation to a representation in the REFINE language.(88) Once in REFINE, the algorithm can be optimized, compiled, and executed.

1.3.1.6 SpecWare. A theory-based system being developed at Kestrel Institute is *SpecWare.*(18, 19, 56, 55) SpecWare is based on the work of Wirsing (128) and Turski and Maibaum (117) among others where a category of specifications and specification morphisms is used to combine specifications into new specifications. One of the benefits of defining a specification development system in terms of category theory is that consistency preserving specification construction operations can be defined (see Chapter III for details). SpecWare is a higher order, functional specification development system that incorporates algorithm theories and specialization techniques. However, architecture theories are not explicitly used to define structure; there is no mechanism in SpecWare for connecting the output of one operation to the input of another outside of using nested function calls. SpecWare includes a collection of mappings from its specification language, *SLANG*, to compilable programming languages such as C or LISP. Because of the benefits offered by its foundation in category theory and its ability to specialize existing specifications, SpecWare was used in this investigation to define functional specifications.

1.3.1.7 Other Approaches. Other research related to this investigation is in area of architecture description languages and architecture definitions. Several papers attempt to formally describe either architecture classes or other aspects of software architecture, such as (74, 6, 38, 39, 76, 1, 78, 64, 114, 94, 93, 4, 5, 72, 60) and (95). Each of these papers provide some insight into the features a formal definition of architecture must support. For example:

- A pipe and filters formal model based on $\mathcal{Z}$ in (4).
- Several classes of architecture including pipeline, object-oriented, layered hierarchy, table driven interpreters, and repositories are described in (94). Although these architecture classes are not formally defined in the paper, the author does describe their identifying characteristics and structural — not semantic — patterns.
- Module interaction is formally described using $\mathcal{Z}$ in (37).

Other clues to the aspects a formal definition of architecture must address can be found by investigating architecture description languages (ADL). One such language is LILEANNA which was previously mentioned as part of the Loral DSSA work. ADLs are described in the following subsection.

1.3.2 Architecture Description Languages.

1.3.2.1 ADLs. An ADL called LIL is described in (43). An extension to LIL for use with Ada, called LILEANNA, is described in (115). LIL is a theory-based specification language with an emphasis on interfaces and data flow. Theories in LIL "declare properties an actual parameter must have to meaningfully substitute for the formal parameter of a generic entity."(43:127) Theories in LIL also define an entity's properties. Tracz states that specification construction operations such as colimits can be defined over LILEANNA specifications.(115) Although both LIL and LILEANNA support inheritance and parameterization, Tracz's work does not include the use of architecture theories. The specifications developed using LILEANNA are not structured to support automated code synthesis, but instead are used for "automated selection, composition, tailoring and instantiation of (existing) Ada code."(120)

Another ADL is MetaH. Although MetaH allows "no functional specification beyond a simple naming of inputs, shared objects, and outputs,"(120) it has several interesting aspects. For example, MetaH has nine different types of components, such as events, ports, and processes. The language also has four different classes of "connection," an example of which is a port connection.

μ-Rapide is an ADL designed for event-driven systems.(64) One of the interesting features of μ-Rapide is its handling of connections between components. "Connections themselves may have complex behaviors specified, and in general, the expressive power of connections and component behaviors is equivalent. In fact, there is a straight-forward way of rewriting a complex connection specification as a component specification ..."(120).

Each of the three ADLs have two classes of entity: a component and a connection. In addition, Vestal notes the following similarities:

- Component interface declarations define types of components, where there may be multiple instances of a declared component type.
- Components have distinct interface and implementation aspects.
- Connections can be made between "things" in component interfaces, not always directly between components themselves.
- Components may be defined in terms of sub-components and connections between them.

ADLs may be used to document or declare possible architectural solutions which — depending on the ADL — may be parameterized. The emphasis of ADLs is quite different than our formalism described in Chapter II. The emphasis of ADLs is on the efficient management of the structure and implementation of large applications. In contrast, our formalism described in Chapter II is concerned with the generation of a consistent architectural and behavioral definition of large applications.

1.3.2.2 Module Interconnection Languages (MIL). MILs, such as Thomas' MIL (113), state what the system modules are and how they fit together to implement the system's functions. MILs are not concerned with what the system does (specification information), how the major parts of a system are embedded into the architecture (analysis information), or how the individual modules implement their function (detailed design information).(86)

According to (86), researchers working in formal models view interconnection in two ways: as a structural model of the resource usage and as a consistency model of the construction of the system. In short, a MIL is a language for "programming in the large with a formal machine-process-able [sic] syntax that provides a means for the designer of a large system to represent the overall system structure in a concise, precise, and verifiable form."(86:309)

Polylith is a MIL used at the University of Maryland.(87) Polylith provides a "packaging system for analyzing configurations and then generating all stubs, build commands and other interfacing structures according to the developer's abstract interfacing deci-

sions," (15:82) and provides a "run-time system providing various forms of communication support." (15, 87) Polylith is used to document the architecture of a family of applications and is designed to facilitate program compilation such that they are consistent with the defined architecture. Polylith is not designed to manipulate formal algebraic specifications.

The main concepts of a MIL are:

- the use of a separate language to describe system design;
- the ability to perform static type-checking at an intermodule level description;
- the ability to consolidate the design and construction process (module assembly) in a single description; and
- the ability to control different versions and families of a system.

MILs are used to *document* the structure of an application; they are not used to *derive* the structure of an application. Although MILs typically provide static type checking, they do not typically provide semantic compatibility checks. For example, a module designed to compute the square root of an integer argument has an implicit input assumption that the argument it receives is greater than or equal to zero. MILs typically do not ensure that input assumptions such as these are satisfied.

1.3.2.3 Specification Languages and Proof Systems. Many specification languages and specification development systems exist (e.g., (2, 53, 59, 89, 92, 13, 9)). Some of the more widely known systems include KBEmacs, which is a specification development system with an emphasis on specialization of code generics,(125) and LaSSIE, which is a frame-based specification system.(29, 28) Other specification development environments include ENCORES (70), OIKOS (7), Clear (23), Φ-nix (12), and DRACO (73). The text *Automating Software Design* (63) contains a description of several of these specification development systems, and (46) contains a survey of several others.

Three specification development systems or languages, *Larch*, *OBJ3*, and *Promela* are described in the following paragraphs.

1. *Larch.* Larch is a specification language which is bundled with a theorem prover.(48) Larch is a two-tiered specification language consisting of a shared language and an interface language.

   (a) The Larch Shared Language is used to specify sorts, operations, and axioms. The Larch theorem prover can be used to prove properties of *traits*, which are the fundamental specification unit of Larch.

   (b) The Larch Interface Language is a programming language specific language that introduces terms, such as exceptions or timers, that are part of the programming language but not part of the shared language. The Larch interface language can also be used to define state. A $\mathcal{Z}$-like notation is used to denote state manipulation.

2. *OBJ3* is a "wide spectrum functional programming language."(44:1) Two types of module are used in OBJ3:

   (a) *objects* encapsulate executable code, and

   (b) *theories* which "specify both syntactic structure and semantic properties of modules and module interfaces."(44:1)

   OBJ3 supports parameterized programming and includes a theorem prover based on an order sorted equational logic.

3. *Promela* is a process-based specification development system consisting of two back ends:(126)

   - One back end generates C++ code "suitable for compilation onto an embedded controller";
   - The second back-end "generates gate level designs suitable for input to conventional silicon compliers or field-programmable gate array ... tools."

   In Promela, static channel definitions are used to define communication. However, the authors incorporate notions of an "unreliable" channel that may corrupt or completely lose messages. Channels are either typed or untyped, and may be declared to be external (e.g., a hardware interface).

All communication in Promela is either message based or via shared data values. Communication can either be synchronous or asynchronous, but synchronous communication need not block if the other party is not ready to communicate. Instead, the communication attempt could return a "fail" value. All Promela processes are concurrent.

1.3.3 Summary of Related Work. No system or language that develops a formal foundation for general architecture theories, develops a specialization process with inference rules and soundness axioms, or applies these concepts to some problem class was found in any paper. Current architecture based development systems such as CARDS, DSSA, or LEAP tend to contain static architectural descriptions of solution space objects. Architecture description languages such as MetaH and module interconnection languages such as Polylith are targeted toward documenting architectural solutions. Algebraic specification languages such as Larch or SLANG do not incorporate architecture specifications. In addition, differing definitions of the term *architecture* can be found in the current literature.

1.4 Assumptions

The overriding goal of this research was to formally define architecture and architecture specifications, and demonstrate how architecture specifications can be used to define software specifications. Given this goal and the background presented in the previous section, this research is predicated on the following assumptions:

Assumption I.1 Nonfunctional Constraints. *Issues associated with sizing and timing, schedulability, priority, periodicity, and real-time constraints are ignored. All processes of an implementation are assumed to be schedulable. Only functional constraints are considered.*

Sizing and timing issues are abstracted away to simplify the development of architecture theories. Including sizing and timing constraints in an architecture theory requires not only representational structures, but would also require the development of reasoning facilities over those structures. Such a reasoning facility would need to determine whether an imple-

mentation or model exists for a given specification such that the non-functional constraints were satisfied, or which given an implementation, could determine if it was consistent with the specification (i.e., if the implementation is a valid model of the specification). There has been some work in the area. For example, (42) touches on the subject of constraint representation, while (116) addresses reasoning with constraints in more detail.

Assumption I.2 Functional Model Definition. *The application developer is responsible for defining the functional model of an application.*

That is, the application developer identifies the data flows within an application. Precedented design — in the form of parameterized specifications — may already contain functional model information.

Assumption I.3 Software Synthesis. *A KIDS-type synthesis system can be used to synthesize implementations for the functional portions of application specifications. If other logics such as modal logic are used to define component behavior, the assumption is made that a synthesis system exists which can find implementations for specifications written in the given logic.*

Assumption I.4 Operation Termination. *All operations identified in a signature are atomic (without duration) and terminating.*

Note that this assumption has significant implications for the class of application that can be developed under this restricted semantic model. Specifically, operations that are not guaranteed to terminate can still be defined in functional specifications and referenced in process specifications, but claims of lack of deadlock or live-lock over processes referencing these operations may no longer be valid. A process that engages in an evaluation of a nonterminating operation is, for all practical purposes, deadlocked.

Assumption I.5 Communication Network. *A static, reliable communication network is assumed. Communication links, such as pipes or connectors, are declared and defined prior to system execution.*

The assumption of reliable communication eliminates the need to define handling of communication errors. Static communication networks are assumed because the language chosen for defining communication does not support dynamic definition of communication channels.

Assumption I.6 Existence of Colimits. *Colimits of specifications are assumed to exist, and specifications are assumed to be consistent.*

Assuming the existence of colimits of specification diagrams simplifies the theoretical development and, according to Srinivas, such an assumption is reasonable.(109) Specifications are assumed to be consistent due to the semi-decidable nature of first order theorem proving: A specification can be proved to be inconsistent if such an inconsistency exists, but in general it cannot be proven that a specification is consistent.

1.5 Contributions

Based on these assumptions, the contributions of this investigation include:

1. Development of a category of process-based specifications.
2. Definition of a mathematical relationship between functional and process-based specifications.
3. Development of a formal definition of software architecture, including functional, process-based, and component-based architectures.
4. Definition of a specialization process for software architectures.
5. Incorporation of architecture specifications into a software development framework.
6. The ability to develop system level specifications using different logics, separately considering functional and process aspects of an application.
7. Demonstration of the feasibility of developing software specifications using formal architectures.

1.6 Summary

This chapter has provided a brief introduction to the research goals and objectives of this investigation and has presented a summary of related work. The remainder of this dissertation is organized as follows:

- Chapter II introduces a framework for developing system level specifications.
- Chapter III discusses specification construction techniques.
- Chapter IV describes an early attempt to combine functional (stateless) and process-based specifications, and provides a more detailed description of components and connectors.
- Chapter V establishes the mathematical foundation of our framework and formally defines components and connectors.
- Chapter VI defines architecture, and introduces and defines several architecture specifications.
- Chapter VII contains an analysis of the relative expressive power of the process-based architecture theories defined in Chapter VI.
- Chapter VIII demonstrates the feasibility of our framework by developing a process-based specification for a problem drawn from the image recognition domain.
- Chapter IX presents the conclusions of this investigation and provides recommendations for future research.

II. Software Development Framework

This chapter provides an overview of the specification development framework defined as part of this investigation. This discussion is general in nature; a rigorous treatment of the specification development using our framework is in Chapter V.

2.1 Overview of the Framework

A framework for the development of software based on the notion of architecture specifications is depicted in Figure 2.1. Only part of the framework shown in the figure is defined in this research. Specifically, this research establishes the mathematical foundations for the Composition Mechanism (CM) and formally defines architecture specifications including functional, process-based, and component-based architecture specifications. The library of functional specifications shown in the figure exists as part of SpecWare.(18, 19) The remaining portions of the framework, the Design Refinement Mechanism (DRM), the Abstract Target Language (ATL), and the Source Code Generation and Optimization Unit are informally described in this chapter but are left for future research. Additional detail can be found as follows:

1. (81, 105, 99, 101, 100, 104) and (98) contain descriptions of domain specifications and algorithm specifications, and include descriptions of how algorithm specifications can be specialized for specific problems.
2. (56, 57, 55) and (102) contain descriptions of how functional specifications can created using a framework based on Category Theory.
3. (8) and (51) define process logics, and (50, 11, 31, 32) describe implementations or models of process-based specifications.
4. (21) and (75) address optimization in more detail.

Application-level specifications are developed using the CM, where the specification construction operations of the CM are consistency preserving. Existing functional and architectural specifications are retrieved from the specification libraries and used in the development of system specifications. Once a system specification has been developed,

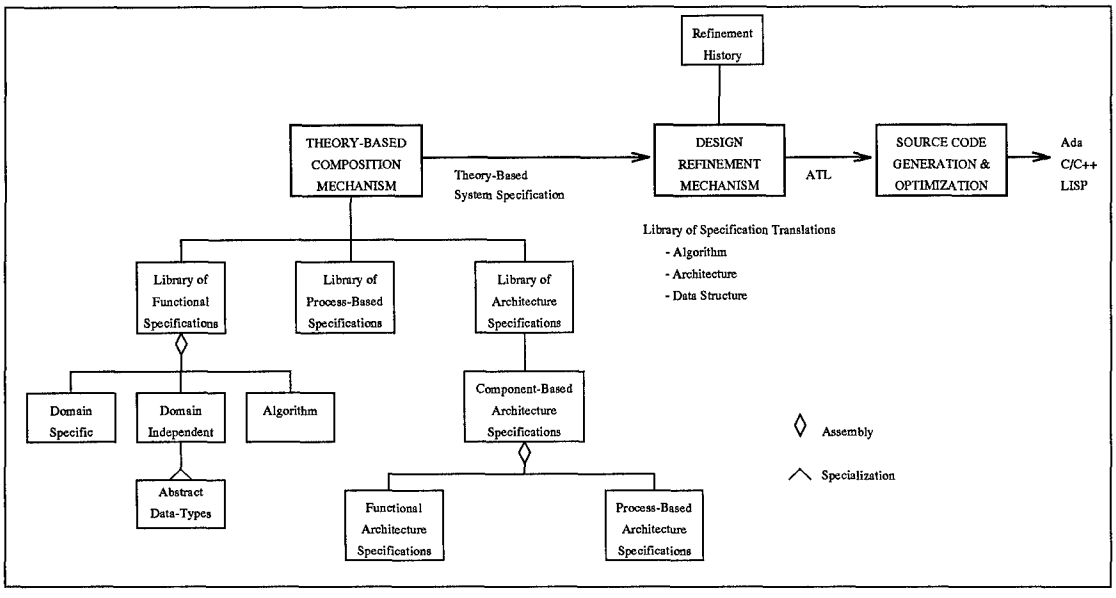


Figure 2.1 Software Development Using Architecture Specifications

the DRM is used to refine the sorts, operations, and architecture(s) of the application. After refinement, the DRM is used to map the specification constructs to an intermediate language, the Abstract Target Language (ATL). A code generator is used to map the ATL representation to a representation in some compilable target language such as Lisp, C, or Ada.

2.2 Composition Mechanism

In Figure 2.1, two types of specifications are used in the development of system specifications:

1. Functional specifications, which are stateless and timeless entities, used to specify sorts and operations.
2. Architecture specifications, used to define structure, state, and interface requirements.

Some system specifications can be developed using only functional specifications, but these systems consist of a single operation that computes $f(x)$ for an input value without regard to state, or a collection of such operations. Process-based specifications use functional op-

erations in the specification of communicating processes; process-based specifications are defined in Chapter V. Architecture specifications allow a developer to better define the structure of an application. Specifically, architecture specifications allow the explicit composition of operators, as in $h = f \circ g$, and provide the flexibility to define communication networks between collections of operations, where such collections may retain state.

The specification composition operations of the Composition Mechanism CM are defined using Category Theory, where the objects of the category are specifications, and the arrows of the category are specification morphisms. Conceptually, a specification morphism defines a relationship between specifications; a specification morphism defines how one specification is contained in another. These terms are defined more precisely in Chapter III. The specification construction operations of the CM allow, for example, a functional specification to be extended with new sorts, operations, and axioms, and they allow sorts and operations of a specification to be refined or specialized. For example, an algorithm specification that requires one of its sorts, say X, to be partially ordered can be specialized to an algorithm specification which requires X to be totally ordered. Specification construction is also addressed in more detail in Chapter III.

2.2.1 Functional Specifications. Functional specifications are used to define the sorts and operations of an application, where sorts are collections of related values. Specifications include statements called *axioms* expressed over the sorts and operations identified in the signature of a specification, where the axioms of a specification are interpreted in some well-founded logical system.

Functional specifications may be defined using reusable specifications drawn from a specification library. These reusable specifications contain definitions of sorts and operations common to variety of problem domains. Domain independent specifications define sorts and operations applicable across multiple problem domains. For example abstract types for sets, sequences, and maps are domain independent, as are the concepts of partial and total order. Domain specific specifications contain definitions of sorts and operators that are not as applicable across multiple problem domains. For example, a domain specific sort for representing complex-valued digital signals could be defined.

```
spec Problem-Theory is
        sorts D, R
        op I : D → Boolean
        op O : D, R → Boolean
end-spec
```

Figure 2.2 Problem Theory

Relationships between specifications are represented using formal objects called *arrows* or *morphisms.* These objects define how the specification at the tail of the arrow is contained within the specification at the head of the arrow. For example, consider the simple functional specification *Problem-Theory* show in Figure 2.2. This specification, written using the syntax of the specification language *SLANG*,(19) introduces two sorts, D and R, and two operations $I : D \rightarrow Boolean$ and $O : D \times R \rightarrow Boolean$, where I defines an input condition and O defines an output condition. This simple specification is contained within the global search algorithm specification shown in Figure 2.3. The specification *Filtered-Global-Search* encapsulates global search as follows:(100, 105)

- D is the input sort.
- R is the output sort.
- The function $I: D \rightarrow Boolean$ constrains the input domain.
- $O : D, R \rightarrow boolean$ is a function defining the output condition. $O(x, z)$ evaluates to true if and only if the output domain value $z \in R$ is a feasible solution with respect to input $x \in D$.
- $\hat{R}$ is the type of the subspace descriptors.
- $\hat{I}$ defines legal search space descriptors.
- $\hat{r}$ and $\hat{s}$ are arbitrary search space descriptors.
- $\hat{r_0}(x)$ is the descriptor of the initial set of candidate solutions.
- $Satisfies(z, \hat{r})$ evaluates to true in case z is in the subspace defined by $\hat{r}$.
- $Split(x, \hat{r}, \hat{s})$ means that $\hat{s}$ is a subspace of $\hat{r}$ with respect to input x.

- *Extract*$(z, \hat{r})$ defines the conditions under which z may be directly extracted from $\hat{r}$.
- *GS0* asserts that the initial space descriptor is a legal space descriptor.
- *GS1* asserts that legal space descriptors split into legal space descriptors.
- *GS2* states that all feasible solutions are contained in the initial space.
- *GS3* gives the denotation of an arbitrary descriptor $\hat{r}$. An output object z is in the set denoted (described) by $\hat{r}$ if and only if z can be extracted after finitely many applications of *Split* to $\hat{r}$.
- *Filter* : $D,\hat{R} \rightarrow$ *boolean* is a feasibility filter used to eliminate spaces from further processing. By construction, the filter only eliminates spaces that do not contain feasible solutions.

External to *Filtered-Global-Search* is a soundness axiom which describes the conditions in which a global search theory $\mathcal{A}$ can be refined into a global search theory satisfying the constraints of a problem theory $\mathcal{B}_F$.

The relationship between *Problem-Theory* and *Filtered-Global-Search* can be defined by the simple arrow *Problem-Theory* $\mapsto$ *Filtered-Global-Search* defined by the map $D_{PT} \mapsto D_{GS}$, $R_{PT} \mapsto R_{GS}$, $I_{PT} \mapsto I_{GS}$, $O_{PT} \mapsto O_{GS}$ where the subscript PT refers to *Problem-Theory* and the subscript GS refers to *Filtered-Global-Search.*

Arrows between functional specifications in the specification library define a hierarchy of related specifications. Although not developed as part of this investigation, an intelligent retrieval system or library manager could be defined. This retrieval system could be used to index into the specification library and retrieve specifications satisfying some set of user defined constraints. For example, such a retrieval system could be used to retrieve all specifications defining or containing a partial order. The developer could then filter this set of specifications using additional constraints, after which, the remaining specifications could be refined or incorporated into a system specification.

Not only does the specification library contain specifications for abstract types, it also contains algorithm specifications. Algorithm specifications represent the structure common to a class of algorithms and abstract out concerns about the specific problem to be

```
spec Filtered-Global-Search is
Sorts D, R, R̂
      op I : D → Boolean
      op O : D,R → Boolean
      op Î: D,R̂ → Boolean
      op r̂0 : D → R̂
      op Satisfies: R,R̂ → Boolean
      op Split: D,R̂, R̂ → Boolean
      op Extract: R,R̂ → Boolean
      op Filter : D,R̂ → Boolean
      axiom GS0 is
            (fa x (implies (I x) (Î x (r̂0 x))))
      axiom GS1 is
            (fa x (fa r̂ (fa ŝ (implies (and (and (I x) (Î x r̂)) (Split x r̂ ŝ))
                                       (Î x ŝ)))))
      axiom GS2 is
            (fa x (fa z (implies (and (I x)(O x z))(Satisfies z (r̂0 x)))))
      axiom GS3 is
            (fa x (fa r̂ (fa z (implies (and (I x)(Î x r̂))
                                       (iff (Satisfies z r̂)
                                            (ex ŝ (and (Split* x r̂ ŝ)(Extract z ŝ))))))))
      axiom filter is
            (fa x (fa r̂ (ex z (implies (and (and (satisfies z r̂) (O x z))
                                            (Î x r̂))
                                       (filter x r̂)) )))
end-spec
```

Figure 2.3 Global Search Theory

solved, the control strategy (top down versus bottom up versus depth-first etc.), the target language and style (e.g., functional versus imperative), and the target architecture.(105) The composition mechanism can be used to define an arrow from the specification *Problem-Theory* to a problem specification in some problem domain, such as the problem *key-search* described in Appendix B. The theory based operations of the CM are used to define an *interpretation* from *Filtered-Global-Search* to the problem specification containing *key-search*. An interpretation defines how *Filtered-Global-Search* can be specialized such that it can be used to find solutions to the *key-search* problem. Specification construction techniques such as this are described in more detail in Chapter III. See (103) or (102) for a discussion concerning algorithm theory interpretation.

A hierarchy of algorithm specifications has been developed by researchers at Kestrel Institute.(105) These algorithm specifications include global search, shown in Figure 2.3, problem reduction,(101) and divide and conquer (99) among others. Also developed at Kestrel Institute is a higher order specification language, *SLANG*,(19) useful for defining functional specifications, and a specification development environment, *SpecWare*,(18) based on Category Theory. Both SpecWare and SLANG are used throughout the remainder of this dissertation; the syntax and semantics of SLANG are described in Chapter III.

Operations identified in the signature of a specification can be combined using a functional architecture specification to define new operations, or the signature of a specification can be used to define a *component*, where a component defines the communication protocol of its operations. Component-based architecture specifications are then used to define communication networks between these operations. This leads to the concept of an architecture specification.

2.2.2 Architecture Specifications. Functional specifications define sorts and operations. Architecture specifications define structure and state, and they define how complex structures can be built out of simpler structures. As shown in Figure 2.1, there are at least three types of architecture specifications:

1. Functional. Functional architecture specifications are used to define functional operators in terms of other operators. That is, functional architecture specifications are used to compose operators to define new operators, or they can be used to decompose an operator into a structured collection of simpler operations. For example, the operation $f : D \rightarrow R$ could be defined by the composition of the operations $g : D \rightarrow E$ and $h : E \rightarrow R$. That is, $f = h \circ g$. Functional architecture theory is formally defined in Section 6.2.1.
2. Process-Based. Process-based architecture specifications define communication protocols, process structure, and state. Process-based architecture specifications define the structure of an application as a collection of processes and a communication network between them. The fundamental building blocks of process-based architecture specifications are processes and communication channels, where process semantics

are provided by C.A.R. Hoare's Communicating Sequential Processes (CSP).(52) Written in a language called *ISLANG*, process specifications and process-based architectures presented in Chapters V and VI respectively.

3. Component-Based. Component-based architecture specifications are an extension of process-based and functional architecture specifications. A component consists of a functional specification defining sorts and operations, and a process specification that defines interface requirements. Component-based architecture specifications define how components can be defined in terms of other components. In this case, a system specification consists of a component specification which defines the processes, communication network, communication protocol, sorts, and operations of an application. Components and component-based architecture specifications are defined in Sections 5.6 and 6.2.3, respectively.

While this section has presented a brief overview of the composition mechanism depicted in Figure 2.1, it has provided limited information on how a developer would use the CM to develop a system specification. Using the composition mechanism to develop system level specifications is the topic of the next section.

2.2.3 Using the Composition Mechanism. There are at least two ways in which the composition mechanism can be used to develop system specifications: a top-down approach (imposed architecture) and a bottom-up approach (constructed architecture).

2.2.3.1 Imposed Architecture. Given a specification S, an architecture specification A can be used to decompose the sorts, operations, or processes of S into a collection of simpler, cooperating elements. For example, a functional architecture specification encapsulating operator composition can be be used to define an operation f of S to be the composition of two, simpler operations as in $f = g \circ h$. An architecture specification defines a structuring of some of the elements of S by defining how operations or processes of S are decomposed into simpler elements. In the case of functional architecture specifications, the elements structured are functional operations. In the case of process-based architecture specifications, the elements structured are processes. See Figure 6.2

of Section 6.2.1 for an example of using a functional architecture specification to define structure.

The advantage of this approach is that complex specifications can be decomposed into simpler structures. For example, it might not be possible to define an interpretation from an algorithm specification in the specification library to a given problem specification, especially if the problem specification involves aspects of more than one algorithm theory. However, if the problem specification were decomposed into a collection of simpler, cooperating operations, such as $f = g \circ h$, then it might be possible to define interpretations from algorithm specifications to g and h, respectively and then use the composition $g \circ h$ for f.

Consider the following problem. Given an unordered sequence A and an element el in the sequence, determine where in A the element el would appear if the sequence were ordered. That is, define an operation f such that $f(A,el) = i$ where the output condition of f is $f(A,el) = i \Leftrightarrow$ *(exists Z (permutation (A,Z)* $\wedge$ *ordered(Z)* $\Rightarrow$ *Z[i] = el). Permutation* evaluates to *true* if and only if its arguments are permutations of each other and *ordered* evaluates to *true* if and only if its argument is ordered. Note that in this case, the conjunct *permutation (A,Z)* $\wedge$ *ordered(Z)* is *true* only if Z is an ordered permutation of the input sequence.

This problem contains aspects of both divide-and-conquer for the sorting of the input sequence and global search for the searching of the ordered sequence for occurrences of the input element. Thus, one possible decomposition of this problem would be *search (sort (A), el)*, where *sort* is a specification for sorting and *search* is a specification for searching an ordered sequence for occurrences of a given element. That is, *f(A,el)* could be defined by *search (sort (A), el)*. A functional architecture specification defining operator composition could be specialized via specification morphism for this purpose. The consistency preserving nature of specification morphisms ensures the resulting specification satisfies the original input specification. Unfortunately, the mathematics required to demonstrate this type of specialization have not yet been developed at this point in this document. A more detailed treatment of using architecture specifications in a top down manner must

therefore wait until the mathematical foundations of specification construction using architecture specifications have been established.

Architecture specifications can also be used to compose larger, more complex specifications from smaller, simpler specifications. That is, architecture specifications can be used in a bottom-up manner to compose simpler specifications to define specifications for complex problems. This bottom-up approach is discussed next.

2.2.3.2 Constructed Architecture. In the bottom-up approach, a software specification is constructed out of a collection of simpler, less complex specifications, rather than decomposing an input specification into a collection of simpler specifications. Architecture specifications are used to define the compositions. For example, a layered architecture specification can be used to compose two process or component specifications such that one is subordinate to the other. For example, a bottom-up construction technique is used in Chapter VIII to specify a segment of an image processing application.

One of the advantages to bottom-up construction is that system specifications can be constructed out of specifications that may be both more reusable and easier to write. A disadvantage of a bottom up construction is that it requires the developer to guide the composition according to his or her perspective of the application. Unlike a top down approach, a system specification does not exist until late in the specification development process. Some properties of the specified system can be investigated as the system specification is developed, but some system level properties might not be well-defined until late in the development process.

Whether employed in a top-down or a bottom-up construction, architecture specifications define application structure.

Once a specification for a given application has been created, the next task is to define or construct an implementation of it. As shown in Figure 2.1, the Design Refinement Mechanism is used for this task.

2.3 Design Refinement Mechanism (DRM)

The Design Refinement Mechanism (DRM) is used to find models of specifications. That is, the DRM is used to translate specifications from a denotational language to an intermediate, executable language such that the translation is behavior preserving. For example, operations defined by a specification to be associative can only be mapped to associative executable operations.

The Library of Specification Translations shown in Figure 2.1 contains a collection of mappings which define how to translate the constructs of the specification language to constructs in a lower-level abstract target language (ATL). More formally, the DRM is responsible for translating expressions in the specification language to expressions in the ATL such that satisfaction is preserved. This topic is addressed in more detail in Section 3.4.

Once a specification has be translated into ATL, it may be optimized using optimization rules that hold in the target language. For example, it may be possible to collapse a process definition found in a component specification into a single operation or a collection of operations within the ATL. Once a specification has been optimized and translated to ATL, a code generator can be used to map ATL expressions to expressions in some compilable target language such as Ada, Lisp, or C.

Although the DRM is a necessary part of the software formalism depicted in Figure 2.1, it is not further defined as part of this investigation.

2.4 Summary

This chapter has provided an overview of one approach to incorporating architecture theories into the development of formal software specifications. Two primary mechanisms are used for this task. The Composition Mechanism (CM) is used to develop system level specifications as well as to specialize algorithm specifications, to define specification interpretations, and to define application structure via architecture specifications. The second mechanism, the Design Refinement Mechanism, is used to translate system specifications created using the CM into an intermediate language which may then be optimized and

mapped via a source code generator and optimization unit to a compilable target language such as Ada, C, or Lisp.

The treatment of the topics presented in this chapter has been informal. A mathematically rigorous treatment of these topics is presented in the remaining chapters of this dissertation. Specifically, the remaining chapters of this dissertation establish a mathematical foundation for the specification of software architecture, where software architecture is defined using a combination of functional and process-based specifications.

III. Combining Theories to Make Theories

> "*Large specifications necessitate structuring mechanisms using which we can build specifications out of smaller, 'mind-sized' chunks.*"(107:46)

3.1 Introduction

The field of mathematics has existed for thousands of years. During this time, specific notation has developed which allows mathematicians, scientists, and engineers to reference complex notions relatively simply and abstractly. For example, "when a mathematician says that he is considering a continuous function or a particular form of a partial differential equation, he is expressing himself on a linguistic level very far removed from the level of natural arithmetic."(117:65) The notion of a linguistic level is comparable to the notion of abstraction. That is, the mathematician is able to reason — using established patterns of reasoning — at a given level of abstraction using, perhaps, named theorems to prove or disprove a given hypothesis. If needed, any given step in a proof at one level of abstraction can be minutely investigated at a lower level.(117)

As stated by Turski and Maibaum (117), there are several significant differences between the development of software as practiced today and the development of mathematical proofs:

- There is a lack of established linguistic levels in the field of software development. The terminology is often chaotic (e.g., object-oriented versus object-based development), basic premises are unlisted, etc.
- There is no established tradition for deductive reasoning.
- The linguistic system is confusingly used, including its simplest, grammatical aspects.
- While the mathematician usually works within a single linguistic level, a software developer almost invariably has to deal with several essentially different linguistic levels.

As noted by Turski and Maibaum, "If the specification and subsequent development of software is to be made rigorous, there is no way that present confusion could be preserved.

On the other hand it is unthinkable that specification writing and software development for millions of applications could be more demanding of the people involved in these activities than mathematics."(117:66)

Instead of attempting to preserve the traditional methods of software development within a rigorous framework, Turski and Maibaum point out that "software specification for application domains is the process of theory formation for these domains. ...If, however, specification construction is in fact theory-making, we must think about means which would make this task manageable."(117:66-67) Software development using a theory-based formalism is shown in Figure 2.1, where algebraic specifications are combined to create an application. This chapter defines some of the terminology used to establish the theoretical foundations of the formalism shown in Figure 2.1.

The software development formalism described in Chapter II is based on concepts drawn from category theory, where category objects are specifications and category arrows are morphisms. Several different types of specification construction operation can be defined in terms of arrows or morphisms. For example specification extension and specification translation can be defined as arrows. Regardless of the type of specification composition operation, all specification morphisms have a common or universal property: they translate the axioms of the source specification into theorems of the target specification. That is, the translated axioms of the source specification logically follow from or are a consequence of the axioms of the target specification. These concepts are made precise in the following section.

Different categories of specification and specification morphism can be defined. For example, a category of first order logic specifications can be defined wherein the definition of logical consequence is based on logical implication. Axioms in such a category could be logical expressions in prenex normal form defined over the sorts and operations of the specification, and logical implication, $\Rightarrow$, is used to define theorems. That is, if Φ is a collection of predicates defined over the sort and operator symbols Σ of a functional specification SP, and if ϕ is also a predicate defined using only the symbols in Σ, then ϕ is a theorem of SP if and only if $\Phi \Rightarrow \phi$. Other categories of specifications and specification

morphisms can be defined. For example, equational logic can be used to define a category of functional specifications.(41)

The approach taken in this investigation is to define different categories of specifications and specification morphisms where specifications in one category may be better able to represent application requirements than specifications in another category. Specifically, a category of functional specifications and functional specification morphisms is used to define sorts and stateless operations, and a category of process-based specifications and process-based specification morphisms is defined and used to define processes and state. Architecture theories are then defined using these categories.

This chapter establishes the mathematical foundations required to define these specification categories. Specifically, the terms introduced and defined in this chapter are used to establish the category **Spec** of functional specifications and functional specification morphisms. Examples of using **Spec** to develop functional specifications can be found throughout this chapter. In Chapter V, a category of process based specifications and process based specification morphisms is defined.

3.2 Basic Definitions

Before categories of specifications and specification morphisms can be meaningfully described, a few terms must be defined. Because category theory is an integral part of specification construction, a definition of category theory is presented below.

Definition III.1 Category.*(106) A category C comprises*

1. *a collection of things called* C-objects*;*
2. *a collection of things called* C-arrows*;*
3. *operations assigning to each C-arrow f a C-object dom f (the domain of f) and a C-object cod f (the codomain of f). If a = dom f and b = cod f we display this as*

$$f : a \to b \quad or \quad a \xrightarrow{f} b$$

4. *an operation,* $\circ$, *called* composition, *assigning to each pair* $\langle g, f \rangle$ *of C-arrows with* $dom\ g = cod\ f$, *a C-arrow* $g \circ f : dom f \rightarrow cod\ g$, *the* composite of f and g *such that we have the* Associative Law*: Given the configuration*

$$a \xrightarrow{f} b \xrightarrow{g} c \xrightarrow{h} d$$

of C-objects and C-arrows, we have

$$h \circ (g \circ f) \quad = \quad (h \circ g) \circ f.$$

5. *an assignment to each C-object* b *a C-arrow* $id_b : b \rightarrow b$, *called the* identity arrow on b, *such that we have the* Identity Law*: For any C-arrows* $f : a \rightarrow b$ *and* $g : b \rightarrow c$

$$id_b \circ f = f \quad and \quad g \circ id_b = g \qquad \square$$

If D is a category, the symbol $|D|$ will be used to refer to the C-objects of D, while D^{op} refers to the category D with its arrows reversed.

Note that the definition of category theory is somewhat broad; for example, directed graphs with self loops can be considered a category. However, the relationship between category theory and specifications (which are technically presentations of theories; see definition III.17) is of more concern.

3.2.1 The Category **Sign.** Functional specifications consist of a collection of sort symbols, operator symbols, and axioms. The sorts and operator symbols of a specification form the *signature* of the specification. This section defines a category of signatures and signature morphisms.

The concept of signature and signature morphism is formalized in the following definition.

Definition III.2 Signature and signature morphisms. *(107) A signature* $\Sigma = \langle S, \Omega \rangle$, *consists of a set* S *of* sorts *and a set* Ω *of* operation symbols. *Associated with each operation*

symbol is a sequence of sorts called its rank. *For example,* $f : s_1, s_2, \ldots, s_n \rightarrow s$ *indicates that* f *is the name of an* n*-ary function, taking arguments of sorts* $s_1, s_2, \ldots, s_n$ *and producing a result of sort* s. *A nullary operation symbol,* $c :\rightarrow s$, *is called a* constant *of sort* s.

Given two signatures $\Sigma = \langle S, \Omega \rangle$ *and* $\Sigma' = \langle S', \Omega' \rangle$, *a signature morphism* $\sigma : \Sigma \rightarrow \Sigma'$ *is a pair of functions* $\langle \sigma_S : S \rightarrow S',\ \sigma_\Omega : \Omega \rightarrow \Omega' \rangle$, *mapping sorts to sorts and operations to operations such that the sort map is compatible with the ranks of the operations, i.e., for all operation symbols* $f : s_1, s_2, \ldots, s_n \rightarrow s$ *in* Ω, *the operation symbol* $\sigma_\Omega(f) : \sigma_S(s_1), \sigma_S(s_2), \ldots, \sigma_S(s_n) \rightarrow \sigma_S(s)$ *is in* Ω'.

The composition of two signature morphisms, obtained by composing the functions comprising the signature morphisms, is also a signature morphism. The identity signature morphism on a signature maps each sort and each operation onto itself. Signatures and signature morphisms form a category, **Sign**, *where the signatures are the* C*-objects and the signature morphisms are the* C*-arrows.* □

Signature morphisms allow the definition of mappings between the signatures of specifications; they allow a developer to refine the definitions of existing sorts, and to rename or restrict the domain and range of operations. However, the rank of the operations defined in the signature remains constant under the signature morphism (signature morphisms in this context are called *signature preserving*). Morphisms are total functions; they define how one specification is related to another. One type of morphism, an injection, can be used to show how one specification is contained in another. This leads to the concept of *extension:*

Definition III.3 Extension. *A signature* $\Sigma_2 = \langle S_2, \Omega_2 \rangle$ *extends a signature* $\Sigma_1 = \langle S_1, \Omega_1 \rangle$ *if* $S_1 \subseteq S_2$ *and* $\Omega_1 \subseteq \Omega_2$. □

For example, consider the signature *Bool*:

sig *Bool* is
    sorts bool
    op and : bool,bool → bool
    op or : bool,bool → bool
    op not : bool → bool

```
    const true :        → bool
    const false :       → bool
end-sig
```

This signature defines a single sort, *bool*, and defines five operations, two of which are the constants (nullary operations) *true* and *false*. The operation *implies* could be added to this signature using a signature extension:

```
sig ExtendedBool is
    import Bool
    op implies : bool,bool → bool
end-sig
```

which is equivalent to the following signature:

```
sig ExtendedBool is
    sorts bool
    op implies : bool,bool → bool
    op and : bool,bool     → bool
    op or : bool,bool      → bool
    op not : bool          → bool
    const true :           → bool
    const false :          → bool
end-sig
```

The keyword **import** indicates that the signature *ExtendedBool* is an extension of another signature and eliminates the need to explicitly list the inherited sorts and operations. The extended signature contains all of the sorts and operations of its parent signature *Bool*.

During the development of a signature for an application, it may be advantageous to apply both signature extensions and signature morphisms to define new signatures. For example, a signature S_B could be extended with some additional sorts and operations to create the signature S_E. In addition, a signature morphism could also be applied to S_B to rename the sorts of the signature, creating a signature S_M. S_E could then be combined with S_M to create a signature which includes the operations of S_E defined over the sorts of S_M. A *pushout* provides the capability to combine two structures such as S_E and S_M along a common part.

Definition III.4 *(106)* Pushout. *Given a pair of C-arrows* $a \xleftarrow{f} c \xrightarrow{g} b$ *with a common domain, their pushout is defined to be a commutative square (shown on the left in Figure 3.1)*

formed with a pair of C-arrows $a \xrightarrow{g'} d \xleftarrow{f'} b$ *(i.e.,* $g' \circ f = f' \circ g$*) such that for any other pair of arrows,* $a \xrightarrow{h} e \xleftarrow{j} b$ *which also form a commutative square (i.e.,* $h \circ f = j \circ g$*) there is a unique C-arrow* $d \xrightarrow{k} e$ *such that the diagram on the right in Figure 3.1 is commutative; i.e.,* $k \circ g' = h$ *and* $k \circ f' = j$*.* □

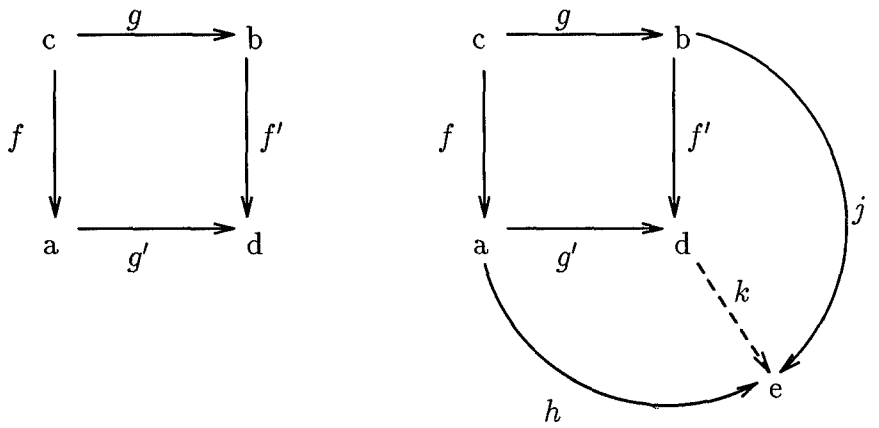


Figure 3.1 Definition of a Pushout. The dashed arrow $k : d \rightarrow e$ indicates that the arrow is unique.

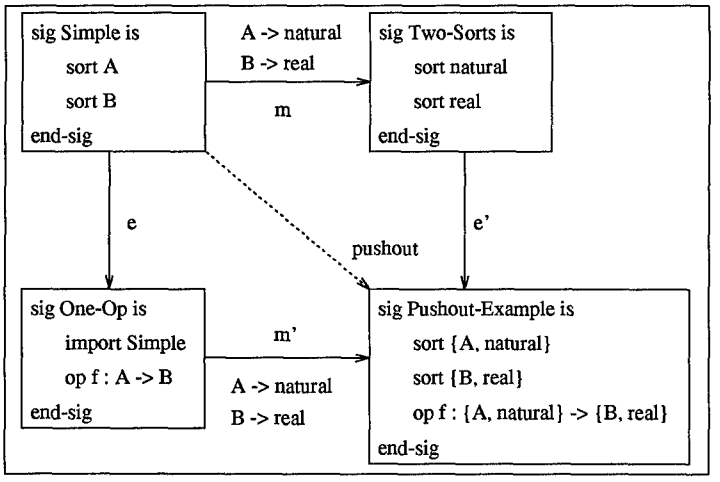


Figure 3.2 Example of a pushout involving signatures

As a simple example, consider the pushout formed from the signatures shown in Figure 3.2. In the figure, the signature *Simple* introduces two sorts, A and B. The signature *Two-Sorts* is defined by the signature morphism m : *Simple* $\rightarrow$ *Two-Sorts* which is defined

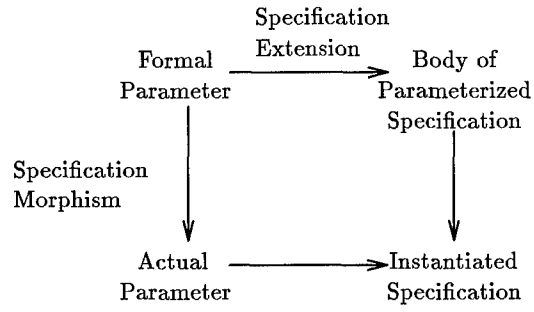


Figure 3.3 Using Pushouts to Instantiate Parameters

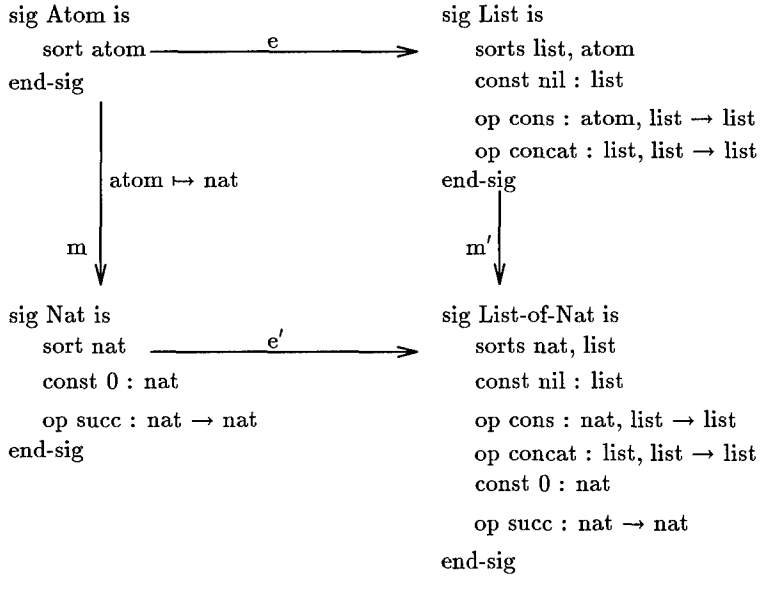


Figure 3.4 Instantiation of a Signature for a List of Natural Numbers

by the map $A \mapsto$ *natural*, $B \mapsto$ *real*. The signature *One-Op* is defined by the extension e which extends *Simple* with the operation $f : A \rightarrow B$. The pushout of the signatures *Simple*, *Two-Sorts* and *One-Op* and the arrows m and e yields the signature *Pushout-Example* containing the operation f defined over the sorts *natural* and *real*. Note that the sort symbols A and *natural* are equivalent in the pushout signature, as are the sort symbols B and *real*. These equivalence classes are represented as sets, and can be removed through translation. Translation is defined later in this chapter.

As the simple example of Figure 3.2 indicates, one of the uses of constructing signatures through pushouts involves parameter instantiation, an example of which is shown in Figures 3.3 and 3.4. Figure 3.3 depicts the general form of using pushouts to instantiate parameters. The formal parameter, be it a signature or a specification, is extended with additional sorts and operations to define the body of a parameterized specification. The formal parameter is also mapped through specification morphism to an actual parameter such that the axioms of the formal parameter are theorems in the actual parameter. The pushout of the resulting diagram defines a specification in which the sorts and operator symbols of the actual parameter are used to complete the definition of the parameterized specification. Figure 3.4 shows an example of using signature parameterization to define a list of natural numbers.

In Figure 3.4, the parameterized signature contains the signature *Atom* and introduces the sort *List* and the operations *nil* and *cons*; the signature is parameterized on the sort *atom*. The actual parameter is created by applying a signature morphism to the signature *Atom* to define the signature *Nat*. A signature for a list of natural numbers is then created by forming the pushout of the formal parameter *Atom*, the actual parameter *Nat*, and the parameterized signature for lists. As a result of the pushout, the parameter *Atom* of the parameterized signature is associated (unified) with the sort symbol *nat*, effectively creating a signature for a list of natural numbers.

Pushouts are useful for combining two C-objects. However, there are situations in which a developer might want to combine more than two objects. For example, suppose that a relation O in some specification S has been defined (the definition of a specification will be given shortly). Further, suppose that S has been extended into three other specifications S_R, S_S, and S_T by adding an axiom over O such that O was reflexive in S_R, symmetric in S_S, and transitive in S_T. If a developer wanted to combine S_R, S_S, and S_T into a single specification and thereby define O to be an equivalence relation, they could form a series of pushouts. However, as the number of objects needing to be combined grows, so too does the number of pushouts required. If a developer needed to combine $n \geq 2$ objects, they would need to form a minimum of $n - 1$ pushouts. A more practical approach would be to generalize the notion of a pushout so that more than two C-objects

could be combined at a time. The generalized notion of a pushout is called a *colimit* and is defined below. The definition of a colimit depends on the definitions of *cone* and *diagram*, so these definitions are presented next.

Definition III.5 Diagram. *A diagram D in a category C consists of a collection D_C of C-objects and a collection D_A of C-arrows such that for any arrow a $\in D_A$,* cod a $\in$ D_C *and* dom a $\in$ D_C. □

For example, Figure 3.4 is a diagram in the category **Sign**. The C-objects of the figure are the signatures *Atom*, *Nat*, *List*, and *List-of-Nat*. The C-arrows are the morphisms e, e', m, and m'.

Definition III.6 Cone. *(106) Given a diagram D in a category C and a C-object c, a cone* from the vertex c to the base D *is a collection of C-arrows* $\{f_i : c \rightarrow d_i \mid d_i \in D\}$, *one for each object* d_i *in the diagram D, such that for any arrow* $g : d_i \rightarrow d_j$ *in D, the triangle shown in Figure 3.5 commutes; i.e.,* $g \circ f_i = f_j$. □

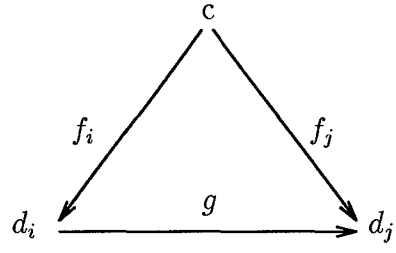


Figure 3.5 Definition of a Cone.

Definition III.7 Colimit. *(106) A colimit for a diagram D in a category C is a C-object c along with a cone* $\{f_i : d_i \rightarrow c \mid d_i \in D\}$ *from D to c such that for any other cone* $\{f_i' : d_i \rightarrow c' \mid d_i \in D\}$ *from D to a vertex c', there is a unique C-arrow* $f : c \rightarrow c'$ *such that for every object* d_i *in D, the diagram shown in Figure 3.6 commutes; i.e.,* $f \circ f_i = f_i'$. □

Note that for the diagram $\cdot \leftarrow \cdot \rightarrow \cdot$, the colimit is the pushout of the diagram. Building colimits is a fundamental theory building operation. Their use will become obvious in the following sections. Appendix A describes some interesting aspects of colimits, such as their initiality.

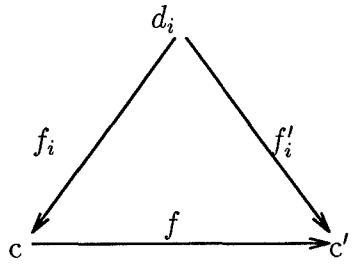


Figure 3.6 Definition of a Colimit.

There are other operations required to help build some of the structures found in specifications. For example, tuples of data types can be defined as the *product* of the sorts involved, and union types can be defined as the *coproduct* of the sorts involved. Products and coproducts are defined below.

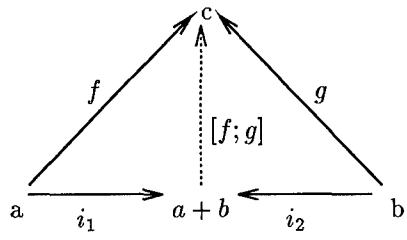
Definition III.8 Coproduct.*(106) A coproduct in a category C of two C-objects a and b is a C-object $a+b$ together with a pair of arrows $a \xrightarrow{i_1} a+b \xleftarrow{i_2} b$ called* injections *such that to any other pair of arrows $a \xrightarrow{f} c \xleftarrow{g} b$ there is a unique arrow $[f;g] : a+b \rightarrow c$ such that the diagram in Figure 3.7 commutes.* □

Definition III.9 Product.*(106) A product in a category C of two C-objects a and b is a C-object $a \times b$ together with a pair of arrows $a \xleftarrow{\pi_1} a \times b \xrightarrow{\pi_2} b$ such that to any other pair of arrows $a \xleftarrow{f} c \xrightarrow{g} b$, there is a unique arrow $\langle f,g \rangle : c \rightarrow a \times b$ such that the diagram in Figure 3.8 commutes.* □

In the specification language SLANG, the functions *embed* and *project* play the role of i and π respectively.

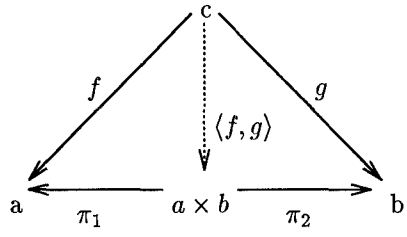
3.2.2 The Category **Spec.** Specifications introduce sort symbols and operator symbols, and define properties that implementations or models of the specification must possess. Specifications are defined below. However, the definition of the term specification depends on the definition of logical consequence, so logical consequence is defined next.

Definition III.10 Logical Consequence.*(107) Given a signature Σ, a Σ-sentence φ is said to be a logical consequence of the Σ-sentences $\varphi_1, \ldots, \varphi_n$, written $\varphi_1, \ldots, \varphi_n \models \varphi$, if each Σ-model that satisfies the sentences $\varphi_1, \ldots, \varphi_n$ also satisfies φ.* □



$[f;g] \circ i_1 = f$ and $[f;g] \circ i_2 = g$

Figure 3.7 Coproduct Definition



$\pi_1 \circ \langle f,g \rangle = f$ and $\pi_2 \circ \langle f,g \rangle = g$

Figure 3.8 Product Definition

Definition III.11 Specification. *(Based on (107)) A specification* SP *is a pair* $\langle \Sigma, \Phi \rangle$ *consisting of a signature* $\Sigma = \langle S, \Omega \rangle$ *and a collection* Φ *of* Σ*-sentences. The collection of all* Σ*-sentences will be denoted by* Sen[Σ].

A model *of a specification* SP $= \langle \Sigma, \Phi \rangle$ *is a* Σ*-model* M *such that* $M \models \psi$ *for each* $\psi \in \Phi$. *The collection of all such models* M *will be denoted by* Mod[SP]. *The sub-category of* **Mod**(Σ) *induced by* Mod[SP] *will also be denoted by* Mod[SP]. □

Σ*-algebras* are used as Σ-models of functional specifications.

Definition III.12 Algebra. *(107) Given a signature* $\Sigma = \langle S, \Omega \rangle$, *a* Σ*-algebra* $A = \langle A_S, F_A \rangle$ *consists of two families:*

1. *a collection of sets, called the carriers of the algebra,* $A_S = \{A_s \mid s \in S\}$; *and*
2. *a collection of total functions,* $F_A = \{f_A \mid f \in \Omega\}$ *such that if the rank of* f *is* $s_1, s_2, \ldots, s_n \rightarrow s$, *then* f_A *is a function from* $A_{s_1} \times A_{s_2} \times \cdots \times A_{s_n}$ *to* A_s. *The symbol* $\times$ *indicates the Cartesian product of sets here.* □

Thus *Mod[SP]* denotes the set of all Σ-algebras of SP in which the axioms of SP are satisfied. For example, consider the specification *Ring* shown in Figure 3.9. The set of integers $Z = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ along with integer addition and integer multiplication form a model of the specification. Note that there are other models for the specification; for example, the set of integers modulo two, $Z_2 = \{0, 1\}$, is also a model of the specification provided the additive inverse function is defined to be the identity function. However the set {*true, false*} representing 1 and 0 respectively with the logical connectives *and, or* and *not* representing $\times$, $+$, and $-$ respectively is not a model of *Ring* since $a + (-a) = 0$ does not hold for any $a \in$ {*true, false*} in this (Boolean) algebra. A specification may have an infinite number of models. For example, consider a specification S developed for some application. *Any* program that satisfies S is a valid model of S. This implies that a variety of *implementations* (programs) that are each valid models of a given specification may be defined.

There is an inverse relationship between the strength of a specification and the size of the set of non-isomorphic models that satisfy the specification. Any model is a valid model of the weakest specification *true*, while only one model exists for the strongest specification *false*. This relationship is made explicit in the following theorem.

Theorem III.1 *Denote by* $SP = \langle \Sigma, \Phi \rangle$ *a specification. If SP contains a contradiction, then Mod[SP] consists of the single model* false.

Proof. *Let* a_1 *and* a_2 *be* Σ*-sentences such that* $\Phi \models a_1$ *and* $\Phi \models a_2$ *and* $a_1 \wedge a_2 \Leftrightarrow$ *false. Denote by* m *an arbitrary model of SP. Because* m *is a model of SP and because* a_1 *and* a_2 *are in* Φ, $m \models a_1$ *and* $m \models a_2$ *which implies* $m \models a_1 \wedge a_2$. *But this simplifies to* $m \models$ *false. By Definition III.10, we get* m *is false.* ■

This theorem has an obvious dual.

Theorem III.2 *Denote by SP an arbitrary well-formed specification. If SP contains the single axiom* true, *then there exists an infinite number of non-isomorphic models of SP.*

Proof. *Follows from theorem III.1 and the principle of duality.* ■

spec *Ring* is
sorts R
ops

_ + _ :	R,R	→ R	/* addition */
0	:	→ R	/* additive identity*/
-_	: R	→ R	/* additive inverse */
_ × _ :	R, R	→ R	/* multiplication */
1	:	→ R	/* multiplicative identity*/

axioms

$\forall a, b, c \in R \cdot a + (b + c) = (a + b) + c$

$\forall a, b \in R \cdot a + b = b + a$

$\forall a \in R \cdot a + 0 = a$

$\forall a \in R \cdot a + (-a) = 0$

$\forall a, b, c \in R \cdot a \times (b \times c) = (a \times b) \times c$

$\forall a \in R \cdot a \times 1 = a$

$\forall a \in R \cdot 1 \times a = a$

$\forall a, b, c \in R \cdot a \times (b + c) = (a \times b) + (a \times c)$

$\forall a, b, c \in R \cdot (a + b) \times c = (a \times c) + (b \times c)$

end

Figure 3.9 A Specification for a Ring (106)

The implication of the first theorem is that if models of specifications are to be developed or defined, such specifications must be free of contradiction. The second theorem implies that these specifications must be "sufficiently strong" to discriminate among non-isomorphic models.

Axioms of a specification restrict the behavior of the operations in Ω. For example, an axiom may restrict the class of models of a specification to those in which the operation is associative. Axioms may also be used to refine the definition of sort symbols in a specification. For example, an axiom could define that one sort was a subsort of another. Another use of sort axioms is to define sorts as products (tuples) or coproducts (unions) of other sorts.

Now that the term specification has been defined, specification morphisms can be defined. However, the definition of a specification morphism depends on the definition of *reduct*, which in turn depends on the definition of *homomorphism*, so these definitions are presented first.

Definition III.13 Homomorphism. *Given a signature $\Sigma = \langle S, \Omega \rangle$ and two Σ-algebras A and B, a Σ-homomorphism $h : A \rightarrow B$ is a family of functions $\{h_s : A_s \rightarrow B_s \mid s \in S\}$ between the carriers of the algebra such that for all operation symbols $f : s_1 \times s_2 \times \cdots \times s_n \rightarrow s$ in Σ and for all $a_1 \in A_{s_1}, a_2 \in A_{s_2}, \ldots, a_n \in A_{s_n}$,*

$$h_s(f_A(a_1, a_2, \ldots, a_n)) = f_B(h_{s_1}(a_1), h_{s_2}(a_2), \ldots, h_{s_n}(a_n)) \ \square$$

Homomorphisms are described in greater detail in Appendix A. Homomorphisms are used to define *reduct.*

Definition III.14 Reduct.*(106) Given a signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ and a Σ'-algebra A', the σ-reduct of A', denoted by $A' \mid_\sigma$, is the Σ-algebra $A = \langle A_S, F_A \rangle$ defined as follows (with $\Sigma = \langle S, \Omega \rangle$):*

$$A_S = A'_{\sigma(s)} \text{ for } s \in S, \quad \text{and} \quad f_A = (\sigma(f))_{A'}, \text{ for } f \in \Omega$$

Given a Σ'-homomorphism $h' : A' \rightarrow B'$ between two Σ-algebras A' and B', the σ-reduct of h' is a Σ-homomorphism $h : A'_\sigma \rightarrow B'_\sigma$, denoted by $h' \mid_\sigma$, and defined by the family of functions $h_s = h'_{\sigma(s)}$ for $s \in S$. $\square$

For example, consider the signature *Group* shown below:

```
sig Group   is
    sorts G
    op _ * _ : G, G → G
    op ϵ     :      → G
    op (_)'  : G    → G /* inverse */
end-sig
```

A signature morphism from *Group* into *Ring* (Figure 3.9) is defined by the map {G $\mapsto$ R, * $\mapsto$ +, $\epsilon \mapsto$ 0, (_)' $\mapsto$ -_}. The σ_{G-R}-reduct of any *Ring*-algebra is obtained by ignoring the extra operations $\times$ and 1 in *Ring.*(107:12) The reduct operation can be viewed as defining inverses to signature morphisms. That is, given two signatures A and B and an arrow a from A to B, a defines how A is contained in B, while $B \mid_a$ defines how B was formed from A.

As given in the following definition, the reduct operation is used to establish the relationship between specifications, specification morphisms, and models.

Definition III.15 Specification Morphism. *(107) A specification morphism from a specification* $SP = \langle \Sigma, \Phi \rangle$ *to a specification* $SP' = \langle \Sigma', \Phi' \rangle$ *is a signature morphism* $\sigma : \Sigma \rightarrow \Sigma'$ *such that for every model* $M \in Mod[SP']$ *we have* $M|_\sigma \in Mod[SP]$. *The specification morphism is also denoted by the same symbol,* $\sigma : \Sigma \rightarrow \Sigma'$.

There are several different types of morphisms, depending on the properties they exhibit. (See Appendix A for a discussion concerning this topic.) For example, a Σ-homomorphism h is a bijection, then h is an *isomorphism.* If an isomorphism exists between specifications A and B, then A and B are said to be isomorphic. The notation $A \tilde{=} B$ is used in case A and B are isomorphic. One property of the relation $\cong$ is that it defines an equivalence relation between specifications. This fact is expressed in the following theorem.

Theorem III.3 *The relation* $\cong$ *between* Σ *algebras is an equivalence relation.*

Proof. *For a proof of this theorem, see Appendix A.* ■

A consequence of this theorem is that structures that are isomorphic cannot be distinguished from each other within the theory framework. That is, if two Σ-algebras are isomorphic, then they are logically equivalent within the theory. If a statement s is valid in a Σ-algebra A, and $A \cong B$, then the isomorphic image of s is valid in B. This fact can be used to simplify specification development, especially when it is easier to reason about or represent statements in an isomorphic algebra. For example, people perform integer addition and subtraction using an algebra whose carrier is the set of integers. However, computer systems perform integer addition and subtraction using an algebra whose carrier is a base-two representation of the set of integers. The two algebras are isomorphic in this case, and each is more widely used in one situation than the other.

When a signature is altered, the axioms over the operations defined in the signature must be altered accordingly. For example, if an operation f in a signature Σ is renamed to h, all references to f in the set of axioms defined over Σ must be replaced by references to h. For example, consider a specification containing an operator symbol $f : D \rightarrow R$ and an axiom *(implies (I x)(O x (f x)))*. Application of the morphism defined by $D \mapsto$ *bag(integer),*

$R \mapsto$ *seq(integer)*, and $f \mapsto$ *sort* results in a specification containing the operator symbol *sort : bag(integer)* $\rightarrow$ *seq(integer)* and the axiom *(implies (I x)(O x (sort x)))*.

Specification morphisms can be used to rename operations in a signature, add axioms to a specification, or add operations to a signature. Specification morphisms can also be used to coalesce elements of a signature. For example, two or more sort symbols can be mapped under specification morphism to a common sort symbol in some target specification. If sort axioms exist, then they must translate to sort theorems in the target specification if the morphism is to be a specification morphism.

Specification morphisms can also be used to add axioms to a specification. Axioms can be added to a specification either by extension or by forming the colimit of a diagram of related specifications. For example, a specification containing a bijective operation f can be formed as the colimit of a collection of specifications and specification morphisms, where f is surjective in one of the specifications and f is injective in another. Then f will be both injective and surjective in the colimit specification.

The following two definitions establish a relationship between specifications and that which they model, theories. The definition of theory depends on the following definition of *closed.*

Definition III.16 Closure, Closed.*(107) Given a signature* Σ*, the closure* $\overline{\Phi}$ *of a set of* Σ*-sentences* Φ *is the set of all* Σ*-sentences which are the logical consequence of* Φ*, i.e.,* $\overline{\Phi} = \{\varphi \mid \Phi \models \varphi\}$*. A set of* Σ*-sentences* Φ *is said to be closed if and only if* $\Phi = \overline{\Phi}$*.* □

Definition III.17 Theory presentation.*(107) A theory* T *is a pair* $\langle \Sigma, \phi \rangle$ *consisting of a signature* Σ *and closed set* ϕ *of* Σ*-sentences. A specification* $\langle \Sigma, \Phi \rangle$ *is said to be a presentation for a theory* $\langle \Sigma, \phi \rangle$ *if* $\overline{\Phi} = \phi$*. A model of a theory is defined just as for specifications; the collection of all models of a theory* T *is denoted* Mod[T]*. Theory morphisms are defined analogous to specification morphisms.* □

Now that a working vocabulary for specifications and specification construction has been established, specification construction techniques can be defined. *Note that a specification is a presentation of a theory, it is not the theory itself.* Because theories are

often infinite while presentations are usually finite,(107) the Composition Mechanism will generally operate over presentations rather than theories. Burstall and Goguen show the soundness of operating on theories by using their counterparts at the presentation level.(40)

3.3 Specification Construction

Five specification construction operations are described in this section. These five operations provide a rich set of construction techniques, including parameterization. These five operations are:

1. Basic Specification. Construct a specification from a signature and a collection of axioms.
2. Translate. Translate a specification via a signature morphism.
3. Colimit. Form a specification by taking the colimit of a collection of specifications and specification morphisms.
4. Import. Importation is similar to the `#include` statement in C and C++. All sorts, operations, and axioms of imported specifications are copied into the importing specification.
5. Interpret one abstract entity using the features provided by others (leading to the notion of a vertical hierarchy of entities).

The last method identified above is the purpose of the Design Refinement Mechanism (DRM) and the library of implementations, but vertical structuring may also take place in the Composition Mechanism. These specification building operations are described in the following section.

Each of these operations is further described in the following subsections. The syntax and semantics of the following operations parallels that used in the SpecWare system being developed at Kestrel Institute.(18, 19)

3.3.1 Basic Specification. Creating a specification using this technique is straightforward: the specification is simply declared. There are no other specification building

operations applied to help create it. Any specification that can be created using any of the other techniques can also be created as a basic specification. However, there is a tradeoff: relatively small and well encapsulated specifications may have a higher potential for reuse than large basic specifications. For example, a specification for sorting complex numbers using problem reduction (99, 101) could be defined as a basic specification, but such a specification might not be well suited for reuse. Instead, an equivalent (isomorphic) specification could be developed from a collection of smaller, less problem specific specifications.

Examples of basic specifications, such as the specification *Bin-Op* listed below, can be found throughout this dissertation. *Bin-Op* introduces a sort E and a binary operation *binop*. The single axiom of *Bin-Op* states that the operation *binop* is associative. Any algebra that includes an associative binary operation over a single sort is a model of this specification. For example, Boolean algebra with the carrier set {*true, false*} is a model of the specification *Bin-Op* because the Boolean operation *or* is an associative binary operation defined over a single sort.

```
spec  Bin-Op  is
      sorts E
      op     binop : E, E → E
      axiom (equal (binop (binop a b) c)
                   (binop a (binop b c)))
end-spec
```

Basic specifications are typically small and relatively domain independent. Basic specifications are usually combined with other specifications in the construction of application specifications.

3.3.2 Translate. The translate operation "creates a copy of a specification with the option of renaming some of the components."(19:13). This implies that specifications created through renaming are isomorphic.

In the semantics of SLANG, if the renaming function maps two or more source elements to a common target, such as mapping the two sort symbols A and B in some specification to a common target sort symbol C, then there will be two distinct sorts in

the target specification with the common name C. Because these two distinct sorts share a common name, the resulting specification may be ambiguous. For this reason, translations will be restricted to be bijections; any translation that attempts to map two or more symbols in a source specification to a common symbol in the target will be considered ill-formed.

Specification translation is defined only over the signature. That is, specification translation maps the symbols in the signature of a source specification to the symbols in the signature of a target specification. The axioms of the source specification are translated by the map defined by the translation. For example, using the SLANG syntax, the expression

```
translate
      spec associative-bin-op is
            sort E
            op       binop : E, E → E
            axiom associativity is
                  (equal (binop (binop x y) z) (binop x (binop y z)))
      end-spec
by { E ↦ F, binop ↦ plus}
```

evaluates to

```
spec associative-bin-op is
      sort F
      op    plus : F, F → F
      axiom associativity is
            (equal (plus (plus x y) z) (plus x (plus y z)))
end-spec
```

Note that although the two specifications above are both named *associative-bin-op*, they are distinct, isomorphic entities.

A typical application of the translate operation is to simplify the denotation of specifications created using the colimit operation. For example, if a specification contained the equivalence class $\{A,\ B,\ int\}$ of sort symbols, a translation of sort symbols defined by the map $\{A,\ B,\ int\} \mapsto int$ could be used to "clean-up" the specification by replacing instances of the equivalence class with the symbol *int*. Equivalence classes of symbols are often produced as the result of colimit operations.

3.3.3 Import. Specifications can import other specifications similar to the way in which programs written in C or C++ can import sort and operator names and definitions using the `#include` construct. All of the structures of included specifications are available to the including specification. For example, a specification for sorting sequences of some total order object may be defined as follows:

```
spec SortSpec is
        import TotalOrder, Seq, Bag
        sorts bag(S), seq(S)
        op ordered : seq(S) → boolean
        op incond : bag(S) → boolean
        op outcond : bag(S), seq(S) → boolean
        axiom (iff (outcond x z)
                    (and (permutation x z) (ordered z)))
        axiom (iff (incond A) true)
        definition definition-of-ordered is
                axiom (ordered [])
                axiom (ordered [a])
                axiom (iff (ordered (concat a b))
                            (and (and (ordered a)(ordered b))
                                 (ordering (last a)(first b))))
        end-definition
end-spec
```

where *(permutation x z)* returns true if and only if z is a permutation of x and *(ordered z)* returns true if and only if z is totally ordered. The operation *outcond* is a boolean function defining the output of the sorting problem, namely that the output is an ordered permutation of the input. The input condition, *incond*, is defined to be the constant *true*. The operations *permutation, [], [_], first, last* and the sort *seq* are defined in the specification *Seq* and are visible within *SortSpec*. The statement *seq(S)* defines a sequence whose elements are of sort S and is "syntactic sugar" for the diagram shown in Figure 3.10. Similarly, the statement *bag(S)* defines a bag whose elements are of sort S. The operation *ordering* and the sort S are defined in the imported specification *TotalOrder* and can be referenced within *SortSpec*. Importation is denoted by the hooked arrow as in *TotalOrder* $\hookrightarrow$ *SortSpec*. All variables referenced in axioms are assumed to be universally quantified unless stated otherwise. Note that the specification *Seq-of-S* shown in the figure was

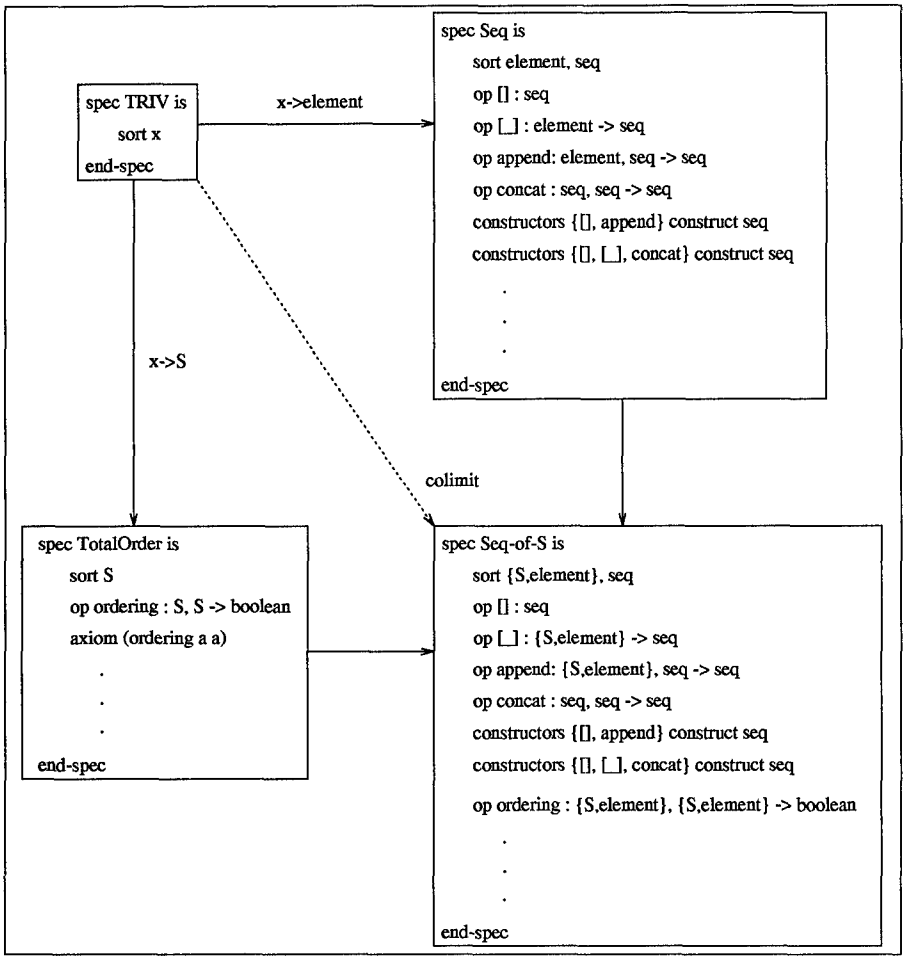


Figure 3.10 Definition of seq(S)

created using the colimit operation; specification construction using colimits is the topic of the next subsection.

Importation is strictly "syntactic sugar" in that importations can be defined using colimits and basic specifications.

3.3.4 Colimit. Informally, a colimit is "a shared union" or a "gluing of specifications along a common boundary." The colimit operation creates a new specification, the colimit specification, and a collection of morphisms referred to as the *cocone morphisms* from each of the source specifications into the colimit specification. As described in (19),

The colimit specification and the associated cocone morphisms are constructed using the standard union-find algorithm for computing the connected components of a graph. The disjoint union of the sorts and operations contained in the specifications attached to all nodes in the diagram is partitioned into equivalence classes according to the mappings given by the morphisms labeling the arcs in the diagram.

To be precise, let the disjoint union U of all signatures in a diagram D be

$$U = \{\langle n, x\rangle \mid n \in \text{nodes}(D) \wedge n : S \wedge (x \in \text{sorts}(S) \vee x \in \text{operations}(S))\},$$

where S is the specification labeling node n.

Define an equivalence relation $\equiv$ on the set U by

$$\langle n_1, x\rangle \equiv \langle n_2, y\rangle \Longleftrightarrow (\exists a) a \in \text{arcs}(D) \wedge a : m \wedge m(x) = y$$

where m is a morphism labeling the arc a.

This equivalence relation partitions the disjoint union U into an equivalence class of sorts or operations (since morphisms map sorts to sorts and operations to operations, each equivalence class will contain only one kind of object). The colimit specification contains one sort or operation corresponding to each equivalence class. The cocone morphism from the specification labeling each node in the diagram is obtained as the map which takes each sort or operation containing to the equivalence class containing it.

Another example of a colimit is depicted in Figure 3.11. The figure depicts a construction of a specification for sorting a bag of natural numbers using the ordering relation *geq*. The common core, the specification *TotalOrder* defines a total order relation *ordering* over a sort *S*. This specification is extended in *SortSpec* to define the problem of sorting a bag of elements. *SortSpec* adds three operations, an input condition *incond*, an output condition *outcond*, and an operation *ordered*. The operation *ordered* is defined to return *true* if the sequence is ordered according to the ordering relation. The sort *S* in *SortSpec* is used to define two other sorts, a sequence whose elements are of sort *S* and a bag whose elements are of sort *S*. A specification morphism from *TotalOrder* to *NaturalNumbers* associates the operation *geq* of *NaturalNumbers* with the operation *ordering* of *TotalOrder*; this morphism defines how the specification *TotalOrder* is contained within *NaturalNumbers*.

The colimit of the diagram defined by the three specifications *TotalOrder*, *SortSpec* and *NaturalNumbers* and the specification morphisms from *TotalOrder* to *SortSpec* and from *TotalOrder* to *NaturalNumbers* results in the specification *SortSeqofNaturals* as shown in the figure. Due to space limitations, only the signature of the colimit specification is

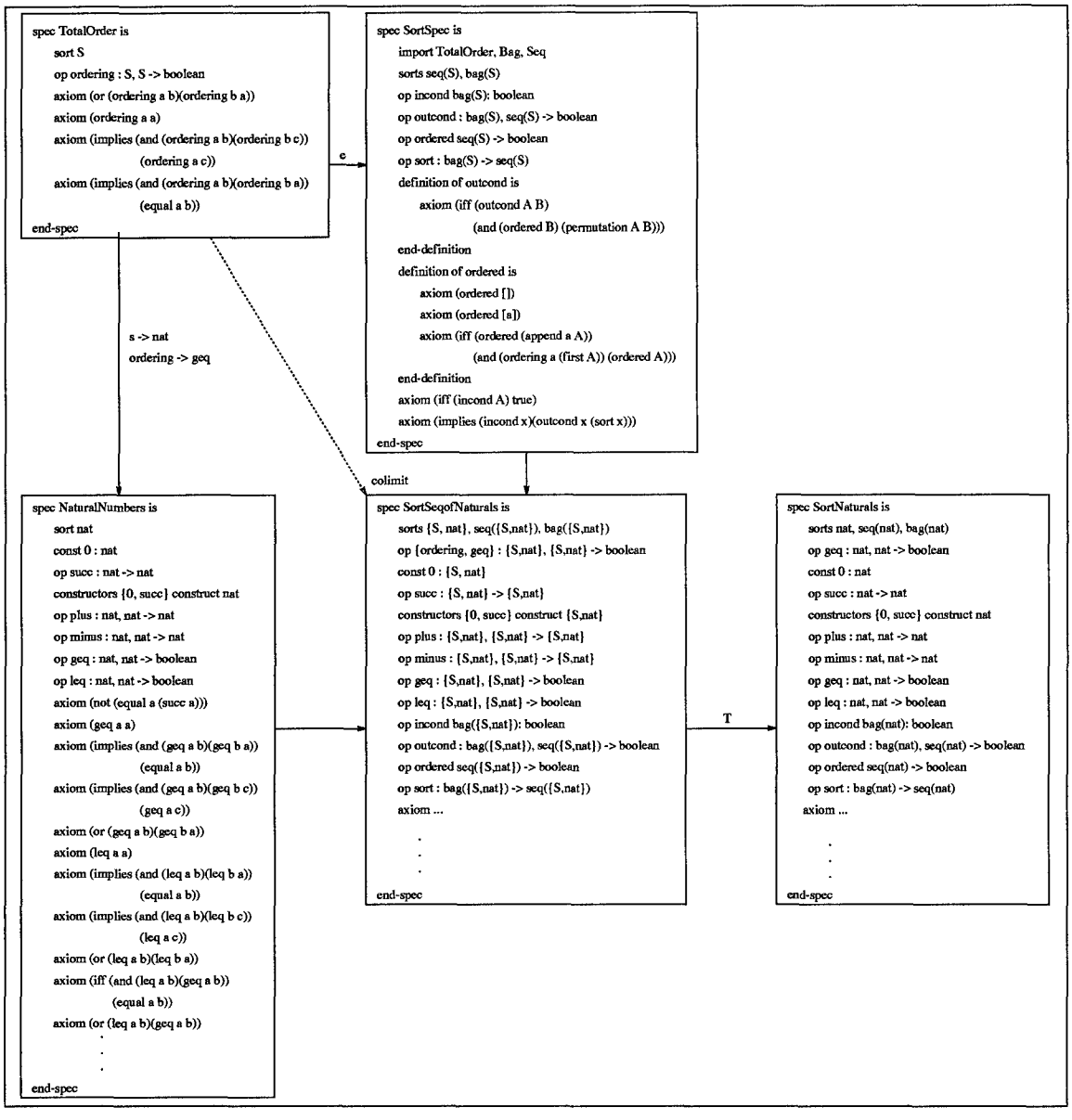


Figure 3.11 Specification Building using the Colimit Operation

shown. Note that the colimit specification contains equivalence classes of symbols. Specifically, the sort symbols *nat* and *S* are equivalent in the colimit object, as are the operation symbols *ordering* and *geq*. A translation from the colimit specification *SortSeqofNaturals* to *SortNaturals* defined by the map $\{\{S,\ nat\} \mapsto nat,\ \{ordering,\ geq\} \mapsto geq\}$ is used to remove these equivalence classes. The specification *SortNaturals* defines the problem of sorting a bag of natural numbers using the relation *geq*. Note that an alternate ordering

in *SortSeqofNaturals* could have been defined by associating the relation *ordering* of *TotalOrder* with the relation *leq* of *NaturalNumbers*. The resulting colimit specification would then contain a definition of the problem of sorting a bag of naturals using the relation *leq*.

The diagram of Figure 3.11 exemplifies a form of parameterization. Specifically, the specification *TotalOrder* is a formal parameter to the specification *SortSpec*. The specification *NaturalNumbers*, which contains a total order, is the actual parameter. The colimit specification is an instantiation of the parameterized specification for the actual parameter.

3.3.5 Specification Interpretation. Specification interpretations define how one abstract entity can be defined using the features provided by others. For example, a specification for sets, *Set*, can be defined using a specification for heaps or arrays. The operations defined in *Set*, such as union or intersection, could be defined using expressions in the specification for heaps. Interpretations generalize the notions of specification morphism:(20:30)

> A morphism from A to B specifies an "embedding" of the [specification] A into the [specification] B; an interpretation from A to B specifies an embedding of A into a *definitional extension* of B, i.e., a specification consisting of B and definitions of further sorts and operations. Both morphisms and interpretations are closed under sequential composition. [T]his allows us to follow one refinement with another.

Interpretations are formally defined below.

Definition III.18 Interpretation. *Given two specifications A and B, an* interpretation *from A to B consists of a pair of arrows, s and d, and a specification,* A-as-B,

$$A \xrightarrow{s} A\text{-}as\text{-}B \xleftarrow{d} B$$

where

1. *s is a specification morphism from A to* A-as-B *which maps the sorts and operators of A to sort expressions and operator expressions in* A-as-B.

2. *d is a specification morphism from B to* A-as-B *such that d extends B with additional sorts and operators where such sorts and operators are defined entirely in terms of the sorts and operators in* B.

3. *The specification* A-as-B *is called a* mediator. □

For example, a interpretation from sets to bags can be defined as follows. Let *Set* denote a specification for sets, and let *Bag* denote a specification for bags. Define operation *no-dupes : bag* $\rightarrow$ *boolean* such that *no-dupes(b)* is true if and only if *b* contains no duplicate elements. Then the operation *empty-set :* $\rightarrow$ *set* of *Set* can be defined using operations from *Bag*. Specifically, the operation *empty-set* can be defined using the operation *empty-bag*: (20)

```
spec Set-as-Bag is
      import Bag
      sort set-as-bag
      sort-axiom set-as-bag = bag | no-dupe
      op no-dupe : bag → boolean
      ⋮
      op empty-set : → set-as-bag
      ⋮
      definition of empty-set is
            axiom (equal
                  empty-bag
                  ((relax no-dupe) empty-set)
      end-definition
      ⋮
end-spec
```

The expression *((relax no-dupe) S)* converts *S* from sort *set-as-bag* to sort *bag*. The single definition shown defines the operation *empty-set* . In this case, under the interpretation *Set* $\rightarrow$ *Set-as-Bag* $\hookleftarrow$ *Bag*, the operation *empty-set* of *Set* is defined using the operation *empty-bag* of *Bag*, where the sort *set* of *Set* is mapped to the sort *set-as-bag* in *Set-as-Bag*. For more information concerning specification interpretations, see (20) or (21).

3.3.6 Summary of Specification Building Operations. The five specification building operations described in this section, *translation*, *importation*, *basic specification*, *colimit*, and *interpretation*, can be used to construct specifications defining functional entities, such

as abstract types. Additionally, it was shown how these construction operations support the notion of parameterized specifications. The following section describes the relationship between the specification construction operations defined in this section and the development formalism described in Chapter II.

3.4 Institution-Based Specification Development

The software development framework depicted in Figure 2.1 shows two primary activities: specification building using the Composition Mechanism (CM) and specification implementation using the Design Refinement Mechanism (DRM). Both of these activities can be supported within an *institution* framework.

Goguen and Burstall developed the theory of institutions after noting that "much of programming methodology is actually *completely independent* of what underlying logic is chosen." An institution is a "precise generalization of a 'logical system.' " An institution based on first order predicate calculus can be defined, as can an institution based on equational logic and an institution based on Horn logic.(41) Each of these institutions can be used to develop specifications in their respective logics. As noted in (107), "an institution is an abstract logical system for specifying algebras." This logical system consists of two parts:

1. syntax, which is defined in terms of signatures and the sets sentences that can be generated over a given signature; and
2. semantics, which is specified in terms of models and a satisfaction relation between models and sentences.

A more formal definition of an institution (taken from (41) as adapted by (107)) is provided below. However, before institutions can be defined, *functors* must be defined.

Definition III.19 Functor.*(66:501) If* $\mathbf{X}$ *and* $\mathbf{X}'$ *are two categories, a functor* $\mathcal{F}$:$\mathbf{X} \rightarrow \mathbf{X}'$ *is a pair of functions, an object function* $\mathcal{F}_O$ *and a mapping function* $\mathcal{F}_M$. *The object function assigns to each object* X *of the first category* $\mathbf{X}$ *an object* $\mathcal{F}(X)$ *of* $\mathbf{X}'$*; the mapping function assigns to each arrow* $f : X \rightarrow Y$ *of the first category an arrow* $\mathcal{F}_M(f) : \mathcal{F}_O(X) \rightarrow \mathcal{F}_O(Y)$ *to the second category* $\mathbf{X}'$. *These functions must satisfy two requirements:*

1. $\mathcal{F}_M(1_X) = 1_{\mathcal{F}_O(X)}$ *for each identity* 1_X *of* $\mathbf{X}$;

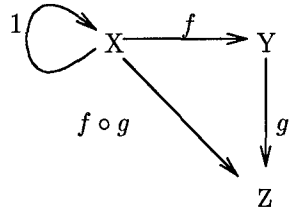
2. $\mathcal{F}_M(g \circ f) = \mathcal{F}_M(g) \circ \mathcal{F}_M(f)$ *for each composite* $g \circ f$ *defined in* $\mathbf{X}$. □

Functors are morphisms between categories.

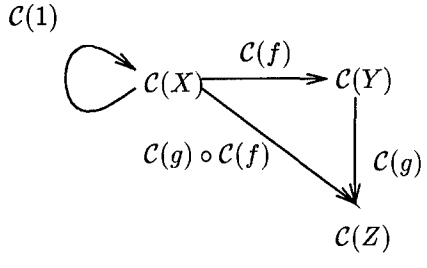
Functors as defined in Definition III.19 are called *covariant* functors. A functor is contravariant when it reverses arrows,(66:147) or more formally:

Definition III.20 Contravariant Functor. *A contravariant functor* $\mathcal{C}$ *on a category* $\mathbf{X}$ *to a category* $\mathbf{X}'$ *is a pair of functions which assign to each object* X *of* $\mathbf{X}$ *an object* $\mathcal{C}(X)$ *in* $\mathbf{X}'$ *and to each arrow* $f : X \to Y$ *in* $\mathbf{X}$ *a morphism* $\mathcal{C}(f) : \mathcal{C}(Y) \to \mathcal{C}(X)$ *in* $\mathbf{X}'$, *assigning to each identity arrow* 1_X *the identity of* $\mathcal{C}(X)$ *and to each composite* $g \circ f$ *of arrows of* $\mathbf{X}$ *the composite* $\mathcal{C}(g \circ f) = \mathcal{C}(f) \circ \mathcal{C}(g)$.*(66:504)* □

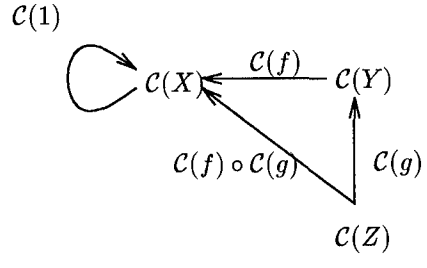
Figure 3.12 highlights the distinction. between covariant and contravariant functors.



Category $\mathbf{X}$ with objects X, Y, and Z



Covariant Functor $\mathcal{C} : \mathbf{X} \to \mathbf{X}'$



Contravariant Functor $\mathcal{C} : \mathbf{X} \to \mathbf{X}'$

Figure 3.12 Covariant and Contravariant Functors.

Now that functors have been defined, institutions may be defined. Institutions include several categories and functors between them. Figure 3.13 was developed to aid the

reader in conceptualizing institutions. Specifically, Figure 3.13 depicts the relationships of the following definition.

Definition III.21 Institution. *An institution I is a 4-tuple* $\langle \mathbf{Sig}_I, \mathbf{Sen}_I, \mathbf{Mod}_I, \models \rangle$ *consisting of*

1. *a category* $\mathbf{Sig}_I$ *of signatures and signature morphisms,*
2. *a functor* $\mathbf{Sen}_I : Sig_I \rightarrow Set$ *(where* **Set** *is the category of sets and functions over sets) which assigns to each signature* Σ *the set of* Σ*-sentences, and to each signature morphism* $\sigma : \Sigma \rightarrow \Sigma'$*, the functor which translates* Σ*-sentences to* Σ'*-sentences,*
3. *a functor* $\mathbf{Mod}_I : Sig_I \rightarrow Cat^{op}$ *(where* $\mathbf{Cat}^{op}$ *is the category of all categories*[1] *and functors between them) which assigns to each signature* Σ *the category of* Σ*-models, and to each signature morphism* $\sigma : \Sigma \rightarrow \Sigma'$*, the functor which translates* Σ'*-models into* Σ *models (note the change in direction), and*
4. *a satisfaction relation* $\models_{I,\Sigma} \subseteq |\mathbf{Mod}_I(\Sigma)| \times \mathbf{Sen}_I(\Sigma)$ *between models and sentences for each signature* Σ

subject to the condition that satisfaction be preserved under change of signature:

Satisfaction Condition. *For any signature morphism* $\sigma : \Sigma \rightarrow \Sigma'$ *in* $\mathbf{Sig}_I$*, for any* Σ*-sentence* $\varphi \in \mathbf{Sen}_I(\Sigma)$*, and for any* Σ'*-model* $M' \in |\mathbf{Mod}_I(\Sigma')|$*,*

$$M' \models_{I,\Sigma'} \mathbf{Sen_I}(\sigma)(\varphi) \quad \Leftrightarrow \quad \mathbf{Mod}_I(\sigma)(M') \models_{I,\Sigma} \varphi$$

Notation. The function $\mathbf{Sen}(\sigma)$*:* $\mathbf{Sen}(\Sigma) \rightarrow \mathbf{Sen}(\Sigma')$ *will be denoted (ambiguously) by* σ*. Objects in the category of models* $\mathbf{Mod}(\Sigma)$ *will be called* Σ*-models and morphisms* Σ*-homomorphisms. The functor* $\mathbf{Mod}(\sigma)$*:* $\mathbf{Mod}(\Sigma') \rightarrow \mathbf{Mod}(\Sigma)$ *will be called the* σ*-reduct functor and denoted by* $-|_\sigma$*. The subscripts for the satisfaction relation will usually be dropped, resulting in the simplified satisfaction relation*

$$M' \models \sigma(\varphi) \quad \Leftrightarrow \quad M' |_\sigma \models \varphi \qquad \square$$

[1] This leads to foundational difficulties similar to Russell's paradox.(65) These difficulties can be avoided if we consider only those categories that are small with respect to some universe.(107)

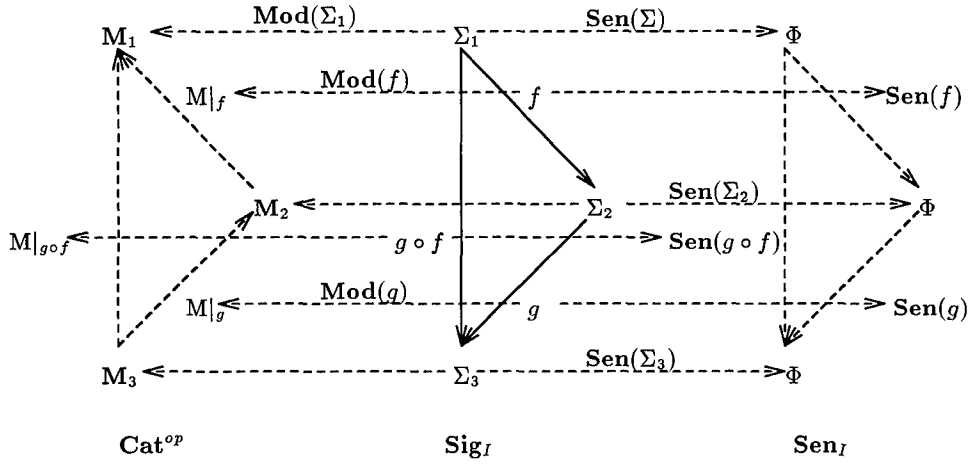


Figure 3.13 An institution.

The specification construction operations defined in the previous section can be defined within an institution framework. Specifically, construction of functional specifications can take place within the categories **Sign** and **Spec**, where signature manipulation such as adding an operation or refining a sort name take place within **Sign** and manipulation of specification axioms takes place within **Spec**. Specifications in **Spec** are defined to include a signature Σ and a collection Φ of Σ-sentences. Given a specification $S = \langle \Sigma, \Phi \rangle$, Σ is a signature in the category **Sign**, while Φ is a collection of sentences from the category $\mathbf{Sen}_I$. That is, **Sign** and **Sen** are categories within an institution of higher order, multi-sorted predicate calculus, denoted *HOPC*. $\mathbf{Sig}_{HOPC}$ is the category of signatures and signature morphisms (Definition III.2). Sentences in this institution are well formed formulas (WFFs). That is, the functor $\mathbf{Sen}_{HOPC}$ assigns to each signature Σ the set of Σ-WFFs, and to each signature morphism σ a translation function σ' which translates Σ-sentences to Σ'-sentences. The combination of a signature Σ and a collection of Σ-sentences from $\mathbf{Sen}_I$ defines a specification. Further, the functor $\mathbf{Mod}_{HOPC}$ assigns to each signature Σ the category of Σ-algebras (Definition III.12), and to each signature morphism σ the σ-reduct functor $\_|_\sigma$ (Definition III.14). These functors are implicitly defined in the SpecWare specification development system. Other logics, such as a logic based on trace semantics, could possibly be used to define an institution supporting a category of process signatures and process specifications. A more detailed treatment of this material may be found in (41).

As the above paragraph indicates, the Composition Mechanism (CM) is used to define specifications within an institution framework. The Design Refinement Mechanism (DRM) is used to find implementations of specifications generated using the CM. However, when (or if) a developer decides to implement the sorts and operations defined in a specification by mapping the specification to a language that can be compiled and executed by a computer, a "semantic ditch" is encountered. Specification languages such as SLANG or Larch can be defined within an institution framework. Similarly, languages such as LISP or C can be defined within (imperative) institutions. To map SLANG to LISP for example, requires that an institution morphism be defined, where an institution morphism must define the following:

1. Translation of signatures. For example, mapping SLANG sort symbols such as *seq* to sort symbols or sort expressions, such as *lists* in LISP. Note that these mappings might not be bijective. The sort `map` in SLANG could be translated, for example, to either association lists or sequences of tuples in LISP.
2. Translation of sentences.
3. Translation of the satisfaction relation.

The first two are syntactic manipulations, where the second translation is obtained as a byproduct of the first. The third item is the hard one, especially if the target institution lacks some necessary structure. The translation must be defined such that the axioms of specifications developed in the first institution are translated into theorems in the second institution. The difficulty lies in the fact that the logic of the second institution may be quite different. For example, translating the satisfaction relation from a first-order predicate calculus institution to an institution whose algebras are state-based may be difficult, but may be possible. However, translating a state-based institution to an institution based on a stateless logic may not in general be possible because the latter lacks necessary structures for representing and reasoning about state. Some work has been done in this area. For example, (50) are researching issues associated with translating a subset of Ada into Milner's CCS.

I have been unable to find any work defining an institution for state-based or process algebras, but Goguen and Burstall (41) conjecture one could be defined, and there are some authors such as (35) who are looking into the problem. For these reasons, mapping expressions in one institution to expressions in another institution is typically done by defining only the first two of the above three transformations, while the translation of the satisfaction relation is simply assumed.(109) That is, once the signature and sentence translations are defined, they are assumed to produce a correct (consistent) representation in the target algebra such that the properties of the sorts and operations are preserved under the transformation. Because specification implementation is not a primary focus of this investigation, the issues associated with translating a specification to an implementation in a compilable target language are not addressed further.

Institutions and the mathematics on which they are founded can seem overwhelming. However, developing specifications within an institution provides a number of benefits, some of which are listed below. The relative complexity of the underlying mathematics of institution-based specification development systems can be hidden from the user of such a system in much the same way that the theoretical foundations of compilers can be hidden from program developers. Some of the benefits of using institutions are:

- they allow a developer to develop specifications in various logics, such as order-sorted logic or equational logic; and
- A user can develop specifications for sorts and operations using a institution whose logic is stateless, such as one based on predicate calculus, while using an institution supporting process based specifications to define the flow of data within an application, thus allowing the developer to preserve the modularity of specifications.

3.5 Summary

This chapter has provided a number of definitions required to support the software development framework depicted in Figure 2.1, including products, coproducts, pushouts, and colimits. Category theory was introduced, and specifications and specification building

operations based on category theory were also introduced and defined, and it was shown how these construction operations support parameterization.

Institutions were introduced and related to the formalism shown in Figure 2.1. Specifically, the category **Sign** of signatures and the category **Spec** of specifications were related to the concept of institutions, and the problems associated with translating specifications written in a language of one institution to a language of another institution were briefly explored.

Although functional specifications can be used to define many useful problems, the category **Spec** of functional specifications and functional specification morphisms lacks the necessary structures for representing and reasoning about processes. In the following chapter, two approaches for defining process-based specifications are explored.

IV. Combining Functional and Process Specifications

4.1 Introduction

The purpose of this chapter is to document the results of two experiments aimed at incorporating architectural information within system specifications. Two techniques were tried:

1. The architecture of an application was defined by incorporating the functional model (in the object oriented sense) of the application within functional specifications.
2. The architecture of an application was defined through expressions written in Hoare's algebra of Communicating Sequential Processes (CSP).

Note that CSP is not the only approach that could be used for defining processes and state. Other approaches such as modal logic or Milner's CCS could have been used for this purpose.

The experimental approach was simple: For each technique listed above, select a simple problem and attempt to develop a system specification for it using the selected technique and observe any weaknesses in the technique. These observations would then be used to either validate the selected technique, or they would be used to define a new method of incorporating or defining application structure. The following sections describe the results of these two experiments.

A pipeline-based application of sorting and searching a sequence was selected for development using the first technique, where structural information (functional model information) is defined within the functional specification. Both this approach to defining the structure of an application and its utility are described in Section 4.2. The second system level specification developed defines a moving average unit. The specification for this second problem was developed using a combination of functional specifications and Hoare's Communicating Sequential Processes (CSP). The results of this approach are described in Section 4.3. Section 4.4 contains a summary of the findings of these two experiments.

```
function KEY-SEARCH
      (A:seq(integer), key1:integer | lt-ordered(A) )
      returns (index:integer |
                              index in [1..size(A)]
                        &     A(index) = key1 )

function ALL-KEYS-SEARCH
      (A:seq(integer), key1:integer | le-ordered(A) )
      returns (keys : set(integer) |
                  keys = { index |
                              (index:integer) index in [1..size(A)]
                        &     A(index) = key1 })
```

Figure 4.1 Specifications for Searching an Ordered Sequence

```
function SORT1 (x:seq(integer) | true)
      returns (z : seq(integer) |
                  bag-equal(elements-of(x), elements-of(z))
            &     le-ordered(z) )
```

Figure 4.2 Specification for Sorting a Sequence of Integers

4.2 Development of a Specification for a Pipeline Application

4.2.1 Introduction. The purpose of this experiment was to determine the utility of defining architecture through functional specification. This experiment was conducted by selecting a simple problem, developing functional specifications for it including the definition of algorithm theory implementations, and observing any shortcomings of the approach. The hypothesis of this experiment was that the structure of an application can be defined using functional specifications. The problem selected for development was a pipeline-based problem of sorting and then searching a sequence of integers.

4.2.2 Problem Description. Given an unordered sequence of integers and an element that appears in the sequence, report a location the element would have if the sequence were ordered by the relation $\leq$. For example, given the sequence $(3, 7, 4, 5, 4, 6, 1, 2)$

and the element 4, the operation will return either 4 or 5. This problem was selected for two reasons:

1. Its pipeline-based structure is well suited to functional specification in that both subproblems — sorting and searching — contain simple interfaces and neither subproblem requires the use of state information.
2. Algorithm theories existed in the Kestrel Interactive Development System (KIDS) algorithm theory library that could be specialized for sorting and searching. Specifically, a global search algorithm theory could be specialized for searching the sequence once it has been sorted, and a divide and conquer algorithm theory could be specialized for the problem of sorting the input sequence.

In addition, KIDS is distributed with several domain theories and problem specifications. One of these domain theories, *Ordered-Search*, contains three problem specifications: *Key-Search, All-Keys-Search,* and *Key-Search-N.* The specifications *Key-Search* and *All-Keys-Search*, shown in Figure 4.1, formally define the problem of searching an ordered sequence A of elements for occurrences of the element *key1.* (Note that the syntax used in Figure 4.1 differs somewhat from the syntax of SLANG. The specifications shown in Figure 4.1 are not written in SLANG, but are instead written in the specification language ReGroup used by KIDS.) All three of these problem theories include a predicate restricting the input sequence to one that is ordered. The predicates *lt-ordered(A)* and *le-ordered(A)* return *true* if the sequence A is ordered by the binary relation $<$ or $\leq$, respectively.

In addition, KIDS includes a domain theory, *Sorting-Theory*, which defines the problem of ordering a sequence of integers using the relation $\leq$. The problem specification *Sort1* from this domain theory is shown in Figure 4.2. The predicate *bag-equal (elements-of (x), elements-of (z))* is used to ensure that the solution returned by *Sort1* includes all of the elements of the original sequence. If *Sort1* lacked this predicate in its output condition, then any ordered sequence, including the empty sequence, would be a valid solution for any input.

4.2.3 Development of the Sort-Search Specification. *Sort1* formally defines the problem of sorting a sequence of integers and *Ordered-Search* contains problem specifica-

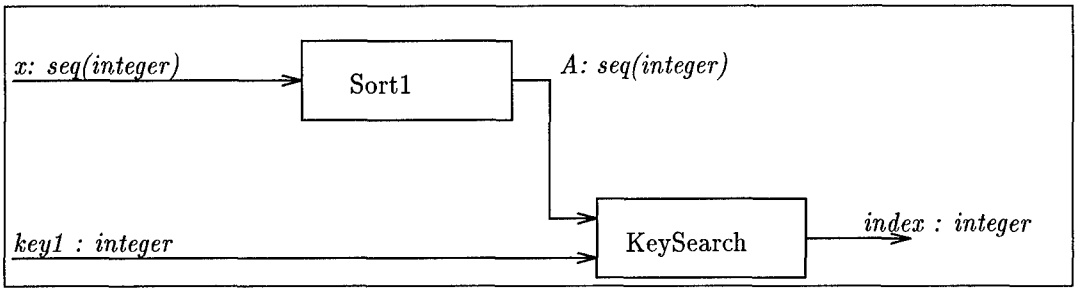


Figure 4.3 Block diagram of the sort-search problem

tions that formally define the problem of searching ordered sequences for occurrences of a given element. Therefore a specification for the problem of sorting and then searching a sequence can be defined by composing *Sort1* with one of the *Ordered-Search* problem theories as shown in Figure 4.3 and defined below.

function FIND-LOCATION (A : *seq(integer)*, key1 : *integer* | *true*)
returns (index : *integer* |
index *in* [1 .. *size(A)*]
& ex(z :*seq(integer)*)
(le-ordered(z)
&bag-equal(elements-of(z), elements-of(A))
&z(index) = key1))

Unfortunately, such a problem specification cannot be used directly by KIDS. This single problem specification includes aspects of both divide and conquer and global search algorithm theories in the sorting and searching of the sequence, respectively. Although the design tactics of KIDS cannot be directly applied to this problem, KIDS can be used to guide the development of a set of problem specifications to which the design tactics can be applied. Specifically, the divide and conquer portion of the above specification can be separated from the global search portion of the specification to obtain a function of the form *Key-Search (Sort1 (A), key1).*

The output condition of *Sort1* is contained within the output condition of *Find-Location.* Specifically, the conjunct *bag-equal (elements-of (z), elements-of (A)) & le-ordered (z)* in the existentially quantified clause of *Find-Location* is equivalent to the output condition of *Sort1.* This conjunct could therefore be replaced with an invocation of *Sort1*

```
function FIND-LOCATION (A : seq(integer), key1 : integer | le-ordered(SORT1(A)) )
    returns (index : integer |
                index in [1 .. size(A)]
            &   SORT1(A)(index) = key1
            &   le-ordered(SORT1(A)) )
```

Figure 4.4 Problem Specification for Find-Location

to obtain $\exists(z : seq(integer))\ (z = SORT1(A)\ \&\ z(index) = key1))$. After substitution and simplification, *Find-Location* can be rewritten as follows:

```
function FIND-LOCATION (A : seq(integer), key1 : integer | true)
    returns (index : integer |
                index in [1 .. size(A)]
            &   SORT1(A)(index) = key1 )
```

Although *Sort1(A)* returns an le-ordered sequence (i.e., *le-ordered(Sort1(A))* is true), this fact will not be discovered by the inference mechanism in KIDS. However, this fact can be made explicitly known. The statement *le-ordered(Sort1(A))* is invariantly true; that is, it is true both as an input and as an output condition of the problem specification *Find-Location.* Adding this invariant to the above specification yields the specification shown in Figure 4.4. This specification is of the form *Find-Location(Sort1(A), key1).*

The specification for *Find-Location* shown in Figure 4.4 separates the global search nature of the problem from the divide and conquer. The derivation of the divide and conquer portion of the problem, as defined by *Sort1*, follows the derivation found in the KIDS manual. However, the derivation of the global search portion of the problem differs from that of *Key-Search* despite the close resemblance between the problem specifications. These differences have a slight impact on the specialization of an algorithm for the searching phase of the problem. Specialization of a global search algorithm theory for the search portion of *Find-Location* is described in Appendix B.

In the specialized algorithm, no provisions were made for *Sort1* to run in parallel with *Find-Location.* Instead, the filter *Sort1* runs to completion before passing its results to the filter *Find-Location.* The static nature of the communication structure of this application

allowed the direct incorporation of an invocation of *Sort1* in the specification *Find-Location.* Alternatively, elements of the sequence could be compared against *key1* as the sequence is being sorted. If the input sequence is denoted A and if S_O and S_U denote an ordered and an unordered subsequence of A such that *bag-equal(elements-of(A), elements-of*(S_O ++ S_U)) is true, then this concurrent sort and search could be terminated when *key1* $\in$ S_O and $\forall x(x \in S_U \Rightarrow key1 \leq x)$. However, the KIDS algorithm specialization process cannot support this type of incremental sort and search.

4.2.4 Observations. Specifications for first sorting and then searching a sequence of integers for occurrences of a given key were successfully developed, and algorithm specifications were successfully specialized for each of the two subproblems. (See Appendix B for details.) Thus, development of functional specifications for this simple problem provides some support that functional specifications can be successfully used to define the architecture of an application. Although incorporating architectural information within individual problem specifications may adversely impact their re-usability, architectural information could be isolated in a separate "structuring specification." For example, a specification of the form shown in Figure 4.5 could be used to compose *Sort1* with *Key-Search.* The single axiom of *Structure*, which is written using the syntax of SLANG, defines the operation h to be a composite of the operations u and v. Note that h has the same structure, $h(a, b) = v(u(a), b)$, as the block diagram of *Find-Location* in Figure 4.3.

The specification *Structure* can be used to compose *Search* with *Sort1* to define an operation having the structure *search(sort1 (A), key)* as follows. First, the "shared union" (or colimit) of the specifications *Sort1* and *Key-Search* is formed. After forming the union of these two specifications, a morphism from the specification *Structure* to the union specification is defined such that u is associated with the operation *sort1* and v is associated with *search.* The colimit of the two specifications and morphisms contains an operation h defined by the equation *key-search ((sort1 A) key).* The colimit diagram is shown in Figure 4.6. The specification *Common-Sorts* contains those sorts that are common between the specifications *Sort1* and *Key-Search.* The union (or colimit) specification, *Sort1+Key-Search*, is extended with an additional operation, *find-location : seq(integer),*

```
spec Structure is
        sorts D, E1, E2, R
        op h : D, E2 → R
        op u : D → E1
        op v : E1, E2 → R
        axiom (fa a (fa b (equal (h a b) (v (u a) b))))
end-spec
```

Figure 4.5 Structuring specification

integer → integer, to form the specification *Extended-Sort1+Key-Search*. The structuring specification is then used to define the operation *find-location* in terms of the operations *sort1* and *key-search*. The colimit of the diagram contained within the dashed box, the specification *Sort-Search*, contains the operation *find-location* in which *find-location (A, key)* equals *key-search (sort1(A), key)*.

The benefit of this modified approach is that architectural information is isolated in the structuring specification, while each problem specification and its associated domain theories remain relatively free of structural information.

One of the problems of using only functional specifications is that they contain no provision for representing or reasoning about state. Hoare's Communicating Sequential Processes (CSP) contains structures useful for representing state, and it can be used to define relatively complex process interfaces. While CSP is good at defining process interfaces, it lacks the ability to specify the behavior of functional operations such as *Sort1*. However, the sorts and operations referenced in CSP process expressions can be defined using functional specifications. The following section describes one attempt at merging the expressive power of CSP with the power of functional specifications to define application specifications.

4.3 Development of the Four Sum Moving Average Unit Specification

4.3.1 Introduction. This section documents the results of an experiment in which functional specifications are combined with expressions in Hoare's Communicating Sequen-

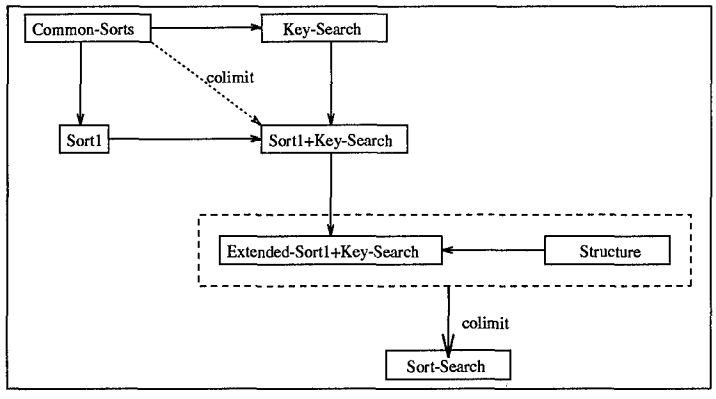


Figure 4.6 Defining structure using a structuring specification

tial Processes (CSP) to define application specifications. The hypothesis of this experiment was that the structure of an application could be defined using CSP while functional specifications could be used to define the sorts and operations of the application. Specifically, sorts and operations of an application are specified using functional specifications, while state and application processes are specified using CSP. Functional specifications are written in SLANG.

4.3.2 Problem Description. The problem selected for development is a moving average unit. Moving average units dampen an input digital signal by averaging an input data value with the previous n inputs. As this brief problem description implies, moving average units require state information. The input, denoted X, consists of a digitized signal represented as a finite sequence of complex numbers. The output, denoted Z, is also a sequence of complex numbers. The relationship between the input X and the output Z for a four-sum unit is shown in Table 4.1. A block diagram for this problem is shown in Figure 4.7. Filters are represented by boxed entities, and arrows denote sorted or typed communication between the filters. The output of the summation unit is a tuple of a complex number representing the sum, and a natural number representing the number of elements used to form the sum. The averaging unit accepts this tuple and forms the value defined by the division of the complex value by the natural value.

This problem was selected in part because of its simplicity, but mainly because of its use of state data. The summation unit accepts individual signal values which it then sums

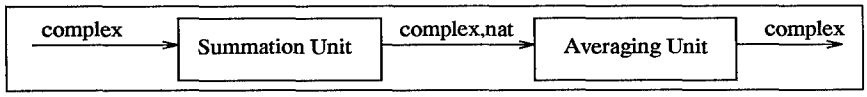


Figure 4.7 Block diagram for the moving average problem

Table 4.1 Output of the Four Sum Moving Average

Output Element	Value
z_1	$\frac{x_1}{1}$
z_2	$\frac{x_1+x_2}{2}$
z_3	$\frac{x_1+x_2+x_3}{3}$
z_4	$\frac{x_1+x_2+x_3+x_4}{4}$
z_5	$\frac{x_2+x_3+x_4+x_5}{4}$
$\vdots$	$\vdots$
z_n	$\frac{x_{n-3}+x_{n-2}+x_{n-1}+x_n}{4}$

with the three previous signal values. The three previous signal values must be maintained. That is, the summation unit is imperative.

For this experiment, the existence of a specification for real numbers is assumed, as is the existence of a specification for complex numbers. For notational purposes, a *component* is a specification containing a combination of a functional specification and a CSP expression written over the sorts and operations defined in the functional specification. A *connector* refers to a specification containing only a CSP expression. A more detailed, yet still informal, description of components and connectors may be found in Appendix C. A formal treatment of components and connectors is contained in Section 5.6.

4.3.3 Development of a Specification for the Moving Average Problem. Specification of the multiplication and divisor units proceeded along the lines of components and connectors. Components were represented as an extension to *Problem-Theory*. As shown in Figure 4.8, a specification incorporating *Problem-Theory* as well as communication or interface structures was defined. This specification, *Communicating Problem Theory*, is defined using a specification *Communicating-Entity*. *Communicating-Entity* introduces two new process-based operations, *rcv* for receiving data and *trx* for transmitting data; four sorts, *P, E, D* and *R*; and includes structures for process names and process axioms. The

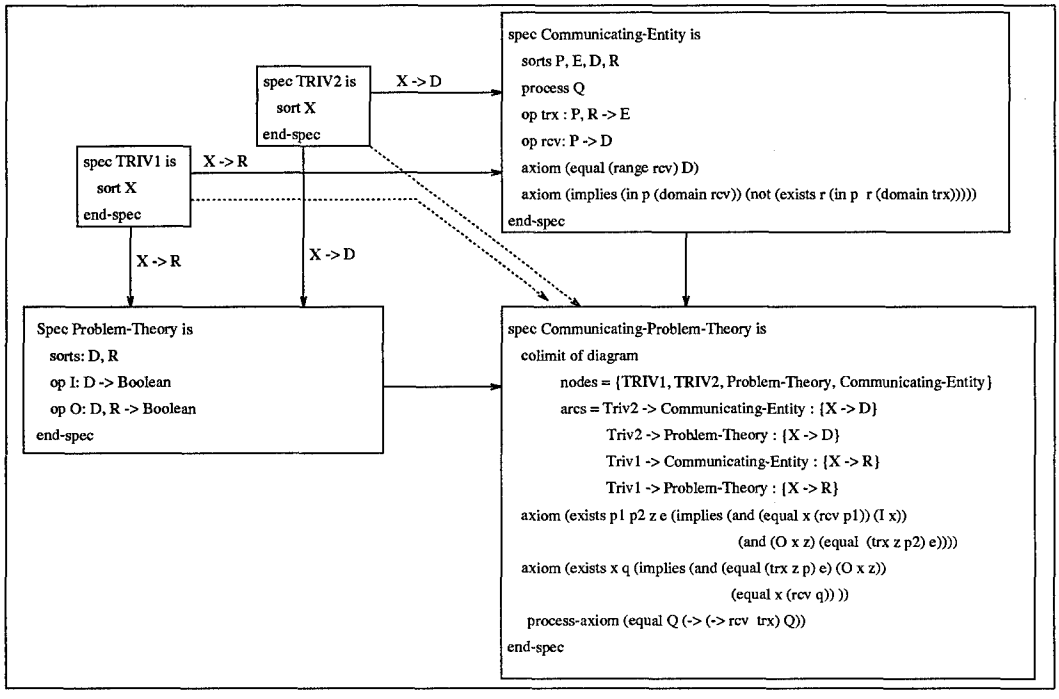


Figure 4.8 Communicating Entity Specification

intent here being that *rcv* and *trx* could be defined according the the needs of the application, where axioms written over *trx* and *rcv* would define the interface or protocol of the component. For example, *rcv* could be specialized to be an asynchronous, unbuffered receiver operation of a connector or a simple receiver operation of a component. Axioms defined over *rcv* or *trx* would follow one of the patterns described in Appendix C.

Although *rcv* and *trx* are defined to be operations, they may be better viewed as communication processes. Specifically, the statement *x:=rcv(c)* can be defined by the CSP construct *c?x*. Note that *rcv* can also perform other communication activities, such as responding with an acknowledgment. In this case, the statement *x:=rcv(c)* could be defined by the CSP statement $c.left?x \rightarrow c.right!acknowledge$. The operation *trx* has as its purpose the communication of a result over a port followed by the receipt of an acknowledgment of sort E over another port dedicated for this purpose. In this case the statement *ack := trx(r,c)* could be defined by the CSP statement $c.left!r \rightarrow c.right?acknowledge$ where *acknowledge* is an event symbol in E, and *ack* is a state variable used to hold the returned event symbol.

The specification *Communicating-Problem-Theory* includes the sorts, P and E, where P is a port sort and E is an event sort. The semantics of the statement $p \in P$ in a specification derived from *Communicating-Problem-Theory* is that p is a port. Note that P is not a true sort in the sense that if $p \in P$ then the sort of p is not P, but is instead the sort of the data communicated over p. Therefore the proper semantic interpretation of P in *Communicating-Problem-Theory* is that P denotes a set of sorted port symbols where the sort of these port symbols is the sort of the data communicated over them. Similarly, the sort symbol E denotes a set of event symbols such as *acknowledge* such that E is the meet of the partial order induced by the sub-sort relations of the functional specification.

The axioms of *Communicating-Entity* serve the following purposes. The first axiom states that the *rcv* operation be capable of receiving any value of type D; and The second axiom states that ports are unidirectional. Three additional axioms are defined in *Communicating-Problem-Theory*:

1. The first axiom states that if a data value x is received over a port P_1 and that data value satisfies the input assumptions of the problem theory, then a value z satisfying $O(x, z)$ will be communicated over a second port P_2.
2. The second axiom of *Communicating-Problem-Theory* states that if a data value is successfully transmitted over a port of a communicating problem theory, then an input value x must have been received on an input port of the component. This axiom eliminates models that may spontaneously transmit data values.
3. The last axiom defines the communication process to be a cycle of receiving an input via *rcv* and transmitting a result via *trx*.

Figure 4.9 shows a diagram for the summation and averaging components. Complex number theory, sequence theory, and tuples of complex and naturals are common to both the *Complex-Adder* and the *Complex-Average* specifications. Thus tuples of complex and natural data values produced by the adder are of the same sort as the tuples of data values consumed by the averaging component. As shown in the figure, problem specifications for the summation and averaging problems are related to *Problem-Theory* by defining a morphism from *Problem-Theory* to *Complex-Adder* and *Complex-Average*, respectively.

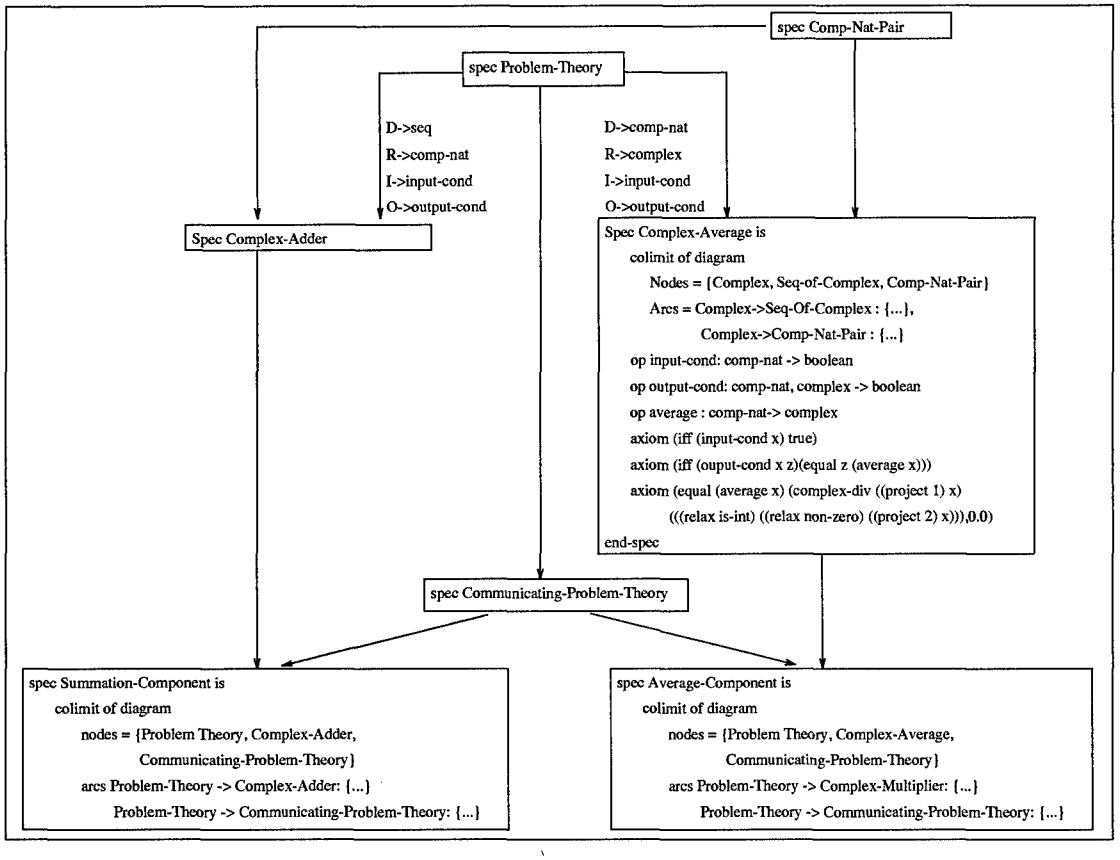


Figure 4.9 Specification of the Summation and Averaging Components

Functional specifications for the averaging component and the summation component are then defined by taking colimits of the diagrams as shown in Figure 4.9.

The resulting specifications, *Summation-Component* and *Average-Component*, include the communication operations *trx* and *rcv*, but these operations are at this point only abstractly defined. The next step in the development is to define these operations.

In Figure 4.10, the specifications *Complex-Average* and *Complex-Adder* have been extended with axioms constraining the operations *trx(c)* and *rcv(c)* to be equivalent to the CSP constructs $c.left!x \rightarrow c.right?ack$ and $c?x$ respectively. The extended specifications also provide a definition of the set of ports for each of these components: one port each for receiving and transmitting data values and one port for receiving acknowledgment events following a successful invocation of *trx*. Also shown in the figure is the specification *Connector*. Connectors are used to manage the communication protocol between the summation and averaging components.

The set of communication events shared between the connector, the summation component, and the averaging component are defined by the specification *System-Events*. By defining the relationship between the set of events defined in *System-Events* and the sort E of the connector and the summation and averaging components, the colimit of the diagram will have a common definition of the sort E across these three entities. Similarly, the simple specification *Channel* is used to provide a unification of port names under the colimit operation. In the semantics of CSP, unification of port names result in the formation of a CSP channel. After taking the colimit, the process-base operations *trx* and *rcv* in the connector are still abstractly defined, and could be specialized using one of the communication paradigms described in Section C.2 of Appendix C.

4.3.4 Observations. There are a number of rather significant problems with this approach that severely limit its practicality:

1. The semantic interpretation of the unification of port symbols *p1* and *p2* where *p1* and *p2* are symbols in two distinct process expressions is that *p1* and *p2* define a CSP channel. This interpretation is distinct from, for example, the semantic interpretation of sorts D and *comp-nat* belonging to a common equivalence class.
2. Neither *trx* nor *rcv* are implementable as functions because they are not timeless operators. That us, the values returned by *rcv(c)* and *trx(c,v)* vary over time. In essence, the semantic interpretation of the operations *trx* and *rcv* differs from the semantic interpretation of any other operation.
3. The structure for defining a communication processes in component specifications such as *Communicating-Entity* is awkward. Specifically, statements such as *(iff (equal x rcv(c))(c?x))* are not well formed. The statement *c?x* is not a boolean valued predicate, so its use in the statement *(iff (equal x rcv(c))(c?x))* is not defined.

The problem is that this approach to defining the architecture of an application by incorporating CSP statements in functional specification results in a mixing of logics. Specifically, this approach results in axioms containing both predicate calculus expressions and CSP process expressions (process logics). Mixing stateless and timeless expressions of

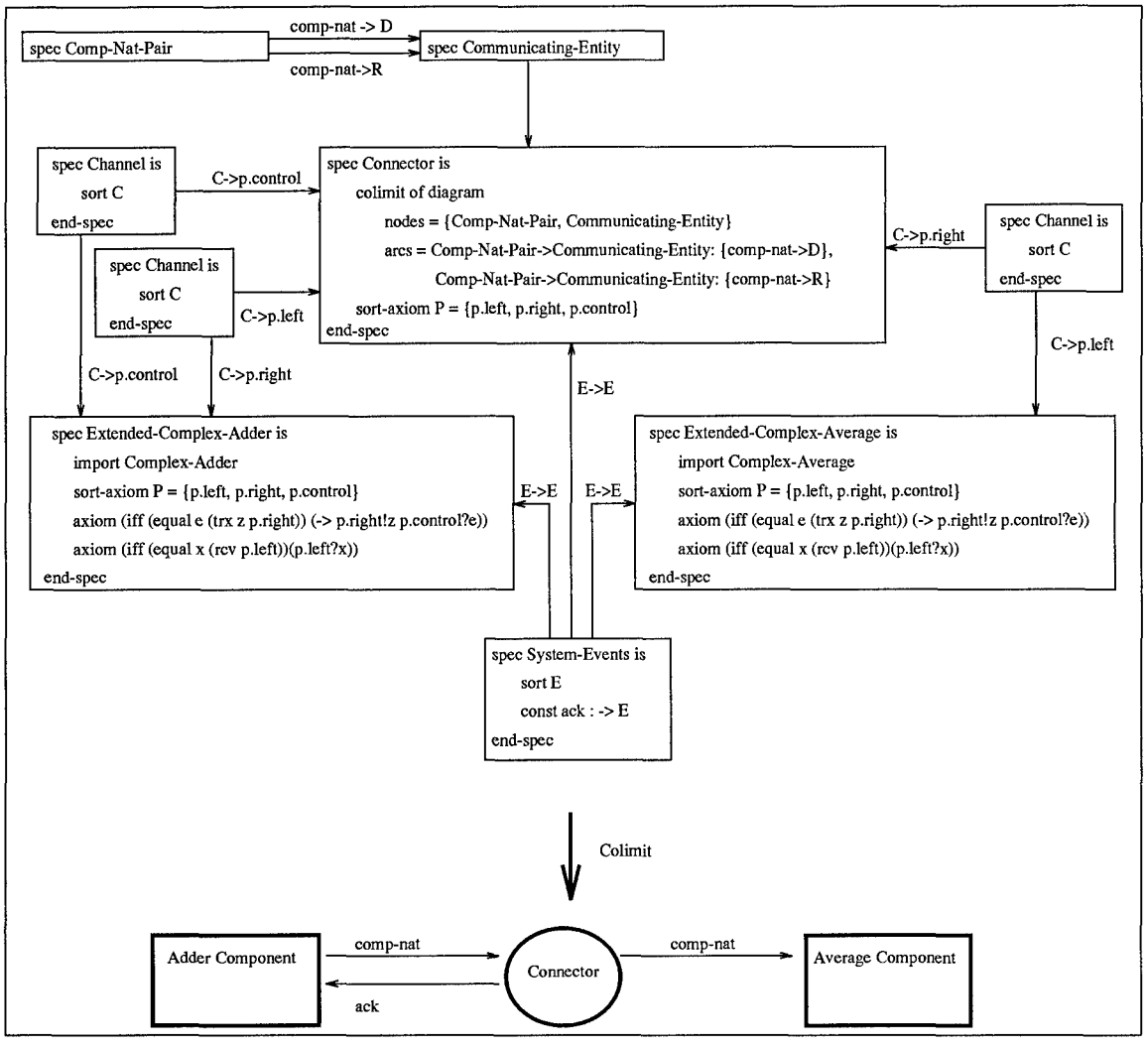


Figure 4.10 Specification of a Pipeline Segment for Finding the Moving Average

predicate calculus with state-based, time-dependent expressions of CSP results in axioms that are not well formed.

Note however that this finding does not necessarily invalidate the approach of using functional specifications to define sorts and operations and using CSP to define process structures. Indeed, the following chapter describes a specification technique in which CSP expressions are used to define application processes and in which functional specifications are used to define sorts and operations. However, the technique described in the following chapter segregates the process logic of CSP from the higher order functional logic of SLANG and SpecWare.

4.4 Summary

Development of specifications for the two applications described in the previous sections leads to the following conclusions:

1. Architecture can be defined within functional specifications. However, such specifications remain stateless and timeless.
2. Directly incorporating CSP expressions in functional specifications allows definition of state and allows definition of processes, but leads to ill-formed axiomatization. The process based logic of CSP must either be segregated from the higher order logic of the functional specifications, or the process logic of CSP must be expressed within a first order framework.

According to Shoham, any modal logic can be replaced by a first order logic.(96:28) This implies that if the process logic of CSP can be redefined as a modal logic, then CSP can be interpreted within a first order logic framework. The integration of these two logics, if possible, would eliminate many of the problems identified in Section 4.3.4. Another approach, the approach taken in the remaining chapters of this dissertation, is to segregate the process logic of CSP from the logic used in support of functional specification. Specifically, the remaining chapters of this dissertation establish a mathematical foundation for the specification of software architecture in which functional specifications are developed in a logic separate from the process logic of CSP. Integrating the process logic of CSP with the higher order logic of SLANG is left for future research.

V. Mathematical Foundations

5.1 Introduction

C.A.R. Hoare in his text *Communicating Sequential Processes* (52) provides a quasi-algebraic definition of communicating sequential processes (CSP). CSP constructs, although not formally defined, were used in the preceding chapter in an unsuccessful attempt to merge the ability of functional specifications to define sorts and operations with the ability of CSP to define state, communication, and processes. The primary failing of the work of the preceding chapter was the incorporation of process expressions in functional axioms. Functional logic and process logic were not well segregated; this lack of segregation led to ill-formed axioms. This chapter develops a formal relationship between the process logic of Hoare's CSP and the higher order logic of functional specifications such that logical expressions are both well-formed and interpreted in their respective logical system. An overview of the relationships defined in this chapter may be found in Figure 5.1. The figure references two institutions, an institution of process logic and an institution supporting functional specifications. Each of these institutions, as well as specification languages within them, are described in the following paragraphs.

Goguen has established the existence of institutions supporting functional specifications, among them are an institution of equational logic and an institution of predicate calculus.(41) As stated in Definition III.21, an institution supplies definition to the categories **Mod** of models, **Sign** of signatures, and **Set** of sentences, and provides definition of the satisfaction relation $\models$. Once an institution supporting functional specification has been defined, any number of specification languages within that institution can be defined. As shown in Figure 5.1, SLANG is a specification language within an institution supporting functional specification. Specifically, SLANG is a specification language within a multi-sorted, higher order predicate calculus institution. Other functional specification languages, such as *OBJ* and *Larch* can also be defined within an institution framework.

Also shown in Figure 5.1 is an institution of process logic. Process-based languages, such as Milner's CCS or Hoare's CSP, could be used as the basis of a process-based specification language. In fact, Hoare's CSP is used in this chapter in the definition of the

process-based specification language *ISlang.* Although Figure 5.1 makes reference to an institution of process logic, I have been unable to find a definition of such an institution in any of the current literature. No attempt is made in this chapter to define such an institution. Instead, an institution of process logic is assumed to exist.

Assumption V.1 *Given Milner's definition of CCS, Hoare's quasi-algebraic definition of communicating sequential processes (CSP), and Hennessy's deductive system for process logic, an appropriate generalization effort could be undertaken to define an institution of process logic. Within such a process logic institution, process-based specification languages could be defined. The process-based specification language* ISlang *is one such language.*

Support. *In their paper* Introducing Institutions, *Burstall and Goguen conjecture that an institution for process logic could be developed.(41) In addition, in his text* Algebraic theory of processes, *Hennessy defines a process algebra similar to Hoare's CSP and develops a complete deduction system for it.(51) The closure of the set of axioms of Hennessy's process algebra would establish a theory of communicating processes which could be generalized to define an institution of process logic.* ■

ISlang is a specification language based on Hoare's definition of CSP. Specifically, ISlang uses CSP expressions to define processes. CSP expressions serve to restrict the set of models of ISlang specifications in much the same way that functional axioms serve to restrict the class of models of SLANG specifications. Development of the specification language ISlang proceeds in three steps:

1. Before CSP can be used in the definition of a specification language, CSP must first be stated as a theory presentation. That is, the signatures of the event and process operators defined by Hoare must be defined, and the behavior of these operations must be defined. A theory presentation for Hoare's CSP is developed in Section 5.2. This theory presentation is denoted CSP_A.

2. CSP_A does not include the algebra of traces defined by Hoare. To provide a trace semantic for expressions in CSP_A, CSP_A is extended in Section 5.3 to define CSP structures. CSP structures provide language based definitions of the trace operators

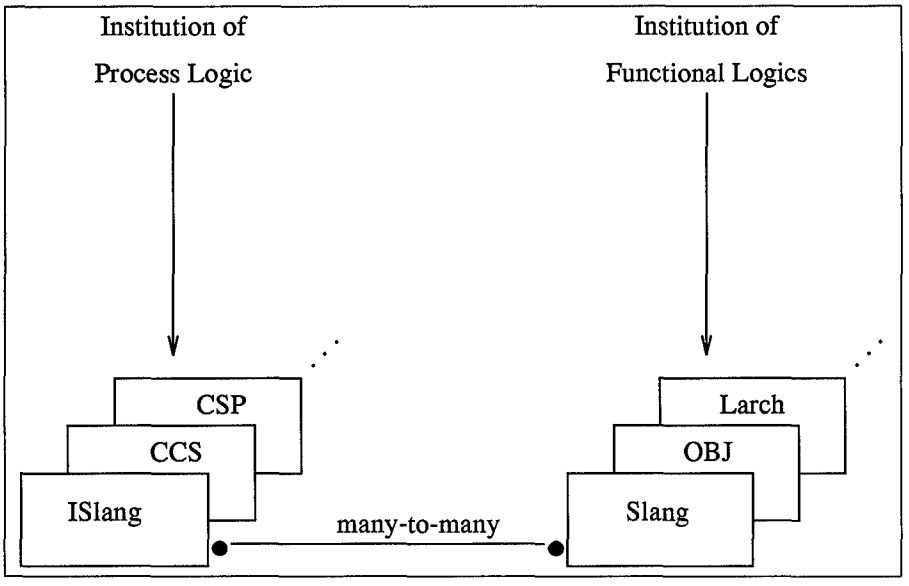


Figure 5.1 Mathematical Overview

defined by Hoare. It is through CSP structures that investigation of the behavior of models of process expressions in CSP_A is possible.

3. Next, a category **PSign** of process signatures and process signature morphisms is defined. **PSign** is defined in Section 5.4.

4. Finally, a category **PSpec** of process specifications and process specification morphisms is defined. The language ISlang is used to denote process specifications within this category. **PSpec** is defined in Section 5.5.

ISlang specifications define processes which may make use of sort symbols and functional operator symbols. Definitions of these sort and operator symbols are provided by functional specifications. That is, the semantics of some of the sort symbols and operator symbols in ISlang specifications are provided by companion functional specifications. In Figure 5.1 the many-to-many relationship between ISlang specifications and SLANG specifications highlights this relationship. The sorts and functional operators of an ISlang specification define a functional signature Σ. Any SLANG specification having the signature Σ can be used to constrain the set of models of these sorts and operations. Conversely, the sorts and operations of a single SLANG specification may be referenced in several ISlang specifications.

Later in this chapter, constructs are introduced which can be used to associate an ISlang specification with a SLANG specification. Specifically, *components* are used to associate an ISlang specification with a SLANG specification such that the SLANG specification defines some of the sorts and operations referenced in the ISlang specification. Furthermore, it is shown that components and component morphisms form the category **App**. The relationship between process specifications and functional specifications is further described in Section 5.6.

5.2 A Theory Presentation for CSP

The algebra of communicating sequential processes as defined by Hoare consists of a number of related algebras, among them are an algebra of traces, an algebra of events, an algebra of sets, and an algebra of processes. For example, the event algebra can be used to define event symbols such as $c?x$, and the process algebra can be used to define processes in terms of events and other processes. This section develops a theory presentation for the event and process algebras defined by Hoare.

Before defining a theory presentation for Hoare's CSP, an algebra for defining terms is presented. Term algebras define carrier elements for data values referenced in CSP expressions. Term algebras are defined using indexed collections of sets, so a definition of indexed collections of sets is provided first.

Definition V.1 Indexed collection of sets. *(110:91) If $\mathcal{D}$ is a set and a set A_d has been defined for each $d \in \mathcal{D}$, then d is called the* index *of A_d, the collection $C = \{A_d \mid d \in \mathcal{D}\}$ is called an* indexed collection of sets, *and $\mathcal{D}$ is called the* index set *of the collection. When $\mathcal{D}$ is the index of a collection C, the notation $\bigcup_{d\in\mathcal{D}} A_d$ denotes the set $\{x \mid \exists d[d \in D \wedge x \in A_d]\}$, and for $\mathcal{D} \neq \{\}$, $\bigcap_{d\in\mathcal{D}} A_d$ denotes $\{x \mid \forall d[d \in \mathcal{D} \Rightarrow x \in A_d]\}$.* □

For example, if $\mathcal{D} = \{int, real, bool\}$, and $A_{int} = \{i_1, i_2\}$, $A_{real} = \{r_1, r_2, r_3\}$, and $A_{bool} = \{b\}$, then $\bigcup_{d\in\mathcal{D}} A_d = \{x \mid \exists d[d \in \{int, real, bool\} \wedge x \in A_d]\} = \{i_1, i_2, r_1, r_2, r_3, b\}$, and $\bigcap_{d\in\mathcal{D}} A_d = \{x \mid \forall d[d \in \{int, real, bool\} \Rightarrow x \in A_d]\} = \{\}$. State variables and communication ports are defined using indexed collections of sets, where the index sets are sets of sort symbols. The notation $x : D$ is used as a shorthand notation for $x \in A_d$ where x is a

variable of sort D. Now that indexed collections of sets have been defined, terms can be defined.

Definition V.2 Terms. *(Adapted from (107).) Given a signature $\Sigma = \langle S, \Omega \rangle$, and an indexed collection of sets $X = \{X_s \mid s \in S\}$ of* variables *indexed by S, i.e., each variable is associated with a sort s of Σ, the set of terms generated by the signature using the variables X, denoted $T_\Sigma(X)$, is defined by the indexed family*

$$T_\Sigma(X) \quad = \quad \{T_{\Sigma,s}(X) \mid s \in S\}$$

where $T_{\Sigma,s}(X)$, the set of terms of sort s, is defined inductively as follows:

1. *if x is a variable of sort s in X_s, then $x \in T_{\Sigma,s}(X)$;*
2. *if c is a constant symbol of sort s with $c \in \Omega$, then $c \in T_{\Sigma,s}(X)$;*
3. *if f is an operation of sort s and rank $s_1, s_2, \ldots, s_n$, and $t_1, t_2, \ldots, t_n$ are terms in $T_{\Sigma,s_1}(X), T_{\Sigma,s_2}(X), \ldots, T_{\Sigma,s_n}(X)$ respectively, then $f\_(t_1, t_2, \ldots, t_n)\_ \in T_{\Sigma,s}(X)$.* □

When X is empty, then $T_{\Sigma,s}(X)$ is called the set of *ground terms* of Σ and is denoted T_Σ. The underscores above are used to highlight the fact that the *symbol* $f\_(t_1, t_2, \ldots, t_n)\_$ is in $T_{\Sigma,s}(X)$. The underscores will be dropped if the meaning is clear based on the context of the expression. Value can now be defined.

Definition V.3 Value. *Given a functional signature $\Sigma = \langle S, \Omega \rangle$, a value v of sort s is an element of the set $T_{\Sigma,s}$.* □

Now that term algebras and value have been defined, they can be used in the definition of the theory presentation CSP_A.

Definition V.4 *A theory presentation of CSP, denoted* CSP_A*, has a multi-sorted signature $\Sigma_{CSP} = \langle S, \Omega \rangle$, where*

1. *The set S of sort symbols is the set* {event, process, channel, value, variable, label}, *and*

2. Ω is the collection of operators of Figures 5.2 and 5.3.

The set of Σ_{CSP} sentences constraining the set of Σ_{CSP}-models is given in (52). □

Although the signatures of the operations in Figures 5.2 and 5.3 appear to be SLANG signatures, the sorts and operations of these two figures are defined in a process logic, not a functional logic. Definition of a term algebra for the sort *value* is provided through association of a functional signature with statements in CSP_A. This relationship is described in Section 5.6. Expressions of sort *process* defined using the operators of CSP_A are defined using terms.

Definition V.5 *The set of terms generated by the signature of CSP_A using the variables X is denoted $T_{CSP}(X)$. A term in $T_{CSP}(X)$ is called a* CSP expression *or a* process expression. □

The semantics of the functions of Figure 5.3 are described in (52). The semantics of the operations $\surd$ and $\_ \xrightarrow{CSP} \_ : \textit{event, process} \rightarrow \textit{process}$ of Figure 5.2 are also described in (52), where $\surd$ is the successful termination event, and $a \xrightarrow{CSP} P$ has the same semantics as *a then P*; the arrow $\xrightarrow{CSP}$ is used in an effort to eliminate confusion with the arrow $\rightarrow$ used in the definitions of operators. The symbol $\not\rightarrow$ of Figure 5.2 denotes the catastrophic event. The other operations of Figure 5.2 are used to define communication events. That is, the operation $c!x$ where c is of sort *channel* and x is of sort *value*, defines a CSP communication event that can be used in an expression such as $P = a \xrightarrow{CSP} (c!x \xrightarrow{CSP} P \mid d!x \xrightarrow{CSP} P)$.

The operations of Figure 5.3 can be used to combine processes and events to define processes. For example, the operator $\_ : \_ : \textit{set(label),process} \rightarrow \textit{process}$ defines a collection of labeled processes operating in parallel. That is, the statement *{0,1,2}:P* where P is a process expression defines the process $0 : P \parallel 1 : P \parallel 2 : P$. Remote subordination can be defined using the operations listed in Figure 5.3. For example, the CSP expression *doub:($\|_{i<27}$ (i:DOUBLE)) // $\cdots$* can be represented in CSP_A by the semantically equivalent expression *doub:({0,1,...,26}:DOUBLE)) // $\cdots$*.

Each process composition operator defined by Hoare in (52) is included in CSP_A. This fact is expressed in the following theorem.

```
sig CSP-Event-Sig is
      sort event
      sort channel
      sort value
      sort variable
      const √:  → event
      const ↯:  → event
      op _!_ :  channel, value → event
      op _?_ :  channel, variable → event
      op _!_?_ :  channel, value, variable → event
end-sig
```

Figure 5.2 Signature of CSP Event Operators

```
sig CSP-Process-Sig is
      sort label
      sort event
      sort process
      const       STOP :→ process
      const       SKIP :→ process
      op _:=_ :  variable, value → process
      op _ --CSP--> _ :  event, process → process
      op _ --CSP--> _ | _ --CSP--> _ :  event, process, event, process → process
      op _||_       :  process, process → process
      op _:_ --CSP--> _ :  event, set(event), process → process
      op _≫_        :  process, process → process
      op _⊓_        :  process, process → process
      op _[]_       :  process, process → process
      op _|||_      :  process, process → process
      op _;_        :  process, process → process
      op _//_       :  process, process → process
      op _⊀_⊁_      :  process, boolean, process → process
      op _*_        :  boolean, process → process
      op _ⓧ_        :  process, process → process
      op _\_        :  process, set(event) → process
      op _:_        :  label,process → process
      op _:_        :  set(label),process → process
      op _^_        :  process, process → process
      op △          :  process → process
      op _ ^↯ _     :  process, process → process
end-sig
```

Figure 5.3 Signature of CSP Process Operators

Theorem V.1 *The theory presentation CSP_A is complete with respect to the class of CSP process operators as given by Hoare.*

Proof. *The validity of this claim is established by the following mapping:*

Hoare Construct and Section of (52)	*Corresponding CSP_A Construct*
a *then* P *(1.1.1)*	*op* _ $\overset{CSP}{\longrightarrow}$ _ : *event, process* $\rightarrow$ *process*
Choice (1.1.3)	*op* _ $\overset{CSP}{\longrightarrow}$ _ | _ $\overset{CSP}{\longrightarrow}$ _ *event, process, event, process* $\rightarrow$ *process*
Choice of (1.1.3)	*op* _ : _ $\overset{CSP}{\longrightarrow}$ _ : *event, set(event), process* $\rightarrow$ *process*
Success (1.9.7)	*const* $\surd$: $\rightarrow$ *event*
Parallel Composition (2.3)	*op* _||_ : *process, process* $\rightarrow$ *process*
Named Process (2.6.2)	*op* _ : _ : *label,process* $\rightarrow$ *process*
Named Process (2.6.4)	*op* _ : _ : *set(label),process* $\rightarrow$ *process*
Non-deterministic Choice (3.2)	*op* _ $\sqcap$ _ : *process, process* $\rightarrow$ *process*
Choice (3.3)	*op* _[ ]_ : *process, process* $\rightarrow$*process*
Concealment (3.5)	*op* _ \ _ :*process, set(event)* $\rightarrow$ *process*
Interleaving (3.6)	*op* _||| _ : *process, process* $\rightarrow$ *process*
Output Communication (4.1)	*op*_!_ : *channel, value* $\rightarrow$ *event*
Input Communication (4.1)	*op* _?_ : *channel, variable* $\rightarrow$ *event*
Chained to (4.4)	*op* _ $\gg$ _ : *process, process* $\rightarrow$ *process*
Subordination (4.5)	*op* _//_ : *process, process* $\rightarrow$ *process*
Sequential Composition (5.1)	*op* _;_ : *process, process* $\rightarrow$ *process*
Interrupts (5.4)	*op* _^_: *process, process* $\rightarrow$ *process*
Catastrophic Event (5.4.1)	*const* $\looparrowright$: $\rightarrow$ *event*
Catastrophe (5.4.1)	*op* _ $\overset{\wedge}{\looparrowright}$ _ : *process, process* $\rightarrow$ *process*
Restartable (5.4.2)	*op* $\triangleq$: *process* $\rightarrow$ *process*
Alternation (5.4.3)	*op* _ $\textcircled{x}$ _ : *process, process* $\rightarrow$ *process*
P *if* b *else* Q *(5.5)*	*op* _ $\not<$ _ $\not>$ _ : *process, boolean, process* $\rightarrow$ *process*
While-Do (5.5)	*op* _ * _: *boolean, process* $\rightarrow$ *process*
Assignment (5.5)	*op* _ := _ : *variable, value* $\rightarrow$ *process*
Subroutine (6.2)	*op* _!_?_ : *channel, value, variable* $\rightarrow$ *event*

The Hoare construct $*P$ *(Section 5.1) is equivalent to* true$*$P. *Furthermore, Hoare's CSP expression* doub:($\|_{i<27}$ (i:DOUBLE)) // $\cdots$ *defining remote subordination (Section 6.4) can be represented in CSP_A by the semantically equivalent expression* doub: ({0,1,...,26} : DOUBLE)) // $\cdots$ ■

Although any well-formed CSP process expression can be represented by an expression in CSP_A, such process expressions cannot be semantically investigated in CSP_A

because CSP_A does not include the trace algebra defined by Hoare. CSP_A defines the syntax for process expressions; the semantics of process expressions in CSP_A are provided through *CSP structures.* CSP structures and their use in defining the semantics of process expressions in CSP_A is described in the following section.

5.3 CSP Structures

5.3.1 Syntax. The theory presentation CSP_A defined in the previous section contained operations for defining processes using event symbols and process symbols, but lacked the ability to investigate semantic properties of those processes. Investigation of process expressions of CSP_A could be accomplished by extending CSP_A with a trace algebra which would include operators such as *traces(_) : process → set (seq (event)).* operators. Instead, the semantics of CSP_A expressions are provided by relating CSP_A to formal languages through CSP structures. That is, CSP structures provide the ability to define collections of related process expressions and when related to language theory, provide the capability to investigate the trace behavior of the collection.

Definition V.6 *(Based on (10).) A CSP structure is the 7-tuple* $\langle \mathcal{P}, \mathcal{A}, \mathcal{C}, \mathcal{V}, \Sigma, O_P, \mathcal{S} \rangle$ *where*

1. $\mathcal{P}$ *is a finite sequence of process symbols denoting communicating sequential processes such that each element in the sequence is unique.*
2. $\mathcal{A}$ *is a finite sequence of nonempty sets of process alphabets such that* A_i *contains the alphabet of process* P_i*. Each* A_i *is further partitioned as follows:*
   (a) *Process events — Those non-communication events generated either by the environment or by the process* P_i*.*
   (b) *Communication events — Those events of the form c <op> v where c is a CSP channel, <op> is either '!' or '?', and v is the value communicated over c.*
3. $\mathcal{C}$ *is a sequence of sets of labeled communications channels connecting the processes in P. Each channel* $c \in C_i$ *and* $c \in C_j$ *connects exactly two processes* $P_i, P_j \in P$

such that c is used exclusively for input to P_i and exclusively for output from P_j. C_i denotes the set of labeled communications channels of P_i.

4. *$\mathcal{V}$ is a sequence of sets of labeled variables of the form $v : s$ such that V_i contains the variables of process P_i.*

5. *Σ is a functional signature containing sorts symbols and operation symbols such that*

   (a) *If the sort of a channel $c \in C_i$ is s for any $C_i \in \mathcal{C}$, then s is a sort symbol in Σ.*

   (b) *If $f : u \rightarrow v$ is an operation referenced in any process expression $S_i \in \mathcal{S}$ or in O_P, then $f : u \rightarrow v$ is an operation symbol in Σ and u and v are sort symbols in Σ.*

   (c) *The sort of any term in $T_\Sigma(X)$ is* value.

6. *O_P is a CSP expression formed over the symbols in $\mathcal{P}, \mathcal{V}, \Sigma$ and $\bigcup_{A \in \mathcal{A}} A$ such that the sort of the expression is* process. *O_P denotes the process expression of the overall CSP process.*

7. *$\mathcal{S}$ is a sequence of well-formed process expressions such that S_i is written over the symbols in A_i, V_i, Σ, and $\mathcal{P}$ such that the sort of the expression is* process. *S_i denotes the process expression of process P_i* $\square$.

Symbols in the sets of $C_i \in \mathcal{C}$ and $V_i \in \mathcal{V}$ can be combined via operations of CSP_A to define communication events.

The expressions in $\mathcal{S}$ define individual CSP processes. For example, for a process symbol $P_n \in \mathcal{P}$ with $A_n = \{e, f, g\}$, the statement $e \xrightarrow{CSP} (f \xrightarrow{CSP} P_n \mid g \xrightarrow{CSP} P_n)$ could be written.

Now that CSP structures have been defined, they can be used to define the semantics of CSP expressions.

5.3.2 Semantics. This section explores the semantics of CSP expressions through an analysis of the language accepted by the corresponding processes. The following discussion establishes the mathematical foundations of trace semantics for process expressions, and includes a brief discussion concerning the refusal sets defined by Hoare. Languages

are defined in terms of labeled transition systems and automata, each of which is defined below.

Definition V.7 Labeled transition system. *A labeled transition system (abbreviated LTS) is a 3-tuple* $L \triangleq \langle N, L, \delta \rangle$ *where*

- N *is a finite nonempty set of* nodes *or* node symbols,
- L *is a finite nonempty set of* labels, *and*
- $\delta : N \times L \rightarrow N$ *is a transition relation between nodes and labels such that if* $\delta(n_1, l) = n_2$, *then* n_1 *and* $n_2 \in N$ *and* $l \in L$. □

The transition relation need not be defined for all combinations of node and label symbols. Nondeterminism is supported in that the transition relation need not be functional, e.g., $\delta(n_1, l) = n_2$ and $\delta(n_1, l) = n_3$, $n_2 \neq n_3$ with $n_1, n_2 \in N$ and $l \in L$ is a valid transition relation for some LTS. Note that the semantics of nondeterminism are not defined as part of an LTS. For example, if $\delta(n_1, l) = n_2$ and $\delta(n_1, l) = n_3$ with $n_2 \neq n_3$ for some LTS, then one interpretation of $\delta(n_1, l)$ could be either n_3 or n_4 with the choice made non-deterministically, while another interpretation could be that δ is multi-valued with $\delta(n_1, l)$ equal to the set $\{n_3, n_4\}$.

Constraints placed on the transition relation of an LTS, e.g., that δ be functional, one-to-one, and onto, or that δ be a total function, restrict the semantics of the transition system. For example, requiring δ to be bijective eliminates nondeterminism. Placing additional structure in an LTS allows discussion of the *language* accepted by an LTS. Language is defined using a specific type of LTS, an *automaton.*

Definition V.8 Automaton. *An automaton is a labeled transition system* $L = \langle N, L, q_0, N_F, \delta \rangle$ *where*

1. N *is a nonempty set of node symbols,*
2. L *is a finite nonempty set of labels called the* alphabet *of the automaton,*
3. $q_0 \in N$ *is the* initial node,

4. $N_F \subseteq N$ *is a set of* acceptance nodes, *and*

5. $\delta : N \times L \rightarrow N$ *is a transition relation such that if* $\delta(n,l) = m$, *then* $m,n \in N$ *and* $l \in L$.

Given an automaton $A = \langle N, L, q_0, N_F, \delta \rangle$ *and a sequence* $s = \langle s_1, s_2, \ldots, s_n \rangle$ *of symbols from* L, *the sequence is accepted by* A *if there exists a sequence* $\langle m_1, m_2, \ldots, m_n \rangle$ *of node symbols from* N *such that* $m_1 = q_0$ *and* $m_n \in N_F$ *and for all* $i \in \{1, 2, \ldots, n-1\}, \delta(m_i, s_i) = m_{i+1}$. *The set of all such sequences is called the* language *of* A *and is denoted* $L(A)$. □

The interpretation or semantics of the relationships defined by δ is external to the automaton itself. In other words, an automaton defines only syntax; semantics are defined externally. For example, consider the process definition $P = e \xrightarrow{CSP} ((f \xrightarrow{CSP} P) \sqcap (f \xrightarrow{CSP} (g \xrightarrow{CSP} P)))$ graphically depicted in Figure 5.4. Node symbols have been added to the figure to facilitate discussion. For this process, the transition relation δ for node 2 has two next states for event f, i.e., $\delta(2,f) = 1$ and $\delta(2,f) = 3$. In the semantics of Hoare's CSP, there are many possible implementations of this process.(52) One implementation may select $\delta(2,f) = 1$, and would deadlock on the input sequence $\langle e, f, g \rangle$. Another implementation could select $\delta(2,f) = 2$, and could therefore properly recognize the input sequence $\langle e, f, g \rangle$, but would deadlock on the sequence $\langle e, f, e \rangle$. A third implementation could delay making the determination of the next state of node 2 until a point in the execution of P is reached in which one branch deadlocks but the other does not. (Hoare refers to this as "angelic nondeterminism".(52:105)) That is, both branches would tentatively be taken following the event sequence $\langle e, f \rangle$ and the decision as to which branch is the proper branch to take would be made based on which event, e or g, occurred next. The point here is that the semantics of a LTS are provided by a set of behavior rules that is external to the LTS itself. The behavior rules of CSP, and hence of CSP_A, are defined in (52).

The semantics of process expressions can be defined in terms of automata theory. As noted in Definition V.8, the development of such a definition requires definitions of the state, the labels, the transition relation, the final states, and the initial state of a CSP process. The state of a CSP process is given below.

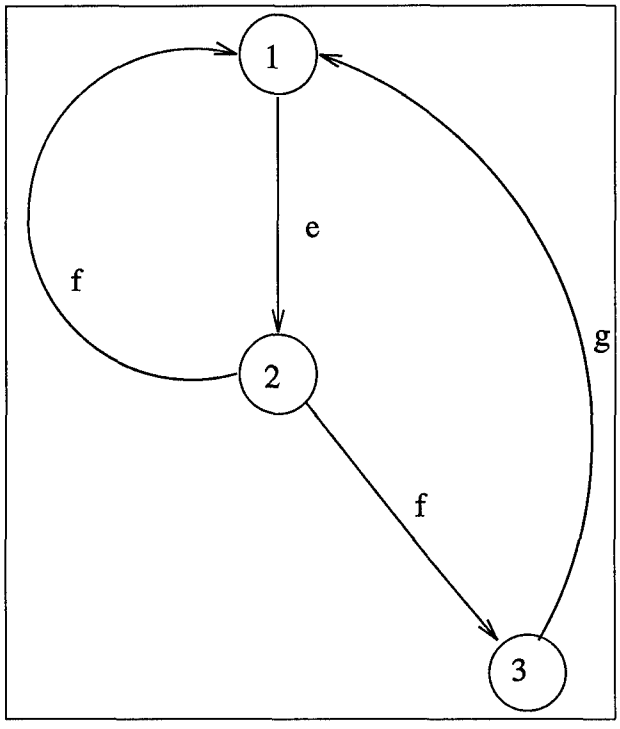


Figure 5.4 Pictorial Representation of the Process Defined by the Expression $P = e \xrightarrow{CSP} ((f \xrightarrow{CSP} P) \sqcap (f \xrightarrow{CSP} (g \xrightarrow{CSP} P)))$

Definition V.9 *(10)* CSP process state. *The state of the execution of a CSP process at time t can be described by the 3-tuple* $\langle C_{E_t}, P_{E_t}, E_{A_t} \rangle$, *where*

1. $C_{E_t} \in \mathcal{P}(C)$ *is the set of enabled communication channels at time t. That is,* C_{E_t} *is the set of communication channels that have an expression available for input.*
2. $P_{E_t} \in \mathcal{P}(P)$ *is the set of enabled processes at time t. That is, the set of processes whose guards are true allowing the process to engage in events.*
3. $E_{A_t} \in \mathcal{P}(A)$ *is the set of events for which the enabled processes in* P_{E_t} *are prepared to engage at time t. That is, those events whose occurrence would cause a change in state.* □

Note that the refusal set of an enabled process P in P_{E_t} as defined by Hoare is the set $\alpha P - E_{A_t}$.

The initial state of execution of a CSP process is defined using the above structures.

Definition V.10 *(Based on (10))* Initial State. *The initial state of execution of a CSP process can be described by the 3-tuple* $\langle C_E, P_E, E_A \rangle$, *where*

1. $C_E \in \mathcal{P}(C)$ *is the set of communication channels of the processes in* P_E *which initially have expressions ready for input.*
2. $P_E \in \mathcal{P}(P)$ *is the set of processes whose guards are initially true.*
3. $E_A \in \mathcal{P}(E)$ *is the set events of the processes in* P_E *such that* $e \in E_A$ *if and only if there is a process* $P_i \in P_E$ *such that* P_i *is initially prepared to engage in* e. *The set of events initially offered by a process* P_i *is denoted* **initial-events**$(\mathbf{P}_i)$.

The initial state of execution of a CSP process P is denoted **initial(P)**. □

The dual to the notion of the initial state of a CSP process is the notion of a final state.

Definition V.11 *(Based on (10))* Final State. *The final state of CSP structure* $S_{CSP} = \langle P, \mathcal{A}, O_P, S, C, V \rangle$ *occurs when the event* $\surd$ *is engaged. That is, when P successfully terminates. The final state of P is denoted* **final(P)**. □

Now that state has been defined, the next state relation (transition relation) can be defined.

Definition V.12 *(Based on (10))* Next-State Relation. *The next-state relation of a CSP structure* $S_{CSP} = \langle P, \mathcal{A}, O_P, S, C, V \rangle$ *with alphabet* $A = \bigcup_{A_i \in \mathcal{A}} A_i$ *is a relation* $\triangle : \mathcal{P}(C), \mathcal{P}(P), \mathcal{P}(A), A \rightarrow \mathcal{P}(C)\mathcal{P}(P)\mathcal{P}(A)$ *such that* $\triangle(C_{E_t}, P_{E_t}, E_{A_t}, a) = (\widehat{C}_{E_t}, \widehat{P}_{E_t}, \widehat{E}_{A_t})$, *where*

1. $\widehat{C}_{E_t}$ *is the new set of enabled communication channels,*
2. $\widehat{P}_{E_t}$ *is the new set of enabled processes,*
3. $\widehat{E}_{A_t}$ *is the new set of possible next events, and*
4. a *is an event in* $\mathcal{A}$,

subject to the following conditions:

1. $\triangle(\{\}, P_{E_t}, \{\}, a) = (\{\}, \{\}, \{\})$, *for any* a *and any* P_{E_t},

2. $\triangle(C_{E_t}, P_{E_t}, E_{A_t}, a) = (\{\}, \{\}, \{\})$ *for any* $a \in A \setminus E_{A_t}$,

3. $\triangle(C_{E_t}, P_{E_t}, E_{A_t}, a\beta) = \triangle(\triangle(C_{E_t}, P_{E_t}, E_{A_t}, a), \beta)$, *and*

4. $\triangle(C_{E_t}, P_{E_t}, E_{A_t}, \lambda) = (C_{E_t}, P_{E_t}, E_{A_t})$ *for any empty sequence* λ. □

The first two conditions placed on the next state relation $\triangle$ address the issue of deadlock. The first condition states that a process that can engage in no further events is deadlocked and that a deadlocked process remains deadlocked. The second condition states that a process will deadlock if an event in the alphabet of the process occurs when the process is not prepared to engage in it. The third condition defines the behavior of $\triangle$ over sequences of events, and the last condition states that no change in state will occur unless an event occurs.

A definition of CSP processes in terms of automata theory may now be given.

Definition V.13 *(Based on (10).)* CSP Automaton. *A CSP automaton is the 5-tuple* $\langle S_{CSP}, Q_{CSP}, initial(S_{CSP}), final(S_{CSP}), \triangle \rangle$ *where*

1. S_{CSP} *is a well formed CSP structure,*

2. Q_{CSP} *is the realizable state space of* S_{CSP} *recursively defined by*

   (a) **initial**$(S_{CSP}) \in Q_{CSP}$,

   (b) *if* $(c,p,e) \in Q_{CSP}$, *then* $\triangle(c,p,e,x) \in Q_{CSP}$ *where* $x \in \bigcup_{A_i \in \mathcal{A}} A_i$.

3. **initial**(S_{CSP}) *is the initial state,*

4. **final**(S_{CSP}) *is the final state,*

5. $\triangle : \mathcal{P}(C), \mathcal{P}(P), \mathcal{P}(A), A \rightarrow \mathcal{P}(C)\mathcal{P}(P)\mathcal{P}(A)$ *is the next state relation.*

Given a CSP automaton $W = \langle S_{CSP}, Q_{CSP}, initial(S_{CSP}), final(S_{CSP}), \triangle \rangle$ *and a sequence* $e = \langle e_1, e_2, \ldots, e_n \rangle$ *of event symbols from* $A = \bigcup_{A_i \in \mathcal{A}} A_i$, *the sequence is accepted by* W *if there exists a sequence* $\langle m_1, m_2, \ldots, m_n \rangle$ *of states from* Q_{CSP} *such that* $m_1 = initial(S_{CSP})$, *for all* $i \in [1..n-1], \delta(m_i, s_i) = m_{i+1}$, *and* $m_n = final(S_{CSP})$. *The*

set of all sequences accepted by a CSP automaton W is called the language *of W, and is denoted* language(W) *or L(W).*

The traces *of a CSP automaton P, denoted* traces(P), *consists of the set* $\{t \mid \exists(s)(t^\wedge s \in$ *language(P))*$\}$ *of sequences of event symbols from P, where* $t^\wedge s$ *denotes the concatenation of the sequences t and s.* □

The definitions above define the semantics of process expressions such as $P = (b \xrightarrow{CSP} SKIP \mid a \xrightarrow{CSP} (P; (c \xrightarrow{CSP} SKIP)))$ through an analysis of the languages they accept. In the case of the process expression P, the language accepted is $a^n bc^n$.(52:173)

Researchers have done some work to classify the types of languages accepted by CSP structures, with initial results presented in (52). However, a characterization of the class of languages generated or accepted by CSP automata is not germane to this investigation. In contrast, a characterization of the expressive power of CSP is important to this research for the following reason: CSP is used in Section 5.5 to define process specifications, and in Chapter VI, several architecture theories are defined in terms of process specifications. Any limitation on the expressive power of CSP automata limits the set of architectures that can be expressed using CSP-based process specifications. The expressive power of CSP structures is explored in the following theorem.

Theorem V.2 *(10:11-46) CSP automata have the generative power of Turing machines.*

Proof. *This proof is based on the results contained in Böhm and Jacopini.(17) Böhm and Jacopini show that every Turing machine is reducible (or equivalent) to a program written in a language which allows only sequence and iteration as formation rules. The syntax of CSP allows for sequence, branch-on-condition (if-then-else), and iteration. Since branch-on-condition can be written as iteration, a CSP structure meets the requirements stated in (17). Thus, every Turing machine can be reduced to a CSP structure.* ■

What functions cannot be computed by a Turing machine? *Church's hypothesis* or the *Church-Turing thesis* implies that any well-defined algorithm can be computed by a Turing machine.(26). Taken in conjunction with the above theorem, this implies that CSP is sufficiently powerful to specify any computable algorithm. That is, CSP is sufficiently powerful to specify any computable architecture.

5.3.3 Summary of CSP Structures. This section has provided a trace semantic for process expressions in the term algebra of CSP_A by defining CSP structures and defining the relationship between CSP structures and automata theory. Furthermore, it was shown that CSP automata have the generative power of Turing machines. CSP structures provide a means for comparing collections of process expressions through an analysis of the traces of the processes they define. The trace semantic provided by CSP structures for CSP statements is used in the following sections in the definition of a category of process specifications.

5.4 The Category of Process Signatures and Process Signature Morphisms

As stated in Assumption V.1, assuming an institution of process logic exists, a category of process specifications and process specification morphisms can be defined. This section defines such a category. Specifically, this section defines the category **PSign** of process signatures and process signature morphisms. **PSign** is used in the next section to define the category **PSpec** of process specifications and process specification morphisms.

CSP structures have several characteristics which must be identifiable in a process signature. For any CSP structure $S_{CSP} = \langle \mathcal{P}, \mathcal{A}, \mathcal{C}, \mathcal{V}, \Sigma, O_P, \mathcal{S} \rangle$, these characteristics include

1. the set $\mathcal{P}$ of process names,
2. the sequence $\mathcal{A}$ of sets of events,
3. the sequence $\mathcal{C}$ of sets of labeled port symbols, and
4. the sequence $\mathcal{V}$ of sets of labeled variables,
5. the sorts and operator symbols of Σ.

Each of these five entities is represented in a process signature, however, note that O_P and $\mathcal{S}$ define behavior and as such are not part of any process signature.

Definition V.14 Process signatures. *A process signature* $\Pi = \langle \Sigma, E, P, V, \kappa \rangle$ *consists of*

- *a* signature $\Sigma = \langle S, \Omega \rangle$ *with sorts* S *and functional operations* Ω*;*

- *a set E of event symbols;*
- *an indexed collection of sets $P = \{P_s \mid s \in S\}$ of* port symbols *indexed by S;*
- *an indexed collection of sets $V = \{V_s \mid s \in S\}$ of* state variable symbols *indexed by S; and*
- *a set κ of* process symbols.

Associated with each process symbol $Q \in \kappa$ is a four tuple of maps, ⟨events, var, act, chan⟩ *where*

- events : $\kappa \rightarrow$ E *maps process symbols to their alphabets such that for any process symbol $Q \in \kappa$,* events(Q) $\subseteq$ E*;*
- var : $\kappa \rightarrow$ V *maps process symbols to indexed collections of sets of variables indexed on S such that for any process symbol $Q \in \kappa$, $x : s \in var(Q) \Rightarrow \exists(V_s)(x \in V_s \wedge V_s \in V)$;*
- act : $\kappa \rightarrow \Omega$ *maps process symbols to sets of operator symbols such that for any process symbol $Q \in \kappa$,* act(Q) $\subseteq \Omega$*; and*
- chan : $\kappa \rightarrow$ P *maps process symbols to indexed collections of sets of port symbols indexed on S such that for any process symbol $Q \in \kappa$, $p : s \in chan(Q) \Rightarrow \exists(P_s)(p \in P_s \wedge P_s \in P)$.* □

Definition V.6 established a CSP structure to be a 7-tuple $\langle \mathcal{P}, \mathcal{A}, \mathcal{C}, \mathcal{V}, \Sigma, O_P, \mathcal{S} \rangle$. The process expressions of O_P and $\mathcal{S}$ of a CSP structure define process behavior and were therefore not represented within process signatures. As given in the following definition, the remaining elements of a CSP structure, $\mathcal{P}, \mathcal{A}, \mathcal{C}, \mathcal{V}$, and Σ can be represented within a process signature.

Definition V.15 Relationship between CSP structures and Process Signatures. *Denote by $S_{CSP} = \langle \mathcal{P}, \mathcal{A}, O_P, \mathcal{S}, \mathcal{C}, \mathcal{V}, \Sigma \rangle$ an arbitrary well-formed CSP structure. Define by the following set of relationships a process signature $\Pi = \langle \Sigma_\Pi, E_\Pi, P_\Pi, V_\Pi, \kappa_\Pi \rangle$ for S_{CSP}:*

- *Define Σ_Π to be the functional signature Σ of S_{CSP}.*

- *For every non-communication event symbol e in the set* $\bigcup_{A_i \in \mathcal{A}} A_i$, *define an event symbol e in E. Thus e is an event symbol in E only if e is a non-communication event contained in one of the sets of event symbols in* $\mathcal{A}$.

- *Define* κ *to be the set* $\{p \mid p \in \mathcal{P}\}$. *Because* S_{CSP} *is well-formed, the sequence of process symbols* $\mathcal{P}$ *in* S_{CSP} *contains no duplicate symbols. Thus every process symbol referenced in* S_{CSP} *is represented in* κ.

- *Define for each* $Q \in \kappa$ *the mapping* $\mathrm{var(Q)} \mapsto \mathrm{V}_i$ *such that* $P_i = Q$. *Define the collection* $V = \{V_s \mid s \in S\}$ *of S-indexed variable symbols of* Π *by the expression*

$$V = \bigcup_{W \in \kappa} \{var(W) \mid var(W) = V_i \Leftrightarrow P_i = W\}$$

  where $V_i \in \mathcal{V}$.

- *For every process symbol Q in* κ, *define* $\mathrm{event(Q)}$ *to be the set of non-communication events in* A_i *where* $P_i = Q$.

- *For every process symbol Q in* κ, *define* $\mathrm{act(Q)}$ *to be the set of operator symbols referenced in* S_i *where* $P_i = Q$. *Note that* $\bigcup_{W \in \kappa} \mathrm{act(W)} \subseteq \Omega$.

- *For every process symbol Q in* κ, *define* $\mathrm{chan(Q)}$ *to be the set of CSP channel symbols* C_i *where* $P_i = Q$. *Then* $\mathcal{C}$ *can be reconstructed by the following sequence former:*

$$\mathcal{C} = [c \mid \forall(i)(i \in [1..size]\mathcal{P}\exists(Q)(Q \in \kappa \wedge Q = P_i \Rightarrow c = chan(Q)))] \; \square$$

Note that process signatures use an alternate form to represent the elements $\mathcal{P}, \mathcal{A}, \mathcal{C}, \mathcal{V}$, and Σ of a CSP structure. For example, *chan(Q)* denotes the set of communications channels used in the definition of process Q; this set can also be defined through inspection of the sequence $\mathcal{C}$ of the corresponding CSP structure.

The syntax for a process signature is shown in Figure 5.5. The convention adopted in the figure and used throughout this dissertation is to associate with every process symbol in a process signature Π a concrete representation of the maps *events, var, act*, and *chan*. However, the map names may be omitted if the meaning of the sets is clear. The rep-

```
psig Simple-Pi-Signature is
        sorts D, R
        event ack
        port input : D
        port output : R
        port handshake : event
        var x : D
        op f : D → R
        op input-condition : D → boolean
        op output-condition : D, R → boolean
        process simple :events:{ack},
                        var:{x : D},
                        act:{f:D→R},
                        chan:{input : D, output : R, handshake : event}
end-psig
```

Figure 5.5 A Simple Process Signature

resentation chosen is *process-name : events:*$\{\cdots\}$, *var:*$\{\cdots\}$, *act:*$\{\cdots\}$, *chan:*$\{\cdots\}$, where *process-name* is a process symbol in κ, and where the ellipses are replaced with the values of the maps. For example, the process signature *Simple-Pi-Signature* shown in the figure has the following features:

- $\Sigma = \langle S, \Omega \rangle$, with $S = \{D, R\}$, and $\Omega = \{f : D \rightarrow R,$ *input-condition* $: D \rightarrow$ *boolean,* *output-condition* $: D, R \rightarrow$ *boolean*$\}$;
- $E = \{ack\}$,
- $P = \{P_D, P_R, P_{event}\}$ with $P_D = \{input\}$, $P_R = \{output\}$, and $P_{event} = \{control\}$;
- $V = \{V_D\}$, where $V_D = \{x\}$; and
- $\kappa = \{\text{simple}\}$.

For any process symbol P in κ, elements of the sets *events(P), var(P), chan(P), act(P),* and κ can be used to construct a process expression for P. For example, if *events(P)* $= \{a,b,c\}$, then the process expression P *sat* $(a \xrightarrow{CSP} P \mid b \xrightarrow{CSP} Skip);(c \xrightarrow{CSP} Skip)$ is a valid process expression for P. Process expressions such as this are represented in process specifications. Process specifications are the topic of the next section.

Now that process signatures have been defined, process signature morphisms can be defined.

Definition V.16 Process signature morphisms. *Given two process signatures* $\Pi = \langle \Sigma, E, P, V, \kappa \rangle$ *and* $\Pi' = \langle \Sigma', E', P', V', \kappa' \rangle$, *a process signature morphism* $\sigma : \Pi \rightarrow \Pi'$ *is a 5-tuple of functions* $\langle \sigma_\Sigma, \sigma_E, \sigma_P, \sigma_V, \sigma_\kappa \rangle$, *where*

- $\sigma_\Sigma : \Sigma \rightarrow \Sigma'$ *is a functional signature morphism (Definition III.2);*
- $\sigma_E : E \rightarrow E'$ *is a function which maps event symbols in* E *to event symbols in* E' *such that if* $e \in E$ *then* $\sigma_E(e) \in E'$;
- σ_P *is a function* $\sigma_P : P \rightarrow P'$ *mapping port symbols to port symbols such that the mapping is consistent with the sort of the port, i.e., for all port symbols* $p : s$ *in* P, $\sigma_P(p) : \sigma_\Sigma(s)$ *is in* P';
- σ_V *is a function* $\sigma_V : V \rightarrow V'$ *mapping state variable symbols to state variable symbols such that the mapping is consistent with the sorts of the variables, i.e., for all state variable symbols* $v : s$ *in* V, $\sigma_V(v) : \sigma_\Sigma(s)$ *is in* V';
- σ_κ *is a function* $\sigma_\kappa : \kappa \rightarrow \kappa'$ *mapping process symbols to process symbols such that if* $X \in \kappa$, *then* $\sigma_\kappa(X) \in \kappa'$ *subject to the following conditions: For all process symbols* $A \in \kappa$,
  - *if* $v : s$ *is a state variable symbol of* A, *i.e.,* $v : s \in \mathrm{var(A)}$, *then* $\sigma_V(v) : \sigma_\Sigma(s)$ *is in* $\mathrm{var}(\sigma_\kappa\ (\mathrm{A}))$;
  - *if* $c : s$ *is a channel symbol of* A, *i.e.,* $c : s \in \mathrm{chan(A)}$, *then* $\sigma_P(c) : \sigma_\Sigma(s)$ *is in* $\mathrm{chan}(\sigma_\kappa\ (\mathrm{A}))$;
  - *if* e *is an event symbol of* A, *i.e.,* $e \in \alpha A$, *then* $\sigma_E(e) \in \alpha(\sigma_\kappa(A))$; *and*
  - *if* $f : s_1, s_2, \ldots s_n \rightarrow s$ *is an operation symbol of* A, *i.e.,* $f : s_1, s_2, \ldots s_n \rightarrow s \in act(A)$, *then* $\sigma_\Sigma(f : s_1, s_2, \ldots, s_n \rightarrow s) \in act(\sigma_\kappa(A))$.

The identity process signature morphism maps each sort, operation, state variable, port, and collection of process symbols back to themselves. □

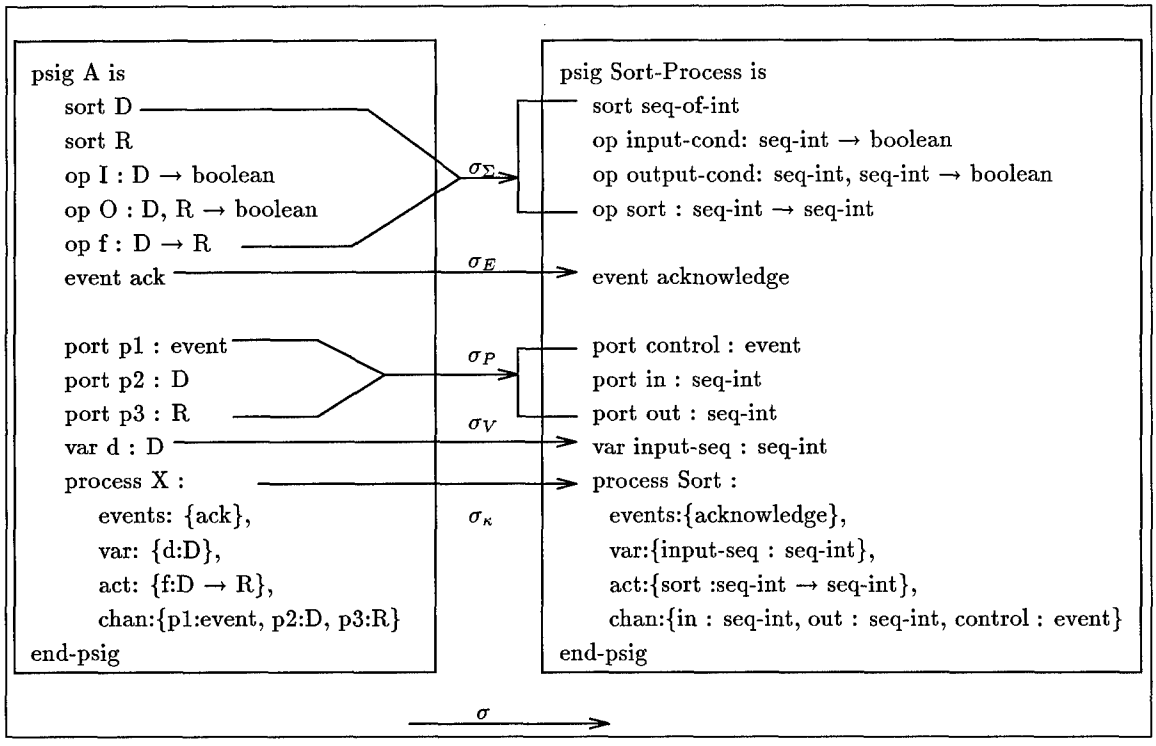


Figure 5.6 Process Signatures and Process signature Morphisms

A simple process signature morphism between two process signatures is shown in Figure 5.6. Each of the four functions defining the process signature morphism is shown in the figure. One of the uses of signature morphisms, as pointed out in Chapter III, is parameter instantiation. Using process signatures and process signature morphisms (and later process specifications and process specification morphisms), the architecture of a system can parameterized, with actual parameters supplied based on the specific application. For example, a system consisting a four stage homogeneous pipeline, e.g. $P \gg P \gg P \gg P$, could be defined as a parameterized specification parameterized on a process specification P. The colimit of the diagram containing the specification P, the pipeline specification, and a specification defining the actual parameter, defines a four stage homogeneous pipeline where the definition of the stages are provided by the actual parameter.

Formation of colimits as described above is predicated on the existence of an appropriate category. As established by the following theorem, process signatures and process signature morphisms define a category.

Theorem V.3 Category **PSign**. *Process signatures and process signature morphisms form a category* **PSign** *where process signatures are the C-objects and process signature morphisms are the C-arrows.*

Proof. *There are two properties that must be shown:*

1. *That each C-object has an identity morphism, and*
2. *That C-arrows compose to form C-arrows.*

Each of these is proven below.

1. *Identity. Identity arrows exist by definition. (See Definition V.16.)*
2. *Composition. Denote by* $\sigma_1 : \Pi \rightarrow \Pi_1$ *and* $\sigma_2 : \Pi_1 \rightarrow \Pi_2$ *two process signature morphisms where* $\Pi = \langle \Sigma, E, P, V, \kappa \rangle$, $\Pi_1 = \langle \Sigma_1, E_1, P_1, V_1, \kappa_1 \rangle$, *and* $\Pi_2 = \langle \Sigma_2, E_2, P_2, V_2, \kappa_2 \rangle$. *It will be shown that* $\sigma_1 = \langle \sigma_{\Sigma_1}, \sigma_{E_1}, \sigma_{P_1}, \sigma_{V_1}, \sigma_{\kappa_1} \rangle$ *and* $\sigma_2 = \langle \sigma_{\Sigma_2}, \sigma_{E_2}, \sigma_{P_2}, \sigma_{V_2}, \sigma_{\kappa_2} \rangle$ *compose to define a process signature morphism* $\sigma_3 : \Pi \rightarrow \Pi_2$ *where* $\sigma_3 = \sigma_2 \circ \sigma_1$.

   (a) *By definition, the functional signature morphisms* σ_{Σ_1} *and* σ_{Σ_2} *compose to define* σ_{Σ_3}.

   (b) σ_{E_1} *and* σ_{E_2} *compose as follows:*

   $\forall(e)\ (e \in E \ \Rightarrow\ \sigma_{E_1}(e) \in E_1)$ *by definition*

   $\forall(e')\ (e' \in E_1 \ \Rightarrow\ \sigma_{E_2}(e') \in E_2)$ *by definition*

   but $\sigma_{E_1}(e) \in E_1$ *implies* $\sigma_{E_2}(\sigma_{E_1}(e)) \in E_2$

   $\therefore$ σ_{E_1} *and* σ_{E_2} *compose to define* $\sigma_{E_3} : E \rightarrow E_2$ *where* $\sigma_{E_3} = \sigma_{E_2} \circ \sigma_{E_1}$.

   (c) *Claim:* $\sigma_{P_1} : P \rightarrow P_1$ *and* $\sigma_{P_2} : P_1 \rightarrow P_2$ *compose to define* $\sigma_{P_3} : P \rightarrow P_2$ *where* $\sigma_{P_3} = \sigma_{P_2} \circ \sigma_{P_1}$. *Proof of claim:*

   $\forall(p : s)(p : s \in P \ \Rightarrow\ \sigma_{P_1}(p) : \sigma_{\Sigma_1}(s) \in P_1)$ *by definition*

   $\forall(p' : s')(p' : s' \in P_1 \ \Rightarrow\ \sigma_{P_2}(p') : \sigma_{\Sigma_2}(s') \in P_2)$ *by definition*

   but $\sigma_{P_1}(p) : \sigma_{\Sigma_1}(s) \in P_1$ *implies* $\sigma_{P_2}(\sigma_{P_1}(p)) : \sigma_{\Sigma_2}(\sigma_{\Sigma_1}(s)) \in P_2$.

   $\therefore$ σ_{P_1} *and* σ_{P_2} *compose to define* $\sigma_{P_3} : P \rightarrow P_2$ *where* $\sigma_{P_3} = \sigma_{P_2} \circ \sigma_{P_1}$.

   (d) *Claim:* $\sigma_{V_1} : V \rightarrow V_1$ *and* $\sigma_{V_2} : V_1 \rightarrow V_2$ *compose to define* $\sigma_{V_3} : V \rightarrow V_3$ *where* $\sigma_{V_3} = \sigma_{V_2} \circ \sigma_{V_1}$. *Proof of claim:*

$\forall(v : s)(v : s \in V \Rightarrow \sigma_{V_1}(v) : \sigma_{\Sigma_1}(s) \in V_1)$ *by definition*

$\forall(v' : s')(v' : s' \in V_1 \Rightarrow \sigma_{V_2}(v') : \sigma_{\Sigma_2}(s') \in V_2)$ *by definition*

but $\sigma_{V_1}(v) : \sigma_{\Sigma_1}(s) \in V_1$ *implies* $\sigma_{V_2}(\sigma_{V_1}(v)) : \sigma_{\Sigma_2}(\sigma_{\Sigma_1})(s) \in V_2$.

$\therefore$ σ_{V_1} *and* σ_{V_2} *compose to define* $\sigma_{V_3} : V \rightarrow V_2$ *where* $\sigma_{V_3} = \sigma_{V_2} \circ \sigma_{V_1}$.

(e) Claim: $\sigma_{\kappa_1} : \kappa \rightarrow \kappa_1$ *and* $\sigma_{\kappa_2} : \kappa_1 \rightarrow \kappa_2$ *compose to define* $\sigma_{\kappa_3} : \kappa \rightarrow \kappa_2$ *where* $\sigma_{\kappa_3} = \sigma_{\kappa_2} \circ \sigma_{\kappa_1}$ *such that the maps* events, act, chan *and* var *are preserved.*

Proof of claim: The proof of this claim consists of two parts. The first part of the proof establishes that the functions σ_{κ_2} *and* σ_{κ_1} *compose, and the second part of the proof establishes that the composition* $\sigma_{\kappa_2} \circ \sigma_{\kappa_1}$ *preserves the maps* events, act, chan *and* var.

i. Proof that σ_{κ_2} *and* σ_{κ_1} *compose:*

$\forall(W)\ (W \in \kappa \Rightarrow \sigma_{\kappa_1}(W) \in \kappa_1)$ *by definition*

$\forall(W')\ (W' \in \kappa_1 \Rightarrow \sigma_{\kappa_2}(W') \in \kappa_2)$ *by definition*

but $\sigma_{\kappa_1}(W) \in \kappa_1$ *implies* $\sigma_{\kappa_2}(\sigma_{\kappa_1}(W)) \in \kappa_2$

$\therefore$ σ_{κ_2} *and* σ_{κ_1} *compose to define the function* $\kappa_3 : \kappa \rightarrow \kappa_2$ *where* $\kappa_3 = \sigma_{\kappa_2} \circ \sigma_{\kappa_1}$.

ii. Items 2a, 2b, 2c, 2d and 2(e)i imply that the maps events, chan, act *and* var *are preserved.*

$\therefore$ $\sigma_1 : \Pi \rightarrow \Pi_1$ *and* $\sigma_2 : \Pi_1 \rightarrow \Pi_2$ *compose to define the signature morphism* $\sigma_3 : \Pi \rightarrow \Pi_2$ *where* $\sigma_3 = \langle \sigma_{\Sigma_2} \circ \sigma_{\Sigma_1}, \sigma_{E_2} \circ \sigma_{E_1}, \sigma_{P_2} \circ \sigma_{P_1}, \sigma_{V_2} \circ \sigma_{V_1}, \sigma_{\kappa_2} \circ \sigma_{\kappa_1} \rangle$. ■

5.4.1 Summary. This section has introduced and defined the category **PSign** of process signatures and process signature morphisms. A relationship between CSP structures and process signatures was defined such that all elements of CSP structures, except for process expressions, could be represented in a process signature. Because process signatures and process signature morphisms define a category, process signatures can be grown using operations such as colimits drawn from Category Theory in much the same way that the function signatures of the category **Sign** can be grown.

Process signatures, like functional signatures, do not define behavior. Process behavior is defined within process specifications. Process specifications are defined in the next section.

5.5 The Category of Process Specifications and Process Specification Morphisms

The previous section introduced and defined process signatures and process signature morphisms, and used them to define the category **PSign**. This section uses the category **PSign** to define the category **PSpec** of process specifications and process specification morphisms. Process specifications define the behavior of the processes identified in a process signature. The relationship between process signatures and process specifications is formalized by the following definition.

Definition V.17 Process specification. Process specification. *A process specification* pSP *is a tuple* $\langle \Pi, \Xi \rangle$ *where* Π *is a process signature and* Ξ *is a collection of process expressions such that* $\Xi : \kappa \rightarrow T_{CSP}(\{\Omega, E, P, V, \kappa\})$ *maps process symbols in* κ *to CSP expressions of sort* process *such that for any process symbol* $W \in \kappa$, $\Xi(W)$ *is an expression over the symbols in* var(W), act(W), chan(W), events(W) *and* κ. $\Xi(W)$ *denotes the process expression of the process symbol* W.

1. *If* $\Xi(W)$ *is an expression written only over the symbols in* chan(W), act(W), var(W), events(W) *and* W, *then* W *is a* simple process *and* $\Xi(W)$ *is a* process description. *If* $\Xi(W)$ *includes process symbols from* κ *in addition to* W, *then* W *is a* compound process.
2. *A model of a process specification* $pSP = \langle \Pi, \Xi \rangle$ *is a* Π*-model* M *such that* $M \models \Xi(W)$ *for each* $W \in \kappa$. *The collection of all such models* M *will be denoted Mod[pSP]. The subcategory of* **Mod**[Π] *induced by Mod[pSP] will also be denoted by Mod[pSP].*
3. *The notation* W sat <expression> *will be used to denote* $\Xi(W)$, *where* expression *is a well-formed expression of sort* process *in the term algebra of* CSP_A. □

Note that the use of the term *sat* in the above definition differs from its use in (52). In Hoare's text, process specifications are relations defined over the set of traces of a

process. For example, using Hoare's notation, a process W with alphabet $\{a, b\}$ could have a specification of W *sat* $0 \leq ((tr \downarrow a) - (tr \downarrow b)) \leq 1$ where tr is an arbitrary trace of W, and $tr \downarrow b$ is the count of the number of occurrences of the event b in the trace tr. In this case, the specification requires that the number of occurrences of the events a and b in any trace differ by at most one. Several distinct processes satisfying this specification could be defined, among them are $W = STOP$, $W = (a \xrightarrow{CSP} (b \xrightarrow{CSP} W))$, and $W = a \xrightarrow{CSP} (b \xrightarrow{CSP} STOP)$.

Hoare uses the *sat* construct to define constraints that a model of a process must satisfy. For example, a process W that accepts the language $a^n b^n$ could be specified by Hoare as W *sat* $0 \leq ((tr \downarrow a) - (tr \downarrow b)) \leq 1$, where tr is an arbitrary trace of W. In contrast, the use of the term *sat* in Definition V.17 associates a process symbol with an expression *defining* a process. Thus from the CSP_A statement W *sat* $(a \xrightarrow{CSP} (b \xrightarrow{CSP} W))$, it could be shown that $0 \leq ((tr \downarrow a) - (tr \downarrow b)) \leq 1$ for any trace $tr \in traces(W)$. Hoare uses the term *sat* to denote constraints, while the term *sat* is used in ISlang to denote process expressions.

A simple process specification is shown in Figure 5.7. The figure extends the process signature of Figure 5.5 by adding a process expression defining the process *simple*. Process *simple* accepts input on the channel *input* and determines if the input value satisfies the input condition. If the input condition is satisfied, then *simple* communicates the value of $f(x)$ on channel *output*, waits for a an acknowledgment over channel *handshake*, and then repeats. If the input condition is not satisfied, then *simple* communicates the event **undefined** over the output channel, and then repeats.

Definition V.17 of process specifications includes a reference to a satisfaction relation $\models$ between process specifications and process models. This satisfaction relation has not yet been defined. For the category **Spec** of functional specifications, the satisfaction relation was defined to be logical implication. As given below, the satisfaction relation of **PSpec** is based on the trace semantic defined in Section 5.3.

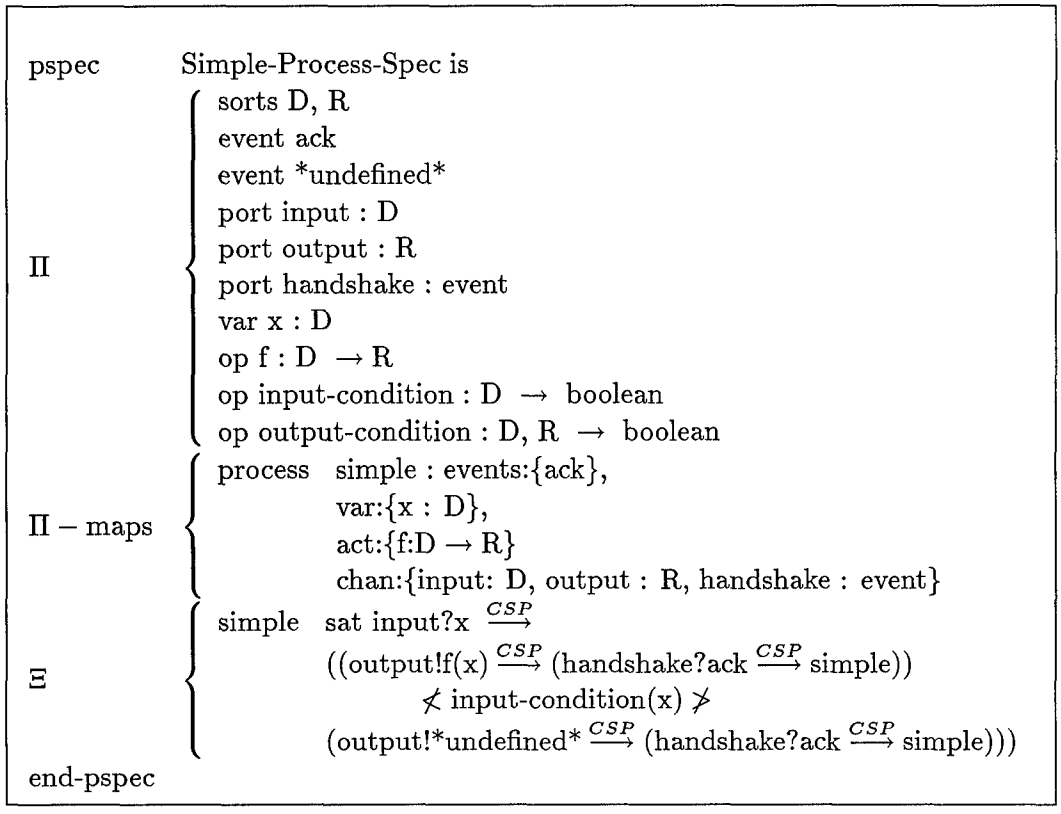


Figure 5.7 A Simple Process Specification

Definition V.18 Satisfaction. *Given two CSP expressions P and Q, $P \models Q$ if and only if* traces(P) $\subseteq$ traces (Q)$\restriction \alpha$ P. *That is, Q can do at least that required of P and maybe more.*

If $P \models Q$ and $Q \models P$ then P and Q are trace equivalent. *The fact that P and Q are trace equivalent will be denoted $P \equiv_T Q$.* □

This definition parallels the definition of *trace semantics* found in (118), and is a relatively weak type of satisfaction. It is a weaker semantic than the trace semantic with refusal sets defined by Hoare.

Trace semantics cannot be used to distinguish between distinct processes that happen to have a common set of traces yet are behaviorally quite different. For example the process expressions P *sat* $a \xrightarrow{CSP} (b \xrightarrow{CSP} P \mid c \xrightarrow{CSP} P)$ and W *sat* $(a \xrightarrow{CSP} (c \xrightarrow{CSP} W)) \sqcap (a \xrightarrow{CSP} (b \xrightarrow{CSP} W))$ are behaviorally distinct, but are trace equivalent; a model of W can choose to deadlock on the sequence $\langle a, b \rangle$ where a model of P will not. Other more powerful forms

of satisfaction (or equivalence) have been defined, such as *observational equivalence* and *bisimulation equivalence* which can distinguish between processes that are trace equivalent. See (118) for an overview of various semantics for process expressions.

The above definition of satisfaction permits trivial terminal models for process specifications. This fact is expressed in the following theorem.

Theorem V.4 *For any process P,* STOP $\models$ P.

Proof. *Because* traces(STOP) $= \{\langle\rangle\}$ *and because* $\{\langle\rangle\} \in traces(P)$ *for any process P, we get* traces(STOP) $\subseteq$ traces(P). ■

Note that the above theorem allows the trivial terminal model as a valid model of any process expression. That is, the process that does nothing is a valid terminal model of any process expression. Goguen recognized a similar problem with his work on Larch, and adopted a "no collapse" rule to address it. That is, he disallows a terminal model as an implementation of a specification unless the terminal model is the only model of that specification.

The satisfaction relation $\models$ of Definition V.18 has a number of important properties. These properties are stated in the following theorem.

Theorem V.5 *The relation* $\models$ *between process expressions has the following properties:*

1. *For any process expression A,* $A \models A$. *That is,* $\models$ *is reflexive.*
2. *For any two process expressions A and B, if* $A \models B$ *and* $B \models A$ *then* $A \equiv_T B$. *That is,* $\models$ *is antisymmetric.*
3. *For any three process expressions A, B, and C, if* $A \models B$ *and* $B \models C$, *then* $A \models C$. *That is,* $\models$ *is transitive.*

Proof.

1. *Proof of 1. For any process expression A,* traces(A) $\subseteq$ traces(A)$\uparrow \alpha$A *which implies* $A \models A$.
2. *Proof of 2. Follows from the definition of* $\equiv_T$.

3. *Proof of 3.* A $\models$ B $\Rightarrow$ traces(A) $\subseteq$ traces(B)$\restriction \alpha$A, *and* B $\models$ C $\Rightarrow$ traces(B) $\subseteq$ traces(C)$\restriction \alpha$B. *Substituting, we get* traces(A) $\subseteq$ (traces(C)$\restriction \alpha$B)$\restriction \alpha$A. *However,* traces(A) $\subseteq$ traces(B)$\restriction \alpha$A *implies* $\alpha A \subseteq \alpha B$. *So* (traces(C)$\restriction \alpha$B)$\restriction \alpha$A *simplifies to* traces(C)$\restriction \alpha$A *which yields* traces(A) $\subseteq$ traces(C)$\restriction \alpha$A. ■

Note that because $\models$ is reflexive, antisymmetric, and transitive, $\models$ defines a partial ordering over process expressions. This is important for it permits an investigation of the relative expressive power of process-based architecture theories.

As expressed in the following theorem, the satisfaction relation of Definition V.18 preserves models.

Theorem V.6 *Given two process expressions A and B in the term algebra of* CSP_A, *if* $A \models B$ *then* m $\in$ Mod[A] $\Rightarrow$ m $\in$ Mod[B].

Proof. *If m is a model of A, i.e., if* m $\in$ Mod[A], *then by the definition of model,* $m \models A$, *which implies* traces(m) $\subseteq$ traces(A)$\restriction \alpha$A. *Because* $A \models B$, *we get* traces(A) $\subseteq$ traces(B)$\restriction \alpha$A. *Substituting,* traces(m) $\subseteq$ (traces(B)$\restriction \alpha$A)$\restriction \alpha$m. *But* traces(m) $\subseteq$ traces(A)$\restriction \alpha$A *implies* $\alpha m \subseteq \alpha A$. *So* traces(m) $\subseteq$ (traces(B)$\restriction \alpha$A)$\restriction \alpha$m *simplifies to* traces(m) $\subseteq$ traces(B)$\restriction \alpha$m), *which implies* m $\in$ Mod[B]. ■

Now that process specifications and satisfaction between process specifications have been defined, process specification morphisms can be defined.

Definition V.19 Process specification morphisms. *A process specification morphism from a process specification* $pSP = \langle \Pi, \Xi \rangle$ *with* $\Pi = \langle \Sigma, E, P, V, \kappa \rangle$ *to a process specification* $pSP' = \langle \Pi', \Xi' \rangle$ *with* $\Pi' = \langle \Sigma', E', P', V', \kappa' \rangle$, *is a process signature morphism* $\sigma : \Pi \rightarrow \Pi'$ *such that for every model* $M \in Mod[pSP']$ *we have* $M|_\sigma \in Mod[pSP]$. *The process specification morphism is also denoted by the same symbol,* $\sigma : pSP \rightarrow pSP'$. □

Consider a process signature morphism σ $pSP \rightarrow pSP'$. If the process symbol W has a process expression $\Xi(W)$ associated with it in pSP, then when the symbols in $\Xi(W)$ are translated according to the mapping defined by σ, the traces of the translated expression $\sigma(\Xi(W))$ restricted to the alphabet of W, must be a superset of *traces*$(\Xi(W))$ if σ is to be

specification morphism. That is, σ is a specification morphism if σ is a process signature morphism such that for every process symbol $W \in \kappa$, $traces(\Xi(W)) \subseteq traces(\sigma(\Xi(W)))\,|_{\sigma}$.

In the higher order logic of SpecWare, a signature morphism is also a specification morphism if the translated axioms are theorems in the target specification. In the process logic of **PSpec**, a process signature morphism is a process specification morphism if the processes defined by the target specification can do at least that required of the processes defined by the source specification.

As expressed in the following theorem, process specifications and process specification morphisms form a category.

Theorem V.7 *Process specifications and process specification morphisms form a category* **PSpec** *where process specifications are the C-objects and process specification morphisms are the C-arrows.*

Proof. *There are two properties that must be shown:*

1. *That each C-object has an identity morphism, and*
2. *That C-arrows compose to form C-arrows.*

Each of these is proved below.

1. *Identity. Exists by definition of process signature morphism.*
2. *Composition. Let W_1, W_2, and W_3 be three process specifications, and let $\sigma_{12} : W_1 \rightarrow W_2$ be a process specification morphism from W_1 to W_2, and let $\sigma_{23} : W_2 \rightarrow W_3$ be a process specification morphism from W_2 to W_3. We need to show that $\sigma_{23} \circ \sigma_{12}$ is a process specification morphism. By Theorem V.3, process signature morphisms compose to form process signature morphisms. So σ_{23} and σ_{12} compose to form the process signature morphism $\sigma_{13} = \sigma_{23} \circ \sigma_{12}$.*

   By the definition of process specification morphism, $\sigma_{13} : W_1 \rightarrow W_3$ is a specification morphism if and only if $M' \in \mathrm{Mod}[W_3] \Rightarrow M'\,|_{\sigma_{13}} \in \mathrm{Mod}[W_1]$. Because $\sigma_{12} : W_1 \rightarrow W_2$ is a specification morphism, $\mathrm{M} \in \mathrm{Mod}[W_2] \Rightarrow \mathrm{M}|_{\sigma_{12}} \in \mathrm{Mod}[W_1]$. Similarly, because $\sigma_{23} : W_2 \rightarrow W_3$ is a specification morphism, $M' \in \mathrm{Mod}[W_3] \Rightarrow M'\,|_{\sigma_{23}} \in \mathrm{Mod}[W_2]$.

But $M' \mid_{\sigma_{23}} \in \text{Mod}[W_2] \Rightarrow (M' \mid_{\sigma_{23}}) \mid_{\sigma_{12}} \in \text{Mod}[W_1]$. *Because signature morphisms compose,* $(M' \mid_{\sigma_{23}}) \mid_{\sigma_{12}} = M' \mid_{\sigma_{23} \circ \sigma_{12}}$. *But* $\sigma_{23} \circ \sigma_{12} = \sigma_{13}$, *so* $M' \mid_{\sigma_{23} \circ \sigma_{12}} \in \text{Mod}[W_1] \Rightarrow M' \mid_{\sigma_{13}} \in \text{Mod}[W_1]$, *which implies that* $\sigma_{13} = \sigma_{23} \circ \sigma_{12}$ *is a specification morphism.* ■

The above theorem allows process specifications to be grown in the same manner that functional specifications are grown. Like their functional counterparts, process specifications can be parameterized. For example, a process specification defining a client-server architecture could be defined, where the process expressions in the specification define the minimum required features of the server process and the client processes. This type of parameterization is described in more detail in Chapter VI.

Although process specification morphisms require the preservation of models under a trace semantic equivalence, other constraints could be placed on the development of process specifications in an effort to prevent the specification of degenerate or unrealizable processes. These constraints could take one of the following two forms:

1. Constraints expressed over the use of functional operations within a process specification. For example, Basic Problem Theory specifications of Chapter III use boolean operations to characterize the range of acceptable input values and to characterize the output condition for an abstractly defined operation. Suppose f and g are operations defined using Problem Theory specifications, where I_f, I_g, O_f and O_g denote the input and output conditions of f and g, respectively. If the structure of the application is such that f is an actual parameter of g as in $g \circ f$, then I_g is satisfied if $O_f \Rightarrow I_g$. Any specification construction that violates this constraint could be flagged as an invalid construction. The expression $O_f \Rightarrow I_g$ defines a constraint between the specifications of f and g.

   This constraint could be generalized as follows. Denote by g an operation defined using Problem Theory, and denote by $F = \{f_1, f_2, \ldots, f_n\}$ a collection of operations defined using Problem Theory such that each f in F is used as an actual parameter of g. Then the constraint becomes $\bigwedge_{f \in F} O_f \Rightarrow I_g$, where $\bigwedge_{f \in F} O_f$ denotes the conjunction of the output conditions of the operations in F.

2. Constraints expressed over process expressions. For example, process expressions should be free of both deadlock and live-lock.

Appendix D addresses constraints in greater detail.

5.6 Relationship Between Functional and Process-Based Specifications

As shown in Figure 5.1, signatures of functional specifications are mapped to process specifications for use in process expressions. The semantics of these sort and operator symbols are provided by the axioms of their associated functional specifications. In essence, the many-to-many relationship depicted in Figure 5.1 establishes a relationship between the semantics of sort symbols and operator symbols and their use in process specifications.

The relationship between ISlang and SLANG specifications is defined through *components.* Appendix C presents an informal treatment of components, where the definition of components was based on the results of the experiments described in Chapter IV. Now that the mathematical foundations of process specifications have been established, a more formal treatment of components may be presented as follows:

1. Sub-section 5.6.1 defines components, and describes how components are used to bridge the gap between the process logic of ISlang and the functional logic of SLANG; and
2. Sub-section 5.6.2 defines a category of components and component morphisms.

5.6.1 Components. The higher order logic of **Spec** and the process logic of **PSpec** share a common core of mathematical principles, such as equivalence. Boolean algebra is also common to both logic systems. The higher order predicate calculus of SpecWare and the process logic of **PSpec** share common mathematical principles, and they share a common set of symbols (in the form of operator and sort symbols) such that the semantic interpretation of the symbols is consistent between the logics. Because of this, expressions formed over the common symbols and common concepts can be exchanged between the logics. For example, the statement that sort symbols X and Y belong to the

same equivalence class has the same interpretation in the logic of ISlang as it does in the logic of SLANG.

Denote by C a mathematical concept shared between the logics of SLANG and ISlang, and denote by $\bar{X}$ a sequence of symbols shared between a SLANG and an ISlang specification such that $C(\bar{X})$ is a well-formed expression. For example, suppose C denoted the concept of sort equivalence, and suppose x and y were two sort symbols referenced in an ISlang specification and defined in a SLANG specification. Then $C(\bar{X})$ would in this case be the predicate $x \cong y$. If $C(\bar{X})$ is generated within the logic of ISlang, then the logic system of SLANG can be used to evaluate the expression. A *proof schema* can be used to define a homomorphism from ISlang to SLANG over the common symbols and concepts such that the proof mechanism of SLANG can be used to evaluate the expression. Such capability is important in that ISlang lacks sort and operator axioms, and as such is unable to answer semantic questions posed over those structures.

Invocation of the SpecWare theorem prover over a conjecture lacking axioms is of little value. Axioms are included in SLANG specifications, but are lost when just the signatures of the specifications are mapped into ISlang specifications. What is needed is a structure that "remembers" the source of the SLANG signatures used in an ISlang specification. This structure is the *component.*

Components include the structure necessary to associate a functional signature of an ISlang specification with the functional specification from which it was drawn. Thus expressions over common concepts and common symbols can be generated in ISlang, paired with axioms from SLANG specifications, proved within the logical system of SLANG, and the results of the proof, in the form of Boolean *true* or *false*, can be interpreted in ISlang. This relationship is graphically depicted in Figure 5.8. In the figure, C is a common mathematical concept such as equivalence or implication, X is a sequence of sort and operator symbols common between SLANG and ISlang, and $C(X)$ is a well-formed expression over X and concept C. Additional knowledge, represented by K, has been added to the expression by associating the symbols X with the SLANG specification in which they are defined.

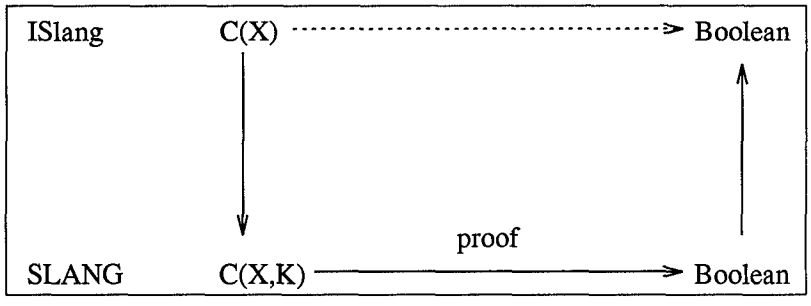


Figure 5.8 Proof Schemas

As described earlier in this section, proof obligations resulting from consistency constraints levied between process specifications and functional specifications require a means to identify the functional specifications from which the process specifications draw their operator and sort definitions. Components contain the structure necessary to provide this identification.

Definition V.20 Component. *A component C is a 5-tuple $\langle \mathcal{D}, S_D, \mathcal{I}, S_I, i \rangle$ where $\mathcal{D}$ is a diagram of functional specifications, S_D is a functional specification such that $S_D \in \mathcal{D}$, $\mathcal{I}$ is a diagram of process specifications, S_I is a process specification such that $S_I \in \mathcal{I}$, and i is a functor from the category* **Sign** *of functional signatures to the category* **PSign** *of process signatures such that $S_I \mid_i$ is isomorphic to the signature of S_D.* □

Every operator symbol or sort symbol referenced in a process specification of a component is defined in the functional specification of the component.

A simple component is shown in Figure 5.9. On the left side of the figure is a functional specification, *Sort-Spec*, defining the problem of sorting a bag of integers. The signature of *Sort-Spec* is mapped via the functor i to the process specification *Sort-Process*. *Sort-Process* defines a single process. The process, *Sort*, reads a bag of integers over the input port *in*, and if the input value x satisfies the input condition, *Sort* outputs the value *sort(x)*. If the input value does not satisfy the input condition, then *Sort* is defined to output the event symbol **undefined**.

The requirement that for any component $C = \langle \mathcal{D}, S_{\mathcal{D}}, \mathcal{I}, S_{\mathcal{I}}, i \rangle$, $S_{\mathcal{I}} \mid_i$ be isomorphic to the signature of $S_{\mathcal{D}}$ ensures that the set of sorts and operator symbols referenced in the process specification $S_{\mathcal{I}}$ are defined in the functional specification of the component.

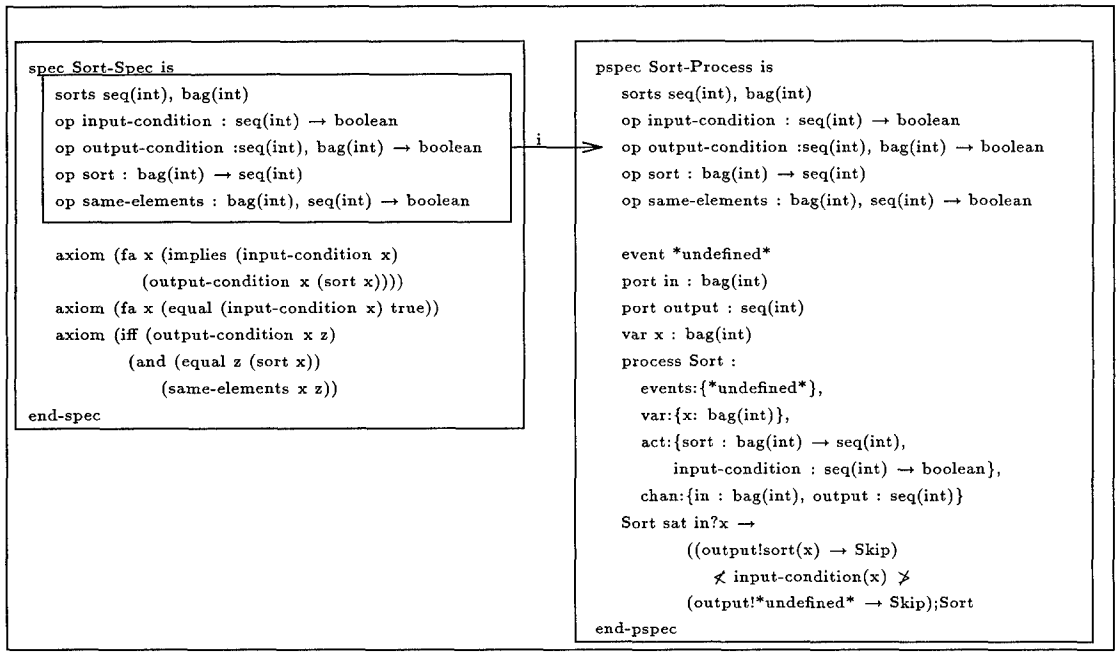


Figure 5.9 A Simple Component

Note however that there is no requirement that the two specifications share models of the sorts and operations. That is, given a component C, there is no requirement that models of $S_{\mathcal{D}}$ be submodels of $S_{\mathcal{I}}$. The relationship between models of functional specifications and models of the sorts and operations of process specifications is formalized in the next section.

Not only do components contain the structure necessary to associate models of functional specifications with models of process specifications, components also contain the structure necessary to define the proof schemas used to maintain consistency between developing functional and process specifications. For example, components permit expression of conjectures involving input condition satisfaction. That is, if the operations $f : u \rightarrow v$ and $g : v \rightarrow w$ are defined using Problem Theory, the condition $O_f \Rightarrow I_g$ must hold if f is to be used as an actual parameter of g as in $g \circ f$. The following definition formally expresses this condition.

Definition V.21 Consistency. *Denote by C a component $\langle \mathcal{D}, S_D, \mathcal{I}, S_I, i \rangle$ such that S_I contains a process expression P where $f : s_1, s_2, \ldots, s_n \rightarrow s$ is an operation defined by a problem specification such that f is used as an argument in an output event $c!f(x_1, x_2, \ldots, x_n)$*

in P. Let $c_i?x_i, 1 \leq i \leq n$ *be a collection of input communication events in P preceding* $c!f(x_1, x_2, \ldots, x_n)$ *in any trace* t $\in$ traces(P), *and denote by* P_i *a collection of process expressions in* S_I *such that* c_i *is a port symbol in* P_i *where* $c_i!h_i(y_i)$ *is an output event in* traces (P_i).

If every h_i *has a defined boolean valued output condition* O_{h_i}, *then* S_I *is consistent with respect to* f *in* P *if, for all occurrences of* f *in every trace of* P,

$$\bigwedge_{i=1..n} O(y_i, h_i(y_i)) \Rightarrow I_f(x_1, x_2, \ldots, x_n) \tag{5.1}$$

where $h_i(y_i) = x_i$. □

Equation 5.1 defines the satisfaction of the input condition of an operation $f : u \rightarrow w$ by the output conditions of the operations used to supply values for the arguments of f. The structure of components allows elaboration of this equation. Specifically, input conditions and output conditions of operations can be found algorithmically through exploration of the diagram of functional specifications. The diagram of functional specifications $\mathcal{D}$ of a component contains the information describing if and how operations of the specification $S_D \in \mathcal{D}$ are defined in terms of Problem Theory. That is, $\mathcal{D}$ contains the information necessary to extract the input and output conditions (if any) of the operations in S_D, while the specification S_I contains the information necessary to determine the flow of data between the operations. This information can then be used to elaborate Equation 5.1. Although it is claimed that such an algorithm could be defined, no attempt is made here to define it. The definition of an algorithm used to ensure input condition satisfaction is left for further research.

5.6.2 The Category **App**. This section defines the category **App** of components and component morphisms. Application specifications involving process specifications and functional specifications can be developed within this category. Definition of the category **App** further formalizes the relationship between functional specifications and process specifications shown in Figure 5.1.

App is defined by first defining what is meant by a model of a component. As expressed in the following definition, models of components are restricted to those that have consistently implemented event signatures.

Definition V.22 Component Models. *A model of a component* $C = \langle \mathcal{D}, S_D, \mathcal{I}, S_I, i \rangle$ *consists of a model* $m \in Mod[S_D]$ *of the sorts an operations of* S_D*, along with a model* $m' \in Mod[S_I]$ *of the process specification* S_I *such that* $m' \mid_i = m$. □

Sharing models between functional specifications and the sorts and operations of process specifications simplifies the task of mapping the specification languages to an implementation language, and simplifies the task of ensuring the sorts and operations are treated consistently within a component definition.

Now that models of a component have been defined, component morphisms can be defined.

Definition V.23 Component Morphisms. *A component morphism from a component* $C = \langle \mathcal{D}, S_D, \mathcal{I}, S_I, i \rangle$ *to a component* $C' = \langle \mathcal{D}', S'_D, \mathcal{I}', S'_I, i' \rangle$ *is a 4-tuple of morphisms*

$$\langle \sigma_{\mathcal{D}}, \sigma_{S_D}, \pi_{\mathcal{I}}, \pi_{S_I} \rangle$$

where

1. $\sigma_{\mathcal{D}} : \mathcal{D} \rightarrow \mathcal{D}'$ *is a total, covariant functor from the diagram* $\mathcal{D}$ *to the diagram* $\mathcal{D}'$ *such that if* $\sigma_{\mathcal{D}}(SP) = SP'$ *for any specifications* $SP' \in \mathcal{D}'$ *and* SP $\in \mathcal{D}$*, then if* m $\in$ Mod[SP'] *we have* $m \mid_{\sigma_{\mathcal{D}}} \in$ Mod[SP].
2. $\sigma_{S_D} : S_D \rightarrow S'_D$ *is a functional specification morphism.*
3. $\pi_{\mathcal{I}} : \mathcal{I} \rightarrow \mathcal{I}'$ *is a total, covariant functor from the diagram* $\mathcal{I}$ *to the diagram* $\mathcal{I}'$ *such that if* $\pi_{\mathcal{I}}(SP) = SP'$ *for any specifications* $SP' \in \mathcal{I}'$ *and* SP $\in \mathcal{I}$*, then if* m $\in$ Mod[SP'] *we have* $m \mid_{\pi_{\mathcal{I}}} \in$ Mod[SP].
4. $\pi_{S_I} : S_I \rightarrow S'_I$ *is a process specification morphism.*
5. *For all operations* $f : s_1, s_2, \ldots, s_n \rightarrow s$ *in* Ω *of* S_D *where* $f : s_1, s_2, \ldots, s_n \rightarrow s$ *is in the domain of* i, $i'(\sigma_{S_D}(f : s_1, s_2, \ldots, s_n \rightarrow s)) = \pi_{S_I}(i(f : s_1, s_2, \ldots, s_n \rightarrow s))$.

The identity component morphism takes each component structure back to itself. □

Thus component morphisms include mappings for each of the four elements of a component. Although the mappings $\sigma_{\mathcal{D}}$ and $\sigma_{\mathcal{I}}$ may be complex, it is anticipated that in practice they will be defined by the identity map. More complex mappings, such as those created when defining an interpretation of one component to another, may be defined.

Now that component models and component morphisms have been defined, the category **App** can be established.

Theorem V.8 *Components and component morphisms form a category* **App** *where components are the C-objects and component-morphisms are the C-arrows.*

Proof. *There are two parts to the proof, the first part is to prove the existence of an identity arrow for each object, and the second part is to prove that component morphisms compose to form component morphisms.*

1. *Identity component morphisms exist by definition.*
2. *Given two component morphisms* $M_{1,2} : C_1 \rightarrow C_2$ *and* $M_{2,3} : C_2 \rightarrow C_3$ *where* C_1, C_2 *and* C_3 *are components with* $M_1 = \langle \sigma_{\mathcal{D}}, \sigma_{S_D}, \pi_{\mathcal{I}}, \pi_{S_I} \rangle$ *and* $M_2 = \langle \sigma'_{\mathcal{D}}, \sigma'_{S_D}, \pi'_{\mathcal{I}}, \pi'_{S_I} \rangle$, *we need to show* $M_{2,3} \circ M_{1,2}$ *composes to form a component morphism* $M_{1,3}$. *This proof is shown in two parts. The first part establishes the composition of the functors contained in component morphisms.*

   (a) *Total covariant functors compose to form total covariant functors. Thus the functors* $\sigma_{\mathcal{D}} : \mathcal{D}_1 \rightarrow \mathcal{D}_2$ *and* $\sigma_{\mathcal{D}'} : \mathcal{D}_2 \rightarrow \mathcal{D}_3$ *compose to form the functor* $\sigma_{\mathcal{D}_3} : \mathcal{D}_1 \rightarrow \mathcal{D}_3 = \sigma_{\mathcal{D}'} \circ \sigma_{\mathcal{D}}$. *Similarly, the functors* $\pi_{\mathcal{I}} : \mathcal{I}_1 \rightarrow \mathcal{I}_2$ *and* $\pi_{\mathcal{I}'} : \mathcal{I}_2 \rightarrow \mathcal{I}_3$ *compose to form the functor* $\pi_{\mathcal{I}_3} : \mathcal{I}_1 \rightarrow \mathcal{I}_3$.

   (b) *Because process specification morphisms compose to form process specification morphisms and functional specification morphisms compose to form functional specification morphisms, we have* $\sigma'_{S_D} \circ \sigma_{S_D}$ *is a functional specification morphism and* $\pi'_{S_I} \circ \pi_{S_I}$ *is a process specification morphism.*

   Thus because total contravariant functors compose to form total contravariant functors and because specification morphisms compose to form specification morphisms,

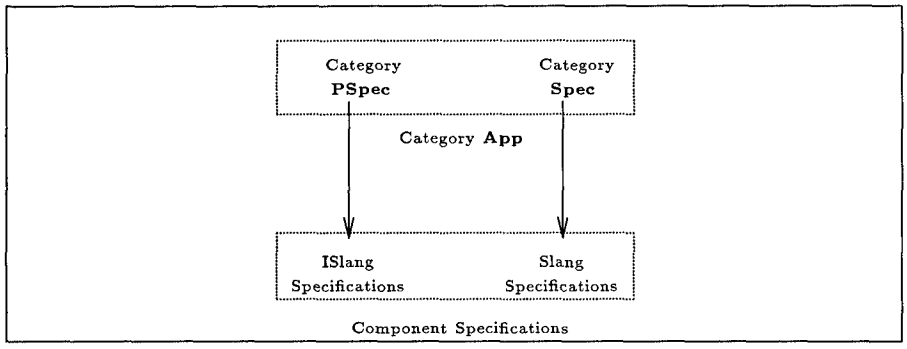


Figure 5.10 Mathematical Summary

we can conclude that component morphisms compose to form component morphisms. That is, $M_{1,3} = M_{2,3} \circ M_{1,2}$, *where* $M_{1,3} = \langle \sigma'_{\mathcal{D}} \circ \sigma_{\mathcal{D}}, \sigma'_{S_D} \circ \sigma_{S_D}, \pi'_{\mathcal{I}} \circ \pi_{\mathcal{I}}, \pi'_{S_I} \circ \pi_{S_I} \rangle$. ■

The category **App** formally establishes the relationship between functional specifications and process specifications of Figure 5.1. As shown in Figure 5.10, the categories **Spec** and **Spec** are complete sub-categories of **App**.

5.7 Summary

This chapter established the mathematical foundations of the specification development system described in Chapter II and shown in Figure 2.1. The mathematical foundations were established through the definition of several related categories. Figure 5.10 depicts an overview of these categories and the relationships between them. **Spec** was defined in Chapter III to be a category of functional specifications.

The primary focus of this chapter was on the development of a category of process specifications, **PSpec**. Hoare's definition of communicating sequential processes (CSP) was used in the definition of process expressions. However, before a category of process specifications could be defined, Hoare's CSP needed to be expressed as a theory presentation using a process logic. Section 5.2 developed such a theory presentation. A trace semantic for expressions in this theory presentation, denoted CSP_A, was developed by relating CSP_A to formal language theory through CSP structures. It was shown in Theorem V.2 that CSP automata have the generative power of Turing machines, which implies that any computable architecture can be defined using CSP structures.

After presenting Hoare's CSP as a theory presentation, a category **PSpec** of process specifications and process specification morphisms was defined. The satisfaction relation of the category **PSpec** was defined using a trace semantic; specifications in **PSpec** define communicating sequential processes. This combination of category theory and the theory of communicating sequential processes as defined by Hoare allows process-based specifications to be grown from other process-based specifications in a manner similar to the way functional specifications are grown from other functional specifications.

Finally, a formal relationship in the form of components was established between functional specifications and process specifications. Components define a relationship between ISlang specifications and SLANG specifications such that the sorts and operations of an ISlang specification are defined in an associated SLANG specification. It was shown that components and component morphisms define a category **App**. The following chapter builds on the mathematical foundations established in this chapter by providing a formal definition of architecture using functional specifications, process specifications, and components.

VI. Software Architecture

6.1 Introduction

The preceding chapters of introduced the concept of architecture and introduced mechanisms for defining formal specifications for both stateless and state bearing entities. Although the treatment of functional and process based specifications has been formal, the treatment of architecture has thus far been informal. This chapter presents a formal treatment of architecture. Specifically, this chapter develops a formal definition of *architecture theory*, as well as three classes of architecture theories:

1. Functional-based, where operations defined in functional specification are composed into well defined structures;
2. Process-based, where process specifications are used to define the architecture; and
3. Component-based, where the fundamental building blocks of the architecture are components.

Section 6.2 contains a definition of both architecture and design, and defines three different architecture theories. Section 6.3 defines several process based architecture theories, including pipelined and layered architecture theories. A taxonomy of process-based

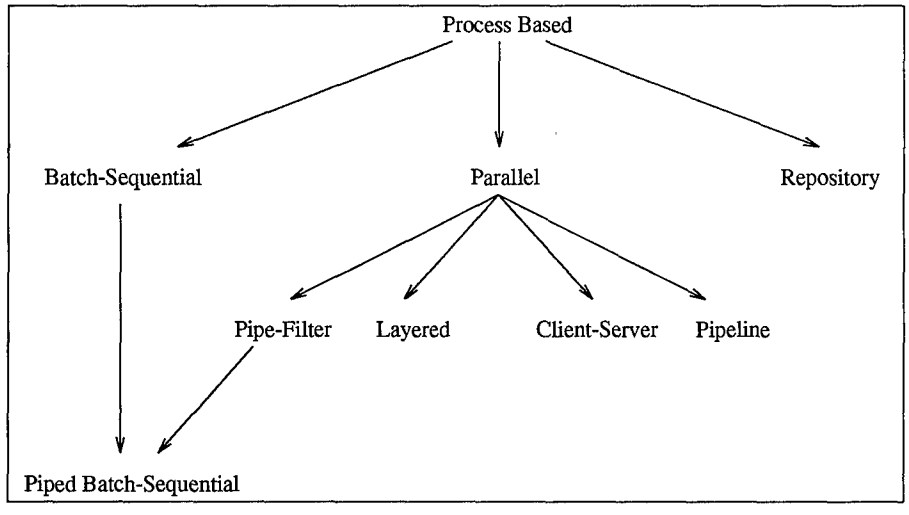


Figure 6.1 Architecture Taxonomy

architecture theories is also developed. Included in Section 6.3 are a few simple examples of using architecture theories in the construction of application specifications.

6.2 Architecture Defined

As pointed out in Chapter I, different authors have different definitions of architecture. A variety of authors— most notably Garlan and Shaw (e.g., (94, 93, 39, 6) and (37)) — have attempted to define software architecture, but their definitions are informal. Some example definitions include the following:

- "In a pipe and filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order. This is usually accomplished by applying a local transformation to the input streams and computing incrementally so output begins before input is consumed. Hence components are termed 'filters'. The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to the inputs of another. Hence the connectors are termed 'pipes'. ... A degenerate case of a pipeline architecture occurs when each filter processes all of its input data as a single entity. In this case the architecture becomes a 'batch-sequential' system."(38:5)
- "A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below."(38:9)
- "A *batch transformation* is a sequential input-to-output transformation, in which inputs are supplied at the start, and the goal is to compute an answer; there is no ongoing interaction with the outside world."(90:212)
- "A *continuous transformation* is a system in which the outputs actively depend on changing inputs and must be periodically updated. Unlike a batch transformation, in which the outputs are computed only once, the outputs in an active pipeline must be updated frequently (in theory continuously, although in practice they are computed discretely at a fine time scale). ...Typical applications include signal processing, windowing systems, incremental compilers, and process monitoring systems."(90:213)

An architecture theory identifies the building blocks of structure and defines how these building blocks may be assembled to define more complex structures. The semantics of an architecture theory are provided by a set of rules used to both interpret the meaning of structures and to identify equivalent or included sub-structures. As a simple example, an architecture theory could be defined wherein the objects are bricks and mortar, and the composition rules define how bricks and mortar may be assembled to define structures such as arches, floors, walls, or bridges. Such an architecture theory includes rules defining, for example, the load bearing capacity of various types of arches. These rules provide the semantics of the architecture, and can be used to determine if a given assembly of bricks and mortar is equivalent in some way — such as load bearing per unit area — to other assemblies where the notion of equivalence may be context dependent.

In the bricks and mortar example, various types of arches can be defined. Although different types of arches share many of the same characteristics, they each have their own unique structural qualities which manifest themselves in either their aesthetics or in their load bearing capability or both. An architecture theory identifying these structures contains rules for determining if a given collection of bricks and mortar form an arch as well as rules defining how arches can be used to define other structures such as bridges. Additionally, it includes rules for interpreting how one arch-structure is equivalent to another. Although this simple example contained physical real world objects, other, more abstract architecture theories can be defined.

All architecture theories have a common theme: they identify fundamental building blocks, they define operations or rules for creating structure, and they define the semantics of those structures. These notions are formalized in the following definition.

Definition VI.1 Architecture Theory. *An architecture theory is a 3-tuple* $\langle O, R, \models \rangle$ *where* O *is a collection of objects,* R *is a collection of relations defined over* O *such that* R *defines the syntax of object composition, and* $\models$ *is a satisfaction relation between* O*-sentences and* O*-models such that* $\models$ *defines a well order.* □

A related concept is that of *design.*

Definition VI.2 Design. *A* design *is a well-formed sentence in an architecture theory.* □

Definition VI.1 is somewhat abstract in that it permits a wide variety of entities to be classified as architecture theories. For example, the category **Spec** of functional specifications written in the higher-order logic of SpecWare can be viewed as an architecture theory:

1. The objects of the theory are functional specifications;
2. The set R of composition relations define signature and specification morphisms;
3. The satisfaction relation $\models$ defines the relationship between models of functional specifications and the specifications themselves. Note that the definition of model allows $\models$ to be extended for use between specifications. That is, $SP \models SP' \Leftrightarrow \forall(m)(m \in Mod[SP] \Rightarrow m \in Mod[SP'])$.

An architecture theory for functional specifications defines how functional specifications can be built out of other functional specifications, and it defines how to interpret the specification diagrams which form the structures of the architecture theory. A more interesting architecture theory based on functional specifications uses the operations of functional specifications as the objects and uses higher order operations to define structure. This architecture theory is defined next.

6.2.1 Functional Architecture Theory. Functional specifications may include the signatures and definitions of multiple operations. For example, a functional specification may contain signatures for the operations $f_{u,v}$, $g_{x,u}$ and $h_{x,v}$. Suppose a definition of h in terms of f and g is desired. That is, suppose an axiom such as $h = f \circ g$ defining h to be the composition of f and g is desired. Such an axiom can be used to define the structure of h as follows. If $F : A_{u_1}, A_{u_2}, \ldots, A_{u_m} \rightarrow A_v$ is a model of the operation $f_{u,v}$ and $G : A_{x_1}, A_{x_2}, \ldots, A_{x_n} \rightarrow A_u$ is a model of the operation $g_{x,u}$ where $A_u = A_{u_1} \times A_{u_2} \times \cdots \times A_{u_m}$, then $F \circ G : A_{x_1}, A_{x_2}, \ldots, A_{x_n} \rightarrow A_v$ is a model of the operation $h_{x,v} = f_{u,v} \circ g_{x,u}$.

The relation $\circ$ is a higher order operation used to define structure; as such, $\circ$ could be viewed as a composition operator of an architecture theory whose objects are functional operations. An architecture theory whose objects are functional operations and whose composition relations are defined over functional operations is presented below.

Definition VI.3 Functional Architecture Theory. *A functional architecture theory (FAT) is a 2-tuple* $\langle \mathcal{D}, A_F \rangle$ *where* $\mathcal{D}$ *is a diagram of functional specifications and* A_F *is an architecture theory* $\langle O, R, \models \rangle$ *where*

1. O *is a collection of operators of the form* $f : s_1, s_2, \ldots s_n \rightarrow s$ *contained in and defined by a functional specification* $SP = \langle \Sigma, \Phi \rangle$ *in* $\mathcal{D}$, *where* $\Sigma = \langle \Omega, S \rangle$ *such that* $O \subseteq \Omega$ *(i.e.,* $f : s_1, s_2, \ldots, s_n \rightarrow s \in O \Rightarrow f : s_1, s_2, \ldots, s_n \rightarrow s \in \Omega$*);*

2. R *consists of the higher order operator* $\_ \circ \_$: operator, operator $\rightarrow$ operator *defining operator composition, (i.e.,* $f \circ g(x) = f(g(x))$*) and the higher order operator* $\_ \cdot \_$: operator, operator $\rightarrow$ operator *defining operator product;*

3. $\models$ *is defined to be the relation* $\models$ *between functional specifications and models of functional specifications such that if* $F : A_{S_1}, A_{S_2}, \ldots, A_{S_n} \rightarrow S$ *is a model of the operation* $f : s_1, s_2, \ldots s_n \rightarrow s$ *and* $G : A_{Y_1}, A_{Y_2}, \ldots, A_{Y_m} \rightarrow A_Y$ *is a model of the operation* $g : y_1, y_2, \ldots, y_n \rightarrow y$ *with* F *and* $G \in m \in$Mod[SP], *then*

   (a) *if* $s = y_1, y_2, \ldots y_m$, *then* $G(F) : A_{S_1}, A_{S_2}, \ldots, A_{S_n} \rightarrow A_Y$ *is a model of* $g \circ f$, *and*

   (b) $G \times F : A_{Y_1}, A_{Y_2}, \ldots, A_{Y_m} \times A_{S_1}, A_{S_2}, \ldots, A_{S_n} \rightarrow Y \times S$ *is a model of* $g \cdot f$. $\square$

As stated in Definition VI.2, functional designs are well-formed sentences of a functional architecture theory. But what does it mean to be a well-formed sentence? Certainly, a well-formed sentence of a functional architecture theory must involve operators with compatible signatures. The following definition states the conditions under which application of composition operation of a FAT results in a well-formed statement.

Definition VI.4 Syntactically well-formed. *Given a functional architecture* $\langle \mathcal{D}, A_F \rangle$ *where* $A_F = \langle O, R, \models \rangle$, *an application of an operator* ρ *in* R *with arguments* $f_{u,v} \in O$ *and* $g_{w,x} \in O$ *used to define an operator* $h_{y,z}$, *denoted* $h = f \ \rho \ g$, *is syntactically well-formed if the following conditions are satisfied:*

1. *The result of the operation,* $h_{y,z}$, *is also in* O.

2. *The resulting operation $h = f\ \rho\ g$ is consistent with respect to the signatures of h, f and g. That is, the arity of h must be consistent with (equivalent to) the arity of $f\ \rho\ g$, and if ρ is the composition operator $\circ$, then the rank of f must be consistent with the sort of g.* □

For example, if the operations $f : u, v, w \rightarrow v$ and $g : x, y \rightarrow z$ are composed using the composition operator $\circ$ to define an operation $h = f \circ g$, then Definition VI.4 implies that z must be isomorphic to the product sort $u \times v \times w$ if h is to be well-formed.

As the above example illustrated, composition of functional operations can have an impact on sort definitions. Note however that Definition VI.4 only addresses the syntactic compatibility of the operations involved in a composition. No mention is made in Definition VI.4 of semantic compatibility. For example, $f \circ g$ may be syntactically well-formed, but may not be semantically well-formed. For example, suppose both f and g are defined using problem theory. Then $I_f \Rightarrow O_f$ and $I_g \Rightarrow O_g$ must hold. If O_g does not imply I_f, i.e., the output condition of g does not satisfy the input condition of f, then not only may f not terminate, but O_f may not be established. Clearly in this case $f \circ g$ is not semantically well formed. The following definition strengthens Definition VI.4 by defining semantically well-formed designs.

Definition VI.5 Semantically well-formed. *Denote by $A = \langle \mathcal{D}, A_F \rangle$ a functional architecture where $A_F = \langle O, R, \models \rangle$, and suppose $f : x \rightarrow y$ and $g : u \rightarrow v$ in O are defined using Problem Theory. Then*

1. *$f \cdot g$ is semantically well-formed, and its input condition is $I_f \wedge I_g$ and its output condition is $O_f \wedge O_g$.*
2. *$f \circ g$ is semantically well-formed if*
   (a) *$f \circ g$ is syntactically well-formed, and*
   (b) *$O_g \Rightarrow I_f$.*
3. *If $f \circ g$ is semantically well-formed, then*
   (a) *$I_{f \circ g}$ is I_g and $O_{f \circ g}$ is O_f; and*

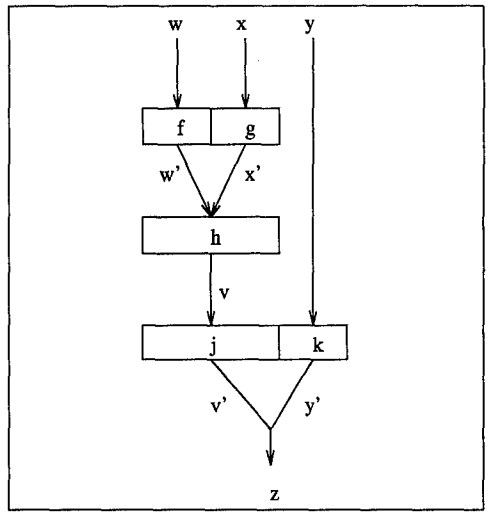


Figure 6.2 Operation Composition Using a Functional Architecture

(b) $I_{f \cdot g}$ *is* $I_f \wedge I_g$, *and* $O_{f \cdot g}$ *is* $O_f \wedge O_g$. □

The operation $f \cdot g$ forms the product of the operations f and g. (See Definition III.9.)

A functional architecture theory allows operators to be combined either horizontally as products or vertically as compositions as shown in Figure 6.2. In the figure, the operations $f : w \rightarrow w'$ and $g : x \rightarrow x'$ have been horizontally combined using the product operation $\cdot$ to define an operation $f \cdot g : w, x \rightarrow w', x'$ where $f \cdot g$ is defined by the product $f(y_w) \times g(y_x)$. This structure has in turn been combined with the operation $h : w', x' \rightarrow v$ to form a simple functional pipeline. The operation $h \circ (f \cdot g) : w, x \rightarrow v$ is in turn combined with the operation $j : v \rightarrow v'$ to define the operation $(j \circ h \circ (f \cdot g)) : w, x \rightarrow v'$. Finally, this operation is combined horizontally with the operation $k : y \rightarrow y'$ to produce the operation $(j \circ h \circ (f \cdot g)) \cdot k : w, x, y \rightarrow z$, where $z = v', y'$.

A design in a functional architecture theory can be likened to a program written in a purely functional programming language. For example, the structure of the LISP statement *(sort (car x) (cdr x))* is *sort* $\circ$ *(car* $\cdot$ *cdr).*

Properties of the higher-order operations $\circ$ and $\cdot$ are shown in Table 6.1. The operation *project* used in the table is defined in Slang and reflects the operation π of Figure 3.8. Based on the properties listed in the table, $(j \cdot k) \circ h$ is not in general equal to $j \cdot (k \circ h)$. This fact is made explicit in the following theorem.

Theorem VI.1 *Denote by* $f : s_1, s_2, \ldots, s_m \rightarrow s$, $g : u_1, u_2, \ldots, u_p \rightarrow v$, *and* $h : w_1, w_2, \ldots, w_r \rightarrow x$ *three arbitrary operations. If* $(f \cdot g) \circ h$ *and* $f \cdot (g \circ h)$ *are both well-formed expressions, then* f *is a nullary operation.*

Proof. *Because the expressions* $(f \cdot g) \circ h$ *and* $f \cdot (g \circ h)$ *are well-formed, we have*

$$\begin{aligned} (s_1, s_2, \ldots s_m) \times (u_1, u_2, \ldots, u_p) &\cong x \qquad (f \cdot g) \circ h \text{ is well-formed} \\ u_1, u_2, \ldots u_p &\cong x \qquad f \cdot (g \circ h) \text{ is well formed} \\ ((s_1, s_2, s_m) \times (u_1, u_2, \ldots u_p)) &\cong u_1, u_2, \ldots, u_p \qquad \text{transitivity of } \cong \end{aligned}$$

which implies that $s_1, s_2, \ldots, s_m$ *is a* **1** *or identity for* $u_1, u_2, \ldots, u_p$. *Because* f, g, *and* h *are arbitrary operations, this implies that* $s_1, s_2, \ldots, s_m$ *is the empty sort, which implies that* f *is a nullary operation.* ■

Thus $(j \cdot k) \circ h$ and $j \cdot (k \circ h)$ might both be well formed if j is a nullary operation. If j is a nullary operation, then $(j \cdot k) \circ h$ is an operation whose rank is defined by h and whose sort is the product of the sorts of j and k. But $j \cdot (k \circ h)$ also has its rank defined by the rank of h and its sort defined by the product of the sorts of j and k. That is, if both $(j \cdot k) \circ h$ and $j \cdot (k \circ h)$ are well-formed, then they have the same signature. If both $(j \cdot k) \circ h$ and $j \cdot (k \circ h)$ have the same signature, do they also define equivalent operations? The answer to this question is yes.

Theorem VI.2 *Denote by* $f : s_1, s_2, \ldots, s_m \rightarrow s$, $g : u_1, u_2, \ldots, u_p \rightarrow v$, *and* $h : w_1, w_2, \ldots, w_r \rightarrow x$ *three arbitrary operations defined in a functional specification where* f, g, *and* h *are defined using problem theory. Denote by* I_j *and* O_j, $j \in \{f, g, h\}$, *the input and output condition respectively of these operations.*

If $(f \cdot g) \circ h$ *and* $f \cdot (g \circ h)$ *are semantically well formed expressions then* $(f \cdot g) \circ h \cong f \cdot (g \circ h)$.

Proof. *By Theorem VI.1,* $(f \cdot g) \circ h$ *and* $f \cdot (g \circ h)$ *are both well-formed implies that* f *is a nullary operation. Because* f *is a nullary operation,* I_f *is* true. *This fact will be used to show that* $I_{(f \cdot g) \circ h} \Leftrightarrow I_{f \cdot (g \circ h)}$ *and* $O_{(f \cdot g) \circ h} \Leftrightarrow O_{f \cdot (g \circ h)}$.

Input Condition. *The input condition of* $f \cdot (g \circ h)$ *is* $I_f \wedge I_{g \circ h}$, *which simplifies to* $I_{g \circ h}$. *Because* $f \cdot (g \circ h)$ *is semantically well-formed,* $O_h \Rightarrow I_g$. *Because* g *and* h *are defined using problem theory, we have* $I_g \Rightarrow O_g$ *and* $I_h \Rightarrow O_h$. *Using the transitive property of implication, this yields* $I_h \Rightarrow O_h \Rightarrow I_g \Rightarrow O_g$, *which implies* $I_h \Rightarrow I_g$, *so* $I_h \wedge I_g$ *simplifies to* I_h.

The input condition of $(f \cdot g) \circ h$ *is* $I_{(f \cdot g) \circ h}$, *which by definition is* I_h.

Output Condition. *The output condition of* $f \cdot (g \circ h)$ *is* $O_{f \cdot (g \circ h)}$, *which equals* $O_f \wedge O_{g \circ h}$. *But* $O_{g \circ h}$ *is, by definition,* O_g, *so* $O_{f \cdot (g \circ h)}$ *simplifies to* $O_f \wedge O_g$.

Similarly, the output condition of $(f \cdot g) \circ h$ *is* $O_{(f \cdot g) \circ h}$, *which by definition is* $O_{f \cdot g}$, *where* $O_{f \cdot g} = O_f \wedge O_g$. ■

Given a functional specification SP and a FAT F whose functional specification diagram contains SP, the application of an operation r in R induces a morphism from SP to a specification SP'. Specifically, if $f : A \rightarrow B$, $g : C \rightarrow D$ and $h : E \rightarrow F$ are operations contained in SP, and r is an operation of the FAT, the statement $f = g\ r\ h$ induces a morphism from SP to a specification SP' containing an axiom defining f in terms of g and h. If r is the operation $\circ$, then the axiom will have the form *(equal (f x) (g (h x)))*, which requires $F \cong C$, $A \cong E$, and $B \cong D$. If the product operation $\cdot$ is used, then the axiom will be of the form *(equal (f x)(product (g ((project 1) x)) (h ((project 2) x))))*, where the binary operation *product* forms the product in the categorical sense of its arguments. Note that in either case, there are proof obligations concerning sort compatibility which must be verified. These obligations are listed in Table 6.1. The use of FATs to create new operators in terms of old operators is described next.

Figure 6.3 depicts the use of an architecture theory to decompose an operation defined by a problem specification into a composition of two simpler operations. The specification $\mathcal{S}$ introduces the sort E, two operations $g : D \rightarrow E$ and $h : E \rightarrow R$, and the single axiom $f = h \circ g$.(57) The specification $\mathcal{B}_F$ is a problem specification for a specific problem, such as the sort-search problem of Chapter IV, and $\mathcal{B}_{FS}$ defines a partition of f into the operations g and h. That is, $\mathcal{B}_{FS}$ contains the axiom *(equal (f x)(h (g x)))*. The specification morphism from $\mathcal{S}$ to $\mathcal{B}_{FS}$ is part of an interpretation from the structure defined in $\mathcal{S}$ to the problem

Table 6.1 Properties of the Operations · and ∘

Property	Expression	Comments
1	f ∘ (g ∘ h) = (f ∘ g) ∘ h	Associativity (Provided $z = w$ and $x = u$)
2	f · (g · h) = (f · g) · h	Associativity
3	f · g ≅ g · f	Commutativity
4	f ∘ (g · h) = f ∘ j	Where j : $\alpha \rightarrow \beta$, ((project 1) α)= w, ((project 2) α)= y, ((project 1) β)= x, ((project 2) β)=z, and $\beta \cong u$.
5	(f · g)∘ h = j ∘ h	Where j : $\alpha \rightarrow \beta$, ((project 1) α)= u, ((project 2) α)= w, ((project 1) β)= v, ((project 2) β)=x, and $\alpha \cong y$
6	f ∘ g · h = f ∘ (g · h)	
7	f · g ∘ h = (f · g) ∘ h	
8	$I_{f \circ g} = I_g$	f, g defined by Problem Theory
9	$O_{f \circ g} = O_f$	f, g defined by Problem Theory
10	$I_{f \cdot g} = I_f \wedge I_g$	f, g defined by Problem Theory
11	$O_{f \cdot g} = O_f \wedge O_g$	f, g defined by Problem Theory
For any operations $f : u \rightarrow v$, $g : w \rightarrow x$, and $h : y \rightarrow z$		

defined in $\mathcal{B}_F$. Note that $\mathcal{B}_F$ may need to be extended before this partition can be defined, and the input and output conditions of g and h will be defined using I_f and O_f. Specifically, $I_f \Rightarrow I_{h \circ g}$ and $O_{h \circ g} \Rightarrow O_f$ must hold, and $h \circ g$ must be semantically well-formed.

As a concrete example, consider once again the sort-search problem described in Chapter IV, where the input and output conditions of *sort-search*, *sort*, *search*, and the identity operation are:

```
I_Sort-Search(el, a-seq) = (in el a-seq)
O_Sort-Search(el, a-seq) =
        (equal (sort-search el a-seq) i)
        ⇔
        (exists i (exists z (implies (and (permutation z a-seq)
                                          (ordered z))
                                     (equal z[i] el))))

I_Sort(a-seq) = true
O_Sort(a-seq) =
        (equal (sort a-seq) v)
        ⇔
        (exists v (and (permutation a-seq v) (ordered v)))
```

I_{Search}(el, v) = (and (in el v)(ordered v))
O_{Search}(el, v) =
(equal (search el v) i)
⇔
(equal v[i] el)

I_{id}(x) = true
O_{id}(x) =
(equal (id x) x)
⇔
true

The specification for Sort-Search plays the role of $\mathcal{B}_F$ of Figure 6.3, and the specification $\mathcal{S}$ defines the composition $h \circ (g \cdot id)$. $\mathcal{S}$ can be used to decompose the operation *sort-search* into *search* ∘ *(sort · id)* as follows:

1. Because *search* ∘ *(sort · id)* must be semantically well-formed, $O_{\text{sort·id}}$ must imply I_{search}. A derived antecedent, (98) P, over the variables *a-seq* and *el* is used to strengthen $I_{\text{search}\circ(\text{sort·id})}$ as follows:

   P(a-seq,el) ⇒ ($O_{\text{sort·id}} \Rightarrow I_{\text{search}}$)
   ⇒ ($O_{\text{sort}} \wedge O_{\text{id}} \Rightarrow I_{\text{search}}$)
   ⇒ (exists v (implies (and (permutation a-seq v)
   (ordered v))
   (and (in el v) (ordered v))))

   Which is true if *(in el v)* is assumed. Because *a-seq* is a permutation of *v*, the derived antecedent P(a-seq,el) becomes *(in el a-seq)*. This implies that *search* ∘ *(sort · id)* can be used for *sort-search* provided the input condition $I_{\text{search}\circ(\text{sort·id})}$ is strengthened to *(in el a-seq)*.

2. $O_{\text{search}\circ(\text{sort·id})} \Rightarrow O_{\text{sort-search}}$ as follows:

   $O_{\text{search}\circ(\text{sort·id})}$ = O_{search}
   = (equal v[i] el)

   But *v* is an ordered permutation of *a-seq*. Therefore $O_{\text{search}\circ(\text{sort·id})} \Rightarrow O_{\text{sort-search}}$ if $z = v$.

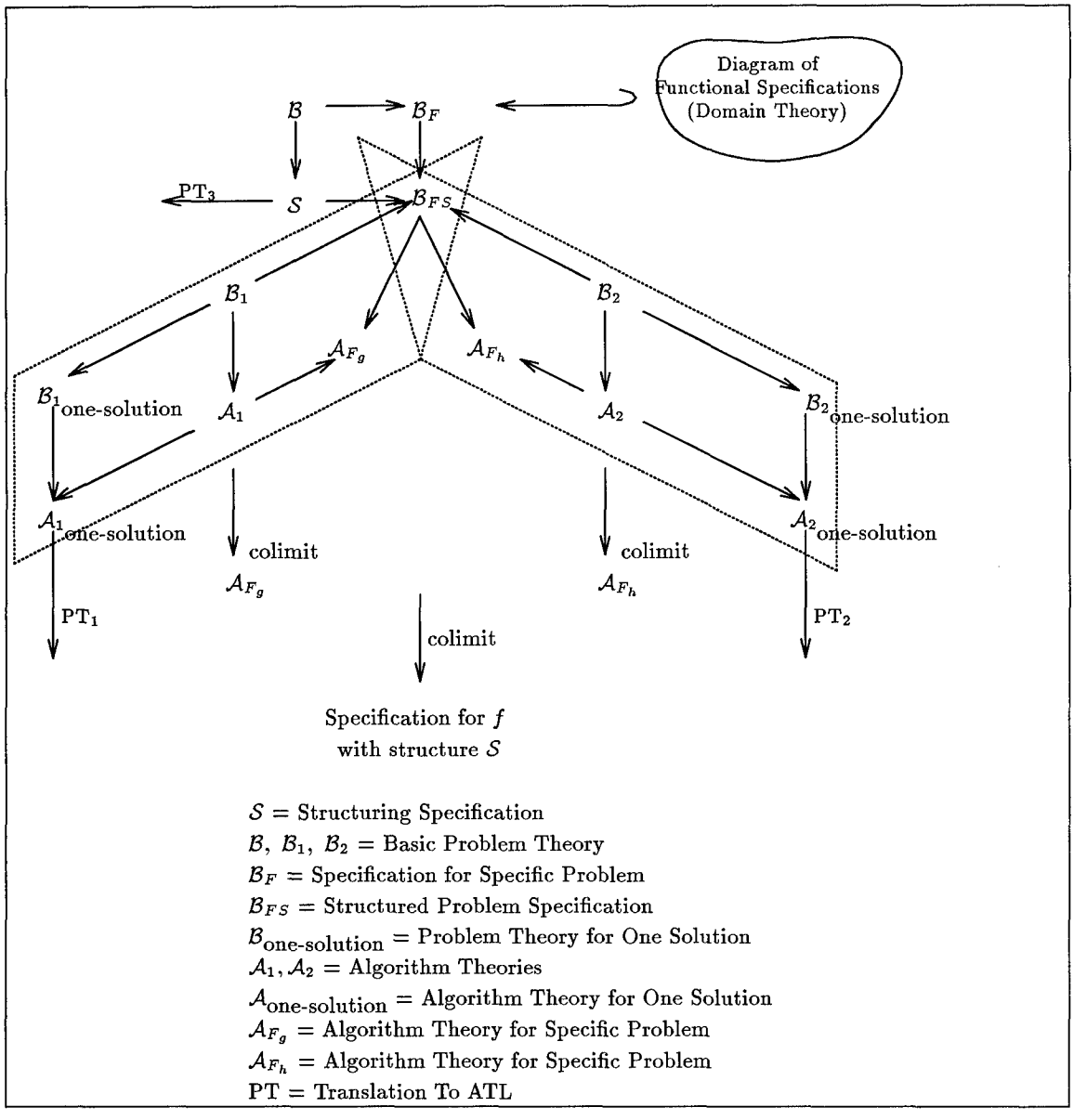


Figure 6.3 Using Functional Architecture Theory to Decompose an Operation (Based on (57))

Items 1 and 2 above imply that *sort-search* can be decomposed into *search* $\circ$ *(sort* $\cdot$ *id).* The resulting specification, denoted $\mathcal{B}_{FS}$, contains a definition of *sort-search* as the composition *search* $\circ$ *(sort* $\cdot$ *id).*

After an interpretation from $\mathcal{S}$ to $\mathcal{B}_F$ has been defined, the resulting subproblems of f, the operations g and h, can each be addressed as separate problems. Continuing

with the *sort-search* problem, this means that *search* and *(sort · id)* can be associated with problem specifications as shown abstractly in Figure 6.3. For example, *search* can be associated with a problem specification $\mathcal{B}_1$ by defining a specification morphism from $\mathcal{B}_1$ to the definition of *search* in $\mathcal{B}_{FS}$. Similarly, *sort* can be associated with a problem specification $\mathcal{B}_2$. After each subproblem has been associated with a problem specification, Figure 6.3 indicates that algorithm theory interpretations for these subproblems can be defined. In the case of *search*, an interpretation from a global search algorithm theory, denoted $\mathcal{A}_1$ in the figure, to a specification for search can be defined, and in the case of *sort*, an interpretation from a divide and conquer algorithm theory, $\mathcal{A}_2$, can be defined. The colimit of the resulting diagram contains both *sort-search* defined using *search* $\circ$ *(sort · id)* as well as refined algorithm specifications for *sort* and *search.*

In Figure 6.3, the arrows PT_i denote translations or interpretations from the structures of the specifications from which they emanate to structures in the abstract target language (ATL). Definition of these arrows to ATL is left for future research.

6.2.2 Process Based Architecture Theory. An architecture theory whose objects are CSP process symbols and whose operations can be used to define processes in terms of other processes can be defined. For example, such an architecture theory could be used to define a process P to be the parallel composition of processes Q and R. A definition of such a process-based architecture theory is presented below.

Definition VI.6 Process-Based Architecture Theory. *A process based architecture theory (PAT) is a 2-tuple* $\langle \mathcal{I}, A_P \rangle$ *where* $\mathcal{I}$ *is a diagram of process specifications and* A_P *is an architecture theory* $\langle O, R, \models \rangle$ *where*

1. *O is a collection of process symbols contained in and defined by a process specification* $pSP = \langle \Pi, \Xi \rangle$ *in* $\mathcal{I}$*, where*

   (a) $\Pi \triangleq \langle \Sigma, E, P, V, \kappa \rangle$ *is a process signature and* $\Sigma \triangleq \langle S, \Omega \rangle$ *is a functional signature,*

   (b) $\Xi : \kappa \rightarrow T_{CSP}(X)$ *is an injective relation between process symbols and process expressions,*

such that $O \subseteq \kappa$,

2. *R is a subset of process operators shown in Figure 5.3 and defined in (52),*

3. $\models$ *is the satisfaction relation of Definition V.18.*

An operation r in R can be used to combine processes P and Q in O to define a process W provided

1. *W is in O, and*

2. *PrQ is a well-formed expression in* $T_{CSP}(O)$.

Application of an operation r in R to processes P and Q in O to define a process W in O will be denoted W sat P r Q, where

1. $\Xi(W) = P\ r\ Q$,

2. chan(W)= chan(P) $\cup$ chan(Q),

3. var(W) = var(P) $\cup$ var(Q),

4. act(W) = act(P) $\cup$ act(Q), *and*

5. α(W) = α(P) $\cup$ α(Q). $\square$

Note that this definition could be generalized in that CSP is a specific type of process logic, and the relation $\models$ of Definition V.18 defines a very weak form of process equivalence. References to CSP_A expressions in the above definition could be replaced with references to process logics of which CSP is one instance. Furthermore, the relation $\models$ could be replaced with a set of process equivalence relations defining a well ordering over processes. For example, the satisfaction relation $\models$ of the above definition could be based on observational equivalence or bisimulation semantics rather than trace semantics.

In the definition above, CSP process definitions in conjunction with the process composition operators of Figure 5.3 define the structure of the architecture, while the behavior rules in (52) provide the semantics of the structures. Equivalence between process structures is defined in this case to be trace-equivalence. This type of architecture theory defines *global structure* in that the structures of the architecture theory — CSP process structures — are relatively large building blocks which may contain multiple functional

operations. The functional operations of these structures may be composed or decomposed into assemblies of equivalent operations through the use of a FAT.

Definition VI.6 does not introduce any process operators that are not already included in the process logic of CSP. That is, CSP process specifications and the PAT of Definition VI.6 are equally powerful. However, if the set of operators R of a PAT is restricted to a subset of process operators, such as the sequential composition operator $\_ ; \_ : process, process \rightarrow process$, then the structures or designs that can be generated by such a PAT are more constrained. The PAT of Definition VI.6 defines a meta-class of process-based architecture theories because it contains the full set of CSP process operators. That is, any well-formed CSP_A expression can be composed using these operators. Other, more structurally constrained process-based architecture theories can also be defined in terms of the PAT of Definition VI.6. These architectures are described in Section 6.3.

The structures created using a PAT define process specifications. The process symbols in O represent CSP processes, and the composition operators in R applied to the symbols in O result in process expressions which extend Ξ for some process specification pSP.

The process symbols combined using a PAT may have process expressions associated with them. There is no requirement in a PAT that these expressions be formed using only the operations contained in R. That is, for any process symbol W in O where $\Xi(W)$ is defined, the expression $\Xi(W)$ need not be written using only the process operations in R. $\Xi(W)$ may have been defined using another PAT. The set of process operators R serves to restrict how the processes referenced in O may be combined to define the structure of other processes referenced in O. This leads to the notion of heterogeneous architectures, where the structure at one level of abstraction — as constrained by the operators of one PAT — may differ from the structure at another level of abstraction as constrained by the operations of another PAT. This concept is further explored in the following sections.

As is the case with application of operations of FAT, application of operations of a PAT defined over a process specification pSP induce morphisms as well, in this case from the process specification pSP to a process specification pSP'. If an operation r contained

in the set R of a PAT is used to combine the process symbols P and Q to define the process W, then $\Xi(W)$ in pSP' will have the form W *sat* P r Q. Note that this implies that the induced morphism must also include mappings establishing *act(W), chan(W), var(W)*, and αW. Specifically, the application of an operation r in R of a PAT $\langle \mathcal{I}, A_P \rangle$ defined over a process specification pSP in $\mathcal{I}$ induces a morphism $\pi : pSP \rightarrow pSP'$ such that if $W = P\ r\ Q$, where P, R, and Q are in O, then:

1. $\pi(\Xi(W)) = \Xi'(W)$, where $\Xi'(W) = P\ r\ Q$,
2. $\pi(var(W)) = var(P) \cup var\ (Q)$,
3. $\pi(chan(W)) = chan(P) \cup chan\ (Q)$,
4. $\pi(act(W)) = act(P) \cup act\ (Q)$,
5. $\pi(\alpha(W)) = \alpha(P) \cup \alpha(Q)$, and
6. $\pi(X) = id(X)$ for any other X in pSP.

Provided only finite automata are generated by the expressions in Ξ and Ξ', this induced morphism may be checked to determine if it is a process specification morphism. That is, a check can be made to determine if $traces(\Xi(W)) \subseteq traces(\Xi'(W))|_\pi$ where $\Xi'(W)$ is the expression $P\ r\ Q$.

PATs and FATs share a common core. Specifically, PATs and FATs share a core of sorts and functional operations. Process specifications lack axioms constraining the set of models of their sorts and functional operations. As defined in Chapter V, these sorts and operations are given semantics through association with functional specifications. The structure used to make this association is a *component.* Component-based architecture theory is defined next.

6.2.3 Component-Based Architecture Theory. Components contain the structure necessary to provide definition to the sorts and functional operations of process specifications. Because components consist of a combination of functional and process specifications, a component based architecture theory is defined as a combination of a functional and process-based architecture theory.

Definition VI.7 Component-Based Architecture Theory. *A component-based architecture theory (CAT) is an architecture theory defined by the 3-tuple* $\langle \mathcal{C}, A_F, A_P \rangle$ *where*

1. $\mathcal{C}$ *is a diagram of components* $\langle \mathcal{D}, S_D, \mathcal{I}, S_I, i \rangle$,
2. $A_F \triangleq \langle O_F, R_F, \models_F \rangle$ *is a functional architecture theory where the set* O_F *of operations are contained in and defined by a functional specification* SP *of a diagram* $\mathcal{D}$ *of a component* C *in* $\mathcal{C}$, *and*
3. $A_P \triangleq \langle O_P, R_P, \models_P \rangle$ *is a process-based architecture theory where the process symbols in* O_P *are contained in and defined by a process specification* pSP *of a diagram* $\mathcal{I}$ *of a component* C *in* $\mathcal{C}$

such that

1. A_F *and* A_P *are applied to the same component* C *in* $\mathcal{C}$, *and*
2. *the models of the process specification* pSP *of a component are restricted to the set of models* $\{m \mid m|_i \in Mod[SP] \wedge m \in Mod[pSP]\}$. *That is, valid models of the sorts and functional operations referenced in a process specification* pSP *of a component are restricted those which are also models of the functional specification* SP. □

A CAT consists of a diagram of components and two architecture theories, one for defining functional operations in terms of other functional operations and the other for defining processes in terms of other processes. Basing an architecture theory on components provides at least some of the structure needed to define and evaluate constraints between functional specifications and process specifications.

The operations in R_P of a CAT place constraints on how process symbols in O_P can be combined to provide definitions for other process symbols in O_P. The application of an operation in R_P to provide a definition of a process in O_P induces a morphism between process specifications, and the application of an operation in R_F to define an operation in terms of other operations induces a morphism between functional specifications. Taken together, this implies that the application of an operation r in $R = R_P \cup R_F$ of a CAT to define structure in terms of other structure induces a morphism between components. The morphism so defined is either a morphism defining structure of functional operations

if $r \in R_F$, or it is a morphism defining process structure if $r \in R_P$. Denote by σ the morphism between components induced by the application of an operation $r \in R$ of a CAT. Then,

1. If the operation R in R_F is applied to the operation symbols f, g, and h in O_F to form the expression $f = gRh$, then

   (a) $\sigma(\mathcal{I}) = \mathcal{I}$;

   (b) $\sigma(S_I) = S_I$;

   (c) $\sigma(S_D)$ is an injection which extends S_D with an axiom of the form *(equal (f) (r (g) (h)))*;

   (d) $\sigma(\mathcal{D})$ is an injection extending $\mathcal{D}$ by adding the object $\sigma(S_D)$ and the arrow $\sigma(S_D)$ from S_D to $\sigma(S_D)$; and

   (e) $\sigma(i) = i$.

2. If the operation r in R_P is applied to the symbols W, V, and T in O_P to form the expression W *sat* V r T, then

   (a) $\sigma(\mathcal{D}) = \mathcal{D}$;

   (b) $\sigma(S_D) = S_D$;

   (c) $\sigma(S_I)$ is an injection extending S_I with an axiom of the form W *sat* V r T and which maps *var(W)* to *var(V)* $\cup$ *var(T)*, *chan(W)* to *chan(V)* $\cup$ *chan(T)*, *act(W)* to *act(V)* $\cup$ *act(T)*, and α*(W)* to α*(V)* $\cup$ α*(T)*;

   (d) $\sigma(\mathcal{I})$ is an injection extending $\mathcal{I}$ by adding the object $\sigma(S_I)$ and the arrow from the specification S_I to the specification $\sigma(S_I)$ as defined by $\sigma(S_I)$.

The diagram $\mathcal{C}$ of a CAT is extended through the application of an operation in $R_F \cup R_P$ by the addition of the component C' defined above and the arrow σ from the component to which the operation was applied to the component C'.

6.2.4 Summary. The reader may be wondering why three different types of architecture theory have been introduced and defined. The reason is simple: **flexibility**.

Defining a CAT in terms of a PAT and a FAT allows relatively independent development of structure within each of these architecture theories. The PAT of a CAT provides the mechanisms necessary for defining global structure at one level of abstraction, while the FAT of a CAT provides the means for defining local structure in the form of operator compositions. In addition, a CAT provides at least some of the necessary structure for expressing and evaluating constraints expressed between functional and process-based specifications.

Note that there are there are at most two composition operators, $\circ$ and $\cdot$, in any functional architecture theory, leading to at most four different classes of functional architecture theory, one for each possible combination of the two composition operators. However, there are several possible composition operators that can used in the definition of process-based architecture theories. In fact, as shown in Figure 5.3, there are at least eight distinct composition operators that can be used to define processes in terms of other processes. Some of these process composition operators can be used to define rather well known homogeneous designs. For example, the sequential composition operator can be used to construct batch-sequential designs.

Because the set of process-based architecture theories that can be defined using the composition operators of Figure 5.3 is rich in comparison to the set of functional architecture theories that can be defined using the composition operators $\circ$ and $\cdot$, the following section on constrained architectures deals exclusively with process-based architecture theories. Extension to component based architecture theories is straightforward. Process-based architecture theories are used in Chapter VIII to develop a specification for a segment of an image recognition application.

6.3 Process Based Architecture Theories

This section addresses process-based architecture theories defined using a subset of the process operators depicted in Figure 5.3. No attempt is made to define all possible architectures. Instead, architecture theories for the well known architectural paradigms are defined, and they are applicable to a wide variety of problem classes. Eight process-based architecture theories falling into four broad categories are developed.

1. Parallel architectures: General parallel architecture is discussed in Section 6.3.2, and five specific architecture theories are defined.

   (a) Parallel architecture.

   (b) Layered architecture, where layered architecture is defined in terms of parallel architecture.

   (c) Pipeline architecture.

   (d) Client-server architecture.

   (e) Pipe-filter architecture.

2. Batch architectures: A batch-sequential architecture theory is defined in Section 6.3.3.

3. Composite architectures: An architecture theory called piped-batch sequential based on the combination of pipe-filter and batch-sequential architectures is defined in Section 6.3.4.

4. Constraint-based architectures: An architecture theory wherein structure is defined through the use of constraints is presented in Section 6.3.5.

Examples of using architecture theories to define structure can be found throughout the following subsections as well.

6.3.1 Structuring Specifications. Architecture theories are used to define structure. Functional architecture theories define structure through the operations "." and "o", while process based architecture theories define structure through the process operations of Figure 5.3. There are at least two ways in which architecture theories can be used to define structure:

1. An architecture theory can be used to define structure through specification extension. For example, consider a process-based specification pSP containing the process symbols T, U, and V. $\Xi(T)$ can be defined as the sequential composition of the process expressions $\Xi(U)$ and $\Xi(V)$ by extending pSP with an axiom of the form T *sat* $U;V$.

2. An architecture theory can be used to define structure through the use of *structuring specifications*, and this is the approach taken here.

A structuring specification for an architecture theory $\langle O, R, \models \rangle$ uses one of the composition operators in R to define a structural relationship between elements in O. For example, a structuring specification for a process-based architecture theory containing the composition operator ";" could be defined. In this case, the structuring specification would define the structural relationship $U;V$ where U and V are parameters. Definition of U and V would be provided through morphism. That is, structuring specifications are parameterized on the process symbols used in the structure defining expression. The notion of a structuring specification is made precise in the following definition.

Definition VI.8 Structuring specification. *Given an architecture theory* $A = \langle O, R, \models \rangle$ *and a binary composition operator* $\rho \in R$, *a* structuring specification S *for* ρ *is a parameterized specification containing three* O*-objects,* O_1, O_2 *and* O_3 *and a single axiom defining* O_1 *to be the composition* $O_2 \rho O_3$. □

Structuring specifications encapsulate architecture theory concepts for use in creating structure using the specification construction paradigm described in Chapter II. That is, architectural designs can be created by forming colimits of diagrams wherein the diagrams include structuring specifications. This concept is depicted in Figure 6.4 and is described in the following paragraphs.

Process-based structuring specifications, such as the one referenced in Figure 6.4, contain three process symbols related through the process composition operator encapsulated by the structuring specification. For example, a structuring specification for the parallel composition operator $\|$ will contain three process symbols, e.g., T, U and V, and a statement T *sat* $U \| V$. In this case, U and V can be given definition through association with process symbols in more concrete specifications. As shown in the figure, association is accomplished through the use of a trivial process specification and two specification morphisms. The trivial specification contains a single process symbol such as *Triv* for which the sets *chan, events, var* and *act* are empty. Of the two specification morphisms involved in an association, one is from the process symbol of the trivial specification to either U

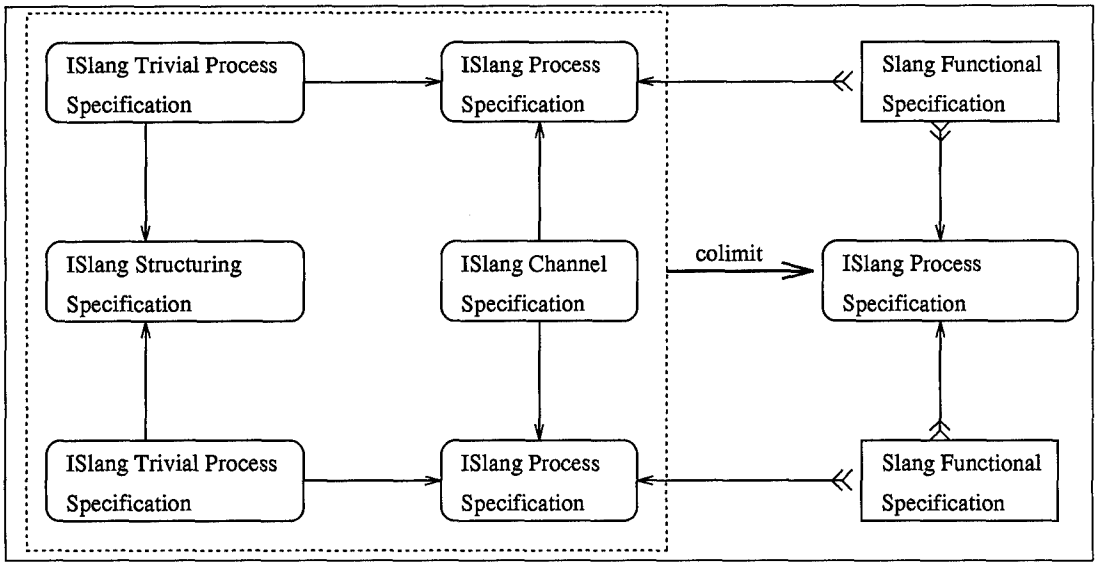


Figure 6.4 Using Process Specifications to Define Structure

or V of a structuring specification, and the other from the process symbol of the trivial specification to a process symbol of a more concrete specification. Once association has been made for both U and V, the colimit of the diagram can be taken. The colimit object will then contain an expression relating the identified process symbols together through the process composition operator embodied in the structuring specification. This relationship is also depicted in Figure 6.4.

Figure 6.4 shows the use of a *channel specification* to unify port symbols. Port symbol unification will result in the formation of CSP channels if the resulting unified port symbols are shared between exactly two concurrent processes. That is, a structuring specification can be used to define process structure, and a channel specification can be used to define communication channels within this structure. A channel specification has the following form:

```
pspec Channel-Spec is
      sort msg
      port c : msg
end-pspec
```

In Figure 6.4, morphisms from a channel specification to each of the process specifications being combined through the structuring specification identify the port symbol

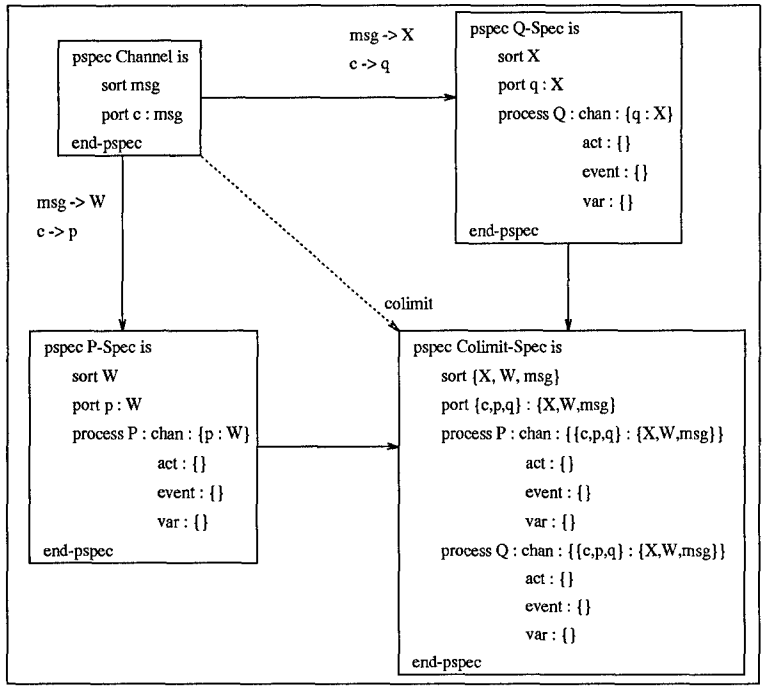


Figure 6.5 Using Channel Specifications to Unify Port Symbols

of the channel specification with a port symbol in each of the process specifications combined under the structuring specification. A colimit of the diagram results in a process specification containing a design based on the architecture theory reflected in the structuring specification, and the use of the channel specification results in individual process structures sharing port symbols.

For example, consider the diagram of process specifications shown in Figure 6.5. In the figure, specification *P-Spec* introduces a process P which has the single port symbol $p{:}W$ in its set of ports. Process Q of specification *Q-Spec* has the port symbol $q{:}X$ in its set of ports. A channel specification is used to unify the port symbol $q{:}X$ of Q with the port symbol $p{:}W$ of P. The colimit of the diagram defined by the nodes *P-Spec*, *Q-Spec* and *Channel-Spec* and the arrows *Channel-Spec* $\rightarrow$ *P-Spec* : $\{c \mapsto p,\ msg \mapsto W\}$, *Channel-Spec* $\rightarrow$ *Q-Spec* :$\{c \mapsto q,\ msg \mapsto X\}$ contains the process symbols P and Q and the port definition *port* $\{p,\ q,\ c\}$: $\{msg,\ X,\ Y\}$. That is, in the colimit object the port symbols p, q, and c are equivalent and the sort symbols msg, X, and Y are equivalent. A proof schema must be used here to ensure that the sorts X and Y are semantically equivalent. Note that sort compatibility of the ports used to define channels requires that the functional specifications

from which sort symbols draw their definitions come from a connected diagram (connected in the graph-theoretical sense). That is, the proof of sort equivalence requires that the sort symbols in question be comparable. Depending on the operator used to combine P and Q, the port symbols p, q and c become synonyms for a CSP channel. For example, if the colimit object of Figure 6.5 was extended with the axiom $P \gg Q$, then p, q and c would be equivalent names for the CSP channel connecting P to Q.

Figure 6.4 also shows how structuring specifications can be used with components. That is, the figure shows Slang functional specifications associated with ISlang process specifications through signature injections. The combination of the signature injection, process specification, and functional specification define a component. In this case, the colimit of the diagram defines a new component. Process-based architecture theories defined using a subset of the operations of Figure 5.3 are defined next.

6.3.2 General Parallel Structures. CSP contains a number of process operators that result in parallel structures. Three of these operators, $\|$, $//$, and $\gg$ are used in this section to define architecture theories. In addition, constraints can be placed on the processes composed using an architecture theory to define additional architecture theories. For example, a buffered architecture theory can be defined by requiring that one of the processes composed using the operation $\|$ be a buffer process and the other process contain no intra-process communication.

6.3.2.1 Parallel Architectures. As given in the following definition, a parallel architecture design consists of a set of process executing concurrently (or in parallel).

Definition VI.9 Parallel processes. *The class of parallel processes is inductively defined as follows:*

1. *(Basis.) Any well-formed expression in CSP_A of sort* process *defines a parallel process.*
2. *(Induction.) If P and Q are well-formed expressions in CSP_A such that P and Q define parallel processes, then the CSP_A expression $P\|Q$ defines a parallel process.*

3. *(Extremal.) No CSP_A expression defines a parallel process unless it can be created through a finite number of applications of clauses 1 and 2.* □

The simplest parallel processes defined by CSP_A expressions are processes such as *SKIP* or *STOP*. However, the basis clause of the above definition admits processes defined by any well-formed process expression in CSP_A. How these process expressions are formed is of little concern. They could, for example, be defined as sequential compositions of other processes. Through the inductive clause, such process expressions can be combined using the parallel composition operator $\|$ to define more complex parallel processes. For example, the process expression $P;Q;R$ could be combined with the process expression $V /\!/ T$ to define the parallel process $(P;Q;R)\|(V /\!/ T)$. In this example, the basis clause states that $P;Q;R$ and $V /\!/ T$ define parallel processes, while the inductive clause states that $(P;Q;R)\|(V /\!/ T)$ defines a parallel process.

Now that the class of parallel CSP processes have been defined, an architecture theory based on Definition VI.9 is presented.

Definition VI.10 Parallel Architecture Theory. *A parallel architecture theory is 2-tuple $\langle \mathcal{I}, A_P \rangle$ where $\mathcal{I}$ is a diagram of process specifications and A_P is a process architecture theory $\langle O, R, \models \rangle$ where the set R of process operators is restricted to contain only the CSP parallel composition operator* $\_ \| \_$: process, process $\rightarrow$ process. □

As formalized by the following theorem, parallel architecture theory is complete with respect to the class of parallel architectures.

Theorem VI.3 *Parallel architecture theory is complete with respect to the class of parallel processes.*

Proof. *A proof by induction is used to establish the claim.*

1. *(Basis). Based on Definitions V.14 and V.17, any well-formed process expression in CSP_A can be represented as a process specification.*
2. *(Induction). Denote by P and Q two arbitrary well-formed process expressions in CSP_A such that P and Q define parallel processes. Then the CSP_A expression $R = P\|Q$ can be expressed using a parallel architecture theory as follows.*

Let PI-P *and* PI-Q *be process specifications where* P_S *is a process symbol in* PI-P *such that* $\Xi(P_S)$ *equals the process expression* P *and where* Q_S *is a process symbol in* PI-Q *such that* $\Xi(Q_S)$ *equals the process expression* Q.

Construct a process specification PI-R *for* R *as follows. Define the process specifications* T1, T2, *and* Parallel-Structure *as shown in Figure 6.6. Note that the structuring specification* Parallel-Structure *is defined using a parallel architecture theory.*

Define the morphisms

(a) $M1$ = T1 $\rightarrow$ Parallel-Structure*:* {Triv1 $\mapsto$ P1},

(b) $M2$ = T2 $\rightarrow$ Parallel-Structure*:* {Triv2 $\mapsto$ P2},

(c) $M3$ = T1 $\rightarrow$ PI-P*:* {Triv1 $\mapsto$ P_S}, *and*

(d) $M4$ = T2 $\rightarrow$ PI-Q*:* {Triv2 $\mapsto$ Q_S},

as shown in Figure 6.6. Then the colimit of the diagram defined by the nodes T1, T2, Parallel-Structure, PI-R *and* PI-Q *and the arrows* M1, M2, M3, *and* M4 *defines a process specification* PI-R *in which*

(a) $\kappa_{PI\text{-}R}$ *is the set* {{P_S, Triv1, P1}, {Q_S, Triv2, P2}, P1-par-P2}, *and*

(b) Ξ ({P_S, Triv1, P1}) *is the process expression* P,

(c) Ξ ({Q_S, Triv2, P2}) *is the process expression* Q,

(d) Ξ(P1-par-P2) *is the process expression* {Q_S, Triv2, P2} $\|$ {P_S, Triv1, P1} *which is isomorphic to the process expression* $P \| Q$. ■

As seen in the proof of Theorem VI.3, creation of parallel designs can be accomplished through use of a structuring specification. In the case of parallel architectures, the structuring specification has the form

```
pspec Parallel-Structure is
      process J : {},{},{},{}
      process S : {},{},{},{}
      process V : {},{},{},{}
      J sat S || V
end-pspec
```

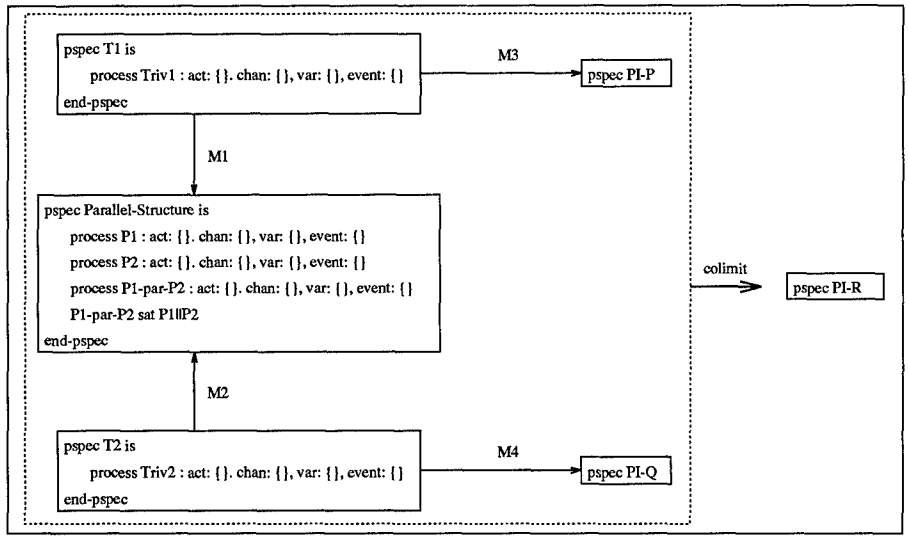


Figure 6.6 Using Parallel Architecture Theory to Construct Specifications

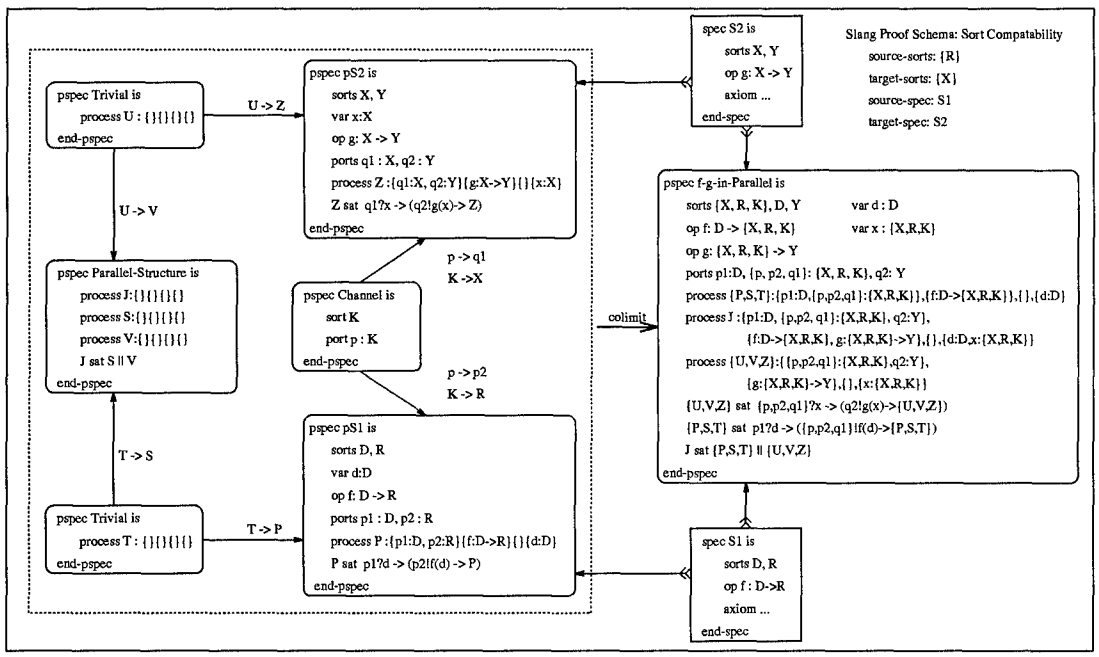


Figure 6.7 Using a Structuring Specification of a Parallel Architecture Theory

where S and V are left abstractly defined. Definitions for S and V are provided through refinement.

For example, Figure 6.7 shows two functional specifications S_1 and S_2 which have been associated with the process specifications *pS1* and *pS2* respectively. The sorts and operations of the functional specifications have been mapped to sort symbols and operation

symbols in the process specifications. The specifications *pS1* and *pS2* of the figure each introduce a single process defining an interface for the functional operations they encapsulate. The single axiom of *pS1* states that the process P consists of at least a repetition of an input event on port p_1 followed by an output event on port p_2, where the output value is obtained from evaluation of the operation f. Specification *pS2* is similarly defined. These two process specifications are combined using a structuring specification for a parallel architecture theory. A channel specification is used to unify ports p_2 of *pS1* and q_1 of *pS2* so that they form a CSP channel. Because a CSP channel is strongly typed, the sort R of functional specification $S1$ and the sort X of the functional specification $S2$ must be compatible. The proof of sort compatibility is carried out in the logic of the functional specification; a proof schema is used for this purpose (see Section 5.6.1 for details). Input condition satisfaction is also required but not shown.

As shown in Figure 6.7, a copy of the specification *Trivial* is used to associate the process symbol S of the structuring specification *Parallel-Structure* with the process symbol P of *pS1*. Another copy of the specification *Trivial* is used to associate the process symbol V of the structuring specification with the process symbol Z of *pS2*. When the colimit is taken over the specifications and arrows shown in Figure 6.7, process symbols T, S, and P are unified as are the process symbols U, V, and Z. Thus the process expression J *sat* $S \parallel V$ of *Parallel-Structure* is translated to the expression J *sat* $\{T,S,P\} \parallel \{U,V,Z\}$ in the colimit object. Similarly, the process expressions for P and Z are translated as follows:

$$P \text{ sat } p1?d \xrightarrow{CSP} (p2!f(d) \xrightarrow{CSP} P)$$
$$\mapsto \{T,S,P\} \text{ sat } p1?d \xrightarrow{CSP} (\{p,p2,q1\}!f(d) \xrightarrow{CSP} \{T,S,P\})$$

$$Z \text{ sat } q1?x \xrightarrow{CSP} (q2!g(x) \xrightarrow{CSP} Z)$$
$$\mapsto \{U,V,Z\} \text{ sat } \{p,p2,q1\}?x \xrightarrow{CSP} (q2!g(x) \xrightarrow{CSP} \{U,V,Z\})$$

The colimit object specifies a CSP process that accepts a value d of sort D over *p1* and outputs the value *g(f(d))* of sort Y over *q2*.

The specification *f-g-in-Parallel* can be combined with another process specification using a structuring specification such as the one in the figure to create a larger design. Specifically, the process symbol J of *f-g-in-Parallel* can be associated with a process sym-

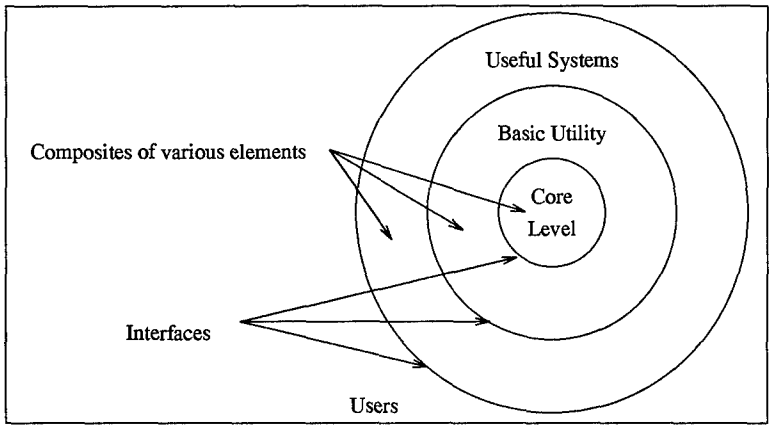


Figure 6.8 Layered Systems (Based on (38))

bol of a structuring specification in the same way in which the process symbol P of the specification *pS1* was associated with the process symbol S of the structuring specification *Parallel-Structure*. A process symbol, say A, of some other process specification can be similarly associated with the other process symbol of the structuring specification. The colimit of the resulting diagram would define the relationship between J and A. For example, J and A could be defined to operate in parallel, resulting in $J \| A$. This procedure is repeated to form even larger designs.

A specific type of parallel structure, a layered structure, can be created using the composition operator //. This operator is defined as a constrained parallel composition.

6.3.2.2 Layered Architecture. Layered systems are organized hierarchically, with inner layers providing services to adjacent outer layers. Each layer builds on the capability of inner layers of the system, and in essence, defines an increasing level of abstraction. The outer layer of a layered system defines the signature of the system. Figure 6.8 is a conceptual representation of a layered system.

The core layer of a layered design contains definitions of primitive operations. The basic utility layer uses the primitive operations of the core layer to define more complex operations, and the outer layer builds on the operations of the utility layer to define system capability. Interfaces between the layers define communication protocols. The interface of the outer layer defines the user interface. To facilitate discussion, the core or inner-most

layer of a layered system S is denoted $L_{S,0}$. The layer immediately adjacent to $L_{S,0}$ is denoted $L_{S,1}$, and so on.

Although only adjacent layers in a layered system may communicate, operations of inner layers may be made available to non-adjacent outer layers. Specifically, an operation $g : X \rightarrow W$ of an inner layer $L_{S,i}$ can be made available to an outer layer $L_{S,j}$, $j > i$, as follows: In each layer $L_{S,a}$, $a = i + 1..j$, define an operation $f_a : X \rightarrow W$ and define f_a in $L_{S,a}$ using the operation f_{a-1} of layer $L_{S,a-1}$. In this way, the operation $g : X \rightarrow W$ can be accessed indirectly in $L_{S,j}$. Optimization can be applied to make this indirect access direct.

In CSP, each layer of a layered system is a process, where the processes defining inner layers are subordinate to the processes defining outer layers. The process composition operator used in CSP to define subordinate processes is the binary operation $/\!/$. Layered systems are formally defined below.

Definition VI.11 Layered processes. *The class of layered processes is inductively defined as follows:*

1. *(Basis.) Any well-formed expression in CSP_A of sort* process *defines a layered process.*
2. *(Induction.) If P and Q are well-formed expressions in CSP_A such that P and Q define layered processes, then the CSP_A expression $P/\!/Q$ defines a layered process.*
3. *(Extremal.) No CSP_A expression defines a layered process unless it can be created through a finite number of applications of clauses 1 and 2.* □

The statement $P/\!/Q$ defines a layered process wherein P defines a layer on which Q is built. The semantics of the operator $/\!/$ are such that if $P/\!/Q$ is well-formed, then Q can "engage independently in the actions of $(\alpha Q - \alpha P)$, without the permission and without the knowledge of its partner P."(52:161) Communication on any channel c between P and Q with $P/\!/Q$ is hidden from the surrounding environment.(52). $P/\!/Q$ is well-formed if

1. $\alpha P \subseteq \alpha Q$, and

2. *chan(P)* $\subseteq$ *chan(Q)*, in which case

   (a) $\alpha(P /\!\!/ Q) = \alpha(P \| Q) \setminus \alpha P$, and

   (b) the set of external ports of $P /\!\!/ Q$ is the set *chan(Q)* $\setminus$ *chan(P)*.

This implies that if $P /\!\!/ Q$ is a well-formed layered design, then it is equivalent to the parallel design $(P \| Q) \setminus \alpha P$. The ability to redefine a design of one architecture theory in terms of a design in another architecture theory has significant impact on optimization and on top down specification construction. For example, a layered implementation may be less efficient than a pipeline implementation of the same problem. The topic of design translation is addressed in Chapter VII.

An architecture theory based on the class of layered processes is presented next.

Definition VI.12 Layered Architecture Theory. *A layered architecture theory is a 2-tuple* $\langle \mathcal{I}, A_P \rangle$ *where* $\mathcal{I}$ *is a diagram of process specifications and* A_P *is a process architecture theory* $\langle O, R, \models \rangle$ *where the set* R *of process operators is restricted to contain only the CSP subordination composition operator* _ // _ : process, process $\rightarrow$ process. $\square$

Layered architecture theory is complete with respect to the class of layered processes of Definition VI.11. That is, any well-formed layered process can be constructed using the above architecture theory. A proof of this claim is similar to the proof of Theorem VI.3.

A simple structuring specification for layered systems is shown below.

```
pspec Layered-Structure is
        process R : {},{},{},{}
        process S : {},{},{},{}
        process T : {},{},{},{}
        R sat S // T
end-pspec
```

CSP contains a syntax for denoting communication across labeled channels. The notation *m:P// Q* defines a process in which Q communicates with P along channels with compound names. Each communication takes the form $m.c.v$, where m is a label, c is a channel name shared between P and Q, and v is a value. The above structuring specification

can be extended to include labels and ports as shown below. In the specification below, process S is defined to be subordinate to T. As such, communication between S and T is hidden from the outside environment. This differs from the semantics of $S||T$ in that communication between S and T in $S||T$ is not hidden from the environment.

```
pspec Extended-Layered-Structure is
        sorts D, R
        port left : D
        port right : R
        port in : D
        port out : R
        label m
        process R : {left:X, right:R, in:D, out:R},{},{},{}
        process S : {left:X, right:R},{},{},{}
        process T : {left:X, right:R, in:D, out:R},{},{},{}
        R sat m:S // T
end-pspec
```

As a simple example of a layered design, consider the problem of computing $g(f(x))$, and suppose a process specification F has been defined such that F encapsulates the operation f. F could be defined to be subordinate to a process H where H includes, for example, error detection mechanisms or handshaking mechanisms not contained in F. Using the layered architecture structuring specification where S is associated with F and T is associated with H, $g(f(x))$ can be computed provided

1. F *sat* $left?x \xrightarrow{CSP} right!f(x)$, and
2. H *sat* $in?d \xrightarrow{CSP} (left!d \xrightarrow{CSP} (right?d \xrightarrow{CSP} (out!g(d) \xrightarrow{CSP} SKIP)))$.

A simple diagram corresponding to these process expressions is shown in Figure 6.9. None of the communication over the channels *left* or *right* is visible to the outside environment. That is, the set of traces over $F/\!/H$ consists of communication events over the channels *in* and *out*, with $0 \leq tr\lceil in - tr\lceil out \leq 1$ for any trace tr in $traces(F/\!/H)$.

6.3.2.3 Pipeline Architecture. Pipelined designs consist of a collection of processes operating in parallel wherein inter-process communication is severely limited. A process P_i in a pipeline design may only receive incoming communication from the process

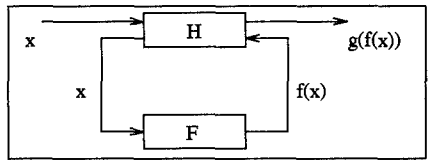


Figure 6.9 Simple Layered Processes

P_{i-1} immediately preceding it in the pipeline and may only communicate results to the process P_{i+1} immediately following it in the pipeline. Each process of a pipeline is called a *stage*. Each stage of a pipeline has exactly two channels: one for input and one for output.

Given a CSP structure S containing the process symbols $P_1, P_2, \ldots, P_n$ such that $P_1 \| P_2 \| \cdots \| P_n$ in S, then $P_1 \| P_2 \| \cdots \| P_n$ is a pipeline design if

1. Each process symbol P_i in S has exactly two channels associated with it, one for input and one for output;
2. A bijective mapping m from the process symbols $P = \{P_1, P_2, \ldots, P_n\}$ to the set $N = \{1, 2, \ldots, n\}$ can be defined such that for all P_i, P_j in $\{P_1, P_2, \ldots, P_n\}$, if $m(P_i) = m(P_j) - 1$ then P_i shares exactly one channel c with P_j such that c is used for output in P_i and for input in P_j; and
3. All communication between processes P_i and P_j with $m(P_i) = m(P_j)+1$ is concealed from the outside environment.

The first condition requires that each process in a pipeline design contain the requisite number of channels. The second condition requires that communication between processes in a pipeline design defines a total order over the process symbols, and the third condition states that internal communication in a pipeline is not visible outside of the pipeline. For example, consider the following process expressions:

A sat p?x $\xrightarrow{CSP}$ (q!f(x) $\xrightarrow{CSP}$ A)
B sat r?x $\xrightarrow{CSP}$ (s!h(x) $\xrightarrow{CSP}$ B)
C sat q?x $\xrightarrow{CSP}$ (r!g(x) $\xrightarrow{CSP}$ C)

Each process, A, B and C has exactly two channels, one for input and one for output. In addition, the bijective mapping m can be defined as follows:

$$\begin{aligned} A &\mapsto 1 \\ B &\mapsto 3 \\ C &\mapsto 2 \end{aligned}$$

such that $\forall$ *(x,y) (x,y* $\in \{A, B, C\} \Rightarrow$ *(m(x) = m(y)-1* $\Rightarrow \exists$ *(c) (c* $\in$ *chan(x)* $\wedge$ *c* $\in$ *chan(y)* $\wedge$ *c is used for output in x and for input in y))).* That is, both $A \| C \| B$ and $A \gg C \gg B$ are well-formed.

Internal communication in a pipelined design occurs over *internal channels*, while observable communication of a pipelined design occurs over *external channels.* Internal and external channels are defined below.

Definition VI.13 Internal and External channels. *A channel c of a CSP structure P is an* internal channel *of P if P contains at least two concurrent processes* P_1 *and* P_2 *such that c is a channel connecting* P_1 *to* P_2*. The set of internal channels of a process P is denoted* $chan_{internal}(P)$*.*

The set of external channels *of a CSP structure P are those channels of P that are not internal channels. The set of external channels of a CSP structure P is denoted* $chan_{external}(P)$*.* □

Communication over internal channels of layered designs or pipeline designs are concealed from the outside environment. In contrast, communication over internal channels of parallel designs is not concealed from the environment. This implies, for example, that **a parallel design will exhibit a wider range of observable behavior than either a functionally equivalent layered or pipelined design.**

CSP contains a process composition operator $\gg$ that can be used to define pipeline designs. This operator is used in the following definition of the class of pipeline designs.

Definition VI.14 Pipelined processes. *The class of pipelined processes can be inductively defined as follows:*

1. *(Basis.) Any well-formed expression in* CSP_A *of sort* process *containing exactly two external ports, one for input and one for output defines a pipeline process.*
2. *(Induction.) If P and Q are well-formed expressions in* CSP_A *such that P and Q define pipeline processes, then the* CSP_A *expression* $P \gg Q$ *defines a pipeline process.*

3. *(Extremal.) No CSP_A expression defines a pipeline process unless it can be created through a finite number of applications of clauses 1 and 2.* □

The above definition allows for pipeline stages to contain internal channels such as those used for communication with subordinate processes.

The class of pipelined processes leads to the following architecture theory.

Definition VI.15 Pipeline Architecture Theory. *A pipeline architecture theory is a 2-tuple $\langle \mathcal{I}, A_P \rangle$ where $\mathcal{I}$ is a diagram of process specifications and A_P is a process architecture theory $\langle O, R, \models \rangle$ where the set R of process operators is restricted to contain only the CSP process composition operator* _ ≫ _ : process, process → process. □

Note that the definition of the operator ≫ requires that the processes being combined contain exactly external two channels, one for input and one for output. In addition, because communication over channels connecting two stages of a pipeline is concealed from the external environment, all communication over internal channels of any stage in a pipeline design must be concealed as well.

Pipeline architecture theory is complete with respect to the class of pipeline processes of Definition VI.14. That is, any well-formed pipeline process can be constructed using the above architecture theory. A proof of this claim is similar to the proof of Theorem VI.3.

Concealment of communication is problematic, because as the following theorem illustrates, application of the concealment operator to an expression in a process specification induces a specification morphism *from* the target specification *to* the source specification.

Theorem VI.4 *Given a process specification $pSP = \langle \Pi, \Xi \rangle$, $\Pi = \langle \Sigma, E, P, V, \kappa \rangle$, such that $\Xi(W)$ is defined for some $W \in \kappa$, an application of the concealment operator* _ \ _ : process, set(event) → process *to the expression $\Xi(W)$ to define a process specification pSP' induces a specification morphism from pSP' to pSP.*

Proof. *Denote by A the set of events to be concealed in the expression $\Xi(W)$ for a process symbol W in a process specification $pSP = \langle \Pi, \Xi \rangle$. pSP' is defined by the concealment operator to be the process specification $\langle \Pi', \Xi' \rangle$ such that*

1. traces(Ξ'(W)) = traces(Ξ(W)\A) *and*

2. $\alpha W' = \alpha W - A$.

pSP' and pSP are identical except for the above two distinctions. Clearly the common structures between pSP and pSP' share a common set of models. What remains to be shown is that models of $\Xi'(W)$ *are also models of* $\Xi(W)$. *That is, we need to show* $\forall m(m \in Mod[\Xi'(W)] \Rightarrow m \in Mod[\Xi(W)])$, *or equivalently, that* traces(Ξ'(W)) $\subseteq$ traces(Ξ(W))$\restriction(\alpha W')$.

By the definition of concealment, traces(Ξ'(W)) = traces(Ξ(W)\A), *which equals* $\{t \restriction (\alpha W - A) \mid t \in traces(\Xi(W))\}$. *But* traces($\Xi$(W))$\restriction(\alpha W')$ = traces(Ξ(W))$\restriction$(αW-A), *which also equals* $\{t \restriction (\alpha W - A) \mid t \in traces(\Xi(W))\}$. *Thus* traces($\Xi'$(W)) $\subseteq$ traces(Ξ(W))$\restriction(\alpha W')$, *which implies that the concealment operator induces a process specification morphism from pSP' to pSP.* ■

The concealment operator can be thought of as the inverse or opposite of specification extension. As a consequence of this theorem, specification morphism arrows for specifications defined using the concealment operator "point the wrong way." In other words, an application of the concealment operator to a process specification specification P to produce a process specification Q induces a specification morphism from Q to P, not from P to Q. In general, this implies that **a parallel design cannot be converted under specification morphism into a pipeline design through concealment of communication** over internal channels because the induced specification morphism would run from the pipeline design to the parallel design. The relationships between pipeline and parallel designs are further explored in Section VII.

A structuring specification for pipeline architecture theory is defined below:

```
pspec Pipeline-Structure is
    sorts x, y, z
    port left : x
    port center : y
    port right : z
    process R : {},{},{},{left:x, center:y, right:z}
    process S : {},{},{},{left:x, center:y}
    process T : {},{},{},{center:y, right:z}
    R sat S ≫ T
```

end-pspec

The sort-symbols x, y and z and the port-symbols *left*, *center*, and *right* are introduced in the structuring specification for the purposes of achieving communication. As their names imply, port *left* is an input port, port *center* is a CSP channel connecting S to T, and port *right* is an output port. (Recall that a port symbol shared between two concurrent process is semantically equivalent to a CSP channel.) The semantics of the operation $\gg$ are such that S uses *center* for output while T uses *center* for input. Refinement of the sorts x, y and z can be accomplished through association with sort symbols defined in a functional specification. Elaboration of the processes S and T can be accomplished using any one of the process specification construction operations.

Because this structuring specification contains port symbols, it may be instructive to demonstrate the creation of multi-stage pipelined designs. In the paragraphs that follow, two approaches to using this structuring specification are presented. The first approach parallels that used in Section 6.3.3 to develop batch-sequential designs, while the second approach combines pipeline segments using channel specifications.

Creating Pipelined Designs through Recursive Application of Structuring Specifications. One method of creating multi-stage pipeline designs is to recursively apply a structuring specification to each stage of a pipeline design. This approach is illustrated in Figure 6.10. In the figure, process *P2* is defined to be the pipelined process $Q1 \gg Q2$. Association of *P2* with the pipelined process $Q1 \gg Q2$ requires that the port symbols *c1* and *l2* be unified, and that the port symbols *r1* and *r2* be unified. Associating *P2* with the process $Q1 \gg Q2$ results in a three stage pipelined design.

Figure 6.11 shows an ISlang diagram corresponding to the recursive application of pipeline structuring specifications. In the figure, a trivial process specification containing a single process symbol and two port definitions is used to unify process *P2* with the pipelined process *QL*. The port symbols of the trivial specification are used to associate ports *c1* and *r1* of *Pipeline-One* with ports *l2* and *r2* of *Pipeline-Two* respectively. The colimit object contains a definition of a three-stage pipeline defined by $P1 \gg (Q1 \gg Q2)$.

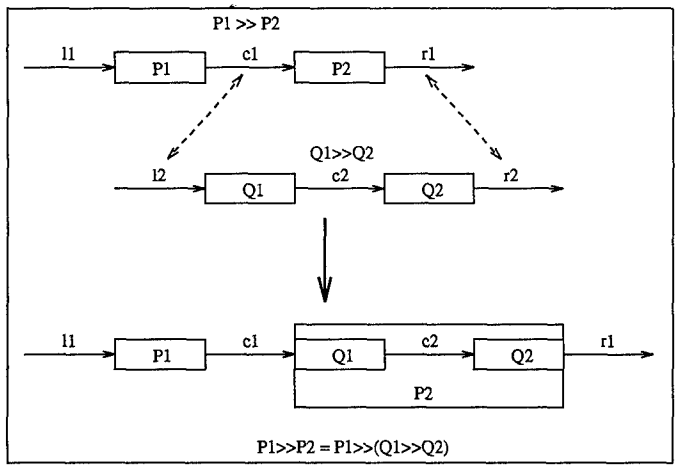


Figure 6.10 Recursive Application of Pipeline Structuring Specification

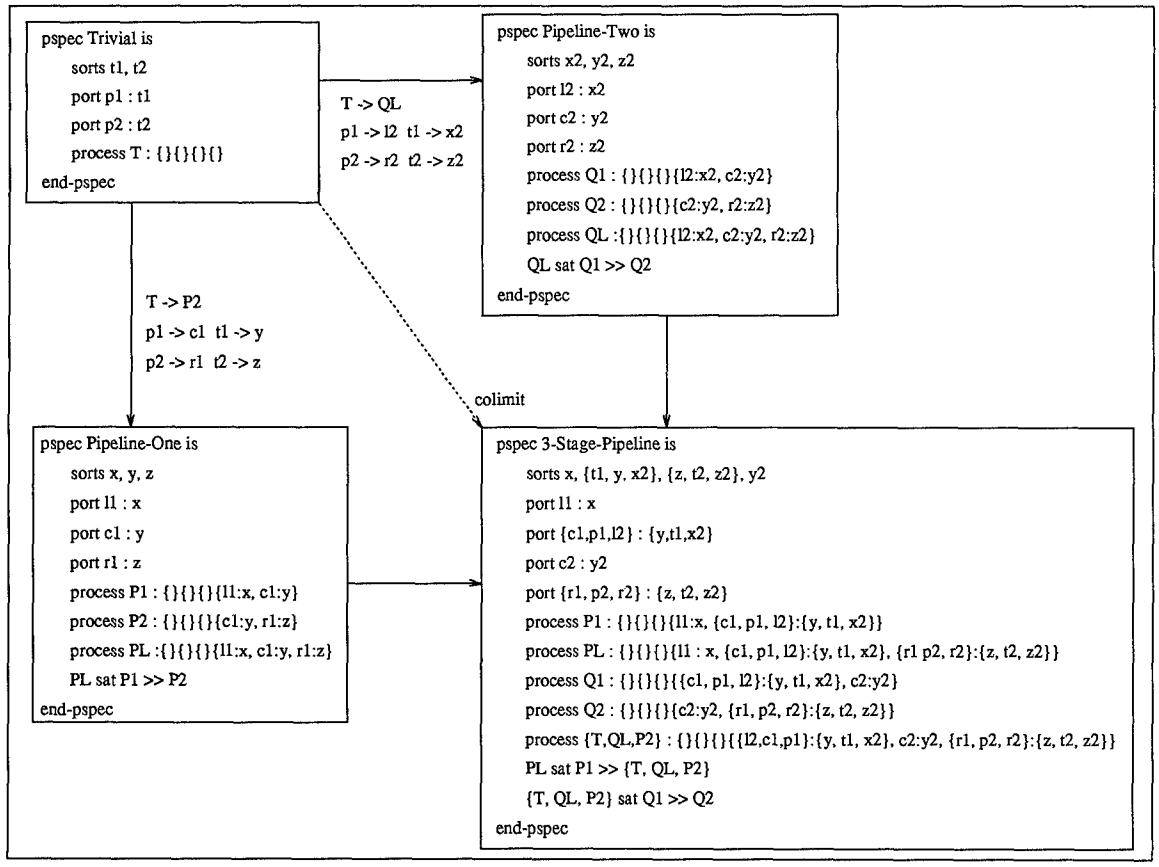


Figure 6.11 ISlang Diagram Depicting Recursive Application of Structuring Specifications

It is readily verified that the morphisms of Figure 6.11 are specification morphisms. This specification construction technique can be applied recursively to define, for example, *Q1* to be the pipelined process T where T *sat* $T1 \gg T2$ to obtain the structure $P1 \gg ((T1 \gg T2) \gg Q2)$, where $Q1$ *sat* $T1 \gg T2$ and $P2$ *sat* $Q1 \gg Q2$.

Note that the colimit object can be made more readable through the application of a specification translation operation. For example, the expression *port* {*c1, p1, l2*} : {*y, t1, x2*} could be mapped under translation to the expression *port p1-2-q1 : y*.

Combining Pipelined Designs via Channel Specifications. Two pipeline structuring specifications P and Q can be combined through unification of the channel *right* of P with the channel *left* of Q. This approach is highlighted in Figure 6.12. In the figure, the two stage pipeline $P1 \gg P2$ is combined with the two stage pipeline $Q1 \gg Q2$ to form the structure $P1 \gg P2 \| Q1 \gg Q2$ through unification of the channels *r1* and *l2*. An ISlang diagram reflecting this construction is shown in Figure 6.13. The statement $P1 \gg P2 \| Q1 \gg Q2$ shown in the figure is a conservative extension of the colimit object. Note that although $P2$ and $Q1$ share exactly one channel identified by the equivalence class $\{c, r1, l2\}$ such that the channel is used for output in $P2$ and input in $Q1$, it is *not* valid to conclude $P2 \gg Q1$. Communication over $\{c, l2, r1\}$ is concealed in $P2 \gg Q1$ but is not concealed in the colimit object. This is an important distinction, for it implies that **this technique cannot be used to form larger pipeline designs out of collections of smaller pipeline designs**. Instead, this technique can be used to define parallel compositions of communicating pipelines. Although the example presented here combined two 2-stage pipelines, this technique may be generalized to combine two pipelines of arbitrary, finite size.

Summary of Pipeline Architecture. As is the case with the other architecture theories, pipelined processes such as *PL* of Figure 6.11 can be combined using other architecture theories to define non-homogeneous designs. For example, several pipelined processes could be combined using a parallel architecture theory to define a design consisting of parallel pipelines. One such example occurs in computer graphics, where multiple parallel pipelines are used to render two and three dimensional images.

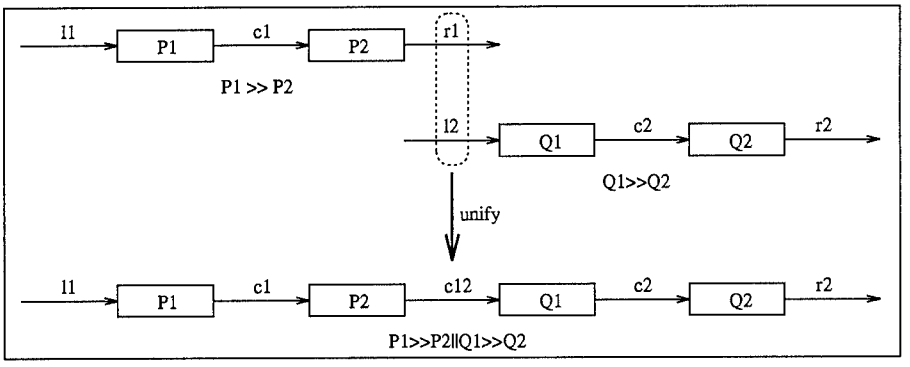


Figure 6.12 Channel Specifications and Pipeline Structure

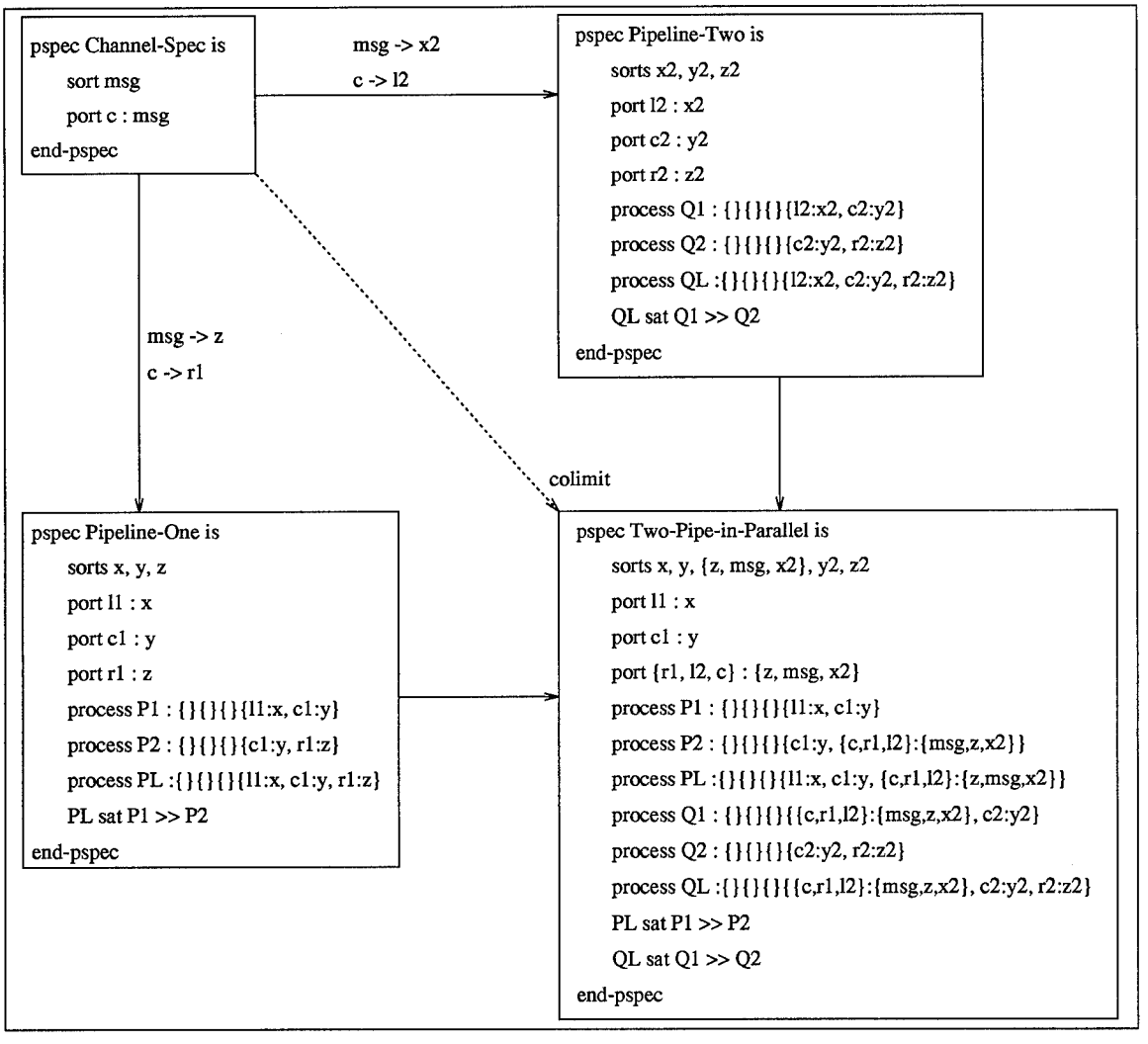


Figure 6.13 An ISlang Diagram for Two 2-Stage Pipelines in Parallel

Further classification of pipelined designs is possible. For example, a constraint expressing the requirement that the processes combined using the pipelined architecture theory be isomorphic leads to the notion of a homogeneous pipeline. If each stage in a n-stage homogeneous pipeline encapsulated a single operation of the form $f : D \rightarrow D$, then the pipeline would produce a value of $\underbrace{f(f(\ldots f(d)))}_{\text{f nested n deep}}$ for argument d. Definition of a taxonomy of pipeline designs is left for future research.

The architecture theory of the following section, pipeline architecture theory, also results in designs containing communication that is hidden from the outside environment.

Relationship between layered and pipelined designs. Consider once again the problem of computing $g(f(x))$. A layered design for this problem was defined in Section 6.3.2.2. A two-stage pipeline can also be defined for this problem. However, the protocol of layered systems differs from that of pipelined systems. If S was defined to be the pipeline $F \gg G$, where F *sat* $in?x \xrightarrow{CSP} (c!f(x) \xrightarrow{CSP} F)$ and G *sat* $c?d \xrightarrow{CSP} (out!g(d) \xrightarrow{CSP} G)$, then the set of traces of H would consist of communication events over the channels *in* and *out*, where $0 \leq tr\restriction in - tr\restriction out \leq 2$ for any trace tr in $traces(F \gg F)$. In contrast, the layered design $S' = (F' /\!/ G')$, where F' *sat* $p?x \xrightarrow{CSP} (q!f(x) \xrightarrow{CSP} F)$ and G' *sat* $in?x \xrightarrow{CSP} (p!x \xrightarrow{CSP} (q?y \xrightarrow{CSP} (out!g(y) \xrightarrow{CSP} G)))$ also computes $g(f(x))$, but $0 \leq tr\restriction in - tr\restriction out \leq 1$ for any trace tr in $traces(S')$.

That is, although communication on internal channels of a pipeline design is hidden from the surrounding environment, an n-stage pipeline design can accept up to n inputs before generating an output. In contrast, a layered design may be defined such that it will completely process an input and generate an output before accepting additional inputs. This distinction is highlighted by the example depicted in Figure 6.14.

As shown in the figure, the processes of the layered design have twice as many ports as the processes of the pipelined design. Although the two designs may be functionally equivalent, they are behaviorally distinct. Both $S \gg R \gg Q \gg P$ and $((S /\!/ R) /\!/ Q) /\!/ P$ may return the value $p(q(r(s(x))))$ for an input x, but the pipeline design may engage in up to four successive input events before engaging in up to four successive output events,

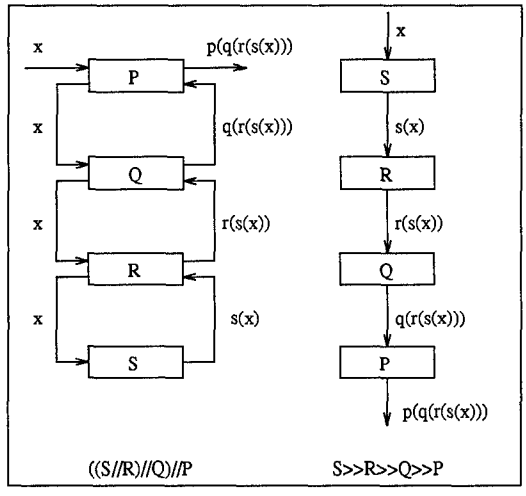


Figure 6.14 Comparison of Layered versus Pipelined Operation

whereas the layered design must engage in an output communication event following the engagement of an input communication event.

As this discussion implies, it is often possible to recast a design of one architecture theory into an equivalent design of another architecture theory, where the notion of equivalence may be context dependent. For example, Figure 6.14 depicts two designs which are functionally equivalent in that they produce equivalent values for equivalent inputs, yet are not trace equivalent. Formalization of this relationship between architectural designs is presented in more detail in Chapter VII.

As discussed in the opening paragraphs of this section, pipeline architecture could be defined as a constrained type of parallel architecture. Instead, the semantics of the process composition operator $\gg$ were used to define pipeline architecture theory. The architecture theory of the following section, *client-server architecture theory*, requires the use of constraints.

6.3.2.4 Client-Server Architecture. A client server design consists of a server process running in parallel with a finite number of client processes. The server process typically encapsulates a resource or collection of resources which are shared among the clients. Individual client processes have access to those resources through the server. For example, a server could encapsulate an UNIX socket and provide operations for reading

from and writing to the socket. Individual client processes have access to the socket via the operations provided by the server. How data obtained from the socket is used within individual clients is not of concern to the server process. A client-server design may contain multiple servers and multiple clients, and a server at one level of abstraction may be a client at another. Client processes and server processes are formally defined below.

Definition VI.16 Server. *A* server *is a process which*

1. *consists of at least two channels,* p_i *and* p_j*, one for input and one for output;*
2. *encapsulates at least one operation of the form* service : msg-in $\rightarrow$ msg-out*; and*
3. *satisfies* p_i?m $\stackrel{CSP}{\longrightarrow}$ (p_j!service(m) $\stackrel{CSP}{\longrightarrow}$ Server).

Client. *A* client *is a process which*

1. *contains at least two ports,* $p.left$ *and* $p.right$*, one for input and one for output; and*
2. *satisfies* $p!x?y$ *for a value* x *and a variable* y. □

Note that the CSP-based definition implies a server of a client-server design contains at least two ports for every client it services.

Client-server processes can be defined using a combination of the parallel composition operator and a constraint.

Definition VI.17 Client-Server processes *The class of client-server processes can be inductively defined as follows:*

1. *(Basis.) Any well-formed expression in* CSP_A *of sort* process *defining a server is a client-server process.*
2. *(Induction.) If* C *and* S *are well-formed expressions in* CSP_A *such that* C *defines a client process and* S *defines a server process, then the* CSP_A *expression* $P||Q$ *defines a client-server process provided* C *and* S *share a common pair of CSP channels, one from* C *to* S *and the other from* S *to* C.
3. *(Extremal.) No* CSP_A *expression defines a client-server process unless it can be created through a finite number of applications of clauses 1 and 2.* □

The basis case above identifies a server process as the simplest client-server process. The inductive case establishes a client-server process to be a parallel composition of a server and a collection of clients such that each client shares a pair of channels with the server and in which the clients are independent, parallel processes. A formal definition of client-server architecture is presented next.

Definition VI.18 Client-Server Architecture Theory. *A* client-server architecture theory *is a 2-tuple* $\langle \mathcal{I}, A_P \rangle$ *where* $\mathcal{I}$ *is a diagram of process specifications and* A_P *is a parallel architecture theory* $\langle O, R, \models \rangle$ *where two process symbols* C *and* S *in* O *can be combined using only the operator* $\|$ *provided*

1. *The process expressions of* C *and* S, $\Xi(C)$ *and* $\Xi(S)$, *both define client processes such that* chan(C) $\cap$ chan(S) $=\{\}$ *and* event(P)$\cap$event(S) $= \{\}$, *or*
2. $\Xi(C)$ *defines a client process and* $\Xi(S)$ *defines a server process, and* chan(C)$\cap$chan(S) $= \{c_1 : s_1, c_2 : s_2\}$, *where* c_1 *is a channel connecting the process defined by* $\Xi(C)$ *to the process defined by* $\Xi(S)$, *and* c_2 *is a channel connecting the process defined by* $\Xi(S)$ *to the process defined by* $\Xi(C)$.$\square$

This architecture theory permits a degenerate case of client-server design consisting solely of a parallel composition of client processes since it does not require that at least one of the process symbols in O define a server process. This requirement could be established through constraints.

Any client-server process can be defined using the above architecture theory. That is, client-server architecture theory is complete with respect to the class of client-server processes of Definition VI.17. A proof of this claim is similar to the proof of Theorem VI.3.

A structuring specification for a client server architecture is presented in Figure 6.15. This structuring specification requires a number of extensions to the ISlang language. For example, the statement *port* $s_{i_{in}}$ *: msg-in, i=1..max* identifies an indexed collection of ports of sort *msg-in.* Similarly, the statement *process Server:* $\{s_{i_{in}}$ *:msg-in,* $s_{i_{out}}$ *:msg-out*$\}$, $\{\}$, $\{\}$, $\{\}$, *i=1..max* identifies a process named *Server* that includes two indexed collection of ports $s_{i_{in}}$ *: msg-in, i=1..max* and $s_{i_{out}}$ *: msg-out, i=1..max.* The statement *process Client*$_i$*:*

pspec Client-Server is
sorts msg-in, msg-out
const max : Nat
op service : msg → msg
var x_i : msg-in, $i = 1..max$
var y_i : msg-out, $i = 1..max$
port $s_{i_{in}}$: msg-in, $i = 1..max$
port $s_{i_{out}}$: msg-out, $i = 1..max$
process Server: chan:{$s_{i_{in}}$:msg-in, $s_{i_{out}}$:msg-out},
events: {}, act: {}, var: {m:msg-in}, $i = 1..max$
process Client$_i$: chan: {$s_{i_{in}}$:msg-in, $s_{i_{out}}$:msg-out},
events: {}, act: {service: msg → msg},
var:{x_i : msg, y_i : msg} $i \in [1..max]$
process CS: chan: {$s_{i_{in}}$:msg-in, $s_{i_{out}}$:msg-out},
events: {}, act:{},
var:{m:msg-in, x_i : msg, y_i : msg}, $i = 1..max$
CS sat Server $\|_{i=1..max}$ Client$_i$
Client$_i$ sat $s_i!x_i?y_i$, $i = 1..max$
Server sat $[\,]_{i=1..max}$ ($s_{i_{in}}$?m $\xrightarrow{CSP}$ ($s_{i_{out}}$!service(m) $\xrightarrow{CSP}$ Server))
end-pspec

Figure 6.15 Client-Server Structuring Specification

{$s_{i_{in}}$:*msg-in*, $s_{i_{out}}$:*msg-out*}, {}, {}, {}, *i in [1..max]* identifies an indexed collection of processes *Client*$_1$, *Client*$_2$, ..., *Client*$_{max}$ where *max* is a natural number identifying the maximum number of clients supported by the server. Each client *Client*$_j$ has associated with it a pair of ports $s_{j_{in}}$ and $s_{j_{out}}$, where the use of the port pair in *Client*$_j$ is defined by the statement *Client*$_i$ *sat* $s_i!x_i?y_i$, where $s_i!x_i?y_i$ is defined to be the atomic process $s_{i_{in}}!x_i \xrightarrow{CSP} s_{i_{out}}?y_i$.

Each client process contains a set of variables, one used as an argument to the operation or resource *service* encapsulated in the process *Server*, and the other used to hold the return value of service invocation. The process *Server* is defined by the operation [ ] to be a process that can engage in communication over the first channel $s_{j_{in}}$ on which a client is prepared to communicate, and returns on the corresponding channel $s_{j_{out}}$ the value obtained from servicing the data supplied by the client. Note that the constant valued operation *max* is defined to be of sort *Nat*, where *Nat* is the natural numbers. However, process specifications lack functional axioms, so the sort *Nat* is unconstrained

in the specification *Client-Server*. Thus a constraint must be placed on the structuring specification expressing the requirement that any model of the sort *Nat* in *Client-Server* be isomorphic to a model of the natural numbers.

As is the case with the other structuring specifications provided in this chapter, the sorts, operations, and processes of the above specification can be refined using specification construction techniques. For example, by extending the process *Server* with two additional port symbols, $p.left$ and $p.right$, and extending the expression Ξ*(Server)* to include $p!x?y$ for some state variables of *Server*, it can be identified as a client of another server process. Extending *Server* to be a client of another server process can be accomplished via process specification morphism.

Also note that the clients may be refined using other structuring specifications. For example, a client may be defined using a process architecture theory to be a loose confederation of communicating processes, or a client may be defined to have a much more rigid structure, such as a pipeline.

Like client-server architecture theory, the architecture theory of the following section, *pipe-filter architecture theory*, is dependent on constraints for its definition.

6.3.2.5 Pipe-Filter Architecture. A pipe-filter architecture consists of two concurrent processes, a *pipe* process and a *filter* process.

Definition VI.19 Pipe. *A CSP_A expression P defines a pipe if P contains exactly two external channels* left *and* right *of a common sort such that* P sat left?x $\xrightarrow{CSP}$ (right!x $\xrightarrow{CSP}$ P).

Filter. *Given a pipe P, a CSP_A expression F defines a filter process in case $F \parallel P$ is well-formed.* □

The class of pipe-filter processes is formally defined below.

Definition VI.20 Pipe-Filter processes. *The class of pipe-filter processes is defined as follows:*

1. *Any well-formed expression in CSP_A defining a pipe process is a pipe-filter process.*

2. *If P and F are expressions of sort* process *in CSP_A such that P defines a pipe process and F defines a filter process, then $P \| F$ defines a pipe-filter process if*

   (a) chan(P) $\cap$ chan(F) $\neq \{\}$; *and*

   (b) $P \| F$ is well-formed. $\square$

That is, pipe and filter designs use pipe processes as buffers between filters. Case 1 above identifies a pipe-filter consisting of a pipe and an empty filter, and case 2 above places a filter in parallel with a pipe process to define a pipe-filter process. The constraint *channels(Pipe)* $\cap$ *channels(Filter)* $\neq \{\}$ can only be satisfied if the pipe process and the filter process share at least one channel. Note that the filter process can have additional channels that are not shared with the pipe process.

Now that the class of pipe-filter processes has been defined, a pipe-filter architecture theory can be defined.

Definition VI.21 Pipe-Filter Architecture Theory. *A* Pipe-Filter Architecture Theory *is a 2-tuple $\langle \mathcal{I}, A_P \rangle$ where $\mathcal{I}$ is a diagram of process specifications and A_P is a parallel architecture theory $\langle O, R, \models \rangle$ where R is restricted to contain only the CSP process composition operator* _ || _ : process, process $\rightarrow$ process *where the process symbols P and Q in O can be used to define a pipe-filter process only if*

1. $\Xi(P)$ *defines a pipe process,*
2. $\Xi(Q)$ *defines a filter process,*
3. chan(P) $\cap$ chan(Q) $\neq \{\}$, *and*
4. $\Xi(P) \| \Xi(Q)$ *is well-formed.* $\square$

The above architecture theory completely characterizes the class of CSP pipe-filter processes. A proof of this is similar to the proof of Theorem VI.3. Note that a filter process in a pipe-filter design may interact with the outside environment. That is, a filter in a pipe-filter design may take inputs from and supply outputs to the environment. A restriction could be placed on pipe-filter designs to preclude this type of filter-environment interaction. Specifically, the expression *chan(P)* $\cap$ *chan(Q)* $\neq \{\}$ of Definition VI.21 could

be replaced with the expression *chan(P) = chan(Q)*, in which case the only interaction a filter could have with the environment is through engagement in non-communication events. Such an architecture theory could produce a class of designs closely matching the informally defined class of batch transformation systems given in the introduction of this section.

A structuring specification encapsulating a pipe-filter architecture theory has the following form:

pspec Pipe-Filter-Structure is
    sort msg
    port left : msg
    port right : msg
    var m : msg
    process Pipe : {left:msg, right:msg},{},{},{m:msg}
    process Filter : {left:msg, right:msg},{},{},{}
    process Pipe-Filter : {left:msg, right:msg},{},{},{}
    Pipe-Filter sat Pipe || Filter
    Pipe sat left?m $\xrightarrow{CSP}$ (right!m $\xrightarrow{CSP}$ Pipe)
end-pspec

The above structuring specification can be defined as an extension to the parallel structuring specification of the previous section. Specifically, the above specification constrains one of the process symbols involved in the parallel composition to be a pipe process. The process *Filter* is defined to share a set of CSP channels with the *Pipe* process. Using specification construction operations, the structure of the process *Filter* can be refined. However, care must be given to ensure that the structure of the process *Filter* is compatible with the structure of *Pipe* when the two are combined. For example, *Filter* could be defined to be a collection of parallel processes wherein only one of them can read from the pipe and only one of them can write to the pipe.

Note that a pipe process acts as a buffer of size one. Larger buffers do not qualify as pipes. A buffer of size greater than one can engage in more than one input event before engaging in an output event. However, the pipe process above cannot engage in two or more input events in succession. A more general architecture theory than pipe-filter could

therefore be *buffer-filter*, where a buffer process is defined to be any first-in first-out buffer process. Clearly a simple pipe is one such process.

A filter in a pipe-filter design could be defined to be a sequential composition of processes. This class of pipe-filter designs is classified as *piped-batch sequential* designs. Compilers are often organized along these lines. Before formally defining piped-batch sequential architecture, batch-sequential architecture is formally defined.

6.3.3 Batch Architectures. A batch-sequential design, as mentioned in Section 6.2, is one where each process contained in the design processes all of its input data as a single entity. That is, a batch sequential system consists of a finite sequence of processes, each of which accept or obtain input values, operate over those values, produce a possibly empty result, and terminate.

The process connective in CSP that matches this informal description is the sequential composition operator $\_\ ;\ \_ : process, process \rightarrow process$. As with the other process composition operators, the semantics of the sequential composition operator are defined in (52). However, the semantics of the sequential composition operator do not preclude the formation of structures of the form $T\ sat\ U;V;W$ where one or more of U, V or W deviates from the input, process, output, terminate paradigm.

Each process expression used in the definition of a batch-sequential system could be restricted to the input, process, output paradigm through the use of constraints. That is, given a batch-sequential structure of the form $T\ sat\ T_1;\ T_2;\ \ldots;\ T_n$, a constraint C could be defined over T such that for any trace t in *traces(T)*, t is either

1. empty,
2. a finite sequence of input events,
3. a finite sequence of input events followed by a finite sequence of non-communication events, or
4. a finite sequence of input events followed by a finite sequence of non-communication events, and terminates with a finite sequence of output communication events.

Further restrictions could be placed on T, for example, the size of the sequence of input and output events could be specified. The point here is that constraints could be used to restrict the structure of each process used to compose a batch-sequential design. In the discussion that follows, no such constraints are implied.

Definition VI.22 Batch-Sequential processes. *The class of batch-sequential processes can be inductively defined as follows:*

1. *(Basis.) Any well-formed expression in CSP_A of sort* process *defines a batch-sequential process.*
2. *(Induction.) If P and Q are well-formed expressions in CSP_A such that P and Q define batch-sequential processes, then the CSP_A expression $P;Q$ defines a batch-sequential process.*
3. *(Extremal.) No CSP_A expression defines a batch-sequential process unless it can be created through a finite number of applications of clauses 1 and 2.* □

In general, $P;Q \ncong Q;T$. An architecture theory based on the above definition is straightforward.

Definition VI.23 Batch-Sequential Architecture Theory. *A batch-sequential architecture theory is a 2-tuple $\langle \mathcal{I}, A_P \rangle$ where $\mathcal{I}$ is a diagram of process specifications and A_P is a process-based architecture theory $\langle O, R, \models \rangle$ where R is restricted to contain only the CSP process composition operator* _ ; _ : process, process → process. □

The above architecture theory completely characterizes the class of batch-sequential designs. A proof of this is similar to the proof of Theorem VI.3.

A structuring specification for batch-sequential architectures has the following form:

```
pspec Batch-Sequential-Structure is
      process A : {},{},{},{}
      process B : {},{},{},{}
      process Batch : {},{},{},{}
      Batch sat A;B
end-pspec
```

This structuring specification can be used to develop batch-sequential designs consisting of a finite number of sequentially composed processes. For example, consider the domain of image recognition. Applications in this domain could be developed using a batch-sequential architecture theory. Figure 6.16 depicts some of the stages involved in image classification. As indicated in the figure, seven of these stages can be executed on digital computers. In the following paragraphs, a seven stage batch-sequential design corresponding to these seven stages is developed. Specification of the structure and operation of each of these stages is presented in more detail in Chapter VIII.

Creation of a seven segment batch-sequential design is straightforward. The structuring specification *Batch-Sequential-Structure* is used to define the structure of each process A and B in the structuring specification. That is, the structuring specification is used to recursively define the structure of each process of the batch-sequential design. Figure 6.17 depicts this recursive application.

Several structuring specifications are used. The names of the process symbols contained within these specifications have been altered to avoid confusion. The colimit of the diagram in Figure 6.17 yields a seven segment batch-sequential design. The following paragraphs describe the formation of the colimit object.

The figure is partitioned into two segments, one containing the process symbols *Batch, Batch2*, and *Batch3*, and the second containing *Batch, Batch4, Batch5*, and *Batch6*. The colimit of the first partition yields a batch-sequential design consisting of three sequentially composed processes. The colimit of the second partition yields a batch-sequential design consisting of four sequentially composed processes. Taken together, the two partitions define a sequential composition of seven processes. The formation of the colimit objects for each partition is further explained in the following paragraphs.

In the first partition, a trivial process specification containing a single process symbol is used to associate process A of specification *Batch-Sequential* with the process *Batch2* of specification *Batch-Sequential2*. *Batch2* is defined to be the sequential composition of the processes C and D. Another trivial specification is used to associate process C with another

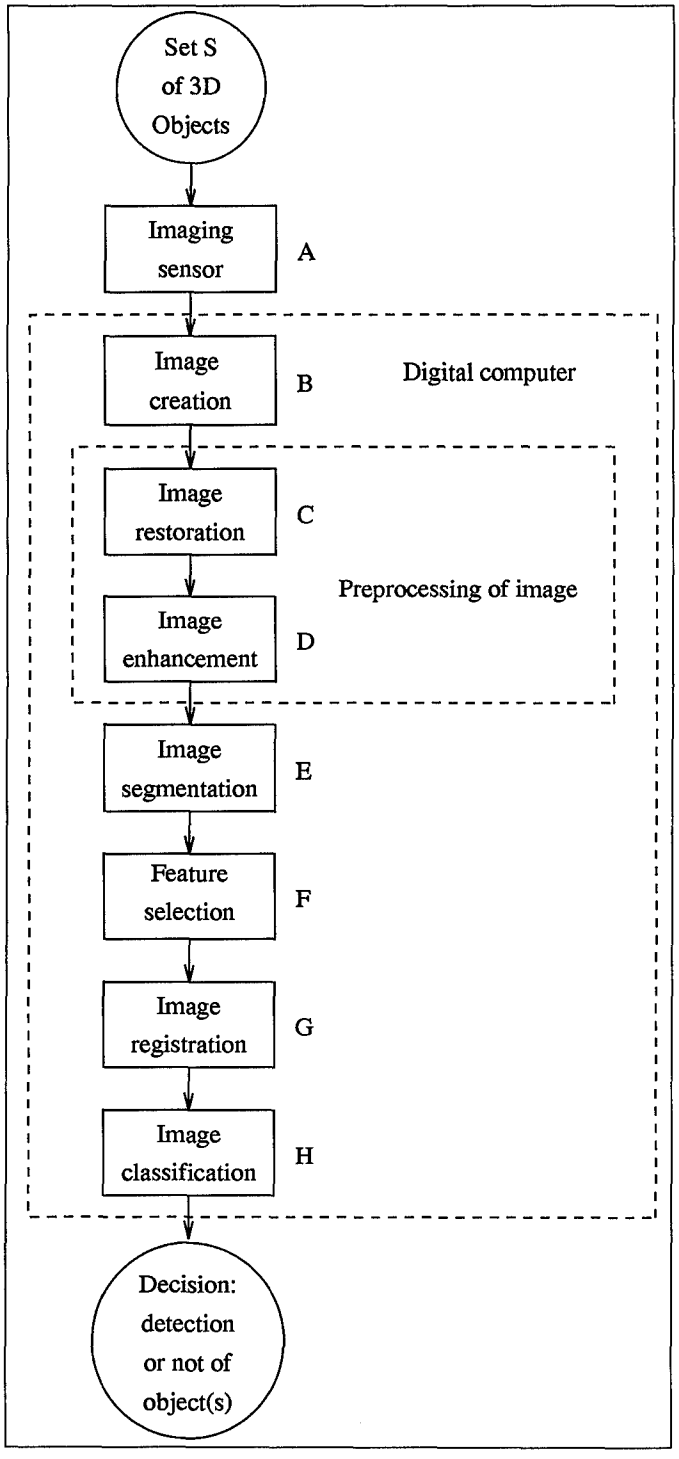


Figure 6.16 Some Stages for Image Recognition Systems (33:295)

batch-sequential process *Batch3*, where *Batch3* is defined to be the sequential composition of processes E and F. Thus the colimit object of the first partition is the following object:

```
pspec Colimit-Partition-1 is
      process {A,Batch2} : {},{},{},{}
      process B : {},{},{},{}
      process {C, Batch3} : {},{},{},{}
      process D : {},{},{},{}
      process E : {},{},{},{}
      process F : {},{},{},{}
      process Batch : {},{},{},{}
      Batch sat {A, Batch2};B
      {A, Batch2} sat {C, Batch3};D
      {C, Batch3} sat E;F
end-pspec
```

Renaming the equivalence class objects $\{A,$ *Batch2*$\}$ and $\{C,$ *Batch3*$\}$ to A and C respectively through specification translation results in a cleaner specification. In addition, the transitive property of *sat* can be exploited to highlight the process structure of the colimit object. Specifically, the expressions *Batch sat A;B*, *A sat C;D*, and *C sat E;F* can be combined to produce the expression *Batch sat ((E;F);D);B*, or equivalently, *Batch sat E;F;D;B*. The second partition of Figure 6.17 defines the structure of B to be a four segment sequence.

In the second partition, process B of the specification *Batch-Sequential* is associated with process *Batch4* of specification *Batch-Sequential4*. *Batch4* is defined to be a sequential composition of processes G and H. However, both G and H are associated with other batch sequential structures. Specifically, G is associated with *Batch6*, which is defined to be the sequential composition of K and L, and H is associated with process *Batch5*, where *Batch5* is defined to be the sequential composition of processes I and J. Thus the colimit object of the second partition is the object:

```
pspec Colimit-Partition-2 is
      process A : {},{},{},{}
      process {B, Batch4} : {},{},{},{}
      process {G, Batch6} : {},{},{},{}
      process {H, Batch5} : {},{},{},{}
      process I : {},{},{},{}
```

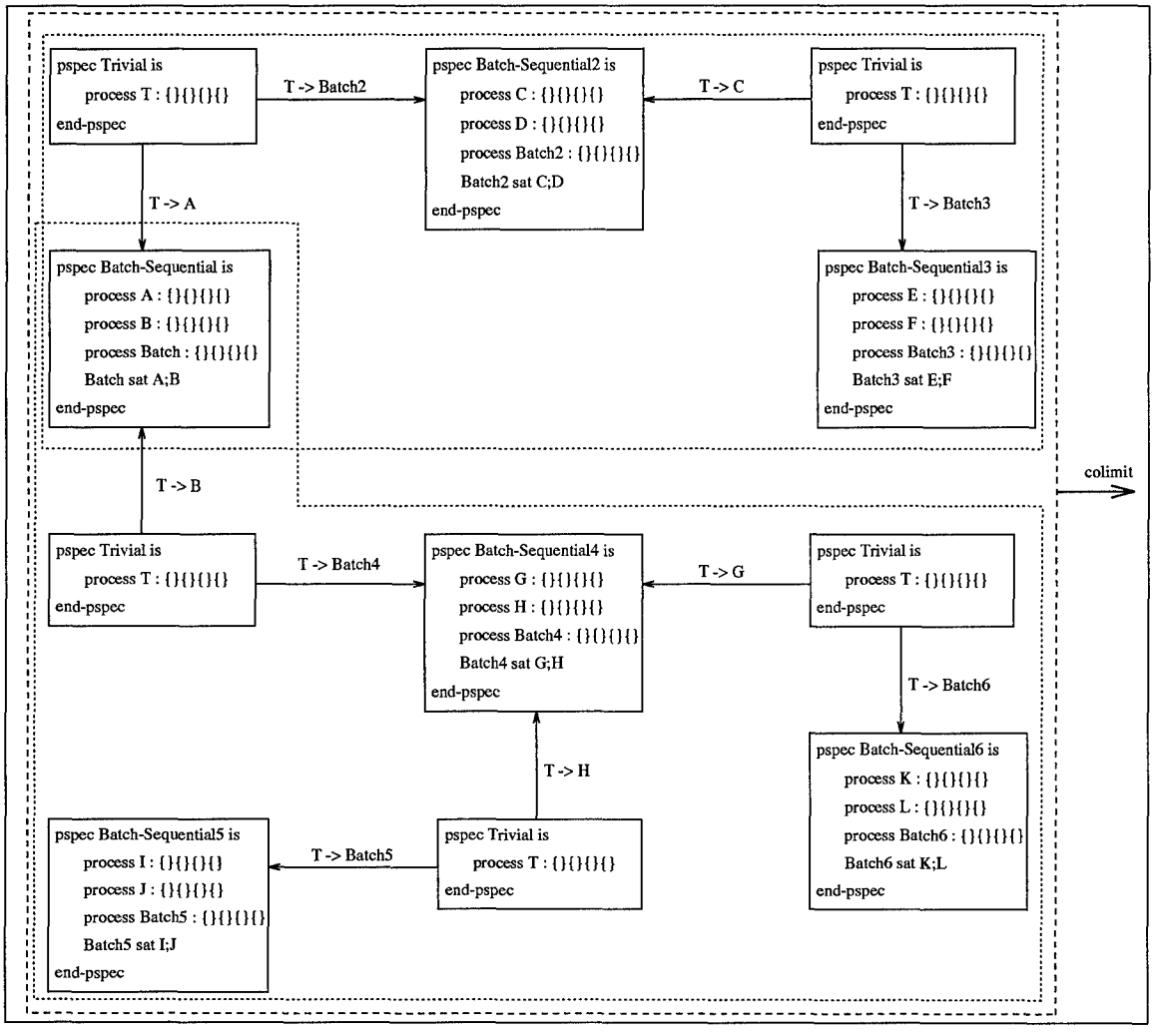


Figure 6.17 Creation of a Seven Segment Batch-Sequential Design

process J : {},{},{},{}
process K : {},{},{},{}
process L : {},{},{},{}
process Batch : {},{},{},{}
Batch sat A;{B, Batch4}
{B, Batch4} sat {G, Batch6};{H, Batch5}
{G, Batch6} sat K;L
{H, Batch5} sat I;J
end-pspec

Again, renaming through specification translation can be used to clean-up the specification.

When the colimit of both partitions is taken, the following specification is created, where the process symbol T has been omitted for clarity:

```
pspec Seven-Segment-Batch is
        process {A, Batch2} : {},{},{},{}
        process {B, Batch4} : {},{},{},{}
        process {C, Batch3} : {},{},{},{}
        process D : {},{},{},{}
        process E : {},{},{},{}
        process F : {},{},{},{}
        process {G, Batch6} : {},{},{},{}
        process {H, Batch5} : {},{},{},{}
        process I : {},{},{},{}
        process J : {},{},{},{}
        process K : {},{},{},{}
        process L : {},{},{},{}
        process Batch : {},{},{},{}
        Batch sat {A,Batch2};{B, Batch4}
        {A, Batch2} sat {C, Batch3};D
        {C, Batch3} sat E;F
        {B, Batch4} sat {G, Batch6};{H, Batch5}
        {G, Batch6} sat K;L
        {H, Batch5} sat I;J
end-pspec
```

The set of process expressions Ξ of the above specification can be simplified to produce the expression *Batch sat ((E;F);D);((K;L);(I;J))*, or equivalently, *Batch sat E; F; D; K; L; I; J*. Renaming these process symbols using the map

- *Batch* $\mapsto$ *ImageRec*,
- *E* $\mapsto$ *Creation*,
- *F* $\mapsto$ *Restoration*,
- *D* $\mapsto$ *Enhancement*,
- *K* $\mapsto$ *Segmentation*,
- *L* $\mapsto$ *Selection*,
- *I* $\mapsto$ *Registration*, and
- *J* $\mapsto$ *Classification*

results in the process specification of Figure 6.18. The specification *Image-Recognition* shown in the figure contains a number of "unused" process symbols. That is, of the thirteen

```
pspec Image-Recognition is
        process A : {},{},{},{}
        process B : {},{},{},{}
        process C : {},{},{},{}
        process Enhancement : {},{},{},{}
        process Creation : {},{},{},{}
        process Restoration : {},{},{},{}
        process G : {},{},{},{}
        process H : {},{},{},{}
        process Registration : {},{},{},{}
        process Classification : {},{},{},{}
        process Segmentation : {},{},{},{}
        process Selection : {},{},{},{}
        process ImageRec : {},{},{},{}
        ImageRec sat A;B
        A sat C;Enhancement
        C sat Creation;Restoration
        B sat G;H
        G sat Segmentation;Selection
        H sat Registration;Classification
end-pspec
```

Figure 6.18 A Batch-Sequential Specification for Image Recognition

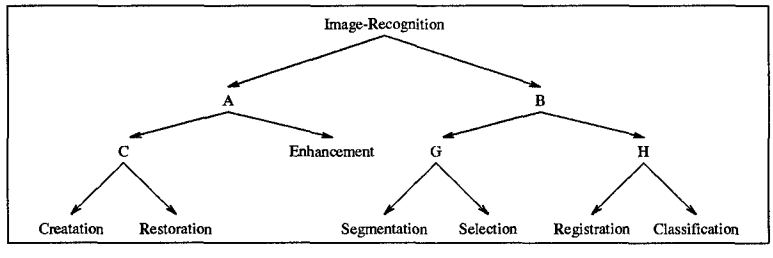


Figure 6.19 Source of Process Symbols

process symbols contained in the specification, only eight are required. The remaining five symbols, *A*, *B*, *C*, *G*, and *H* are a bi-product of the binary composition operator *";"* and can be eliminated through optimization. Figure 6.19 shows the source of these five symbols.

The semantics of the sequential composition operator preclude communication via a CSP channel between sequentially composed processes. That is, if *Q sat T;V*, then no communication over a CSP channel between *T* and *V* can occur. In this example, process *V* can begin execution only after process *T* has terminated. Thus the results of one process

can be passed to another process sequentially composed with the first only through either a shared concurrent process such as a pipe or through shared state. The first approach results in a class of designs called *piped-batch sequential*, while the second approach is a specific instance of a *repository*. Repositories are defined in Section 6.3.5. Piped-batch sequential architectures are defined in the following section.

6.3.4 Composite Architectures. This section examines an architecture theory defined in terms of other architecture theories. Specifically, this section defines a *piped-batch sequential* architecture theory, which is a composite of a the *pipe-filter* and the *batch sequential* architectures.

Processes composed together using a batch-sequential architecture theory have extremely limited inter-process communication options. The results of a process P sequentially composed with a process Q can be made available to Q through a shared concurrent process R provided P *sat* $c!x$, Q *sat* $p?x$, R *sat* $c?x \xrightarrow{CSP} (p!x \xrightarrow{CSP} SKIP)$, and W *sat* $R \parallel (P;Q)$. Process R in this expression acts as a buffer or pipe process between P and Q. R is established to run in parallel with the process defined by $P;Q$. This parallel structure cannot be generated using the batch-sequential architecture theory of the preceding section. However, it can be generated from a composition of the parallel architecture theory — specifically, from the pipe-filter architecture theory — and the batch-sequential architecture theory. A piped-batch sequential design consists of a batch-sequential process running concurrently with a pipe process. The following definition characterizes the class of piped-batch sequential processes.

Definition VI.24 Piped-Batch Sequential Processes. *The class of piped-batch sequential processes is defined as follows: Any process expression* Pipe || Filter *such that* Pipe *is a pipe process and* Filter *is a batch-sequential process is a piped-batch sequential process. Nothing else is a piped-batch sequential process.* □

Implicit in this definition is that the process *Filter* is inductively defined. The structure of a filter process is restricted only by the constraint that its subprocesses be sequentially composed, and when combined in a parallel with a pipe process, the resulting structure is well-formed. A filter in a piped-batch sequential design consists of a finite number

of sequentially composed processes, $P_1;P_2,\ldots,P_n$. Any communication between P_i and P_{i+1} can occur only indirectly through the process *Pipe*. Communication within any P_i, $i \in [1..n]$, may occur on channels contained within P_i. For example, P_i could be defined to be the parallel composition of the processes P_{i_1}, P_{i_2}, ..., P_{i_m}, in which case communication between P_{i_a} and P_{i_b}, $a, b \in [1..m]$ may occur independently of the process *Pipe*. This again highlights the possibility of non-homogeneous designs.

Now that the class of piped-batch-sequential processes have been defined, a definition of the corresponding architecture theory may be given.

Definition VI.25 Piped-Batch Sequential Architecture. Piped-Batch Sequential Architecture. *A piped-batch sequential architecture theory is a pipe-filter architecture theory* $PF \triangleq \langle \mathcal{I}, A_{PF} \rangle$ *and a batch-sequential architecture theory* $BS \triangleq \langle \mathcal{I}, A_{BS} \rangle$ *defined over a common process specification diagram* $\mathcal{I}$ *such that the filter process of the architecture theory* PF *is defined using the architecture theory* BS.

Designs created using a piped-batch sequential architecture theory have the form Pipe || Filter, *where*

1. Pipe *is a pipe process, and*
2. Filter *is a batch-sequential design*

such that Pipe || Filter *is well-formed.* □

The process *Pipe* in the above definition acts as a buffer between successive sequentially composed processes used to define the filter. Note that no restriction is placed on any process F_i used to define the filter concerning communication with the environment. That is, F_i may be defined to include explicit communication with the environment. A constraint restricting communication of filter processes could be defined. For example, the constraint *chan(Filter) = chan(Pipe)* effectively precludes any filter process F_i in *Filter* from engaging in any communication events other than with the process *Pipe*.

Figure 6.20 depicts process specifications from which a piped batch-sequential design may be defined. Note that the process symbols V and Y shown in the figure may be

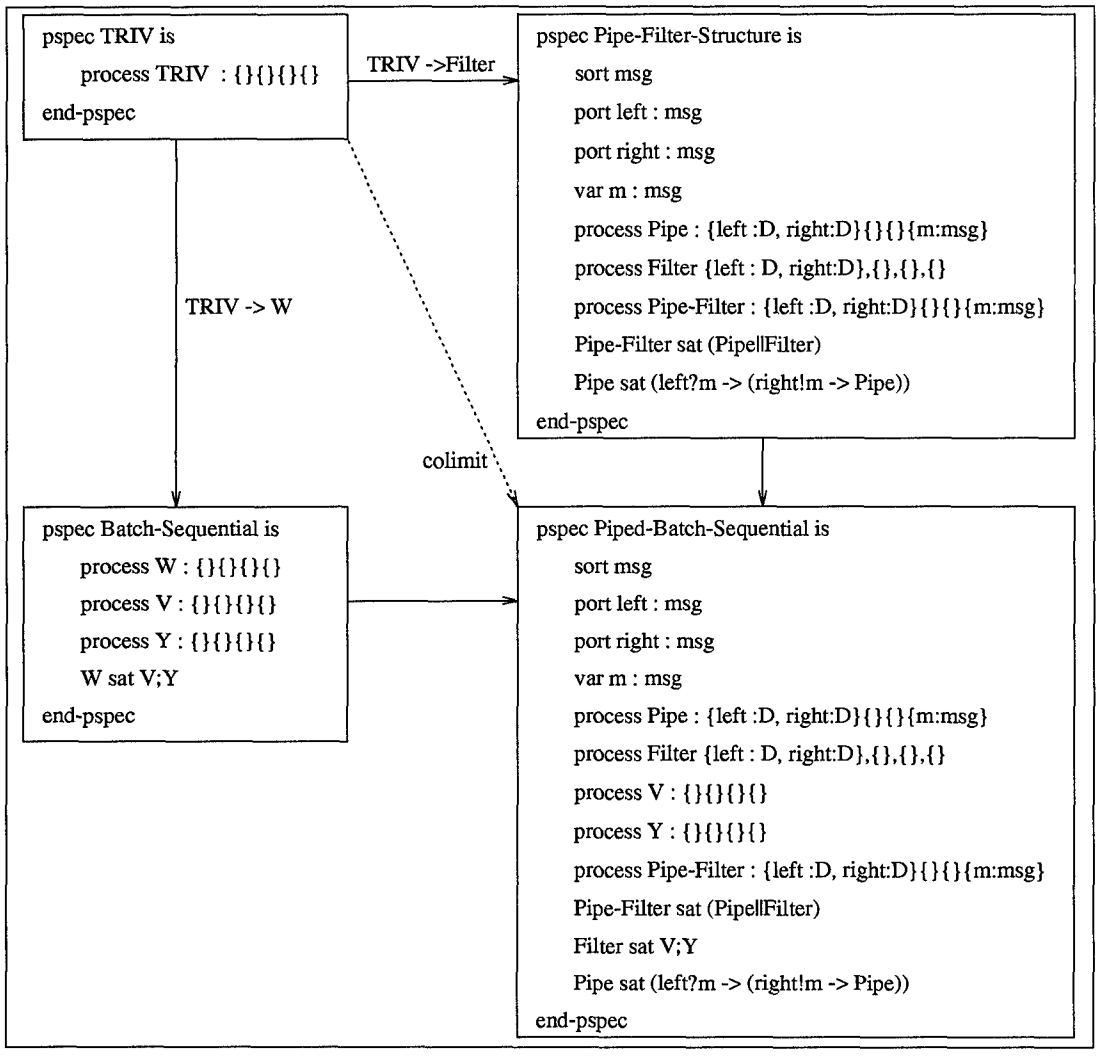


Figure 6.20 Piped Batch-Sequential Structure

given further definition through process specification morphism. In fact, V and Y could be defined to be sequential composition of other processes.

Compilers are often organized around a piped-batch sequential paradigm. The filter process of a compiler could consist of the sequential composition of a parser, a semantic analyzer, an optimizer, and a code generator. The data communicated over the pipe in this case could consist of an annotated abstract syntax tree (AAST) representation of the input program. The parser reads the input program, creates an AAST representation of the program, writes this structure to the pipe, and terminates. The semantic analyzer reads the AAST from the pipe, operates over it, generates output error messages (if any),

writes the modified AAST to the pipe, and terminates. The operation of the remaining filters are similarly defined. In Chapter VIII, the batch-sequential design of Figure 6.18 is extended to a piped-batch sequential design.

This section has presented one approach to achieving communication between sequentially composed processes. The following section introduces another mechanism — one based on shared variables — that can be used to achieve communication between sequentially composed processes. The architecture theory of the following section is defined using constraints.

6.3.5 Constraint-Based Architectures. Architecture theories can be defined through constraints placed on other architecture theories. For example, the layered architecture theory of Section 6.3.2.2 could be defined as a constrained parallel architecture theory. The architecture theory of this section, *repository architecture theory*, is defined as a constrained type of process-based architecture theory. Before defining repository architecture theory, the notion of constraint-based architecture theory is made precise.

Definition VI.26 Constraint-Based Architecture Theory. *A constraint-based architecture theory is a 3-tuple* $\langle \mathcal{I}, A_P, C \rangle$ *where* $\mathcal{I}$ *is a diagram of process specifications,* A_P *is an architecture theory* $\langle O, R, \models \rangle$*, and* C *is a set of constraints defined over* R *and* O *such that each* $c \in C$ *satisfies one of the following:*

1. *c defines the alphabet of a process expression of a process symbol in* O*.*
2. *c defines both the alphabet and a set of traces that a process expression of a process symbol in* O *must exhibit.*
3. *c restricts how a relation in* R *can be used to define structure.* □

That is, a constraint-based architecture theory is an architecture theory in which either some of the process symbols in O have been given partial definition or in which application of the composition operations in R is restricted. Client-server architecture theory and pipe-filter architecture theory are each constraint-based architecture theories. Repository architecture theory is also a constraint-based architecture theory.

A repository design consists of a collection of processes operating over a shared structure. Repositories are ubiquitous in compute science. For instance, the JOVIAL programming language is designed around this theme where the comm-pools of the language play the role of the shared data structures. Common-blocks in FORTRAN-77, header files in C and C++, and Ada package specifications can also be used to define common data pools in their respective languages.

Some repository designs have been given special treatment in the literature (e.g., (74, 7)). For example, Blackboard systems are a constrained class of repository where the shared structure is the blackboard. A collection of processes operating over the shared structure are usually either control units which post changes to the shared structure or knowledge sources which monitor the shared structure and generate requests to alter some subset of the shared structure. Some compilers are based on repository architectures. For example, the shared structure for a compiler or family of compilers could be an abstract syntax tree. As the parser builds the tree, semantic analyzers, optimizers, and translators may begin operating over the tree. The order of execution for repository-based compilers is consistent with the order of execution of batch-sequential or piped-batch sequential implementations, but in a repository based implementation, each stage may be concurrently executing. For example, a section of an abstract syntax tree representation of an input program may be built by the parser then semantically analyzed while other portions of the tree are built.

The following definition characterizes the class of repository designs.

Definition VI.27 Repository processes. *The class of repository processes is inductively defined as follows:*

1. *(Basis.) Any well-formed expression in CSP_A of sort* process *containing at least one state-holding variable defines a repository process.*
2. *(Induction.) Let P and Q be two well-formed expressions in CSP_A such that P and Q define repository processes. If P and Q share at least one variable, then the CSP_A expression $P \rho Q$ defines a repository process where ρ is any binary process composition operator of CSP_A such that $P\ \rho\ Q$ is well-formed.*

3. *(Extremal.) No CSP_A expression defines a repository process unless it can be created through a finite number of applications of clauses 1 and 2.* □

The above definition does not restrict communication over CSP channels. Processes of a repository design can communicate with each other through shared data structures. For example, semaphore variables can be shared between processes to ensure protection of critical regions as follows. Denote by *mutex* a semaphore variable whose initial value is the integer 1, and define the parameterized atomic operations *wait(S) : while (S ≤ 0) do SKIP; S := S - 1* and *signal(S) : S := S + 1* where ":=" is destructive assignment. If each process requiring access to the critical section guarded by the semaphore variable *mutex* includes *mutex* in its set of variables and observes the protocol *wait(mutex); critical-section; signal(mutex)*, then only one process at a time will have access to the critical region.(97) Note that critical section protection could also be provided through a shared, subordinate semaphore process *SEM* where *SEM sat* $p \xrightarrow{CSP} (v \xrightarrow{CSP} SEM)$ wherein p and v are events. If p and v are included in the alphabets of each process P_i of a structure P requiring access to the critical section, and if each such process observes the protocol $mutex.p \xrightarrow{CSP} (critical\text{-}section \xrightarrow{CSP} mutex.v)$ with *mutex:SEM*$/\!/P$, then the critical section will be protected.(52) The point here is that shared data structures, like shared events, permit a form of communication not involving channels.

A constraint-based architecture theory leading to repository designs is defined next.

Definition VI.28 Repository Architecture Theory. *A* repository architecture theory *is a constraint-based architecture theory* $\langle \mathcal{I}, A_P, C \rangle$ *where* C *requires that:*

1. *For any process symbol* P *in* O, var (P) *is not empty; and*
2. *Two process symbols* P *and* Q *in* O *can be related through the process composition operator* r *in* R *only if* $P\ r\ Q$ *is well-formed and if* var(P) ∩ var(Q) ≠ {}. □

Clause 1 requires that every process of a repository design contain state variables, and the statement *var(P)* ∩ *var(Q)* ≠ {} in Clause 2 requires repository designs to share some set of variables. The above architecture theory is complete with respect to the class of repository processes. A proof of this is similar to the proof of Theorem VI.3.

The constraint reflected in the architecture theory can be used to achieve communication between sequentially composed processes. That is, if P *sat* P_1; P_2; ...; P_m where P_1, P_2, ..., P_m share a common variable, the value of the variable when process P_i terminates is the same as the value of the variable when P_{i+1} begins. Continuity of the value of the variable is ensured by process P and the definition of *value*. Note that specification of repository-based applications will be feasible in ISlang only after constraint representation and constraint-based reasoning over process specifications have been defined.

6.4 Summary

This chapter has introduced and formally defined several architecture theories, including functional, process, and component based architecture theories. Each of these architecture theories was defined in terms of category theory.

Functional Architecture Theory is used to define operators in terms of other operators using the the product and coproduct operations of category theory. Application of a functional architecture theory to define the structure of operations defined in functional specifications was explored.

Several process-based architecture theories such as layered, pipeline, and batch-sequential process-based architecture theories were defined. Structuring specifications encapsulating the composition operators of architecture theories were introduced and defined, and their use in defining structure was explored via the development of several simple process-based specifications.

Figure 6.21 depicts a taxonomy of the architecture theories presented in this chapter. The arrows in the figure denote specialization. At the root of the taxonomy is the general process based architecture theory of Definition VI.6. Each of the architecture theories defined in the previous subsections are specializations of this process based architecture theory.

Batch sequential, pipeline, and parallel architecture theories were defined by restricting the set of operators used to combine processes to form other processes. Repositories were defined using constraints; specifically, that each process in the design share a common

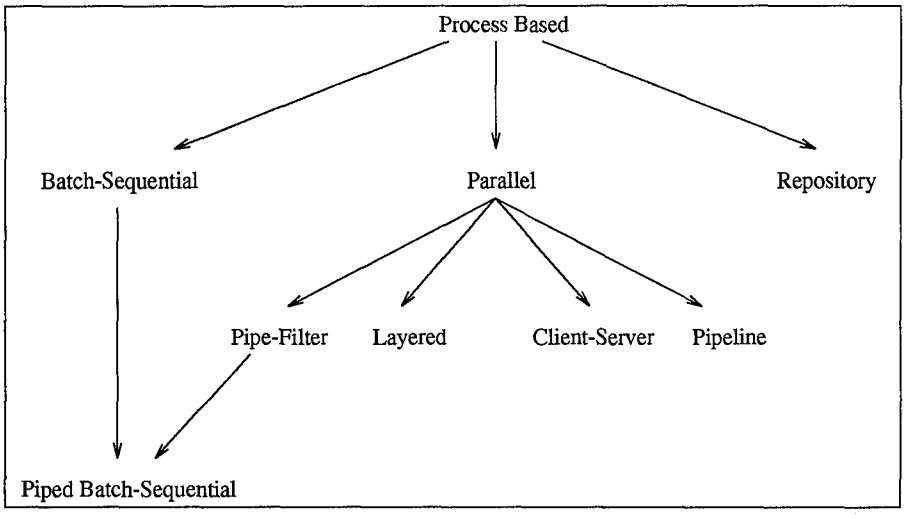


Figure 6.21 Architecture Taxonomy

variable. The architecture theories for client-server, layered, and pipe-filter, were defined as specific types of parallel architecture. But does this imply that any layered design, pipe-filter design, or client-server design is also a parallel design? Is the converse also true? Under what conditions can a parallel design be translated into a pipe-filter design? Into a layered design? Or into a client-server design?

Relationships between some of the process-based architecture theories developed in this chapter were briefly explored. Among the results of this preliminary investigation is that parallel architecture theory appears to be a more expressive architecture theory than any of the other process-based architecture theories. In addition, the use of the concealment operator to define relationships between process specifications was shown in induce a a specification morphism from the target of the concealment to the source of the concealment. The implication of this finding is that application of the concealment operator to a process specification results in a specification whose set of models may be larger than the set of models of the source specification. The next chapter explores in greater detail relationships between the process-based architecture theories defined in this chapter.

VII. Analysis of Process-Based Architecture Theories

7.1 Introduction

The previous chapter defined several architecture theories. Some of the architecture theories were introduced as specializations of more general architectures. For example, a client-server architecture was defined to be a specific type of parallel architecture. Similarly, some of the architecture theories were defined to be combinations of other theories. This chapter highlights relationships between the various architecture theories defined in the proceeding chapter. These relationships are developed through an exploration of the issues involved in translating a design in one architecture theory into a design in another architecture theory such that the translation process is behavior preserving. For example, this chapter investigates whether layered designs can be translated under process specification morphism to pipelined designs.

This chapter is organized into the following subsections:

1. Section 7.2 defines a semantic weaker than trace semantics. This weaker semantic can be used to determine if a translation preserves traces consisting solely of communication events.
2. Section 7.3 describes the relationship between parallel and layered designs.
3. Section 7.4 describes the relationship between parallel and pipeline designs.
4. Section 7.5 describes the relationship between layered and pipeline designs; and
5. Section 7.6 describes other relationships.

7.2 Mathematical Foundations

The definition of successful design translation may be context sensitive. Depending on the application, a translation that preserves trace satisfaction may not be required; perhaps some communication can be concealed from the environment without unduly impacting application functionality. At issue is whether a translation from one design to another can be defined such that it preserves at least a subset of the behavior of the source design.

One measure of successful translation, trace satisfaction, was defined in Chapter VI. A weaker semantic, one that ignores everything except for communication over external CSP channels, is defined next.

Definition VII.1 External Compatibility. *Given two process expressions* D_1 *and* D_2 *where* $\text{chan}_{\text{external}}(D_1) = \{c_1, c_2, \ldots, c_k\}$, D_2 *is* externally compatible *with* D_1 *if for any trace* $t \in \text{traces}(D_1)$ *there is a trace* $t' \in \text{traces}(D_2)$ *such that* $t \restriction \{c_1, c_2, \ldots, c_k\} = t' \restriction \{c_1, c_2, \ldots, c_k\}$. *The expression* $D_1 \sqsubseteq D_2$ *is used to indicate that* D_2 *is externally compatible with* D_1.

If $D_1 \sqsubseteq D_2$ *and* $D_2 \sqsubseteq D_1$ *then* D_1 *and* D_2 *are* externally equivalent *designs. The notation* $D_1 \equiv_e D_2$ *will be used if* D_1 *is externally equivalent to* D_2. □

For example, consider the following process expressions:

P *sat* $P_1 \| P_2$
P_1 *sat* $(q?x \xrightarrow{CSP} Skip \mid c?x \xrightarrow{CSP} (r!f(x) \xrightarrow{CSP} Skip))$
P_2 *sat* $left?y \xrightarrow{CSP} (c!g(y) \xrightarrow{CSP} (r?v \xrightarrow{CSP} (right!h(v) \xrightarrow{CSP} Skip)))$

Q *sat Start;* $Q_1 \gg Q_2 \gg Q_3$
$Start$ *sat* $e \xrightarrow{CSP} Skip$
Q_1 *sat* $left?y \xrightarrow{CSP} (c_{12}!g(y) \xrightarrow{CSP} Skip)$
Q_2 *sat* $c_{12}?x \xrightarrow{CSP} (c_{23}!f(y) \xrightarrow{CSP} Skip)$
Q_3 *sat* $c_{23}?v \xrightarrow{CSP} (right!h(v) \xrightarrow{CSP} Skip)$

Then $chan_{external}(Q) = \{left, right\}$, and $traces(P) \restriction \{left, right\} = \{\langle\rangle, \langle\ left.y\ \rangle, \langle\ left.y, right.h(f(g(y)))\ \rangle\}$. Similarly, $traces(Q) \restriction \{left, right\} = \{\langle\rangle, \langle\ left.y\ \rangle, \langle\ left.y, right.h(f(g(y)))\ \rangle\}$. Thus $Q \sqsubseteq P$. That is, P and Q exhibit the same behavior with respect to the external channels *left* and *right*. Note however $P \not\sqsubseteq Q$. Also note that $traces(P) \not\subseteq traces(Q) \restriction \alpha P$ and $traces(Q) \not\subseteq traces(P) \restriction \alpha Q$.

External compatibility, like trace satisfaction, defines a partial order over designs:

1. Reflexive - A design is externally compatible with itself. That is, for any design D, $D \sqsubseteq D$.
2. Transitive - If $D_1 \sqsubseteq D_2$ and $D_2 \sqsubseteq D_3$, then $D_1 \sqsubseteq D_3$.

3. Antisymmetric - By definition, if $D_1 \sqsubseteq D_2$ and $D_2 \sqsubseteq D_1$, then $D_1 \equiv_c D_2$.

If process P is externally compatible with process Q, then P exhibits at least the external communication behavior of Q. However, P and Q may exhibit quite distinct internal behavior in terms of their trace. That is, $D_1 \equiv_c D_2$ does not imply that D_1 and D_2 are trace equivalent. This fact is formalized in the following theorem.

Theorem VII.1 *Given two well-formed process expressions P and Q, $Q \sqsubseteq P$ does not imply* traces(Q) $\subseteq$ traces(P) $\restriction \alpha$ Q.

Proof. *Consider the simple process expressions P sat $e \xrightarrow{CSP} (c?x \xrightarrow{CSP} q!f(x))$ and Q sat $c?x \xrightarrow{CSP} (e \xrightarrow{CSP} q!f(x))$. Clearly $Q \sqsubseteq P$. However,* traces(P) $= \{\langle\rangle, \langle e\rangle, \langle e, c.v\rangle, \langle e, c.v, q.f(v)\rangle\}$ *and* traces(Q) $= \{\langle\rangle, \langle c.v\rangle, \langle c.v, e\rangle, \langle c.v, e, q.f(v)\rangle\}$, *thus* traces(Q) $\not\subseteq$ traces (P) $\restriction \alpha$Q. ■

This theorem implies that translations that can be shown to preserve external compatibility are not necessarily process specification morphisms.

At a minimum, external compatibility must be maintained when translating a design of one architecture theory to a design of another architecture theory. Ideally, trace satisfaction should be preserved as well. However, it may not be possible to preserve trace satisfaction between designs, yet it may be possible to preserve external compatibility between designs. As formalized in the following theorem, preserving trace satisfaction implies that external compatibility is preserved as well.

Theorem VII.2 *Given the well-formed process expressions P and Q such that $P \equiv_T Q$, then $P \equiv_c Q$.*

Proof. *By definition, $P \equiv_T Q$ if and only if every trace of P is also a trace of Q. Denote by $tr = \langle e_1, e_2, \ldots, e_m\rangle$ an arbitrary trace of P. If tr contains no communication events, then neither input nor output was accepted or produced by P during tr. Because tr is also a trace of Q, this implies that for such traces, $P \equiv_c Q$ is trivially true. On the other hand, if tr includes communication events, $P \equiv_T Q$ implies that both P and Q must have engaged in the same sequence of communication events of the same values over the same channels, implying $P \equiv_c Q$.* ■

This theorem can be generalized as follows.

Theorem VII.3 *Given the well-formed process expressions P and Q such that* traces(Q) $\subseteq$ traces(P)$\restriction \alpha Q$, *then* $Q \sqsubseteq P$.

Proof. *Denote by* $tr = \langle e_1, e_2, \ldots, e_m \rangle$ *an arbitrary trace of* Q. *If* tr *contains no communication events, then neither input nor output was accepted or produced by* Q *during* tr. *This implies that for such a trace,* $Q \sqsubseteq P$ *is trivially true. On the other hand, if* tr *includes communication events, then with* $tr \in$ traces(Q) *and* $tr \in$ traces(P)$\restriction \alpha Q$, *if* Q *can engage in a sequence* $s = \langle c_1.v_1, c_2.v_2, \ldots c_k.v_k \rangle$ *of communication events over the channels* $\{c_1, c_2, \ldots, c_k\}$, *then* traces(Q) $\subseteq$ traces(P)$\restriction \alpha Q$ *implies there exists a trace* t' *in* traces(P) *such that* $t' \restriction \{c_1, c_2, \ldots, c_k\} = s$, *which implies* $Q \sqsubseteq P$. ■

External compatibility can be generalized for use with process specification morphisms as stated in the following definition.

Definition VII.2 External Compatibility. *Given the process specifications* pSP *and* pSP' *and a morphism* σ *from* pSP *to* pSP', pSP' *is externally compatible with* pSP *if for every process symbol* P *in* pSP, $\sigma_\kappa(P) = P'$ *implies* traces $(\Xi(\mathrm{P}))$ $\restriction$chan(P) $\subseteq$ (traces $(\Xi'(P'))$ $\restriction$ chan $(P'))$ $|_\sigma$, *where* traces$(\Xi'(P'))|_\sigma$ *is the set* $\{t|_\sigma \mid t \in traces(\Xi'(P'))\}$. □

There is a subtle distinction between the above definition and Definition VII.1. Specifically, external compatibility in Definition VII.1 required both processes to share a common set of channels, while no such requirement is made in the above definition. Definition VII.2 allows a process defined by a process expression of a process specification pSP' to accept values over channels not contained in pSP and to produce additional results and communicate them over channels not contained in pSP. Definition VII.2 can be used to formalize the relationship between process specification morphisms and external compatibility as stated in the following theorem.

Theorem VII.4 *Given process specifications* pSP *and* pSP', *if a process specification morphism* σ *from* pSP *to* pSP' *exists, then* pSP' *is externally compatible with* pSP.

Proof. *Denote by* σ *a process specification morphism from process specification* pSP *to process specification* pSP'. *Because* σ *is a specification morphism, for any process*

symbol P' in κ' such that $\sigma_\kappa(P) = P'$, traces($\Xi(P)$) $\subseteq$ traces($\Xi'(P')$)$|\ \sigma$. This implies traces(Ξ(P))⌈chan(P) ⊆ (traces($\Xi'(P')$) ⌈ chan(P'))$|_\sigma$. ■

Thus, if a process specification morphism between two specifications can be defined, then the target specification is externally compatible with the source specification.

As stated in the following theorem, if a design D_1 can be translated under specification morphism to a design D_2, and if D_2 can be translated under specification morphism to D_1, then D_1 and D_2 are isomorphic.

Theorem VII.5 *Given two well-formed specifications S_1 and S_2, if a specification morphism from S_1 to S_2 exists and a specification morphism from S_2 to S_1 exists, then $S_1 \cong S_2$.*

Proof. *Denote by σ_{12} a specification morphism from S_1 to S_2, and denote by σ_{21} a specification morphism from S_2 to S_1. Because σ_{12} is a specification morphism, $S_1 = S_2\ |_{\sigma_{12}}$ and $S_2 = S_1\ |_{\sigma_{21}}$, which implies $S_1 = (S_1 \mid \sigma_{21})\ |_{\sigma_{12}}$ and $S_2 = (S_2 \mid \sigma_{12})\ |_{\sigma_{21}}$. These facts will be used to establish that σ_{12} and σ_{21} are bijections. A specification morphism is bijective if and only if it is both one to one and onto.*

1. One-to-one. *Suppose σ_{12} is not one-to-one. Then there exists a collection $\{a_1, a_2, \ldots, a_m\}$ of elements of S_1, $m \geq 2$, such that $\sigma_{12}(a_i) = b$, $1 = i..m$, for some b in S_2. This implies that $b\ |_{\sigma_{12}} = \{a_1, a_2, \ldots, a_m\}$. Because $(b\ |_{\sigma_{12}})\ |_{\sigma_{21}} = b$, this implies that $a_i\ |_{\sigma_{21}} = b$, $i = 1..m$. But then $(a_i\ |_{\sigma_{21}})\ |_{\sigma_{12}} = b\ |_{\sigma_{12}} = \{a_1, a_2, \ldots, a_m\} \neq a_1$, for $i = 1..n$. This implies that σ_{12} must be one-to-one. A similar argument shows that σ_{21} must be one-to-one.*

2. Onto. *Suppose σ_{12} is not onto. Then there exists a b in S_2 such that $b\ |_{\sigma_{12}}$ is undefined. This implies that $(b\ |_{\sigma_{12}})\ |_{\sigma 21}$ is undefined as well, which implies $(b\ |_{\sigma_{12}})\ |_{\sigma 21} \neq b$ which contradicts the requirement that $(b\ |_{\sigma_{12}})\ |_{\sigma 21} = b$. Thus σ_{12} must be onto. A similar argument shows that σ_{21} must also be onto.*

By conditions 1 and 2 above, σ_{12} and σ_{21} are both one-to-one and onto, which implies that they are both bijections. Because $\sigma 12 : S_1 \rightarrow S_2$ is a bijection, $S_1 \cong S_2$. ■

The implication of this theorem is that if every design D of an architecture A can be translated under specification morphism to a design D' of an architecture A', and if every

design D' of A' can be translated to a design D of A, then A and A' are isomorphic architectures. For example, if every layered design can be translated under specification morphism to a pipelined design, and if every pipelined design can be translated under specification morphism to a layered design, then pipeline architecture is isomorphic to layered architecture.

This subsection has introduced some of the desired properties that should be preserved whenever a design is translated. These properties, trace satisfaction and external compatibility, each define partial orders that can be used to determine the relative success of a translation effort. The following subsections describe whether translations between designs of some of the architecture theories of Chapter VI can be defined such that these properties are preserved.

7.3 Relationship Between Parallel and Layered Designs

As shown in Figure 6.21, a layered architecture is a specialized parallel architecture. But can this specialization be defined via a specification morphism? Can a design using a parallel architecture theory be translated under specification morphism to a design using a layered architecture theory? Conversely, can a layered design be translated under specification morphism to a parallel design? This section formally addresses these issues.

7.3.1 Translating Layered Designs to Parallel Designs. Communication between a process and its subordinate in a layered design is concealed from the environment. In other words, a layered design can be viewed as a parallel design in which inter-process communication has been concealed, and in fact, the composition operator // used in the definition of layered architecture theory is itself defined in terms of the parallel composition operator ||. As described in Theorem VI.4, concealment induces a specification morphism from the target specification to the source specification. This implies that specification morphisms from layered designs to parallel designs exist. This fact is expressed in the following theorem.

Theorem VII.6 *Any well-formed layered design can be translated under specification morphism to a parallel design.*

Proof. *Both layered processes and parallel processes are defined inductively. Therefore structural induction is used to prove the claim.*

1. *(Basis). Based on Definition VI.11, any well-formed expression in CSP_A of sort* process *defines a layered process. However, by Definition VI.9, such an expression also defines a parallel process. Thus for the basis case, the translation is simply the identity map.*

2. *(Induction). Based on Definition VI.11, if L_1 and L_2 are layered processes, then so is $L_1 /\!/ L_2$. Based on the definition of the operator $/\!/$, $L_1 /\!/ L_2 = (L_1 \| L_2) \setminus \alpha L_1$. By Theorem VI.4, there exists a process specification morphism from $(L_1 \| L_2) \setminus \alpha L_1$ to $L_1 \| L_2$.* ■

For example, the layered design L *sat* $(P /\!/ Q)$ where P *sat* $in?x \xrightarrow{CSP} out!p(x)$, and Q *sat* $left?x \xrightarrow{CSP} (in!q(x) \xrightarrow{CSP} (out?y \xrightarrow{CSP} (right!r(y) \xrightarrow{CSP} SKIP)))$ has the set of traces $\{\langle\rangle$, $\langle left.x\rangle$, $\langle left.x,\ right.r(p(q(x)))\rangle\}$. Translating to the parallel design LP *sat* $P \| Q$ yields a process which has the set of traces $\{\langle\rangle$, $\langle left.x\rangle$, $\langle left.x,\ in.q(x)\rangle$, $\langle left.x,\ in.q(x),\ out.p(q(x))\rangle$, $\langle left.x,\ in.q(x),\ out.p(q(x)),\ right.r(p(q(x)))\rangle$,$\}$, which when restricted to the alphabet of L, where $\alpha L = \alpha Q - \alpha P$, or $\{left,\ right\}$, yields $\{\langle\rangle$, $\langle left.x\rangle$, $\langle left.x,\ right.r(p(q(x)))\rangle\}$. Thus $traces(L) \subseteq traces(LP)\restriction \alpha L$, which implies that the translation from L to LP is a specification morphism.

7.3.2 Translating Parallel Designs to Layered Designs. The preceding subsection illustrated how a layered design is translated via a specification morphism to a parallel design, and it provided some insight to the problem of translating parallel designs to layered designs. As stated in the following theorem, non-trivial parallel designs cannot be translated under specification morphism to layered designs.

Theorem VII.7 *Given two arbitrary well-formed expressions P_1 and P_2 in CSP_A where $\alpha P_1 \cap \alpha P_2 \neq \{\}$ and $\alpha P_1 \neq \alpha P_2$ such that* traces $(P_1 \| P_2) \neq \{\langle\rangle\}$, $(P_1 \| P_2) \not\models (P_2 /\!/ P_1)$.

Proof. *Communication of all forms between a process and its subordinate is hidden from the outside environment. Thus if processes P_1 and P_2 share a common CSP channel c in $P_2 /\!/ P_1$, communication over c will not appear in any trace of $P_2 /\!/ P_1$, but will appear in*

a trace of $P_1 \| P_2$. *This implies* $\text{traces}(P_1 \| P_2) \not\subseteq \text{traces}(P_2 /\!/ P_1)$, *which yields* $(P_1 \| P_2) \not\models (P_2 /\!/ P_1)$. *Similarly, if* P_1 *and* P_2 *share a non-communication event* e, *then* e *may appear in a trace of* $P_1 \| P_2$ *but will not appear in any trace of* $P_2 /\!/ P_1$, *which implies* $(P_1 \| P_2) \not\models (P_2 /\!/ P_1)$. ■

The above theorem is not very strong. It simply states that a direct mapping of non-trivial subprocesses of a nontrivial parallel process to individual layers of a layered design cannot be accomplished via specification morphism. The stronger theorem provided below states that no non-trivial parallel design can be translated under specification morphism to a layered design of more than one layer.

Theorem VII.8 *Given a well-formed parallel design* P *sat* $P_1 \parallel P_2 \parallel \cdots \parallel P_n$ *consisting of* n *nontrivial subprocesses such that* $\text{traces(P)} \neq \{\langle\rangle\}$, P *cannot be mapped under specification morphism to a layered design* L *consisting of two or more non-trivial layers.*

Proof. *Suppose a specification morphism* σ *from* P *to a layered design* L *consisting of* $m > 1$ *nontrivial layers could be defined. Because only those events of the outer layer* L_{m-1} *of* L *not shared with any subordinate layer* $L_b, b < m-1$ *appear in any trace of* L, *each subprocess of* P *contributing to any trace of* P *must be mapped to the outermost layer* L_{m-1} *of* L. *This implies that for any event* e *in any trace* t *in* traces(P), *if* $e \in \alpha P_i$ *for any subprocess* P_i *of* P, *then* P_i *must be mapped to the outer layer of* L.

If every P_i *shared a common alphabet, i.e.,* $\alpha P_i = \alpha P$ *for every* P_i *of* P, *then each* P_i *must be mapped under* σ *to the outer layer of* L, *otherwise every layer of* L *would share a common alphabet, resulting in* $\text{traces(L)} = \{\langle\rangle\}$. *However, mapping every* P_i *in* P *to the outer layer of* L *results in a layered design of less than two layers. Therefore,* P *cannot be translated to a layered design of more than one layer if* $\forall(P_i)(P_i \in P \Rightarrow \alpha P_i = \alpha P)$.

If $\alpha P_a \neq \alpha P$ *for any* P_a *in* P, *then* $\text{traces(P)} = \{t \mid (t \restriction \alpha P_1) \in \text{traces}(P_1) \wedge (t \restriction \alpha P_2) \in \text{traces}(P_2) \wedge \ldots \wedge (t \restriction \alpha P_n) \in \text{traces}(P_n) \wedge t \in \{\alpha P_1 \cup \alpha P_2 \cup \cdots \cup \alpha P_n\}^*\}$, *where* $\{\alpha Q\}^*$ *denotes the Kleene closure of* αQ. *(26) Distributing* $\cup$ *over* $\wedge$ *produces* $\text{traces(P)} = \{t \mid (t \restriction \alpha P_1) \in \textit{traces}(P_1) \wedge t \in \{\alpha P_1 \cup \alpha P_2 \cup \cdots \cup \alpha P_n\}^*\} \cup \{t \mid (t \restriction \alpha P_2) \in \textit{traces}(P_2) \wedge t \in \{\alpha P_1 \cup \alpha P_2 \cup \cdots \cup \alpha P_n\}^*\} \cup \cdots \cup \{t \mid (t \restriction \alpha P_n) \in \textit{traces}(P_n) \wedge t \in \{\alpha P_1 \cup \alpha P_2 \cup \cdots \cup \alpha P_n\}^*\}$. *Because no* P_i *is a trivial process, no set* $\text{traces}(P_i)$ *for any* P_i *in* P *is*

empty. Suppose for some P_j in P the set $\{t \mid (t \restriction \alpha P_j) \in traces(P_j) \wedge t \in \{\alpha P_1 \cup \alpha P_2 \cup \cdots \cup \alpha P_n\}^*\}$ was empty. Then P_j in P contributes to no trace of P, which again implies that P_j is a trivial process. Because no subprocess of P is trivial, this implies that each set $\{t \mid (t \restriction \alpha P_i) \in traces(P_i) \wedge t \in \{\alpha P_1 \cup \alpha P_2 \cup \cdots \cup \alpha P_n\}^*\}$ for P_i in P is nonempty. This implies that each P_i in P contributes to the traces of P, which in turn implies that each P_i of P must be mapped to the outer layer of L. ■

If layered designs could be translated under specification morphism into parallel designs and parallel designs could be translated under specification morphism to layered designs, then parallel and layered architecture theories would be isomorphic.

Theorem VII.8 implies that arbitrary well-formed parallel designs consisting of non-trivial subprocesses cannot be translated under specification morphism to layered designs of more than one layer. Furthermore, Theorem VII.8 implies that specification construction operations cannot be defined for translating designs of parallel architectures to designs of layered architectures because such translations do not in general preserve trace satisfaction. Although parallel designs cannot be translated under specification morphism to layered designs, this does not preclude the possibility of translation under external compatibility.

Claim VII.1 *Any well-formed design P of a parallel architecture theory can be translated to a design L of a layered architecture theory such that $P \sqsubseteq L$.*

The above claim can be proven using language theory, and is conceptually the same as saying that any parallel design can also be implemented as a sequential design sacrificing only performance. A proof of the above claim is out of scope of this research effort.

7.3.3 Summary of the Relationship Between Parallel and Layered Designs. This section has highlighted the relationship between designs of layered and parallel architecture theories. The result of this comparison is that layered designs can be translated under process specification morphism to parallel designs, but the converse is not true; parallel designs cannot in general be translated under specification morphism to layered designs. When taken into conjunction with Theorem VII.5, this implies that parallel architecture

theory is not isomorphic to layered architecture theory. This is not a surprising result, for subordination is defined to be a restricted form of parallel composition.

7.4 Relationship Between Parallel and Pipeline Designs

This section explores some of the relationships between parallel and pipelined designs.

7.4.1 Translating Parallel Designs to Pipeline Designs. Pipeline architecture theory is shown in Figure 6.21 to be a specialization of parallel architecture theory. However, this specialization — like the specialization from parallel architecture to layered architecture — cannot be defined via specification morphism. This fact is formalized in the following theorem.

Theorem VII.9 *Given two arbitrary well-formed expressions P_1 and P_2 in CSP_A such that $P_1 \gg P_2$ is well-formed and* $\text{traces}(P_1 \| P_2) \neq \{\langle\rangle\}$, *then* $(P_1 \| P_2) \not\models (P_1 \gg P_2)$.

Proof. *Communication between successive stages of a pipeline process is hidden from the outside world. Processes P_1 and P_2 share a common CSP channel in $P_1 \gg P_2$ such that communication over that channel does not appear in any trace of $P_1 \gg P_2$. However, communication over the common channel between P_1 and P_2 will appear in the traces of $P_1 \| P_2$. This implies* $\text{traces}(P_1 \| P_2) \not\subseteq \text{traces}(P_1 \gg P_2) \restriction \alpha(P_1 \| P_2)$, *which yields* $(P_1 \| P_2) \not\models (P_1 \gg P_2)$. ■

Theorem VII.9 addresses only whether the processes of a parallel design can be mapped under a bijective specification morphism to processes in a pipeline design. The theorem can be generalized to address the issue of whether other forms of specification morphisms such as injections can be defined which map the subprocesses of a parallel design to subprocesses of a pipeline design. The following theorem is one such generalization.

Theorem VII.10 *Given a well-formed parallel design P such that P sat $P_1 \| P_2 \| \cdots \| P_n$, if*

1. *For every subprocess P_i, $i = 1..n$, of P* $\text{traces}(P_i) \neq \{\langle\rangle\}$, *and*

2. $\alpha P_h \cap \alpha P_j \neq \{\}$ *for at least two subprocesses* P_h *and* P_j, $h \neq j$, *of* P *such that* $\exists(t)(t \in traces(P) \Rightarrow \exists(e)(e \in \alpha P_h \cap \alpha P_j \Rightarrow e \text{ in } t))$,

then P *cannot be translated under specification morphism to a pipeline process* G *of two or more non-trivial stages.*

Proof. *Suppose a specification morphism* σ *from* P *to an* m*-stage pipeline could be defined, where* $m \geq 2$. *Then* $\alpha G = \alpha left(G_1) \cup \alpha right(G_m)$, *where* left *is the input channel of* G_1 *and* right *is the output channel of* G_m. *Thus, if any trace* $t \in$ traces(P) *contains anything other than communication events, then* traces(P) $\not\subseteq$ traces(G)$|_\sigma$, *because* traces(G)$|_\sigma$ *contains only communication events.*

Suppose then that every trace of P *contained only communication events. Denote then by* P_h *and* P_j, *where* $h \neq j$, *two subprocesses of* P *such that* $\alpha P_h \cap \alpha P_j \neq \{\}$ *and at least one common event between* P_h *and* P_j *appears in some trace* t *of* P. *Such processes* P_h *and* P_j *are guaranteed to exist based on the assumptions of the theorem. Because all events in any trace of* P *are communication events, this implies that* P_h *and* P_j *communicate. However, no communication between any two stages of* G *can appear in any trace of* G. *This implies that* t $\in$ traces(P), *but* t $\notin$ traces(G)$|_\sigma$, *which implies that* σ *is not a specification morphism.* ■

The above theorem states that specification morphism cannot generally be used to translate parallel designs to pipeline designs. However, specification morphism can be used to translate parallel designs to pipeline designs in the degenerate case when the set *traces*$(P_1 \| P_2)$ consists of only the empty trace.

As formalized in the following theorem, there are conditions under which parallel designs can be translated into pipeline designs such that the pipeline design is externally compatible with the parallel design.

Theorem VII.11 *Given a parallel design* P *such that* $P = P_1 \| P_2 \| \cdots \| P_n$, P *can be translated into a pipeline design* L *such that* $P \sqsubseteq L$ *if*

1. *No subprocess of* P *shares an event with any other subprocess of* P *except for possibly the successful termination event* $\surd$.

2. *Each sub-process P_i, $i = 1..n$, of P contains exactly two CSP channels, one for input and one for output; and*

3. *A bijection m : Process $\rightarrow$ Nat mapping each sub-process of P to an element of the set $\{1, 2, \ldots, n\}$ can be defined such that $m(P_i) = m(P_j) - 1$ if and only if P_i and P_j share a single CSP channel such that the channel is used exclusively for output in P_i and exclusively for input in P_j.*

Proof. *The first of the above two conditions restricts the subprocesses of P to be stages. The second condition requires that a total order over the stages of P exist. Because $R \| S = S \| R$, P can be defined by the expression $P'_1 \| P'_2 \| \cdots \| P'_n$ where for $i = 1..n$, $P'_i = P_j$ such that $m(P_j) = i$. Communication over the internal channels of P can be concealed as follows. Denote by C the set of channels used for interprocess communication in P. That is, $C = \{c \mid \exists (i,j)(i,j \in \{1,2,\ldots,n\} \Rightarrow c \in chan(P_i) \wedge c \in chan(P_j) \wedge m(P_i) = m(P_j) - 1)\}$ where P_i and P_j are subprocesses of P. Then $P \setminus C$ defines a process in which all communication over the internal channels of P are concealed from the environment. Then the traces of $P \setminus C$ consists of sequences of input events over the input channel of P'_1 and output events over the output channel of P'_n such that for any trace t in* traces$(P \setminus C)$, $\{0 \leq size(t \downarrow in) - size(t \downarrow out) \leq n\}$ *where* in *is the input channel of P'_1 and* out *is the output channel of P'_n. In this case, $t \in traces(P \setminus C) \Rightarrow t \in traces(P'_1 \gg P'_2 \gg \cdots \gg P'_n)$, which implies $(P_1 \| P_2 \| \cdots \| P_n) \sqsubseteq (P'_1 \gg P'_2 \gg \cdots \gg P'_n)$.* ■

The above theorem does not imply that a parallel design will, when translated to a pipeline design, exhibit the same behavior for the same sequence of enabled events. In fact, a pipeline design and a parallel design whose internal communications are concealed may engage in different sequences of communication events. For example, consider the three stage pipeline $P_1 \gg P_2 \gg P_3$ and the three process parallel design $P_1 \| P_2 \| P_3$ operating in an environment in which inputs are always ready for P_1. Because the semantics of the connective $\gg$ places an emphasis on external communication, no trace of $P_1 \gg P_2 \gg P_3$ will contain an output event unless the output event has been preceded by two input events. However, the semantics of the connective $\|$ places no such emphasis on communication. Thus an output event may appear in the traces of $P_1 \| P_2 \| P_3$ concealed over internal communication following a single input event, which implies that the two designs yield differing

sequences of communicated values when operating in the same environment. However, Definition VII.1 makes no reference to the operating environment. If Definition VII.1 was strengthened to include references to the external environment, then the above theorem would not hold.

As the above theorems illustrate, only trivial parallel designs can be translated under specification morphism to pipeline designs, and only a very restricted class of parallel design can be translated under external compatibility to a pipeline design.

7.4.2 Translating Pipeline Designs to Parallel Designs. Although parallel designs cannot generally be translated under specification morphism to pipeline designs, as pointed out in the following theorem, pipeline designs can be translated under specification morphism into parallel designs.

Theorem VII.12 *Any well-formed pipeline design can be translated under specification morphism to a parallel design.*

Proof. *Both pipeline designs and parallel designs are defined inductively. Therefore structural induction is used to prove the claim.*

1. *(Basis.) A single stage pipeline design P, by definition, contains exactly two channels, one for input and one for output. If P is not chained to any other process, then by Definition VI.9, P is also a parallel process. Thus for the basis case, the translation is the identity specification morphism.*
2. *(Induction.) Consider the pipeline design $P_1 \gg P_2$. Both P_1 and P_2 have exactly two channels, one for input and one for output, with communication between P_1 and P_2 not only concealed from the environment but of lower priority than external communication. Denoting the channels of both P_1 and P_2 by* left *and* right*, with* right *of P_1 and* left *of P_2 defining a channel c between P_1 and P_2, $P_1 \gg P_2$ has the following laws:(52)*

   (a) $(\mathrm{c!v} \xrightarrow{CSP} P_1) \gg (\mathrm{c?v} \xrightarrow{CSP} P_2) = P_1 \gg P_2(v)$*, where $P_2(v)$ is denotes the process P_2 after accepting the value v. (Internal communication is concealed.)*

(b) $(\text{c!v} \xrightarrow{CSP} P_1) \gg (\text{right!w} \xrightarrow{CSP} P_2) = \text{right!w} \xrightarrow{CSP} ((\text{c!v} \xrightarrow{CSP} P_1) \gg P_2)$ *(External communication takes precedence.)*

(c) $(\text{left?v} \xrightarrow{CSP} P_1(v)) \gg (\text{c?w} \xrightarrow{CSP} P_2(w)) = \text{left?v} \xrightarrow{CSP} (P_1(v) \gg (\text{c?w} \xrightarrow{CSP} P_2(w)))$ *(External communication takes precedence.)*

Concealment of internal communication can be accomplished through application of the concealment operator, and the possibility of engaging in external communication over internal communication is offered in parallel designs:

1. *With respect to communication over the internal channel c,* $\text{traces}(P_1 \gg P_2)\restriction\{\text{c}\} = \{\langle\rangle\}$, *but* $\text{traces}((P_1 \| P_2) \setminus \{c\})\restriction \{c\} = \{\langle\rangle\}$. *Thus concerning communication over the internal channel c, we have* $\text{traces}(P_1 \gg P_2)\restriction \{c\} \subseteq \text{traces}((P_1 \| P_2) \setminus \{c\})\restriction \{c\}$.
2. *Concerning external communication,* $((\textit{left?v} \xrightarrow{CSP} P_1(v)) \| (\textit{c?w} \xrightarrow{CSP} P_2(w)))$ *is equivalent to* $(\textit{left?v} \xrightarrow{CSP} (P_1(v) \| \textit{c?w} \xrightarrow{CSP} P_2(w))) \mid (\textit{c?w} \xrightarrow{CSP} (\textit{left?v} \xrightarrow{CSP} P_1(v) \parallel P_2(w)))$, *which clearly offers the possibility to engage first in external communication before engaging in internal communication. Similarly,* $((\textit{c!v} \xrightarrow{CSP} P_1) \| (\textit{right!w} \xrightarrow{CSP} P_2))$ *is equivalent to* $(\textit{c!v} \xrightarrow{CSP} (P_1 \| (\textit{right!w} \xrightarrow{CSP} P_2))) \mid (\textit{right!w} \xrightarrow{CSP} ((\textit{c!v} \xrightarrow{CSP} P_1) \| P_2))$, *which clearly offers the possibility to engage first in external communication before engaging in internal communication. Clearly then,* $\text{traces}(P_1 \gg P_2) \subseteq \text{traces}((P_1 \| P_2) \setminus \{c\})$.

Noting that $\text{traces}((P_1 \gg P_2) \setminus \{c\}) = \text{traces}(P_1 \gg P_2)$ *where c is the channel between* P_1 *and* P_2, *items 1 and 2 above imply that* $\text{traces}(P_1 \gg P_2) \subseteq \text{traces}((P_1 \| P_2) \setminus \{c\})$, *which implies that* $P_1 \gg P_2$ *can be translated under specification morphism to* $(P_1 \| P_2) \setminus \{c\}$. *Theorem VI.4 states that a specification morphism from* $(P_1 \| P_2) \setminus \{c\}$ *to* $P_1 \| P_2$ *exists. Since specification morphism compose to form specification morphisms, a specification morphism exists from* $P_1 \gg P_2$ *to* $P_1 \| P_2$. ■

Because well-formed pipeline designs can always be mapped under specification morphism to parallel designs, TheoremVII.4 states that parallel designs are externally compatible with pipeline designs.

7.4.3 Summary of the Relationship Between Parallel and Pipeline Designs. The result of this comparison is that pipeline designs can be translated under specification

morphism to parallel designs, but not conversely. When taken into conjunction with Theorem VII.5, this implies that parallel architecture theory is not isomorphic to pipeline architecture theory.

7.5 Relationship Between Layered and Pipeline Designs

This section addresses the issue of translating layered designs to pipeline designs and vice-versa.

7.5.1 Translating Layered Designs to Pipelined Designs. The process composition operators $\gg$ used in pipeline architecture theory and // used in layered architecture theory both conceal communication. However, the locus of concealment differs between the two operators.

The operator $\gg$ conceals all communication from the external environment except for communication over the input channel of the first process of the chain and communication over the output channel of the last process in the chain. Thus, the set of traces of an n-stage pipeline P where P sat $P_1 \gg P_2 \gg \cdots \gg P_n$ is the set defined by $\{t \mid (0 \leq \text{size}(t \downarrow \{\text{in}\}) - \text{size}(t \downarrow \{\text{out}\}) \leq n) \wedge t \in \{\text{in}, \text{out}\}^*\}$ where *in* is the input channel of P_1 and *out* is the output channel of P_n. In addition, communication in a pipeline design is acyclic; a stage in a pipeline can communicate data to only the next stage in the pipeline.

The operator // conceals from the external environment all events shared between a process and its subordinate process. Thus the set of traces of a layered design L where L sat $((L_0 // L_1) // \cdots L_{n-1}) // L_n$ is the set *traces*$(L_n \setminus \alpha L_{n-1})$, where $\alpha L_{i-1} \subset L_i$, $\forall i \in \{1, 2, \ldots, n\}$. Communication between a process and its subordinate can be cyclic since there are no restrictions concerning the flow of data between a process and its subordinate.

Because only the communication over the input channel of the first stage and communication over the output channel of the last stage of P appear in any trace of P, the outer layer $L_{S,n}$ of a layered design L_S must map to both the first stage of P if L_S accepts input and must map to the last stage of P if L_S generates output. These restrictions greatly reduce the class of layered designs that can be translated under specification morphism

to a pipeline design. In fact, as the following theorem states, non-trivial layered designs cannot be translated under specification morphism to multi-stage pipeline designs.

Theorem VII.13 *An arbitrary layered design L containing at least one input channel c_{in} and at least one output channel c_{out} such that c_{in} and c_{out} appear in a trace of L cannot be translated under specification morphism to a pipeline design P of two or greater stages.*

Proof. *Suppose a specification morphism σ from a layered design L of $n \geq 1$ layers to a pipeline design P of $m > 1$ stages could be defined. Then* $\text{traces(L)} \subseteq \text{traces(P)}|_\sigma$ *only if every external channel of $L_{S,n}$ can be mapped to either the input channel of P_1 or the output channel of P_m. This is the only means through which communication over these channels will appear in the traces of the pipeline design. Because all external channels of a layered design reside in the outermost layer $L_{S,n-1}$, this results in $L_{S,n-1}$ being mapped to both P_1 and P_m under σ. But* $\text{traces}(L_{S,n-1}) \not\subseteq \text{traces}(P_n)|_\sigma$ *under this map because* $\text{traces}(P_n)|_\sigma$ *includes only communication over the output channel(s) of L, while* $\text{traces}(L_{S,n-1})$ *may include communication over both input and output channels. Thus σ does not preserve trace satisfaction and is therefore not a specification morphism.* ■

Determination of whether an arbitrary well-formed layered design can be translated to an externally compatible multi-stage pipeline design requires analysis of the functional model of the layered design, and is left for future research.

7.5.2 Translating Pipeline Designs to Layered Designs. Communication within a pipeline design also poses some problems for translating pipeline designs to layered designs. Only the communication over the input channel of the first stage and the output channel of the last stage appear in any trace of a pipeline design. In contrast, only the external communication of the outermost layer of a layered design appears in its traces. If a specification morphism from a pipelined design to a layered design could be defined, then the first and last stages of the pipeline must be mapped to the outer layer of the corresponding layered design. This implies, for example, that a two stage pipeline design collapses to a single layer layered design. But what about larger pipelines? Can a specification morphism from an n-stage pipeline, where $n \geq 2$, to a layered design be defined?

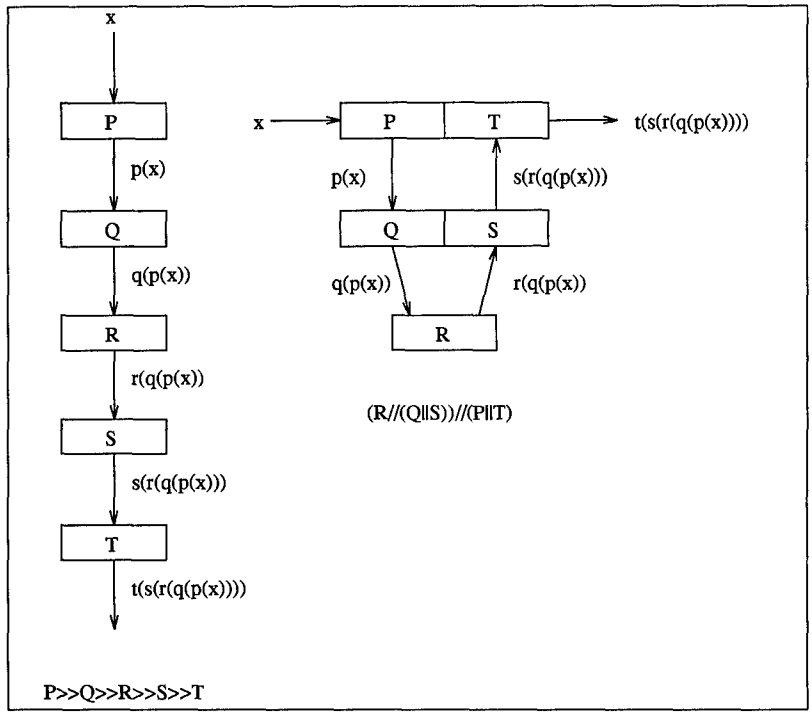


Figure 7.1 Translating a Pipeline Design to a Layered Design

As shown in Figure 7.1 and stated in the following theorem, the answer to this question is yes.

Theorem VII.14 *Given an arbitrary well-formed pipeline design P, such that no two stages share any common non-communication events except for the successful termination event $\surd$, P can be translated under specification morphism to a layered design.*

Proof. *Denote by P sat $P_1 \gg P_2 \gg \cdots \gg P_n$, $n \geq 1$, an arbitrary, well-formed pipeline design, with* in *denoting the input channel of P_1 and* out *denoting the output channel of P_n. If* traces(P) *equals $\{\langle\rangle\}$, then $P \cong STOP_{\alpha P}$. Which by Theorem V.4 implies that $P \models L$ for any well-formed layered design L such that $\alpha L = \alpha P$.*

For the general case when traces(P) $\neq \{\langle\rangle\}$, *a specification morphism σ from P to an m-layer, layered design L_S can be constructed as follows.*

P	*sat*	$P_1 \gg P_2 \gg \cdots \gg P_n$
P	$\mapsto$	L_S
L_S	*sat*	$((L_{S,0} /\!\!/ L_{S,1}) /\!\!/ \cdots) /\!\!/ L_{S,m}$, *where* $m = ((n+1)\ div\ 2) - 1$
P_1	$\mapsto$	$L_{S,m}$

$$
\begin{array}{lll}
P_n & \mapsto & L_{S,m} \\
L_{S,m} & sat & (\Xi(P_1)\|\Xi(P_n))
\end{array}
$$

$$
\begin{array}{lll}
P_2 & \mapsto & L_{S,m-1} \\
P_{n-1} & \mapsto & L_{S,m-1} \\
L_{S,m-1} & sat & (\Xi(P_2)\|\Xi(P_{n-1}))
\end{array}
$$

n *even* $\Rightarrow$

$$
\begin{array}{lll}
P_{n\ div\ 2} & \mapsto & L_{S,0} \\
P_{(n\ div\ 2)+1} & \mapsto & L_{S,0} \\
L_{S,0} & sat & (\Xi(P_{n\ div\ 2})\|\Xi(P_{(n\ div\ 2)+1}))
\end{array}
$$

n *odd* $\Rightarrow$

$$
\begin{array}{lll}
P_{(n\ div\ 2)+1} & \mapsto & L_{S,0} \\
L_{S,0} & sat & \Xi(P_{(n\ div\ 2)+1})
\end{array}
$$

Thus, for any stage P_i *in* P, *if* σ_{P_i} *maps* P_i *to its corresponding layer* $L_{S,b}$ *as defined above,* $\text{traces}(P_i) \subseteq \text{traces}(L_{s,b})|_{\sigma_{P_i}}$. *What remains to be shown is that* traces(P) $\subseteq$ traces(L).

Based on the above mapping, $\text{traces}(L_{S,m}) = \text{traces}((P_1\|P_n) \setminus \{c_{1,2}, c_{n-1,n}\})$. *Because* P_1 *and* P_2 *share no common non-communication events,* $(P_1\|P_n) \setminus \{c_{1,2}, c_{n-1,n}\}$ *equals* $(P_1 \setminus \{c_{1,2}, c_{n-1,n}\}\|P_2 \setminus \{c_{1,2}, c_{n-1,n}\})$, *which simplifies to* $(P_1 \setminus \{c_{1,2}\}\|P_2 \setminus \{c_{n-1,n}\})$. *This implies that* traces($L_{S,m}$) *consists of sequences of communication events over the channels* in *and* out. *What remains to be shown is that for any trace* $t \in \text{traces}(L_S)$, *the number of input events over channel* in *minus the number of output events over channel* out *satisfies the condition* $\{t \mid (0 \leq size(t \downarrow \{in\}) - size(t \downarrow \{out\}) \leq n) \wedge t \in \{in, out\}^*\}$.

L_S *can accept up to* n *inputs before generating an output as follows. Based on the definition of a stage, each stage* P_i *of* P *satisfies* $c_{i-1,i}?y \xrightarrow{CSP} (ci, i+1!p_i(y) \xrightarrow{CSP} P_i)$. *Then each layer* $L_{S,c}$ *contains up to two stages operating in parallel, which implies that each layer can accept up to two inputs, one for each stage, before generating an output. The number of inputs a given stage can accept is dependent on whether* n *is even or odd.*

1. *If* n *is even, there will be* $((n+1)\ div\ 2) - 1 + 1 = (n\ div\ 2)$ *layers, each containing two stages and capable of accepting* $2 \times (n\ div\ 2) = n$ *inputs before generating an output.*

2. *If n is odd, there will be $((n+1)\ div\ 2)$ layers, of which $((n+1)\ div\ 2)-1$ layers contain two stages. layer $L_{S,0}$ can accept only one input before generating an output, so L_S can accept up to $2\times((n+1)div\ 2\ -\ 1)+1$ inputs before generating an output. Because n is odd, this simplifies to $2\times(n\ div\ 2)-1=(n+1)-1$ which equals n.*

Thus for any trace $t \in$ traces(L), $(0 \leq size(t \downarrow \{in\}) \leq n) \wedge t \in \{in, out\}^*$.

Conversely, L_S can generate up to n outputs in succession as follows. Suppose L_S has accepted n inputs as described above. Then each layer of L except possibly layer $L_{S,0}$ has two outputs ready for communication. Due to the definition of a stage, communication over channel out *of P_n of layer $L_{S,m}$ is the only communication that can occur. Once communication over channel* out *has occurred, then P_n of $L_{S,m}$ can accept an output from P_{n-1} of $L_{S,m-1}$, which can then accept an output from P_{n-2}, and so on. This process can continue until the n values contained in L have been processed by the stages contained in L and output over channel* out. *In addition, based on the definition of a stage, each stage contained in any layer of L can generate only one output per input. Thus for any trace* $t \in$ traces(L), $(0 \leq size(t \downarrow \{out\}) size(t \downarrow \{in\})) \wedge t \in \{in, out\}^*$, *which implies that* traces(L) = traces(P). ■

Figure 7.1 depicts a translation from a pipeline design to a trace equivalent layered design. As shown in the figure, the layered design has just over half the number of layers as compared to the number of stages of the pipeline design. The functional model of the pipeline design is intact in the layered design; adjacent stages of the pipeline design are mapped to adjacent layers in the layered design. Also shown in the figure is a value x as it propagates through both the pipeline design and the layered design.

7.5.3 Summary of the Relationship Between Layered and Pipeline Designs. This subsection has highlighted some of the relationships between pipeline and layered designs, and as such compliments the discussion relating pipeline designs and parallel designs contained in Subsection 7.4 and compliments the relationship between layered and parallel designs presented in Subsection 7.3. The result of this comparison is that pipeline designs can be translated under specification morphism to layered designs, but not conversely.

When taken into conjunction with Theorem VII.5, this implies that layered architecture theory is not isomorphic to pipeline architecture theory.

7.6 Other Relationships

The previous subsections have formally described relationships between parallel, pipeline, and layered designs. This subsection takes an informal look at relationships between designs of some of the other architecture theories, beginning with the relationship between piped-batch sequential designs and pipeline designs.

Based on the definition of piped-batch sequential architecture, any well-formed piped-batch-sequential design is also a pipe-filter design. Similarly, any well-formed pipe-filter design is also a parallel design. Because specification morphisms compose to form specification morphisms, for any piped-batch sequential design D, a specification morphism from D to a parallel design P can be defined. This means that piped-batch-sequential designs can be refined through specification morphism to define pipe-filter designs or parallel designs. However, Theorem VII.10 states that non-trivial parallel designs cannot be translated under specification morphism to pipeline designs of more than one stage. This means that piped-batch sequential designs cannot be translated under specification morphism to pipeline designs of more than one stage.

Theorem VII.11 identified three conditions under which a parallel design P can be translated under external compatibility to a pipeline design PL. Because piped-batch sequential designs can be translated under specification morphism to parallel designs, if it can be shown that if a piped-batch sequential design PBS satisfies the three conditions of Theorem VII.11, then PBS can be translated under external compatibility to a pipeline design. Fortunately, two of the three conditions of the theorem are satisfied by the definition of piped-batch sequential designs. The remaining condition, that each sub-process P_i, $i = 1..n$, of P contains exactly two CSP channels, one for input and one for output, is also readily verifiable.

Translating a pipeline design PL to a piped-batch sequential design PBS can be accomplished by translating PL to a parallel design P under a specification morphism

$\sigma_{PL,P} : PL \rightarrow P$, and then translating P to a pipe-filter design PF using a morphism $\sigma_{P,PF} : P \rightarrow PF$, and finally, PF can then be translated to PBS using a morphism $\sigma_{PF,PBS} : PF \rightarrow PBS$. However, some of these morphisms might not be specification morphisms. For example, an n-stage pipeline design PL where PL *sat* $P_1 \gg P_2 \gg \cdots \gg P_n$ can engage in up to n input events in succession before engaging in an output event. But if each P_i, $i \in [1..n]$, is mapped to a filter process F_i, $i \in [1..n]$, of a piped-batch sequential design PBS such that $P_i \sqsubseteq F_i$, then PBS can accept only one input from the outside environment before generating an output for consumption by the external environment. In PBS there is only one pipe used to communicate intermediate results between successive filter processes while in PL there are $n-1$ pipes used to communicate intermediate results between concurrent stages. This implies that although it might be possible to translate a pipeline design under external compatibility to a piped-batch sequential design, it might not be possible to translate a pipeline design to a piped-batch sequential design using specification morphisms.

Other translation possibilities exist. For example, it should be possible to translate repository designs under specification morphism to parallel designs simply by making the implicit communication of repository designs explicit. Similarly, it is easy to see that a piped-batch sequential design PBS can be made into a repository design R by replacing the explicit communication encapsulated in the pipe process of PBS with implicit communication via shared data structures. Clearly such a translation would not be a specification morphism. And finally, the definitions of piped-batch sequential architecture and batch sequential architecture indicate that batch-sequential designs are included in piped-batch sequential designs.

Examination of the relationships between the remaining process-based architecture theories of Chapter VI is left for future research.

7.7 Summary

This chapter has formally described relationships between several architecture theories through an exploration of the existence of specification morphisms between them. A summary of these relationships is shown in Figure 7.2, where the arrows in the figure rep-

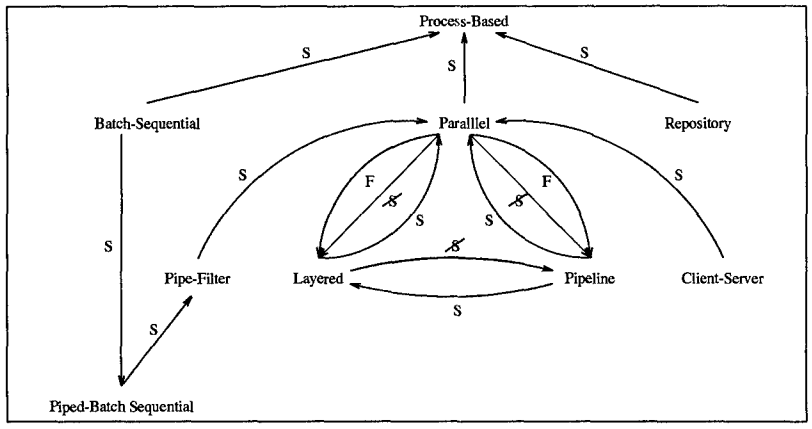


Figure 7.2 Design Translation

resent morphisms. Arrows annotated with an S are specification morphisms, while arrows annotated with an F are morphisms which preserve external compatibility. The strongest condition shown to hold is depicted in the figure. Relationships that have been shown not to hold are annotated with a slash. For example, Theorem VII.13 states that arbitrary well-formed layered designs cannot be translated under specification morphism to pipeline designs of two or more stages, so the arrow from *Layered* to *Pipeline* in Figure 7.2 is annotated with the symbol $\not S$. Note that the lack of an arrow between architectures is not necessarily significant. Also note that the arrows compose. For example, an arrow exists between *Piped-Batch Sequential* and *Parallel*, but it is not shown in the figure.

Relationships between process-based architecture theories were explored through an examination of design translation, where the goal of design translation is to translate a design D of an architecture theory A_D into a design D' of an architecture theory $A_{D'}$. Two definitions of successful translation were introduced. The first, trace satisfaction, was defined in Chapter V. The second, external compatibility, was defined in Section 7.2.

A partial order over architecture theories can be defined based on the existence of specification morphisms between designs of the respective theories. A partial elaboration of this partial order was developed in this chapter. The relationship between four of the architecture theories is shown in Figure 7.3, where the existence of specification morphisms from designs of an architecture theory A to designs of an architecture theory B is is represented by B appearing above A in the diagram.

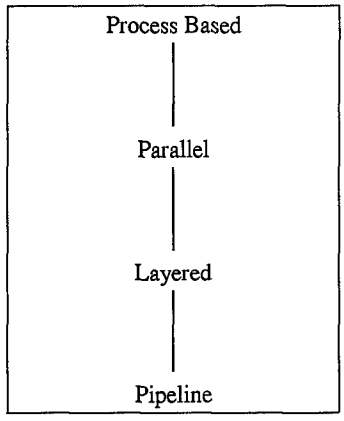


Figure 7.3 Relative Expressive Power of Process Based Architecture Theories

Much analysis work still remains, especially with respect to design translation. For example, it may be possible to define relationships between external compatibility and functional models (in the object-oriented sense) where these relationships could be used to establish necessary and sufficient conditions for demonstrating that a morphism preserves external equivalence.

This chapter concludes the theoretical development portion of this investigation. The feasibility of using process-based architecture theories to define structure is demonstrated in the next chapter where a specification for a segment of an image processing application is developed.

VIII. Feasibility Demonstration

8.1 Introduction

In this chapter, formal process based specifications for a portion of an image processing application are developed using the framework defined in the preceding chapters. Specifically, one of the stages of the image recognition application depicted in Figure 6.16 and partially specified in Figure 6.18 is further refined. (Recall that the image recognition application of Figure 6.16 consists of series of eight segments used to create, process, filter, and classify images.) A batch sequential design for this application was developed in Section 6.3.3. This batch-sequential design is extended in this chapter to a piped-batch sequential design so that data may be communicated between successive filter processes. In addition, a process based specification for one of the sequentially composed processes — *Selection* — is developed.

This chapter is organized as follows:

1. In Section 8.2, the batch sequential design of Figure 6.18 is extended to a piped-batch sequential design.
2. Section 8.3 describes the problem selected for the *Selection* stage, that of extracting the *skeleton* of a two dimensional image.
3. Process-based specifications for *Selection* are developed in Section 8.4; and
4. Section 8.5 contains an qualitative evaluation of the specification methodology.

Process specification names and process symbols will be denoted in typewriter font as in *Selection*. The use of a process symbol in a statement refers to either the process it names or to its process expression depending on the context of the statement. That is, for a process symbol P, the symbol P may be used to refer to the process expression $\Xi(P)$. Models of a process specification P will be denoted P_{mod}, and functional operations will be denoted in italics. The subscript *mod* will be dropped if the meaning is clear based on the context of the expression.

This chapter does not seek to demonstrate the utility of all previously defined architecture theories, nor does it attempt to demonstrate the feasibility of design translation.

The goal of this chapter is to establish the feasibility of using architecture theories to combine simple, process based specifications using specification morphism to define more complex, application level specifications, and to show that reusable, problem independent designs can be defined and used effectively in the construction process.

8.2 Creating a Piped-Batch Sequential Design

8.2.1 Introduction. Figure 6.16 depicts some of the stages for image recognition systems. The figure hints that pipeline architecture theory could be used to define the top level design of such systems in that each stage of Figure 6.16 has exactly one input and one output channel and a total ordering over the stages is apparent. However, piped-batch sequential designs can also be used for image recognition systems wherein successive stages of Figure 6.16 correspond to sequentially composed processes. Communication between these batch processes is achieved through extension of the batch-sequential design to a piped-batch sequential design.

The relative merits of these two approaches, e.g., time versus space, are not germane to this chapter and are therefore not debated herein. Instead, the batch-sequential design of Figure 6.18 is used as the starting point for the development of a piped-batch sequential design.

This section discusses the extension of the specification of Figure 6.18 to form a piped-batch sequential design.

8.2.2 Adding Communication. The specification of Figure 6.18 lacks communication between the successively composed processes. However, Figure 6.16 clearly shows that communication between successive stages is required. Therefore the first task in further development of a specification for this class of problem is to extend the specification of Figure 6.18 with communication. This is accomplished by associating the batch sequential specification *ImageRec* of Figure 6.18 with the process symbol *Filter* of a piped-batch sequential structuring specification. Figure 6.20 depicts how this is done in general. Figure 8.1 is specific to the problem of image recognition.

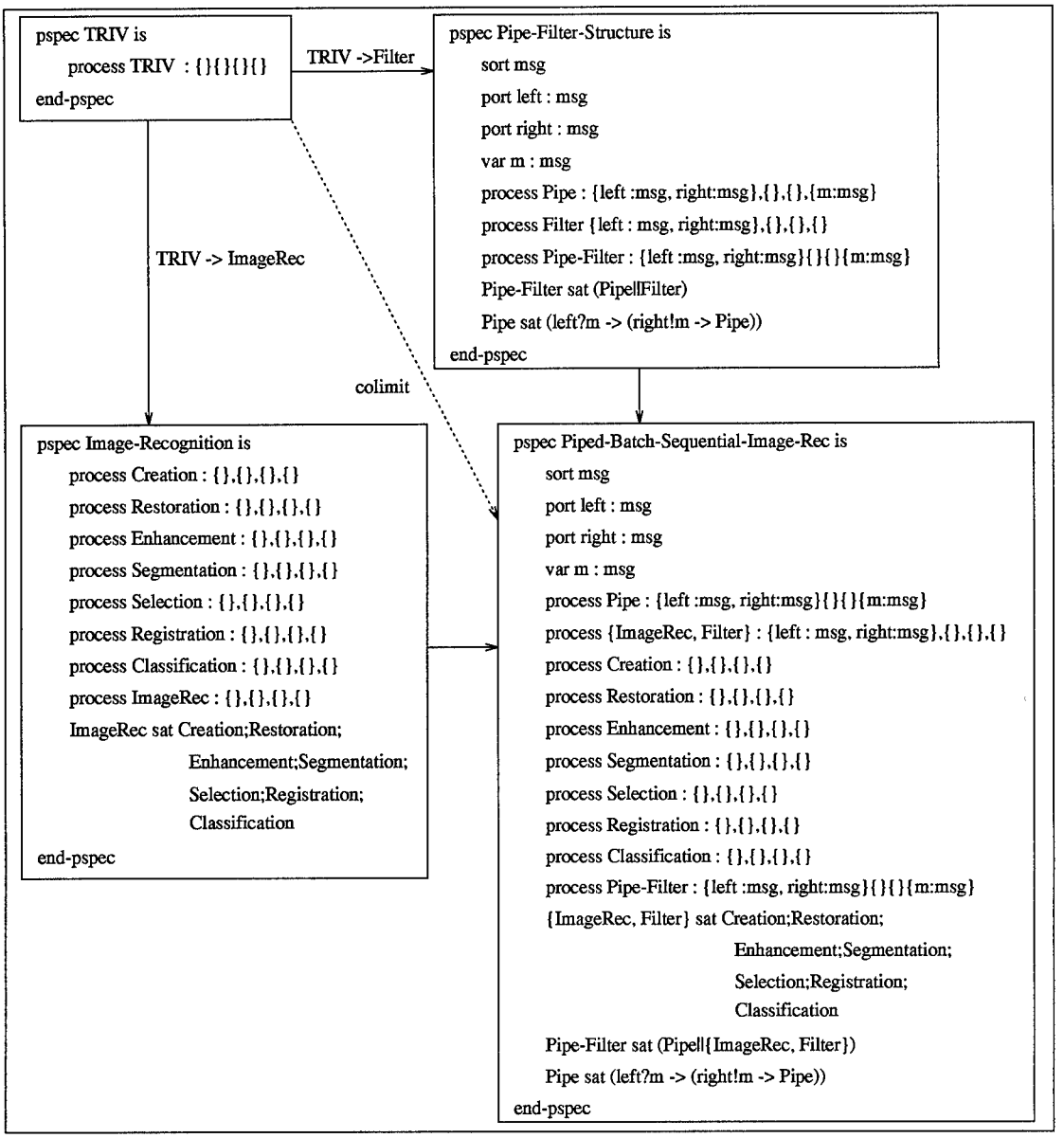


Figure 8.1 Piped-Batch Sequential Image Recognition

The batch sequential specification *Image-Recognition* of Figure 8.1 has been somewhat simplified from that of Figure 6.18. In the colimit object of Figure 8.1, *ImageRec* shares a common set of port symbols with *Pipe*. However, due to an artifact of the definition of colimit of process specifications, namely that alphabet extension of a process does not automatically extend the alphabets of its subprocesses, none of the subprocesses of *ImageRec* contain these port symbols in their alphabets. This implies that even-though *ImageRec* shares common port symbols with the process *Pipe*, no communication between

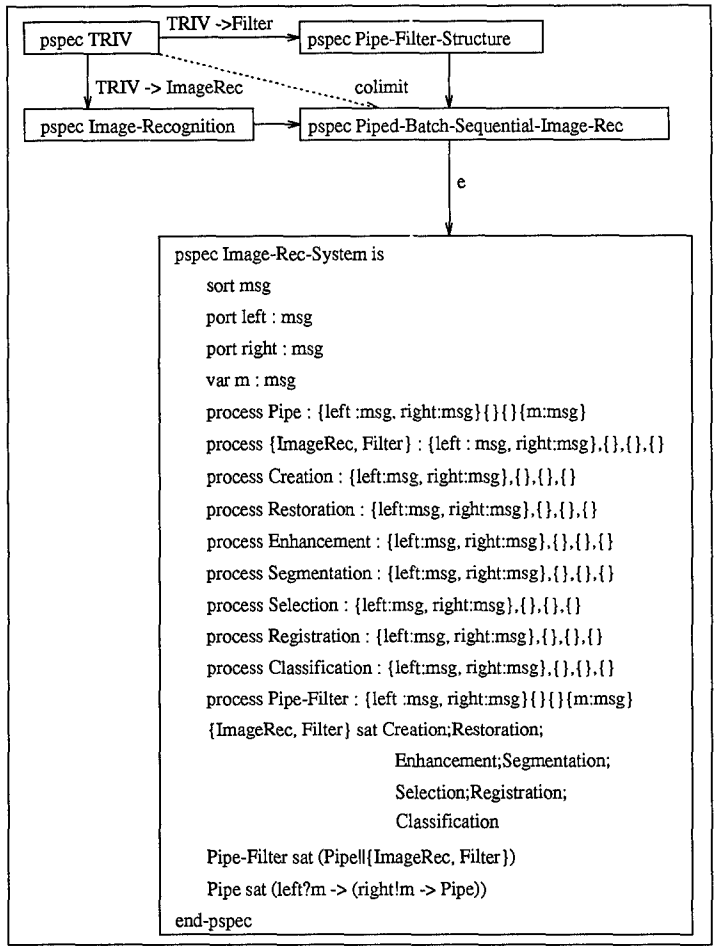


Figure 8.2 Image Recognition Extended with Communication

Pipe and any of the sequentially composed port-less subprocesses defining *ImageRec* can take place. The port symbols of *ImageRec* need to be incorporated into the alphabets of the sequentially composed processes. This simple extension is shown in Figure 8.2, where each subprocess of *ImageRec* has been extended with the port symbols *left : msg* and *right : msg*. Although this results in multiple processes all sharing a common set of port symbols, the semantics of CSP have not been violated in that only one of the sequentially composed processes will be active at any given time. Thus the port symbols *left* and *right* will be shared between the process *Pipe* and exactly one of the filter processes when the process *Pipe-Filter* is executing.

A consequence of the extension depicted in Figure 8.2 is that the functional model of the top level process specification *Pipe-Filter* is now a connected graph, whereas before

the extension is was not. It may be possible to develop or define high level specifications for problem domains such as image processing wherein these high level specifications are partially expressed as constraints over functional models. Definition and expression of such specifications is left for future research.

Now that the sequentially composed processes of the specification *ImageRec* can communicate with each other via a pipe process, results of one image recognition subprocess such as $Image\text{-}Segmentation_{mod}$ can be passed to the next subprocess in turn.

The following section contains a problem description for one of these subprocesses, *Selection*. After the feature selection problem has been defined, a process based specification for it is developed.

8.3 Feature Selection Problem Description

8.3.1 Introduction. The purpose of feature selection is to extract from an image information that can be used to classify it. For example, feature selection can be used for edge detection or intensity gradient computation. Once features have been selected, they can be used to classify the image. The feature selection problem chosen for specification development in this chapter is one of computing the *skeleton* of an image. This problem was selected because its structure is well suited for specification using a mixture of architecture theories and because its well structured nature is suited for design reuse.

A process defined by the specification *Selection* accepts as input a digitized black and white image represented as a two dimensional matrix, and produces another two dimensional matrix. The entries in these matrices represent pixel values, and come from the set $\{1, 0, *\}$ where 1 corresponds to black, 0 corresponds to white, and $*$ corresponds to a lack of sensory data. Although matrices are used as both an input and an output data sort (as is required by the semantics of piped-batch sequential architecture theory), no attempt is made here to define a functional specification for matrices. For the purposes of this chapter, a functional specification for matrices is assumed to exist.

8.3.2 Skeleton Problem Description. As described in (33), the skeleton operation is a thinning operation where a "...figure is replaced by a *thin* representation of

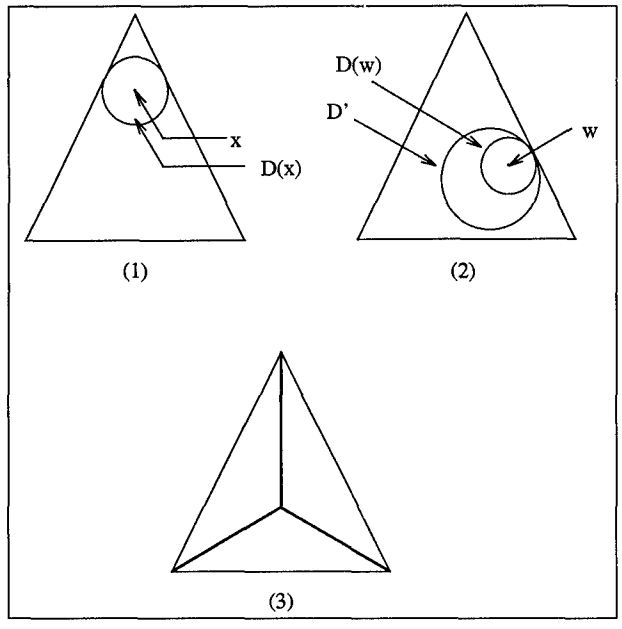


Figure 8.3 Determining skeleton for isosceles triangle (33:352)

itself."(33:351) Informally, the skeleton of a two dimensional image can be defined as follows:

Definition VIII.1 *The* Euclidean skeleton *of a set S is defined in the following manner. For each x in S, let $D(x)$ denote the largest disk centered at x such that $D(x)$ is a subset of S. Then x is in the skeleton of S if there does not exist a disk D_1, not necessarily centered at x, such that D_1 properly contains $D(x)$ and such that D_1 is contained in S. (33:351)* □

The creation of a skeleton for an isosceles triangle is shown in Figure 8.3. The point x shown in the first image is part of the skeleton of the triangle because no other disk D_1 can be defined such that $D(x) \subset D_1$. However, as shown in the second image, the point w is not part of the skeleton because a disk D' can be defined such that $D(w) \subset D'$ and D' is contained within the triangle. The third image in the figure depicts, using heavy lines, the skeleton of the triangle.

The skeleton of several simple pictures is shown in Figure 8.4. As shown in the figure, skeletons are not necessarily unique; several different pictures can all have the same skeleton. Also evident in the figure is the fact that noise can have a significant impact on the resulting skeleton. For example, the disk with a single missing or incorrect pixel value

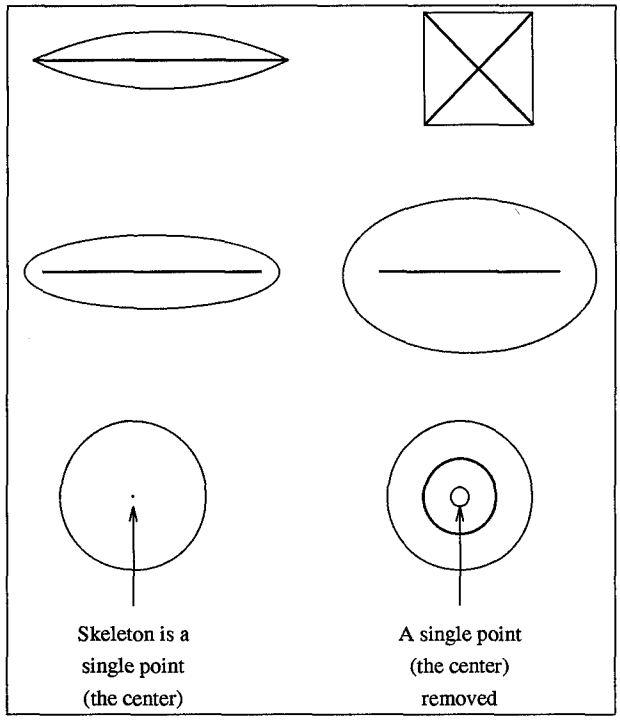


Figure 8.4 Skeleton for various pictures (33:352)

at the origin has a much different skeleton than the noise free disk next to it. However, noise such as this can be mitigated by an image enhancement stage of an image recognition application.

Definition VIII.1 defines skeleton for Euclidean space. However the input to *Selection* is digital, not Euclidean. One of the difficulties of using a digital representation of Euclidean space is that of defining an analogue to Euclidean disks. The solution to this problem proposed in (33) and used here is to use "square disks" as shown in Figure 8.5. In Figure 8.5, the element at the origin is circled. These "square disks" are used in the definition of the digital skeleton operation *SKEL*. The definition of *SKEL* depends on the definition of the operation *TRAN : matrix, index* $\rightarrow$ *matrix*. *TRAN* is defined below.

Definition VIII.2 *(33)* Translate. *If* $f = (a_{pq})_{rt}$ *is a matrix, then for integers* i *and* j, TRAN(f; i, j) $= (a_{pq})_{r+i,t+j}$. □

In other words, *TRAN(f; i, j)* translates f by (i, j). Given the definitions of square disks and *TRAN*, a more formal definition of digital skeleton skeleton can be given:

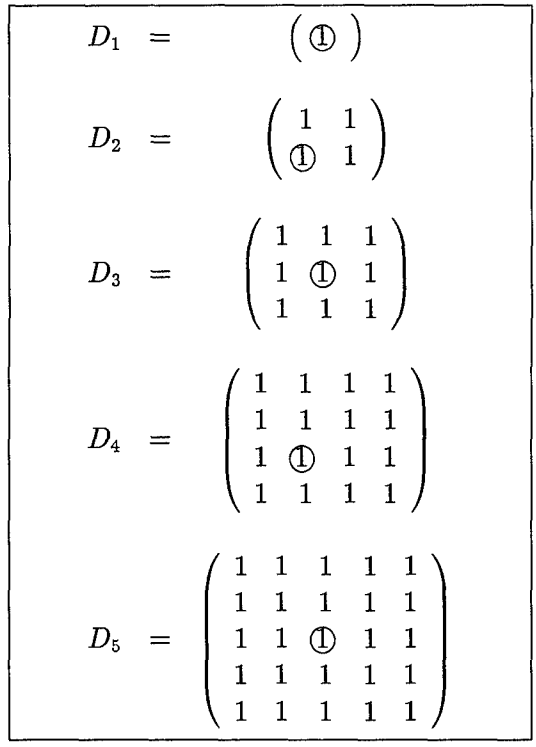


Figure 8.5 Square disks of increasing size (33:353)

Definition VIII.3 Skeleton. *Let T be a constant image (pixel values* 1 *or* $*$*). For any pixel (i,j) in the domain of T, the maximal disk for (i,j),* MAXDISK(i, j)*, is the highest numbered disk D_k, translated so that its new center is at (i,j), such that* TRAN(D_k; i, j) *is a sub-image of T. The skeleton of T,* SKEL (T)*, is a constant image (*1*'s and* $*$*'s) such that a pixel lies within the domain of* SKEL (T) *if and only if its maximal disk is not a proper sub-image of any other translated disk that is itself a sub-image of T.(33:353)* □

A block diagram for *SKEL(T)* is depicted in Figure 8.6, where the skeleton of the image T is taken with respect to the disk D_2. The block diagram shown in the figure works by "finding the skeleton pixels that have maximal disk of edge length 1, then those with maximal disk of edge length 2, and so on. It then takes the set theoretic union of those pixel classes."(33:358) As shown in the figure, there are several operations used to compute the skeleton of an image. Among them are the operations *ERODE* and *OPEN*, and a few matrix valued logical operations. *ERODE* is a thinning operation described in Section 8.3.3. *OPEN* is a smoothing operation described in Section 8.3.4. *OR* and *COMPLEMENT* are

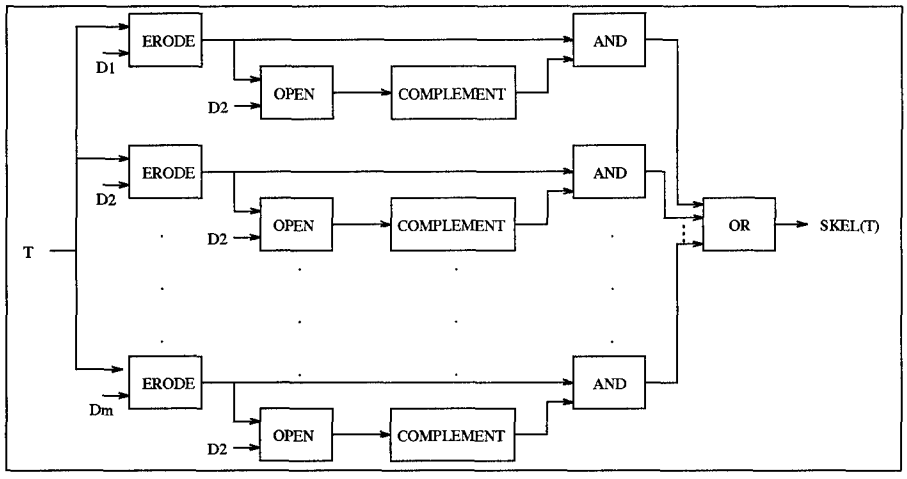


Figure 8.6 Block diagram of SKEL (33:355)

logical operations; *OR* is defined in Definition VIII.8, and *COMPLEMENT* is defined below.

Definition VIII.4 Complement. *Given a constant* $(1, *)$ *valued matrix* f*, the complement of* f *is defined by the following equation.*

$$[COMPLEMENT(f)](i,j) = \begin{cases} 1 \text{ if } f(i,j) = * \\ * \text{ if } f(i,j) = 1 \end{cases} \quad \square$$

8.3.3 Erode. When a human analyzes an image, they do so by filtering the image to form *precepts*, which become "the raw material for analysis."(33:327) Computer based image recognition systems can be developed around this same paradigm. That is, digital images can be filtered to extract relevant information which is then used to categorize the image. One class of filters used for this purpose are *morphological* filters.

"The morphological approach is generally based on the probing of a two-valued [1, $*$] image by some predetermined geometric shape known as a *structuring element.*"(33:327) Morphological filters can be used for such purposes as edge detection, segmentation, and image enhancement.(33). One of the advantages morphological filters have over linear filters is that morphological filters preserve much of the underlying geometric form of an image. One such morphological filter, *ERODE* is defined in this section.

ERODE is defined in terms of a simpler morphological filter. Specifically, *ERODE* is defined in terms of *Minkowski difference.*

Definition VIII.5 Minkowski difference *and* Minkowski sum.*(33) Given two images A and B in R^2, the Minkowski difference of A and B is defined set-theoretically as*

$$A \ominus B = \bigcap_{b \in B} A + b$$

and the Minkowski sum or dilation *of A and B, denoted $\mathcal{D}(A, B)$, is defined set-theoretically as*

$$A \oplus B = \bigcup_{b \in B} A + b$$

□

For example, consider the square A with vertices $\{(0,0), (0,2), (2,0), (2,2)\}$ and the structuring element B defined by the line segment with endpoints $\{(1,1), (2,2)\}$. Then $A \oplus B$ is the union of the sets $A + (i,i)$, $1 \leq i \leq 2$, which defines an object with vertices $\{(1,1), (1,3), (2,4), (4,4), (4,2), (3,1)\}$, and $A \ominus B$ is the intersection of the sets $A + (i,i)$, $1 \leq i \leq 2$, which defines the unit square with vertices $\{(2,2), (2,3), (3,2), (3,3)\}$. These operations are graphically depicted in Figure 8.7. Now that Minkowski subtraction has been defined, a definition for *ERODE* can be given.

Definition VIII.6 Erosion.*(33) Given two sets A and B in R^2, where $-B$ is defined to be the set $\{-b \mid b \in B\}$, where $-b$ is defined to be the scalar multiple of the vector b by -1, the* erosion of *A* by *B, denoted $\mathcal{E}(A, B)$ is defined to be $A \ominus (-B)$.* □

Note that the above definitions of erosion, Minkowski addition and Minkowski subtraction are based on Euclidean geometry, while image recognition systems operate over digitized images. Generalization of the above definitions for use in a digital environment is straightforward. However, before defining digital Minkowski operations, a few other fundamental digital operations must be defined. These operations are defined below.

Definition VIII.7 *(33)* Domain. *If A denotes a constant $(1,*)$ digital image, then the* domain *of A, denoted D_A, is the set $\{(i,j) \mid a_{ij} \in A \wedge a_{ij} = 1\}$.* □

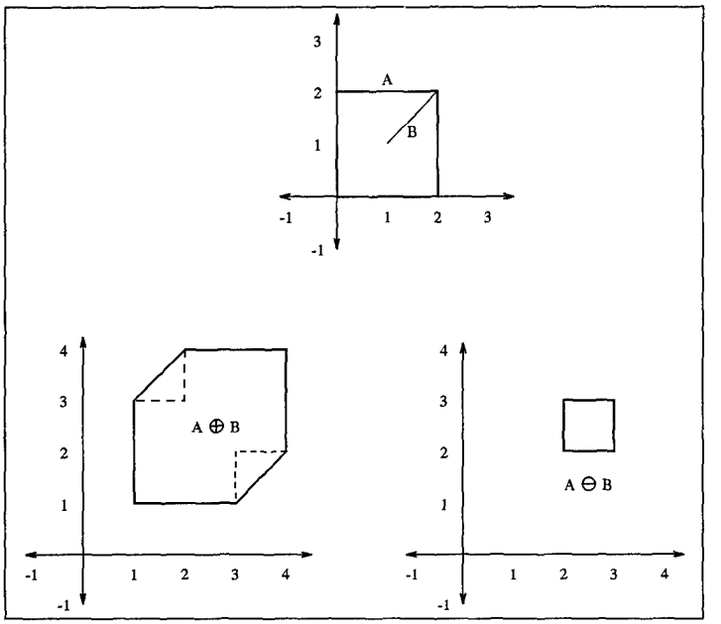


Figure 8.7 Minkowski addition and subtraction

Definition VIII.8 *(33)* And, Or. *If* $S_1, S_2, \ldots, S_n$ *denote constant* $(1,*)$ *images, then*

$$[\bigvee_k S_k] = \begin{cases} 1, & \text{if there exists at least 1 } k' \text{ for which } S_{k'}(i,j) = 1 \\ *, & \text{if } S_k(i,j) = * \text{ for all } k \end{cases}$$

$$[\bigwedge_k S_k] = \begin{cases} 1, & \text{if } S_k(i,j) = 1 \text{ for all } k \\ *, & \text{if there exists at least 1 } k' \text{ for which } S_{k'}(i,j) = * \end{cases} \quad \square$$

The term *AND* will sometimes be used for $\bigwedge$, and *OR* will sometimes be used for $\bigvee$. Now that these simple operations have been defined, digital Minkowski addition and subtraction can be defined.

Definition VIII.9 *(33) The* Minkowski addition *or* dilation *of* S *by* E *where* S *and* E *are constant* $(1,*)$ *valued images is denoted* $S \boxplus E$ *and is defined by the following equation:*

$$\begin{aligned} S \boxplus E &= \bigvee_{(i,j) \in D_S} TRAN(E;i,j) \\ &= DILATE(S,E) \end{aligned}$$

The Minkowski subtraction *of* S *by* E*, denoted* $S \boxminus E$*, is defined by*

$$S \boxminus E = \begin{cases} 1, & \text{if } TRAN(-E;i,j) \bigvee S = S \\ *, & \text{otherwise} \end{cases} \quad \square$$

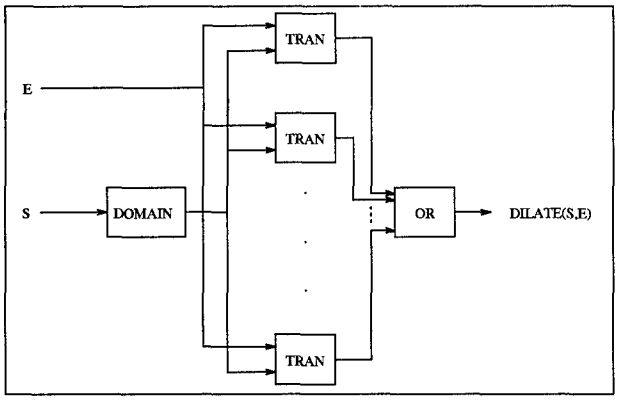


Figure 8.8 Block diagram of DILATE (33:342)

The domain of $S \boxplus E$ equals the union of the domains of the translates *TRAN(E; i,j)*. A block diagram for *DILATE* is shown in Figure 8.8.

Erosion can be described intuitively as "template translation." As noted in (33), "If, for a given pixel, say (i,j), the translated copy of E, *TRAN(E;i,j)*, is a sub-image of S, then (i,j) is activated [in the erosion of S by E]; otherwise (i,j) is given the value $*$ in the image. ...[E]rosion eliminates those parts of the image that are small in comparison to the structuring element."(33:343)

Because $\mathcal{E}(A,B) = A \ominus (-B)$ in Euclidean space, the corresponding digital equation for *ERODE* is *ERODE(A,B)* $= A \boxminus$ *(-B)*. One way to compute $-B$ is to rotate B 180 degrees around the origin. A primitive matrix operation, *NINETY*, can be used for this purpose. Specifically, *-B = NINETY[NINETY(B)]*, where *[NINETY(f)](i,j) = f(j,-i)*.(33) This leads to the alternate formulation of *ERODE*:

Definition VIII.10 ERODE. *Given two constant* $(1,*)$ *valued matrices* S *and* E, *the digital erosion of* S *by* E, *denoted* ERODE(S,E), *is defined as follows:*

$$ERODE(S,E) = \bigwedge_{(i,j)\in DOMAIN[NINETY^2\ (E)]} TRAN(S;i,j)$$

S is called the image or picture, and E is called the structuring element. □

This alternate definition of erosion has the block diagram shown in Figure 8.9. The erosion morphological filter is used in the computation of *OPEN*. *OPEN* is described in the following section.

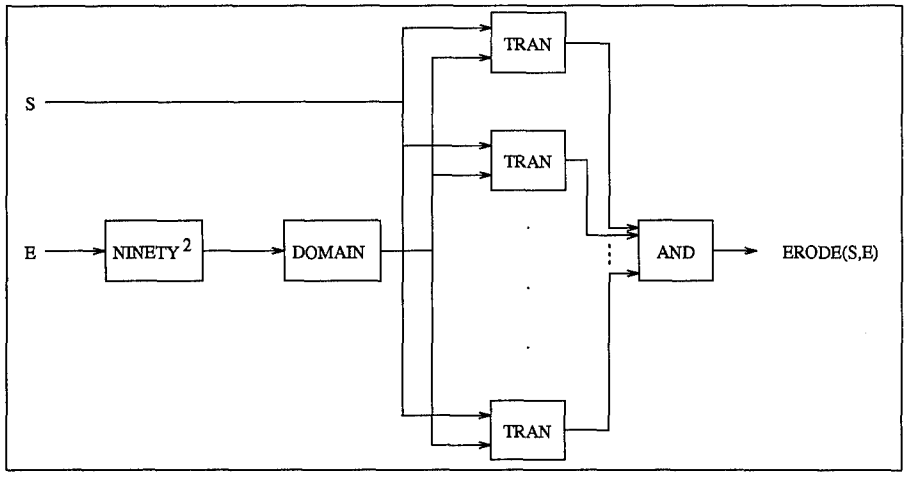


Figure 8.9 Block diagram of ERODE (33:345)

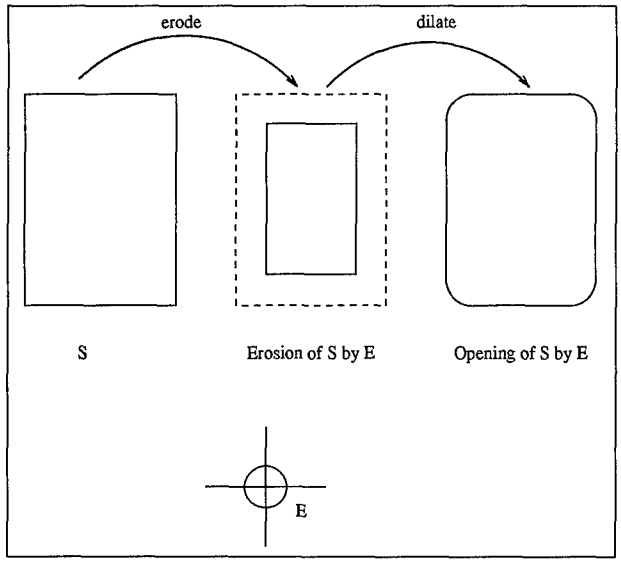


Figure 8.10 Opening in terms of dilation and erosion (33:334)

8.3.4 Open. *OPEN* is a morphological filter that, depending on the structuring element used, has a smoothing effect on the input image and has the effect of "expanding" the image in a manner defined by the structuring element.(33) Open can be defined using the Euclidean operations of erosion and dilation.

Definition VIII.11 *(33)* Opening. *Given two images A and B in* R^2, *the* opening *of A by B, denoted O(A,B), is defined by the equation* $\mathrm{O(A,B)} = \mathcal{D}[\mathcal{E}(A,B),B]$. □

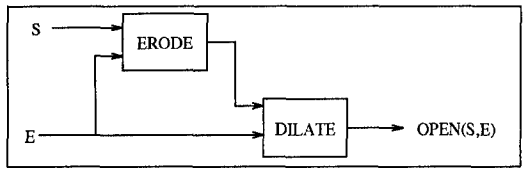


Figure 8.11 Block diagram of OPEN (33:347)

As a simple example, consider the series of images depicted in Figure Figure 8.10. The input image, a rectangle, is opened by the second image, a disk. The erosion of the rectangle by the disk yields an image that has been thinned or shrunk. Dilation of the eroded image yields an image in which the corners of the rectangle have been rounded.

Definition VIII.11 defines Euclidean open. Digital opening is defined below.

Definition VIII.12 OPEN. *Given constant* $(1,*)$ *valued images A and B, the* opening *of A by B, denoted OPEN(A,B), is defined by the equation*

$$\begin{aligned} OPEN(A,B) &= [A \boxminus (-B)] \boxplus B \\ &= DILATE[ERODE(A,B), B] \qquad \square \end{aligned}$$

DILATE is the digital analog to Euclidean dilation $\mathcal{D}$; see Definition VIII.9. Now that *ERODE*, *OPEN*, *COMPLEMENT*, *AND* and *OR* have been defined, a definition for digital erosion can be given.

Definition VIII.13 SKEL. *Given a constant* $(1,*)$ *valued image T, the* skeleton *of T, denoted* SKEL(T), *is defined by the equation*

$$\begin{aligned} SKEL(T) = OR_{\mathcal{D}}\,[AND\ \ & (COMPLEMENT(OPEN(ERODE(T,D_m),D_2)), \\ & ERODE(T,D_m))] \end{aligned}$$

where $\mathcal{D} = \{D_i : \exists(i,j)((i,j) \in DOMAIN(T) \Rightarrow TRAN(D_i;i,j) \subseteq T)\}$. *That is,* $\mathcal{D}$ *is the maximal set of square disks such that any disk* D_i *in* $\mathcal{D}$ *can be translated such that it is a sub-image of T.* $\square$

This definition of *SKEL* is based on the block diagram of Figure 8.6, and it will be used in conjunction with Figure 8.6 to guide the development of a process based specification for this morphological filter.

8.3.5 Summary. This section has defined a variety of morphological filters useful for a feature selection stage of an image recognition application, including *DILATE*, *OPEN*, *ERODE*, and *SKELETON*. Block diagrams for these operations were presented. As evidenced by the block diagrams, these filters can be defined using a relatively small set of building blocks. For example, *OPEN* was defined in terms of *DILATE* and *ERODE*. This indicates that once specifications for the simple filters have been developed, they can be used as building blocks in the construction of specifications for the more complex filters.

Now that these filters have been defined, formal process based specifications for them can be developed. These specifications are developed in the following section.

8.4 Specification Development

In this section, process based specifications for the morphological filters defined in the preceding section are developed. One of these filters, *SKEL*, is used to define the feature-selection stage of the image processing application depicted in Figure 6.16 and partially specified in Figure 6.18. As part of the specification development effort, the batch sequential design of Figure 6.18 is extended to a piped batch sequential design so that the results of one filter process such as *Selection* can be made available to the next filter process in turn.

This section is organized as follows:

1. In Section 8.4.1 a reusable design based on a partition-solve-compose paradigm is developed.
2. In Section 8.4.2 a process-based specification for *ERODE* is developed.
3. In Section 8.4.3 diagrams leading to process-based specifications for *DILATE* and *OPEN* are developed.
4. In Section 8.4.4 a diagram leading to a process-based specification for the filter *SKEL* is developed.

5. In Section 8.4.5 the process *SKEL* is associated with the process *Selection* to produce a piped batch sequential design for an image recognition system which uses the skeleton of an image for classification purposes.

8.4.1 Specification Development for Partition-Solve-Compose. The morphological filters depicted in Figures 8.6, 8.8, and 8.9 each show several concurrent sub-designs. For example, the block diagram for the filter *ERODE*, pictured in Figure 8.9, shows several concurrent filters, *TRAN*, each encapsulating a translate operation. However, a careful examination of the definitions for *SKEL*, *DILATE*, and *ERODE* reveals that the number of concurrent sub-designs is data dependent. Furthermore, Assumption I.5 states that the framework defined in the previous chapters can be used to develop process based specifications containing only static communication networks; CSP does not support dynamic, data-dependent specification of communication networks. Therefore, process based specifications for the morphological filters *ERODE*, *DILATE*, and *SKEL* must be defined using static communication networks.

Although data dependent parallelism cannot be achieved within this framework, some degree of flexible parallelism can be provided. Specifically, an inspection of Figures 8.6, 8.8, and 8.9 reveals a common architectural pattern: each figure contains a one or two to m partitioning of the incoming data, i.e., m parallel sub-designs used to compute partial solutions, and an m to one composition of the partial solutions. Because a partition-solve-compose design can be reused in the construction of process based specifications for each of the three morphological filters *ERODE*, *DILATE*, and *SKEL*, a process based specification for this partition, solve, compose approach is developed in this section and will be used in the development of the specifications for the individual filters.

The partition-solve-compose approach has the block diagram shown in Figure 8.12. The arrows in the figure represent communication channels between processes. Port names and port sorts have been added to the figure to highlight the relationship between the block diagram and its process based specification; a specification based on Figure 8.12 will be presented shortly. As shown in the figure, the process *Partition* accepts set-valued inputs, where the sort of individual set elements is left abstractly specified. *Partition*

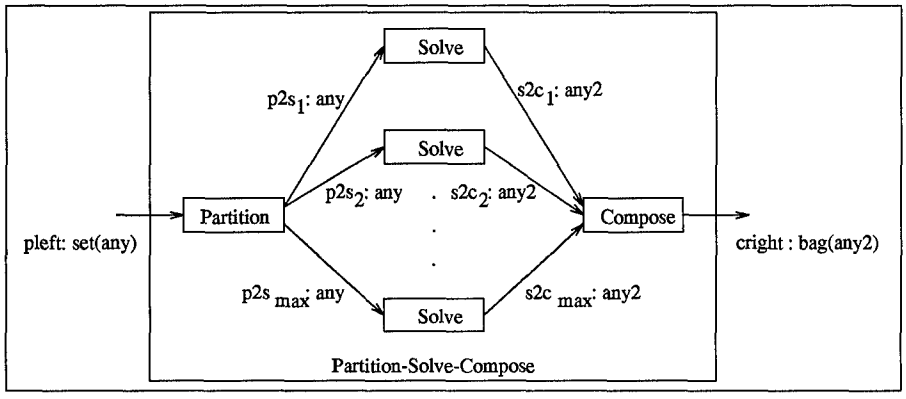


Figure 8.12 Block diagram of Partition-Solve-Compose

shares a channel with each *Solve* process, and each *Solve* shares a channel with the process *Compose*. The output port of the specification *Compose*, *cright*, is used to communicate the results collected from the individual *Solve* processes. The operation of the processes of Figure 8.12 are described in the next several paragraphs.

Partition defines a process that accepts a set valued input over an input channel. After accepting the input, $Partition_{mod}$ enumerates over the input set, communicating individual set elements over its output channels such that each element in the input set is communicated exactly once. The output channels of $Partition_{mod}$ are input channels of *Solve* processes. A *Solve* process is defined to have exactly two external ports, one for input and one for output. Each *Solve* process reads incoming data from its input channel, operates over that data, and communicates its results over its output channel. The outputs of the *Solve* processes are collected by the process *Compose*. The output of the process *Compose* is the complete set or bag of solutions generated by the *Solve* processes. Bags are used as the output sort of *Compose* in case duplicate solutions are significant. A process specification for these processes is shown in Figure 8.13.

As shown in Figure 8.13, *Partition* contains an input port, *pleft*, and a collection of output ports, $p2s_i$, one for each *Solve* process. The process expression of *Partition* defines a process that accepts a set-valued input over *pleft*, and while the set is not empty, communicates an arbitrary element, *el*, of the input set to one of the *Solve* processes, removes that element from the input set, and repeats. After all elements of the input set have been communicated to *Solve* processes, *Partition* is defined to engage in the event

pspec Partition-Solve-Compose is
sorts any, set(any), any2, bag(any2)
const max : Natural
op empty : set(any) → boolean,
op arb : set(any) → any,
op less : set(any), any → set(any)
op with : any2, bag(any2) → bag(any2)
var b : any2
var accum : bag(any2)
var el : any
var set-any : set(any)
port pleft : set(any)
port p2s$_i$: any, $i = 1..max$
port s2c$_i$: any2, $i = 1..max$
port cright : bag(any2)
event done

process Partition : events: {done}
chan: {pleft : set(any), p2s$_i$: any $i = 1..max$ }
act: { empty : set(any) → boolean, arb : set(any) → any,
less : set(any), any → set(any) }
var: {set-any : set(any), el : any}
process Solve$_i$: events: {done}
chan: {p2s$_i$: any, s2c$_i$: any2 }
act: {}
var: {}, $i = 1..max$
process Compose : events: {done}
chan: {s2c$_i$: any2 $i = 1..max$, cright : bag(any2) }
act: {with : any2, bag(any2) → bag(any2)}
var: {accum : bag(any2), b : any2 }
process Part-Solve-Comp : events: {done}
chan : ...
act : ...
var : ...

Partition sat {pleft?set-any $\xrightarrow{CSP}$
[not(empty(set-any))∗[el:= arb(set-any);
(x:{p2s$_i$!el | i ∈ [1..max]} $\xrightarrow{CSP}$ set-any := set-any less el; Skip)]]} ;
done $\xrightarrow{CSP}$ Skip
Solve$_i$ sat done $\xrightarrow{CSP}$ Skip
Compose sat (x:{s2c$_i$?b | i ∈ [1..max]} $\xrightarrow{CSP}$ accum := accum with b; Compose))
| (done $\xrightarrow{CSP}$ (cright!accum $\xrightarrow{CSP}$ Skip))
Part-Solve-Comp sat (Partition $\|_{i=1..max}$ Solve$_i$)|| Compose
end-pspec

Figure 8.13 Specification for Partition-Solve-Compose

done. The event *done* is used by *Partition* to signal that there are no further elements of the input set left to communicate. Similarly, each process *Solve* is defined to engage in the input event *done*, as is the process *Compose*. *Compose* is defined to accept inputs over the channels it shares with the *Solve* processes, and accumulates these inputs in a bag data structure. When *Partition*, *Compose*, and each *Solve* process are ready to engage in the event *done*, they do so simultaneously. After engaging in *done*, *Compose* is defined to output over the port *cright* the bag of values it has accumulated. The construct *(x:B* $\xrightarrow{CSP}$ *P(x))* is used in the definition of *Partition* and *Compose* so that these processes can engage in communication with any enabled *Solve* process. That is, when *Partition* is ready to communicate with a *Solve* process, it will communicate an element from the input set to the first *Solve* process ready to receive input, and when *Compose* is ready to accept an input from a *Solve* process, it will accept input from the first *Solve* process ready to report its results.

Note that in Figure 8.13, the alphabet of the process *Part-Solve-Comp* has been denoted with ellipses for purposes of brevity. Also note that the notation *process* $Solve_i$ *: events :* {} ... *var :* {}, *i=1..max* declares an indexed collection of process symbols, each of which has the alphabet given. Individual solve processes can be referenced through subscripting. For example, $Solve_3$ references the third solve process. Similarly, the notation *port* $p2s_i$ *: any i=1..max* declares an indexed collection of port symbols of sort *any*.

The specification *Partition-Solve-Compose* can be refined through specialization of the sort symbols *any* and *any2*, and through specification of the constant *max*. Additional port and sort symbols can be added to the specification through specification morphism to define a partition-solve-compose design in which each process $Solve_i$ accepts more that one input. Such a specification is developed later in this section.

Solve is left abstractly specified in Figure 8.13; the intent is that *Solve* will be refined through process specification morphism. The values generated by *Partition-Solve-Compose*$_{mod}$ are dependent on the definition of *Solve*; any discussion concerning the value generated a model of *Partition-Solve-Compose* will have to wait until *Solve* is refined. If a more detailed process expression for *Solve* were given, then refinement of any $Solve_i$ through process specification morphism must preserve the traces of that expression. For

example, if each $Solve_i$ was defined to satisfy $(p2s_i?x \xrightarrow{CSP} (s2c_i!f_i(x) \xrightarrow{CSP} Skip)) \mid (done \xrightarrow{CSP} Skip)$, then any refinement of $Solve_i$ through specification morphism must preserve the set of traces $\{\langle\rangle, \langle done\rangle, \langle p2s_i.x\rangle, \langle p2s_i.x, s2c_i.f_i(x)\rangle\}$. Because $f_i(x)$ is an output value referenced in a trace of $Solve_i$, any refinement of $Solve_i$ through specification morphism must contain $f_i(x)$ as an output value. This greatly restricts the options available for defining f_i. However, because $Solve_i$ is only defined to satisfy $done \xrightarrow{CSP} Skip$, $Solve_i$ can be refined through process specification morphism to be a collection of communicating processes.

Note that *there is no requirement that the collection of* Solve *processes be heterogeneous.* For example, a pipeline design P can be associated with one of the *Solve* processes, say $Solve_i$, by defining morphisms from a trivial specification T to $Solve_i$ and from T to P. Similarly, a layered design L can be associated with one of the other solve processes, say $Solve_j$. The colimit object of the resulting diagram will be a partition-solve-compose specification in which the pipeline design P and the layered design L are each used in parallel to find solutions for elements of a common input data set.

Homogeneous designs, such as those required for *Dilate* or *Erode* can also be defined. A homogeneous design in which *Tran* is used for each process $Solve_i$ is developed later in this section. However, before such a design can be developed, *Partition-Solve-Compose* must be extended so that partition accepts two inputs from the environment and provides two outputs to each *Solve* process. This extension is defined next.

It is a simple matter to extend *Partition* and *Solve* through specification morphism so that they share a pair of channels, such that *Partition* accepts two inputs from the outside environment and provides two outputs to each *Solve* process. Specifically, the sort s and the ports $pleft2 : s$ and $p2s2_i : s$, $i=1..max$ can be added to the specification *Partition-Solve-Compose.* The alphabet of *Partition* can then be extended with the port *pleft2* of sort s and with the indexed collection of ports $p2s2_i$ of sort s, $i = 1..max$. Similarly, the alphabet of each $Solve_i$ can be extended with the port $p2s2_i$ of sort s. This extension effectively defines a collection of channels from *Partition* to each process $Solve_i$ for $i=1..max$. In addition, the process expression for *Partition* can be extended to include communication over these new ports. Figure 8.14 depicts such an extension. Only those portions of Figure 8.13 that have been modified are shown. It is easily verified that this

pspec Two-Partition-Two-Solve-Compose is
sorts any, set(any), any2, bag(any2), s
port pleft2 : s
port $p2s2_i$: s, $i = i..max$
var m : s
$\vdots$

process Two-Partition : events: {done}
chan: {pleft : set(any), pleft2 : s,
$p2s_i$: any, $i = 1..max$
$p2s2_i$: s, $i = 1..max$ }
act: { empty : set(any) $\rightarrow$ boolean,
arb : set(any) $\rightarrow$ any,
less : set(any) $\rightarrow$ set(any) }
var: {set-any : set(any), m : s}

process Two-Solve$_i$: events: {done}
chan: {$p2s_i$: any, $p2s2_i$: s, $s2c_i$: any2 }
act: {}
var: {}, $i = 1..max$

Two-Partition sat {pleft2?m $\xrightarrow{CSP}$ pleft?set-any $\xrightarrow{CSP}$
[not(empty(set-any))*[el:= arb(set-any);
(x:{$p2s_i$!el | i $\in$ [1..max]} $\xrightarrow{CSP}$
($p2s2_i$!m $\xrightarrow{CSP}$ set-any := set-any less el; Skip))]]};
done $\xrightarrow{CSP}$ Skip

$\vdots$

Two-Part-Solve-Comp sat (Two-Partition $\|_{i=1..max}$ Two-Solve$_i$)|| Compose
end-pspec

Figure 8.14 Specification for Two-Partition-Solve-Compose

extension defines a process specification morphism from *Partition-Solve-Compose* to *Two-Partition-Two-Solve-Compose.* Note that the process expression for *Two-Partition* defines a process that accepts a matrix-sorted value before it will accept a value of sort *set(index).* Attempts to communicate these values in the reverse order will lead to deadlock. This explicit ordering places some constraints on the use of communication channels in any process defined by some $Solve_i$. A more general expression for *Two-Partition* would permit incoming communication to be in either order.

Two-Partition-Two-Solve-Compose is used in the following sections in the development of specifications for the morphological filters described in the preceding section.

8.4.2 Specification of Erosion. The erosion of S by E was defined in Section 8.3.3 by the equation $ERODE(S,E) = \bigwedge_{(i,j)\in DOMAIN[NINETY^2\ (E)]} TRAN(S;i,j)$. This section develops a process based specification for this morphological filter. The specification for *ERODE* developed in this section is based on the block diagram of Figure 8.9. Each box in Figure 8.9 is treated as a separate process; there are four such boxes, $Ninety^2$, *Domain*, *Tran*, and *And.* Specifications for these processes are straightforward and are shown in Figures 8.15, 8.16, 8.17, and 8.18 respectively. The partition-solve-compose specification of Figure 8.14 is refined for use in a specification for *ERODE*, leading to the block diagram of Figure 8.20. This block diagram is used to guide the development of a specification for digital erosion.

In the paragraphs that follow, the specifications *Two-Partition-Two-Solve-Compose*, *And*, *Tran*, *Domain* and *Ninety-Sq* are combined through specification morphism to define the specification *Erode.* Construction of a specification for the filter *ERODE* proceeds in three steps:

1. *Domain* and *Ninety-Sq* are combined using a pipeline structuring specification to produce the specification *Domain-Ninety-Sq.*
2. The specification *Two-Partition-Two-Solve-Compose* is refined for use with the filter *TRAN.*

```
pspec Ninety-Squared is
        sort matrix
        var m : matrix
        op ninety : matrix → matrix
        port left : m
        port right : m
        process Ninety-Sq :
                events: {}
                chan: {left : m, right : m}
                act: {ninety : matrix → matrix}
                var: {m : matrix}
        Ninety-Sq sat left?m ⟶(CSP) right!(ninety(ninety(m)))
end-pspec
```

Figure 8.15 Specification for Ninety^2

```
pspec Domain-Spec is
        sorts matrix, index, set(index)
        var m-in : matrix
        op domain : matrix → set(index)
        port left : matrix
        port right : set(index)
        process Domain :
                events: {}
                chan: {left : matrix, right : set(index)}
                act: {domain : matrix → set(index)}
                var: {m-in : matrix}
        Domain sat left?m-in ⟶(CSP) right!domain(m-in)
end-pspec
```

Figure 8.16 Specification for Domain

3. The resulting specifications from the above two steps are combined with *And* to yield a specification for erode having the structure depicted in Figure 8.20.

Development proceeds by defining a pipeline structure which returns the value *domain (ninety (ninety (E)))* for an input E.

```
pspec Tran is
        sorts matrix, index
        op tran : matrix, index → matrix
        var tran-mat: matrix
        var ij : index
        port tleft1 : index
        port tleft2 : matrix
        port tright : matrix
        process Tran:
                chan : {tleft1 : index, tleft2 : matrix,
                        tright : matrix }
                var : { tran-mat : matrix, ij : index}
                event : {}
                act : { tran : matrix, index → matrix}
        Tran sat (tleft1?ij --CSP--> tleft2?m --CSP-->
                tright!tran(m, ij) --CSP--> Skip);Tran
end-pspec
```

Figure 8.17 Specification for Translate

```
pspec And-Or-Spec is
        sorts matrix, bag(matrix)
        var and-matrix-bag : bag(matrix)
        var or-matrix-bag : bag(matrix)
        op and : bag(matrix) → matrix
        op or : bag(matrix) → matrix
        port and-left : bag(matrix)  port or-left : bag(matrix)
        port and-right : matrix      port or-right : matrix
        process And :
                events: {}
                chan: {and-left : bag(matrix), and-right : matrix}
                act: {and : bag(matrix) → matrix}
                var: {and-matrix-bag : matrix}
        And sat and-left?and-matrix-bag --CSP--> and-right!and(and-matrix-bag)
        process Or :
                events: {}
                chan: {or-left : bag(matrix), or-right : matrix}
                act: {or : bag(matrix) → matrix}
                var: {or-matrix-bag : matrix}
        Or sat or-left?or-matrix-bag --CSP--> or-right!or(or-matrix-bag)
end-pspec
```

Figure 8.18 Specification for the Logical Operations And and Or

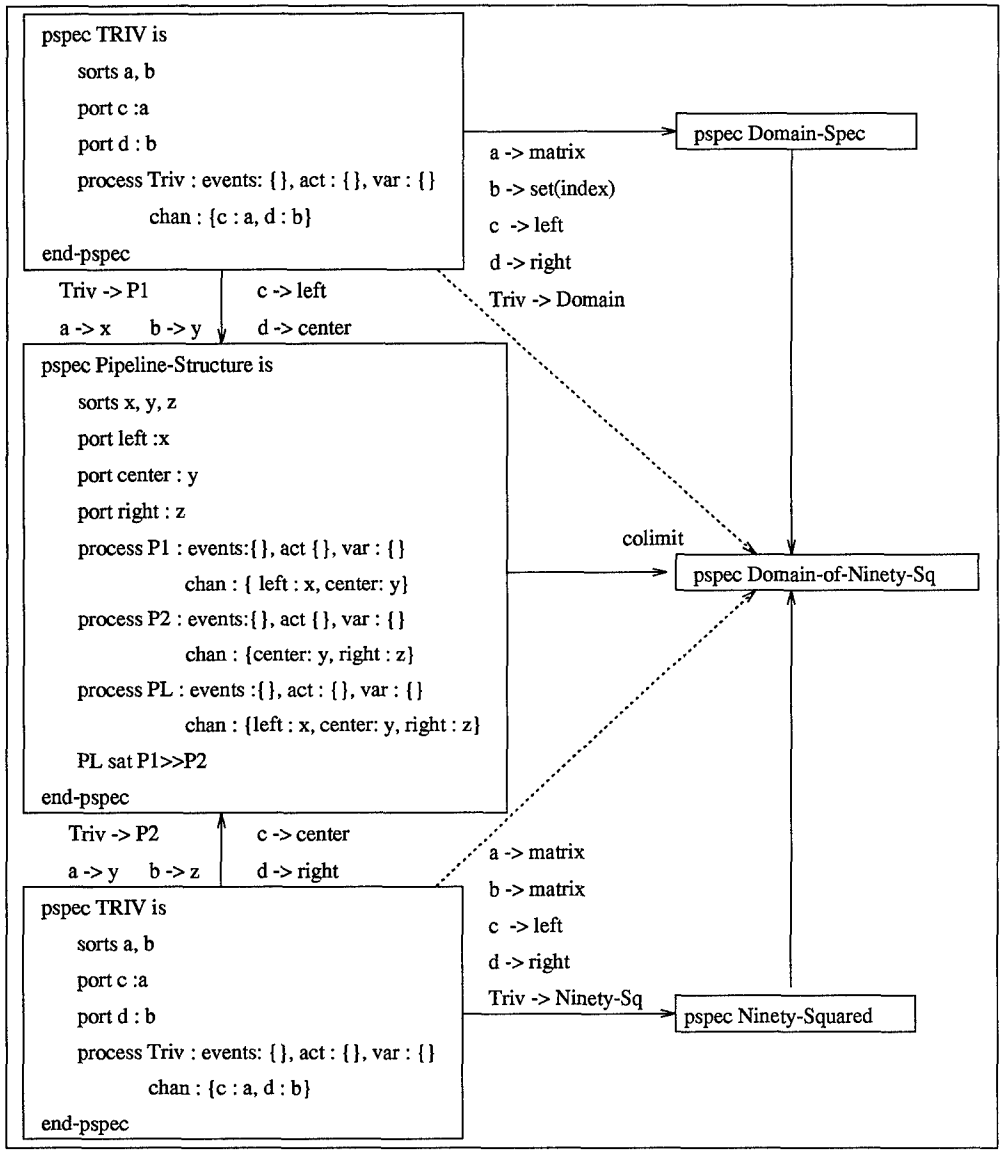


Figure 8.19 Specification Construction for Domain≫Ninety-Sq

Process specifications for the filters *DOMAIN* and $NINETY^2$ are shown in Figures 8.16 and 8.15 respectively. Because both *Domain* and *Ninety-Sq* define stages, one way to compose *Domain* with *Ninety-Sq* is through a structuring specification for a pipeline architecture as shown in Figure 8.19; note that morphisms used to unify the sort symbol *matrix* of *Domain-Spec* with the sort symbol *matrix* of *Ninety-Squared* are present but for purposes of brevity are not shown. The colimit object of Figure 8.19, the specification *Domain-of-Ninety-Sq*, is shown in Figure 8.21. For purposes of clarity, equivalence classes

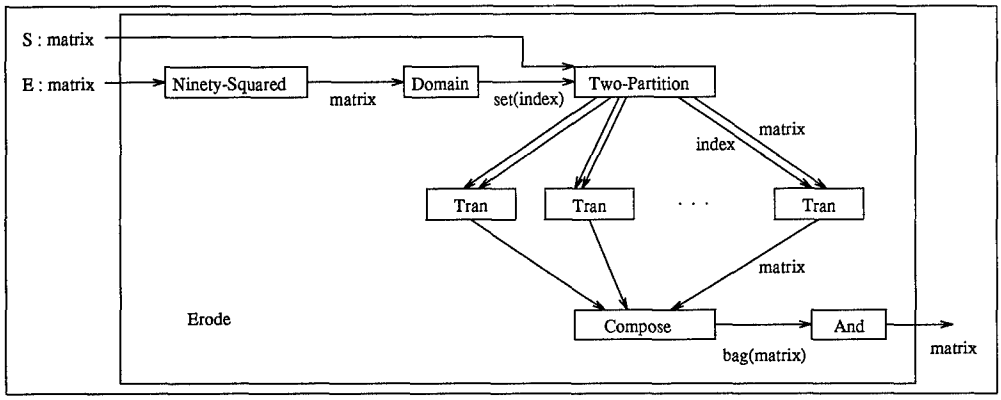


Figure 8.20 Structure of Erode

of symbols such as $\{x,\ y,\ matrix,\ a\}$, are not shown in the colimit object. It is easily verified that the pipeline process, *Domain-Ninety-Sq*, of Figure 8.21 produces the value *domain(ninety(ninety(E)))* of sort *set(index)* for an input value E. It is also easily verified that the morphisms shown in Figure 8.19 are process specification morphisms.

The specification *Domain-of-Ninety-Sq* defines part of the erosion process. The next part of erosion is specified through a specialization of the process specification *Two-Partition-Two-Solve-Compose*. Specifically, the process *Tran* is extended for use in the partition-solve-compose specification of Figure 8.14.

Refining the specification *Two-Partition-Two-Solve-Compose* so that each process $Two\text{-}Solve_i$ computes the translation of an input matrix by a index is relatively straightforward. Refinement can be accomplished by defining an *implementation* of *Two-Solve* by *Tran*, where "an implementation of a specification S by a specification T is a realization of the behavior specified in S using the concepts from T."(56:2) The concept of interpretation is formalized in the following definition.

Definition VIII.14 *(Based on (56).) An interpretation of a specification S to a specification T is a pair of specification morphisms* S $\rightarrow$ S-as-T $\hookleftarrow$ T *where the arrow* $\hookleftarrow$ *denotes extension by definition.* □

Define the specification *Two-Solve-as-Tran* as shown in Figure 8.22. Then an implementation of *Two-Solve* by *Tran*, denoted *Two-Solve* $\rightarrow$ *Two-Solve-as-Tran* $\hookleftarrow$ *Tran*, can be defined as follows:

```
pspec Domain-of-Ninety-Sq is
    sorts matrix, index, set(index)
    var m : matrix
    var m-in : matrix
    op domain : matrix → set(index)
    op ninety : matrix → matrix
    port left : matrix
    port center : matrix
    port right : set(index)

    process Domain :
        events: {}
        chan: {center : matrix, right : set(index)}
        act: {domain : matrix → set(index)}
        var: {m-in : matrix}

    process Ninety-Sq :
        events: {}
        chan: {left : matrix,center : matrix}
        act: {ninety : matrix → matrix}
        var: {m : matrix}

    Process Domain-Ninety-Sq :
        events: {}
        chan: {left : matrix, center : matrix, right : set(index)}
        act: {ninety : matrix → matrix, domain : matrix → set(index)}
        var: {m : matrix, m-in : matrix}

    Domain sat center?m-in --CSP--> right!domain(m-in)
    Ninety-Sq sat left?m --CSP--> center!(ninety(ninety(m)))
    Domain-Ninety-Sq sat Ninety-Sq ≫ Domain

    end-pspec
```

Figure 8.21 Specification for Ninety^2-Domain Pipeline

```
pspec Two-Solve-as-Tran is
      sorts matrix, index
      op tran : matrix, index → matrix
      var tran-mat: matrix
      var ij : index
      port tleft1 : index
      port tleft2 : matrix
      port tright : matrix
      event done
      process Two-Solve-as-Tran:
            chan : {tleft1 : index, tleft2 : matrix,
                    tright : matrix }
            var : { tran-mat : matrix, ij : index}
            event : {}
            act : { tran : matrix, index → matrix}
      Two-Solve-as-Tran sat (done ⟶CSP Skip) |
            (tleft1?ij ⟶CSP tleft2?m ⟶CSP
                  tright!tran(m, ij) ⟶CSP Skip]);Two-Solve-as-Tran
end-pspec
```

Figure 8.22 Specification for Two-Solve-as-Tran

- The specification morphism *Two-Solve* → *Two-Solve-as-Tran* is defined by the map $\{done \mapsto done,\ any \mapsto index,\ any2 \mapsto matrix,\ p2s \mapsto tleft1,\ p2s2 \mapsto left2,\ s \mapsto matrix\}$.

- The specification morphism *Two-Solve-as-Tran* ↩ *Tran* is defined by extending the alphabet of *Tran* with the event symbol *done* and by extending the process expression of *Tran* with the nondeterministic choice of $done \overset{CSP}{\longrightarrow} Skip$.

Because *Two-Solve* can be extended through process specification morphism to define *Two-Partition-Two-Solve-Compose*, and because process specification morphisms compose to form process specification morphisms, the implementation of *Two-Solve* by *Tran* can be extended through process specification morphism to define the specification *Two-Partition-Tran-Compose* as shown in Figure 8.23. The arrows in the figure denote specification morphisms, with e denoting an extension, m denoting a morphism, and d denoting an extension by definition. The specification *Two-Partition-Tran-Compose* is shown in Figure 8.24. For purposes of clarity, equivalence classes are represented by a single symbol from that class.

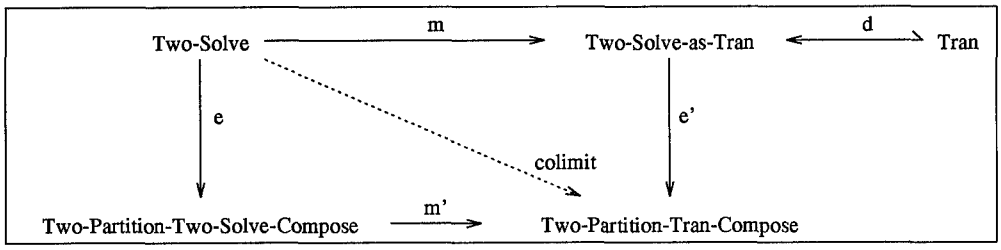


Figure 8.23 Specification Diagram for Two-Partition-Tran-Compose

For example, the equivalence class of sort symbols {*any, index*} is represented by the symbol *index*. Specifically, the specification of Figure 8.24 is the colimit object of Figure 8.23 translated by the following map:

{tleft1, p2s}	$\mapsto$	p2t
{tleft2, p2s2}	$\mapsto$	p2t2
{s2c,tright}	$\mapsto$	t2c
{any2, s, matrix}	$\mapsto$	matrix
{any, index}	$\mapsto$	index
{Two-Solve, Tran, Two-Solve-as-Tran}	$\mapsto$	Solve-as-Tran
set-any	$\mapsto$	index-set
Part-Solve-Comp	$\mapsto$	Two-Partition-Tran-Compose

where the rest of the map is defined by the identity map.

Now that *Solve* has been refined, the values returned by the processes defined by *Two-Partition-Tran-Compose* can be determined. Specifically, for an input matrix S and an input set of indices E, $Two\text{-}Partition\text{-}Tran\text{-}Compose_{mod}$ generates the value { *tran(S, ij) : ij* $\in S$ as follows:

1. *Partition* accepts the values S and E, and enumerates over E communicating both S and an element of E to individual *Tran* processes.
2. *Tran* accepts the matrix S and the index ij and generates the value *tran(S, ij)*. Because *Partition* enumerates over the entire set E, there will be one value *tran(S,ij)* generated by a *Tran* process for every ij in E.
3. *Compose* accepts the values generated by individual *Tran* processes and collects them in a bag data structure. *Compose* then communicates the accumulated bag of values over its output channel.

pspec Two-Partition-Tran-Compose is
sorts index, set(index), matrix, bag(matrix) }
const max : nat
port pleft : set(any) port pleft2 : matrix
port p2t$_i$: index, $i = i..max$ port p2t2$_i$: matrix, $i = i..max$
port t2c−i : matrix, $i = 1..max$ port cright : bag(matrix)
var m : matrix var tran-mat : matrix
var b : index var accum : bag(matrix)
var el : index var index-set : set(index)
op empty : set(index) → boolean,
op arb : set(index) → any,
op less : set(index), index → set(index)
op with : matrix, bag(matrix) → bag(matrix)
op tran : matrix, index → matrix
event done

process Two-Partition : events: {done}
chan: {pleft : set(index), pleft2 : matrix,
p2t$_i$: index $i = 1..max$, p2s2$_i$: matrix $i = 1..max$ }
act: { empty : set(index) → boolean, arb : set(index) → index,
less : set(index) → set(index) }
var: {index-set : set(index), m : matrix}
process Solve-as-Tran$_i$: events: {done} chan: {p2t$_i$: index, p2t2$_i$: matrix }
act: {tran : matrix, index → matrix}
var: {tran-mat$_i$: matrix, ij : index}, $i = 1..max$
process Compose : events: {done} chan: {t2c$_i$: matrix $i = 1..max$, cright : matrix }
act: {with : bag(matrix), matrix → bag(matrix) }
var: {b : matrix, accum : bag(matrix)}
process Two-Partition-Tran-Compose : ...

Two-Partition sat {pleft2?m $\xrightarrow{CSP}$ pleft?index-set $\xrightarrow{CSP}$
[not(empty(index-set))*[el:= arb(index-set);
(x:{p2t$_i$!el | i∈[1..max]} $\xrightarrow{CSP}$
(p2t2$_i$!m $\xrightarrow{CSP}$ index-set := index-set less el; Skip))]]};
done $\xrightarrow{CSP}$ Skip
Solve-as-Tran$_i$ sat (done $\xrightarrow{CSP}$ Skip) |
(p2t$_i$?ij $\xrightarrow{CSP}$ p2t2$_i$?tran-mat $\xrightarrow{CSP}$ t2c$_i$!tran(m, ij) $\xrightarrow{CSP}$ Skip]);Solve-as-Tran$_i$
Compose sat (x:{t2c$_i$?b | i∈[1..max]} $\xrightarrow{CSP}$ accum := accum with b; Compose)
| (done $\xrightarrow{CSP}$ (cright!accum $\xrightarrow{CSP}$ Skip))
Two-Partition-Tran-Compose sat (Two-Partition $\|_{i=1..max}$ Solve-as-Tran$_i$)|| Compose
end-pspec

Figure 8.24 Specification for Two-Partition-Tran-Compose

The specification of Figure 8.24 defines the last major building block for the specification *Erode*. What remains to be done to complete the process based specification *Erode* is to connect the output of the process *Domain-Ninety-Sq* of Figure 8.21 to the set-sorted input of the process *Two-Partition-Tran-Compose* of Figure 8.24, and to connect the output of *Two-Partition-Tran-Compose* to the input of the process *And* of Figure 8.18. This can be accomplished through port symbol unification as described in Section 6.3.1. Figure 8.25 is specific to this case.

In Figure 8.25, port symbols *pleft* of the specification *Two-Partition-Tran-Compose* and *left* of the specification *Domain-Ninety-Sq* are unified using a simple channel specification. Similarly, the port symbols *cright* of *Two-Partition-Tran-Compose* and *and-left* of *And* are also unified. The colimit of the diagram, *Connected-Erode* contains specifications for the processes *Domain-of-Ninety-Sq*, *Two-Partition-Tran-Compose*, and *And*, including specifications of their subprocesses. In addition, port unification has resulted in the formation of two CSP channels, one from *Domain-Ninety-Sq* to *Two-Partition-Tran-Compose* and one from *Two-Partition-Tran-Compose* to *And*. However, the colimit object does not yet fully define how these processes are related. Therefore, the colimit object is extended with the process symbol *Erode* and the process expression *Erode sat Domain-Ninety-Sq* || *Two-Partition-Tran-Compose* || *And*. Translation of the colimit object is used to clean up the specification by replacing equivalence classes with a representative element, and to rename the external channels of *Erode* to *s-erode*, *e-erode*, and *erode-out*, where *s-erode* is the matrix sorted input port of *Two-Partition-Tran-Compose*, *e-erode* is the set-sorted input port of *Domain-Ninety-Sq*, and *erode-out* is the output port of *And*. The resulting process expression for *Erode* defines a collection of processes having the structure depicted in Figure 8.20. This nearly completes the generation of a process based specification for the morphological filter *ERODE*; all that remains is to provide a definition for the constant *max*.

For values S and E, where S is an image and E is a structuring element, *Erode* defines a process encapsulating the filter *ERODE* as follows:

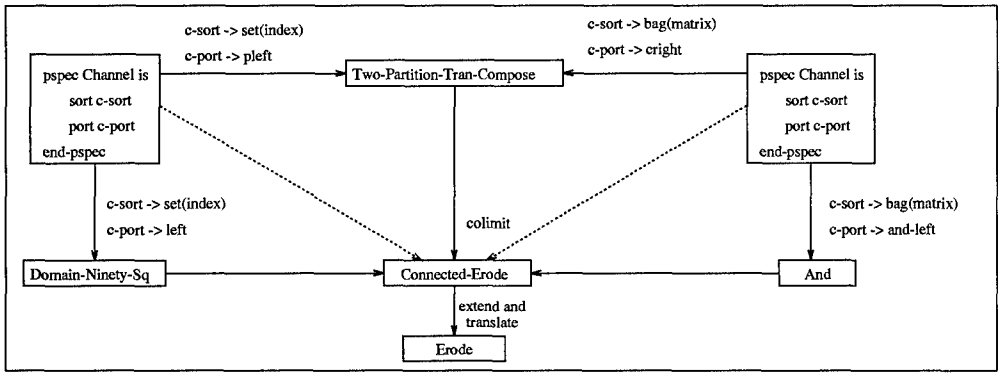


Figure 8.25 Specification for Erode

1. The process *Domain-Ninety-Sq* accepts the value E from the environment and generates the value .

2. The process *Two-Partition-Tran-Compose* accepts the value S from the environment and accepts the value *domain (ninety (ninety (E)))* of sort *set(index)* from *Domain-Ninety-Sq*. After accepting both inputs, *Two-Partition-Tran-Compose* then generates the value $\{tran(S, ij) : ij \in domain\ (ninety\ (ninety\ (E)))\}$.

3. The process *And* accepts the value $\{tran(S, ij) : ij \in domain\ (ninety\ (ninety\ (E)))\}$ from *Two-Partition-Tran-Compose* and generates the value $and(\{tran(S, ij) : ij \in domain\ (ninety\ (ninety\ (E)))\}$, which equals *ERODE(S,E)*.

Some of the specifications developed in this section are referenced in the next several sections. Specifically, *Two-Partition-Tran-Compose* is used as a major building block for the specification *Dilate*, and *Erode* is used in the construction of the specifications *Open* and *Skel*. A process based specification for the morphological filters *OPEN* and *DILATE* are developed in the following section.

8.4.3 Specification of Open and Dilate. As shown in Figure 8.6, the filter *OPEN* is used in the computation of the skeleton of an image, and as shown in Figure 8.11, *OPEN* is defined using the morphological filters *ERODE* and *DILATE*. The previous section developed a process based specification *Erode* encapsulating the filter *ERODE*. This section develops a specification *Open* encapsulating the filter *OPEN*. Because *OPEN* is defined in terms of *DILATE*, a process based specification *Dilate* encapsulating the filter *DILATE*

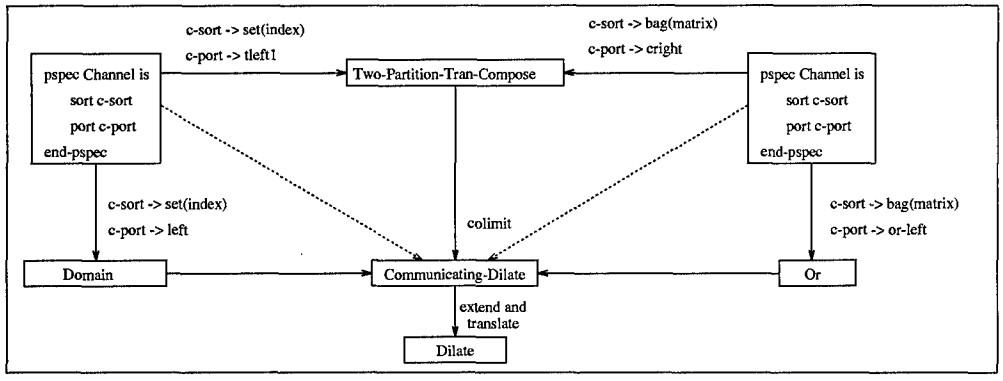


Figure 8.26 Specification for Dilate

is developed in the following paragraphs. After *Dilate* is developed, it is combined with *Erode* to define the specification *Open.*

The equation given in Definition VIII.9 for the morphological filter *DILATE* is $DILATE(S,E) = \bigvee_{(i,j)\in D_S} TRAN(E;i,j)$. As indicated by this equation for *DILATE* and as shown in its block diagram, Figure 8.8, *DILATE* can be specified using multiple, concurrent translate processes, each translating the input image by an index element drawn from the structuring element. The specification *Two-Partition-Tran-Compose* of Figure 8.24 can be reused for this purpose. In addition, the process specifications *Domain-Spec* of Figure 8.16 and *Or* of Figure 8.18 have also already been developed and can be used in the construction of *Dilate.* To define *Dilate* from these three specifications, all that needs to be done is to unify port symbols to define CSP channels and to introduce the process symbol *Dilate* and to define a process expression for *Dilate* as a parallel composition of the processes *Domain* of *Domain-Spec*, the process *Two-Partition-Tran-Compose* of the specification with the same name, and the process *Or.*

Figure 8.26 contains a specification diagram used to construct the process-based specification *Dilate.* As shown in the figure, the output port of the process *Domain* is associated with the set-valued input port of the process *Two-Partition-Tran-Compose* via a simple channel specification. Similarly, the output port of the process *Two-Partition-Tran-Compose* is associated with the input port of the process *Or* via a simple channel specification. The colimit of this diagram, the process specification *Communicating-Dilate*, contains process expressions for *Domain*, *Two-Partition-Tran-Compose*, and *Or* such that

a CSP channel exists from *Domain* to *Two-Partition-Tran-Compose* and a CSP channel exists from *Two-Partition-Tran-Compose* to *Or*.

The colimit object of Figure 8.26 is extended with the process symbol *Dilate* and is translated to clean up the specification by replacing equivalence classes with a representative element. Part of the translation morphism involves renaming the external ports of *Dilate* to *s-dilate*, *e-dilate*, and *out-dilate*, where *s-dilate* is the matrix sorted external port of the process *Two-Partition-Solve-Compose*, *e-dilate* is the set-sorted input port of the process *Domain*, and *out-dilate* is the matrix sorted output port of the process *Or*. Thus for the inputs S and E where S is an image received over *s-dilate* and E is a structuring element received over *e-dilate*, *Dilate* will return the value *or({tran(S;i,j) : (i,j)* $\in$ *domain(E)}*. This value is generated as follows:

1. *Domain* communicates the value *domain(E)* to the process *Two-Partition-Tran-Compose.*
2. The process *Two-Partition-Tran-Compose* iterates over the elements of the set *domain(E)*, computing and accumulating the bag of values *TRAN(S;i,j)* for every *(i,j)* $\in$ *domain(E)*. This bag of values is then communicated to the process *Or.*
3. *Or* accepts the bag of values *{tran(S;i,j) : (i,j)* $\in$ *domain(E)}* and returns the value *or({tran(S;i,j) : (i,j)* $\in$ *domain(E)})*. This is equivalent to the value returned by the filter *DILATE* for image S and structuring element E. This implies that the specification *Dilate* encapsulates the filter *DILATE.*

As is the case with the other morphological filters defined using the specification *Two-Partition-Tran-Compose*, the constant *max* of specification *Dilate* can be given definition through morphism to establish the number of concurrent translate processes. Now that the specification *Dilate* has been developed, it is a simple matter to use it in the construction of a specification for the filter *OPEN.*

As given in Definition VIII.12 and shown in Figure 8.11, *OPEN(S,E)* equals *DILATE (ERODE (S,E), E)*. Note that both Figure 8.11 and the equation *DILATE (ERODE (S,E), E)* indicate that the input E is used in both *ERODE* and *DILATE.* The lack of a broadcast

semantic in CSP implies that the process based specification *Open* must include a connector that provides a broadcast semantic. A specification for such a connector is given below.

pspec Broadcast is
  const number-of-outputs : nat
  sort any
  var the-item : any
  port b-in : any
  port b-out$_i$: any, i = 1..number-of-outputs
  process Broadcast :
    events : {}
    act : {}
    var : {the-item : any}
    chan : {b-in : any, b-out$_i$: any, i = 1..number-of-outputs}
  Broadcast sat b-in?the-item $\xrightarrow{CSP}$ $\|_{i=1..number-of-outputs}$(b-out$_i$ $\xrightarrow{CSP}$ Skip)
end-pspec

Broadcast defines a process that reads a data value over its input channel and communicates that value over a collection of output ports. The sort of the input and output ports and the number of output ports can be defined through refinement of the specification. In the case of *Dilate*, the sort is *matrix* and the constant *number-of-outputs* is the natural number 2. This refined specification will be referred to as *Broadcast-2.*

Connecting the output of *Erode* to the input of *Dilate* can be accomplished through port symbol unification as shown in Figure 8.27. In the figure, a simple channel specification is used to unify the output port of *Erode*, *erode-out*, with the input port *s-dilate* of *Dilate.* The colimit of this diagram, the specification *Erode-to-Dilate*, contains a CSP channel from *Erode* to *Dilate.* The colimit object is then extended with the process symbol *Open* and the process expression *Open sat Erode* || *Dilate.* However, this specification, *Concurrent-Erode-Dilate* does not yet address the issue of broadcasting the input structuring element to both *Dilate* and *Erode.* Therefore *Broadcast-2* is brought into the specification to provide a broadcast semantic. Specifically, one of the two output ports of the specification *Broadcast-2* is associated with the input port *e-erode* of *Erode*, and the other output port of *Broadcast-2* is associated with the input port *e-dilate* of *Dilate.* The colimit of this diagram, the specification *Cluttered-Open*, contains two external input ports, one from *Broadcast-2* and one from *Erode*, and one external output port, *dilate-out.* Equivalence classes of symbols are prevalent in *Cluttered-Open*, and are removed through

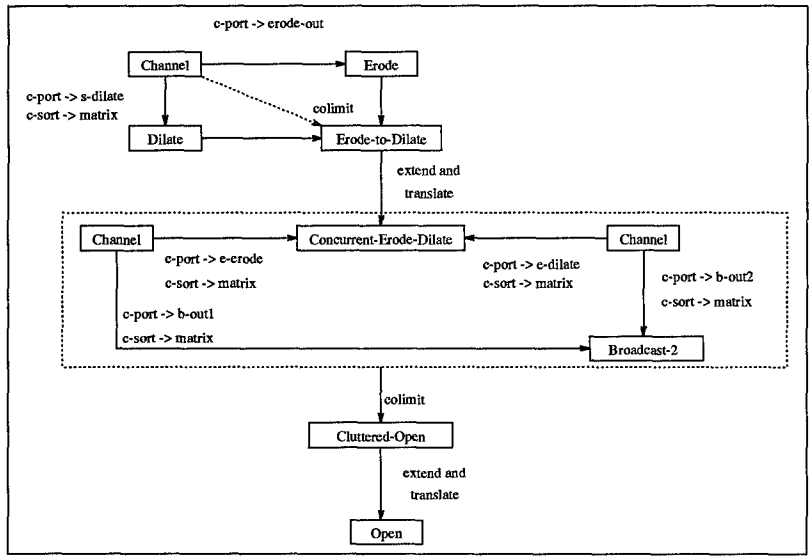


Figure 8.27 Specification for Open

translation. Part of the translation includes renaming the external ports according to the following map: { *b-in* $\mapsto$ *e-open*, *s-erode* $\mapsto$ *s-open*, *dilate-out* $\mapsto$ *open-out*}.

For an image S and a structuring element E, *Open* defines a process that generates a value equivalent to the opening of S by E as follows:

1. E is accepted by *Broadcast-2* which relays it to both *Dilate* and *Erode*.
2. *Erode* accepts the value E from *Broadcast-2* and accepts the value S from the environment. After accepting both of these inputs, *Erode* computes the erosion of S by E and communicates this value to *Dilate*.
3. *Dilate* accepts E from *Broadcast-2* and accepts the value v generated by *Erode*. *Dilate* then returns the value defined by the dilation of v by E.

Now that *Open* is defined, it can be used as a building block for the specification *Skel*.

8.4.4 Specification of Skeleton. In this section, a process based specification for the filter *SKEL* is developed. The specification developed in this section builds on the specifications for the filters *OPEN*, *ERODE*, *OR*, and *AND* presented earlier in this chapter.

The equation given in Definition VIII.13 for the morphological filter *SKEL* is *SKEL(T) = $OR_{\mathcal{D}}$ [AND (COMPLEMENT (OPEN (ERODE (T,D_m), D_2)), ERODE(T,D_m))]*, where T is a constant $(1,*)$ valued matrix and $\mathcal{D} = \{D_i : \exists(i,j)((i,j) \in \text{DOMAIN}(T) \Rightarrow \text{tran}(D_i;i,j) \subseteq T)\}$. Each disk $D_i \in \mathcal{D}$ is a "square disk" of diameter m; see Figure 8.5.

As indicated by both its equation its block diagram (Figure 8.6), the skeleton of an image can be determined using a parallel composition of series of simpler filters. This series, *ERODE, OPEN, COMPLEMENT,* and *AND*, is referred to as *EOCA*. Specifically, *EOCA (T, D_m) = AND (COMPLEMENT (OPEN (ERODE (T,D_m), D_2)), ERODE(T,D_m))*. Using this definition of *EOCA*, *SKEL* can be defined as *SKEL (T) = $OR_{D_m \in \mathcal{D}}$ (EOCA (T, D_m))*. This alternate definition of *SKEL* is used to guide the development of a process based specification *Skel* for this filter.

Development of the specification *Skel* proceeds in three steps:

1. A process based specification *EOCA* encapsulating the filter *EOCA* is developed.
2. An implementation *Two-Solve* $\rightarrow$ *Two-Solve-as-Eoca* $\hookleftarrow$ *EOCA* is developed, which yields a specialization of the specification *Two-Partition-Two-Solve-Compose* such that each *Solve*$_i$ is implemented by *EOCA*.
3. The specification generated by the above action is then extended with the process symbol *Skel* and a process expression defining *Skel* in terms of the other process symbols of the specification.

These steps are described in the following subsections.

8.4.4.1 Development of the Specification EOCA. The specifications *Erode*, *Open*, *Complement*, and *And* have all been developed in the preceding sections and will be used here as building blocks in the construction of the specification *EOCA*.

The specification for the process *And*, shown in Figure 8.18, defines a process that accepts a bag of constant $(1,*)$ valued matrices and generates a matrix defined by the operation *AND* of Definition VIII.8. However, *Erode* generates a single constant valued matrix, as does *Complement*. This indicates that there is a type mismatch between the input sort of the process defined by *And* and the output sorts of the processes defined

by *Erode* and *Complement.* This disparity can be overcome by either extending *And* or by defining a *merge* connector. Specifically, *And* can be extended through specification morphism with the operation *and-two : matrix, matrix* $\rightarrow$ *matrix*, the variables *m1* and *m2* of sort matrix, and the ports *and-two-1 : matrix* and *and-two-2 : matrix.* The process expression for *And* can then be extended with the nondeterministic choice of *and-two-1?m1* $\stackrel{CSP}{\longrightarrow}$ *(and-two-2?m2* $\stackrel{CSP}{\longrightarrow}$ *(and-right!and-two(m1,m2)* $\stackrel{CSP}{\longrightarrow}$ *Skip)).* However, the approach taken here is to define a connector *Merge* as given below.

pspec Merge is
  sort any, set(any)
  var m_i : any, i=1..fan-in
  var accum : set(any)
  const fan-in : nat
  port merge-in_i : any, i=1..fan-in
  port merge-out : any
  process Merge is
    events : {}
    act : {}
    chan : {merge-in_i : any, i=1..fan-in, merge-out : any}
    var : {m : any, accum : set(any)}
  Merge sat
    ($\|_{i \in \{1..fan-in\}}$ merge-in_i?m_i $\stackrel{CSP}{\longrightarrow}$ accum := accum with m; Skip);
    (merge-out!accum $\stackrel{CSP}{\longrightarrow}$ Skip)
end-pspec

Merge defines a process that accepts exactly *fan-in* elements of sort *any* and then outputs a bag containing those values. *Merge* can be refined by providing definition to the sort *any* and by specifying the constant *fan-in.* For the specification *EOCA*, *Merge* can be refined through specification morphism by the following map: {*any* $\mapsto$ *matrix, fan-in* $\mapsto$ *2*}. The resulting specification, *Merge-2*, is used in the construction of *EOCA.*

A specification diagram for *EOCA* is shown in Figure 8.28. The arrows in the figure denote morphisms. Channel specifications are used to unify port symbols and thereby define CSP channels. Specifically, the output port of *Erode*, *erode-out*, is associated with the input port of the broadcast specification *Broadcast-2.* One of the output ports of *Broadcast-2* is associated with the port *s-open* of *Open*, and the other output port of *Broadcast-2* is associated with one of the input ports of *Merge-2.* The other input port of *Open*, *e-open*, is associated with the simple process *DTwo. DTwo* specifies a process

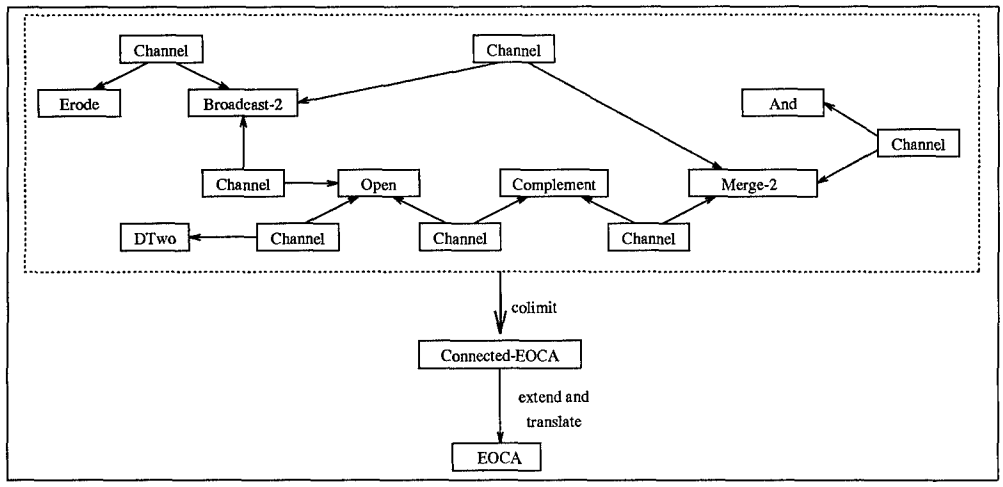


Figure 8.28 Specification for EOCA

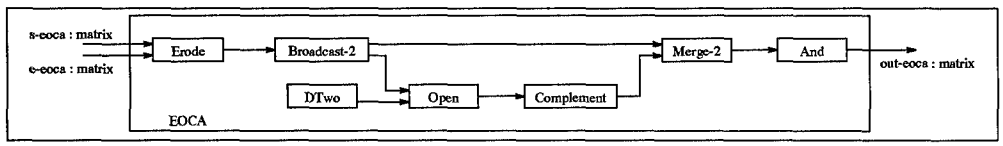


Figure 8.29 Communication Network for EOCA

that outputs the matrix D_2. Thus the *Open* in the colimit specification outputs the value *OPEN (ERODE (S,E), D_2)* for the values S and E accepted by $Erode_{mod}$. Furthermore, the output port of *Open* is associated with the input port of *Complement*, and the output port of *Complement* is associated with the other input port of *Merge-2*. The output port of *Merge-2* is associated with the input port of *And*. The colimit object of this diagram, the specification *Connected-EOCA* defines a collection of processes having the communication network shown in Figure 8.29, where the arrows in the figure denote communication channels.

The colimit object of Figure 8.28, *Connected-EOCA*, is extended through specification morphism with the process symbol *EOCA* and with the process expression *EOCA sat Erode || Open || Complement || And || Broadcast-2 || Merge-2*. The resulting process specification has four external ports; three are input ports and one is an output port. As shown in Figure 8.29, two of the input ports are the input ports of *Erode*. The third input port is the port *e-open* of *Open*. This input port of open *Open*, as shown in Figure 8.6, will be connected to the output port of a simple process that simply outputs the constant

matrix D_2. The external ports of *EOCA* can be renamed through the map {*e-erode* $\mapsto$ *e-eoca, s-erode* $\mapsto$ *s-eoca, e-open* $\mapsto$ *e-open, and-right* $\mapsto$ *out-eoca*}.

Although more efficient specifications are possible, this specification of *EOCA* is sufficient for purposes of demonstration.

8.4.4.2 Implementing Two-Solve *as* EOCA. Defining an implementation of *Two-Solve* as *EOCA* is straightforward. Specifically, two morphisms need to be defined, *Two-Solve* $\rightarrow$ *Solve-as-EOCA* and *Solve-as-EOCA* $\hookleftarrow$ *EOCA*. These two morphisms can be defined as follows.

1. Define the process specification *Solve-as-EOCA* to be the specification *EOCA* extended with the event *done* such that *done* is added to the alphabet of the process *EOCA*. Furthermore, extend the process expression for the process symbol *EOCA* to include the nondeterministic choice of *done* $\stackrel{CSP}{\longrightarrow}$ *Skip*. That is, the extended process expression for *EOCA* is *(done* $\stackrel{CSP}{\longrightarrow}$ *Skip)* $\sqcap$ *(Erode* $\|$ *Broadcast* $\|$ *DTwo* $\|$ *Open* $\|$ *Complement* $\|$ *Merge-2* $\|$ *And)*. Then the traces of the process *EOCA* are contained within the traces of this extended process expression because *traces(P* $\sqcap$ *Q)* = *traces(P)* $\cup$ *traces(Q)* for any processes P and Q. That is, denoting the extended process expression by *Solve-as-Eoca*, *traces(EOCA)* $\subseteq$ *traces(Solve-as-Eoca)*$\restriction \alpha$*EOCA*. Clearly, *Solve-as-EOCA* $\hookleftarrow$ *EOCA*.
2. Define *Two-Solve* $\rightarrow$ *Solve-as-EOCA* by the map { *Two-Solve* $\mapsto$ *Solve-as-EOCA, s* $\mapsto$ *matrix, any* $\mapsto$ *matrix, any2* $\mapsto$ *matrix, p2s* $\mapsto$ *e-eoca, p2s2* $\mapsto$ *s-eoca, s2c* $\mapsto$ *eoca-out*}. Because *traces(Two-Solve)* = {$\langle\rangle$, $\langle$*done*$\rangle$} $\subseteq$ *traces(Solve-as-EOCA)*$\restriction \alpha$*Two-Solve*, Two-Solve $\rightarrow$ Solve-as-EOCA is a specification morphism.

Because *Two-Solve* can be extended through process specification morphism to define the specification *Two-Partition-Two-Solve-Compose*, and because process specification morphisms compose to form process specification morphisms, the implementation of *Two-Solve* by *EOCA* can be extended through process specification morphism to define the specification *Two-Partition-EOCA-Compose*. This extension is shown in the diagram of Figure 8.32. The arrows in the figure denote specification morphisms, with e denoting an extension, m denoting a morphism, and d denoting an extension by definition.

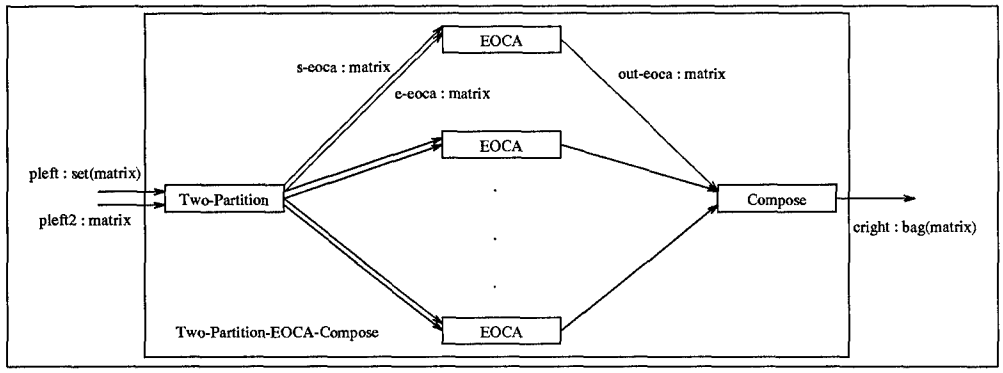


Figure 8.30 Communication Network for Two-Partition-EOCA-Compose

As shown in Figure 8.30, *Two-Partition-EOCA-Compose* defines a collection of processes uses *EOCA* to find solutions to input data values. The arrows in the diagram denote communication channels. *Two-Partition-EOCA-Compose* has three external ports, *pleft*, *pleft2*, and *cright*. The sorts of the three external ports have been refined based on the implementation of *Solve* by *EOCA*. Specifically, because *Two-Solve* $\rightarrow$ *Solve-as-EOCA* is defined by the map { *Two-Solve* $\mapsto$ *Solve-as-EOCA*, *s* $\mapsto$ *matrix*, *any* $\mapsto$ *matrix*, *any2* $\mapsto$ *matrix*, *p2s* $\mapsto$ *e-eoca*, *p2s2* $\mapsto$ *s-eoca*, *s2c* $\mapsto$ *eoca-out*}, and because *Two-Partition-Two-Solve-Compose* is defined as an extension to *Two-Solve*, the colimit of the upper portion of the diagram of Figure 8.32 yields the unification of the sort symbols *any*, *any2* and *s* with *matrix* in the colimit object.

Inputs to *Two-Partition-Two-Solve-Compose* are accepted by *Two-Partition* which is contained within *Two-Partition-EOCA-Compose*. After accepting the inputs, *Two-Partition* enumerates over the input set of matrices, communicating both the single matrix accepted over the port *pleft2* and a matrix value from the input set of matrices to one of the *EOCA* processes. Subprocess *Compose* of *Two-Partition-EOCA-Compose* collects the values generated by the individual *EOCA* processes, and when all output values have been generated, *Compose* is defined to output the bag of values it has accumulated.

The block diagram for *Skel*, Figure 8.6, gives an indication of how the external ports of *Two-Partition-EOCA-Compose* will be used. As shown in the figure, the input image T is replicated and communicated to m parallel *EOCA* processes. Definition VIII.13 indicates that m is the largest natural number such that $TRAN(D_m; i, j) \subseteq T$ for *(i,j)* $\in$

DOMAIN(T). This implies that the set $\mathcal{D}$ should be computed after *Skel* accepts the image T. This set of square disks will be the set valued entity communicated to *Partition* over the port *pleft*.

Two-Partition-EOCA-Compose is the largest building block used in the construction of the specification *Skel*. A diagram used to construct the specification *Skel* is developed in the following subsection.

8.4.4.3 The Specification Skel. In the introduction to this section, the skeleton of an image T was defined by the equation *SKEL (T)* = $OR_{D_m \in \mathcal{D}}$ *(EOCA (T, D_m))*, where $\mathcal{D} = \{D_m : \exists(i,j)((i,j) \in \text{DOMAIN}(T) \Rightarrow \text{tran}(D_m; i, j) \subset T\}$. This implies that the skeleton of T can be computed by taking the disjunction of the set of values $\{EOCA(T,D_i) : D_i \in \mathcal{D}\}$. In other words, the skeleton of an image T can be determined by

1. Defining the set $\mathcal{D} = \{D_i : \exists(i,j)((i,j) \in \text{DOMAIN}(T) \Rightarrow \text{tran}(D_i; i, j) \subseteq T)\}$. That is, $\mathcal{D}$ is the maximal set of square disks such that any disk D_i in $\mathcal{D}$ can be translated such that it is a sub-image of T.
2. Enumerating over the set $\mathcal{D}$ to compute the set of values *and(complement (open (erode (T, D_i)), D_2), erode (T, D_i))* such that $D_i \in \mathcal{D}$.
3. Taking the disjunction of the set values generated in the above step. The resulting value, *or($\{$and(complement (open (erode (T, D_i)), D_2), erode (T, D_i)) : $D_i \in \mathcal{D}\}$)*, equals *SKEL(T)*.

The process specification *Skel* developed in this section is organized around these three steps. Specifically, *Skel* has three main subprocess, *DSet* which is used to compute the set $\mathcal{D}$, *Two-Partition-EOCA-Compose* used to compute the set of values $\{EOCA(T,D_i) : D_i \in \mathcal{D}\}$ for an image T, and *Or* which forms the disjunction of the values generated by *Two-Partition-EOCA-Compose*. Both *Or* and *Two-Partition-EOCA-Compose* have already been developed in preceding sections of this chapter. The specification *DSet* is developed in the following paragraphs. After *DSet* is developed, it is used in the definition of a specification diagram for *Skel*.

```
pspec DSet is
        sorts matrix, set(matrix)
        var t : matrix
        port left-DSet : matrix
        port right-DSet : set(matrix)
        op get-dsets : matrix → set(matrix)
        process DSet : events : {}
                act : {op get-dsets : matrix → set(matrix)}
                chan : {left-DSet : matrix, right-DSet : set(matrix)}
                var : {t : matrix}
        DSet sat left-dset?t ⟶(CSP) right-dset!get-dsets(t)
        end-pspec
```

Figure 8.31 Specification for DSet

DSet specifies a process consisting of two ports, an input port *left-dset* of sort *matrix* and an output port of sort *set(matrix)*, and defines a process that accepts a matrix t and produces the set $\{D_i \ : \ \exists(i,j)((i,j) \in \text{DOMAIN}(t) \Rightarrow \text{TRAN}(D_i; i, j) \subseteq t)\}$. There are many interesting approaches for defining *DSet*, including one based on a layered architecture where *Tran* and *Domain* are subordinate to *DSet*. Another approach would be to define a subordinate partition-solve-compose structure. In any case, the form of the specification of *DSet* is not that critical; any specification for *DSet* that returns the requisite set of values for an input image t is acceptable. Therefore, an operation *get-dsets : matrix → set(matrix)* contained in *DSet* is assumed to exist. This operation has the functional axiom $\forall$ *(T : matrix) (get-disks(T) =* D_i *: ∃(i,j) ((i,j) ∈ DOMAIN(t) ⇒ TRAN(*D_i*;i,j) ⊆ t)})*. The specification *DSet* then takes the simple form shown in Figure 8.31.

Both *DSet* and *Two-Partition-EOCA-Compose* of *Skel* require access to the input variable T. Rather than accepting the input twice, the specification *Broadcast* can be refined through specification morphism to accept values of sort *matrix* and relay them to both *DSet* and *Two-Partition-EOCA-Compose*. The specification *Broadcast-2* can be used for this purpose. Thus the specification *Skel* can be composed from the specifications *DSet*, *Broadcast-2*, *Two-Partition-EOCA-Compose*, and *Or*. A specification diagram composing these specifications to define *Skel* is shown in Figure 8.32.

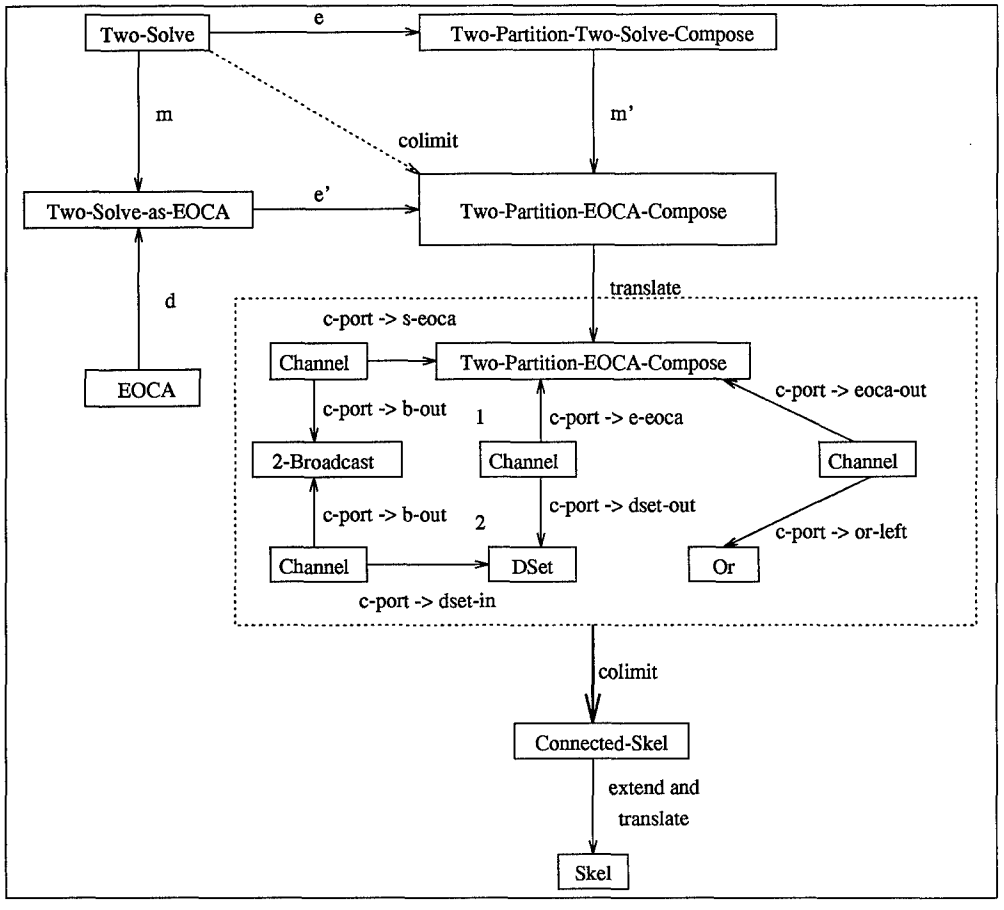


Figure 8.32 Specification for Skeleton

In the portion of Figure 8.32 enclosed by the dashed box, one of the output ports of *Broadcast-2* is associated with the input port of *DSet*, and and other output port of *Broadcast-2* is associated with the matrix-sorted input port *pleft2* of *Two-Partition-EOCA-Compose* (recall that any value received over the port *pleft2* of *Two-Partition* is communicated to each *Solve* process along with an element from the set of values received by *Two-Partition* over the port *pleft*.) Similarly, the output port of *DSet* is associated with the port *pleft* of sort *set(matrix)* of the process *Two-Partition*. Finally, the output port of the subprocess *Compose* is associated with the input port of *Or*. Thus the colimit object, the specification *Connected-Skel*, defines a collection of processes containing two external ports, both of which are of sort matrix. The input port of *Connected-Skel* is the input port *b-in* of *Broadcast-2*, and the output port of *Skel* is the output port of *Or*.

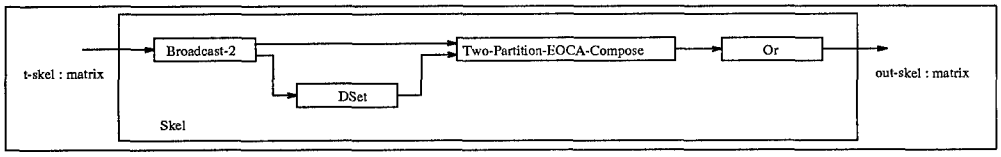


Figure 8.33 Process Communication in Skeleton

The colimit object of the boxed in portion of Figure 8.32 defines a collection of communicating processes, but the specification does not yet define — beyond the communication network defined by the CSP channels between the processes — how the processes it defines interact. Therefore the colimit object is extended with the process symbol *Skel* and the process expression *Skel sat Broadcast-2* || *DSet* || *Two-Partition-EOCA-Compose* || *Or*. The external ports of *Skel* can also be renamed; specifically, *b-in* is renamed to *t-skel* and *or-right* is renamed to *out-skel*. After renaming, the specification *Skel* defines a collection of processes having the communication network shown in Figure 8.33. The arrows in Figure 8.33 represent CSP channels.

Skel generates the skeleton of an input image t as follows. *Broadcast-2* accepts the input t and relays it to both *DSet* and *Two-Partition-EOCA-Compose*. *DSet* accepts t and computes the set $\mathcal{D} = \{D_i : \exists\ (i,j)\ ((i,j) \in DOMAIN(t) \Rightarrow TRAN(D_i;\ i,j) \subseteq t)\}$. This value is communicated to *Two-Partition-EOCA-Compose*, which then generates the set $\{EOCA(t,\ D_i) : D_i \in \mathcal{D}\}$. This set is communicated to *Or*, which forms the disjunction of the elements of the set. That is, *Or* accepts the set of values from *Two-Partition-EOCA-Compose* and outputs the value $OR(\{EOCA(t,\ D_i) : D_i \in \mathcal{D}\})$, which equals $SKEL(t)$.

Now that the specification *Skel* has been defined, it can be used to define the feature selection stage of an image recognition application.

8.4.5 Using Skeleton for Feature Selection. In the previous subsections, a process based specification for the feature selection operation *skeleton* was developed. In this section, the skeleton process specification, *Skel*, is associated with the feature selection stage of the image recognition design shown in Figure 8.1. This association will result in the *skeleton* operation being used for feature selection.

The process specification for *Selection*, shown in Figure 8.1, simply defines a process that has two ports, *left : msg* and *right : msg*. Although there is no process expression associated with the process symbol *Selection*, the process expression for *ImageRec*, *Pipe* || *(Creation; Restoration; Enhancement; Segmentation; Selection; Registration; Classification)*, where *Pipe* shares the port symbols *left* and *right* with the sequentially composed processes, indicates that one of the ports of *Segmentation* is an input port and the other port is an output port.

The specification *Skel* also defines a process that has two external ports, one used for input and one used for output. In addition, the input and output sorts of *Skel* — like the input and output sorts of *Selection* — are identical, so mapping the sort *msg* to the sort *matrix* is a consistent with respect to the process signatures of *Skel* and *Selection*. That is, the interface defined by *Skel* is compatible with the interface defined by *Selection*. (Actually, the interface of *Sel* is isomorphic to the interface of *Selection*.)

Associating *Skel* with *Selection* is straightforward, and can be accomplished either by defining an implementation *Selection* $\rightarrow$ *Selection-as-Skel* $\hookleftarrow$ *Skel*, or through explicit unification *Selection* and *Skel* through the the colimit of the diagram defined by $T \rightarrow$ *Skel* and $T \rightarrow$ *Selection*, where T is a specification containing only a single process symbol whose alphabet includes exactly two ports. In this case, the two approaches are equivalent. That is, $T \cong$ *Selection*. Thus the approach taken here is to define an implementation of *Selection* by *Skel* as follows. Define the specification *Selection-as-Skel* to be a copy of the specification *Skel*. Then

1. *Selection* $\rightarrow$ *Selection-as-Skel* is defined by the mapping $\{msg \rightarrow matrix,\ left \mapsto t\text{-}skel,\ right \mapsto out\text{-}skel,\ Selection \mapsto Skel\}$. Because *traces(Selection)* $= \{\langle\rangle\}$, and because $\{\langle\rangle\}$ is a subset of the set of traces of any CSP process, *Selection* $\rightarrow$ *Selection-as-Skel* is a specification morphism.
2. *Selection-as-Skel* $\hookleftarrow$ *Skel* is trivially defined to be the identity specification morphism.

The implementation of *Selection* by *Skel* is shown in Figure 8.34. Also shown in the figure is the extension of *Selection* to the specification *Image-Rec-System*. Because *Selection* can be extended to *Image-Rec-System* and because *Selection* can be mapped un-

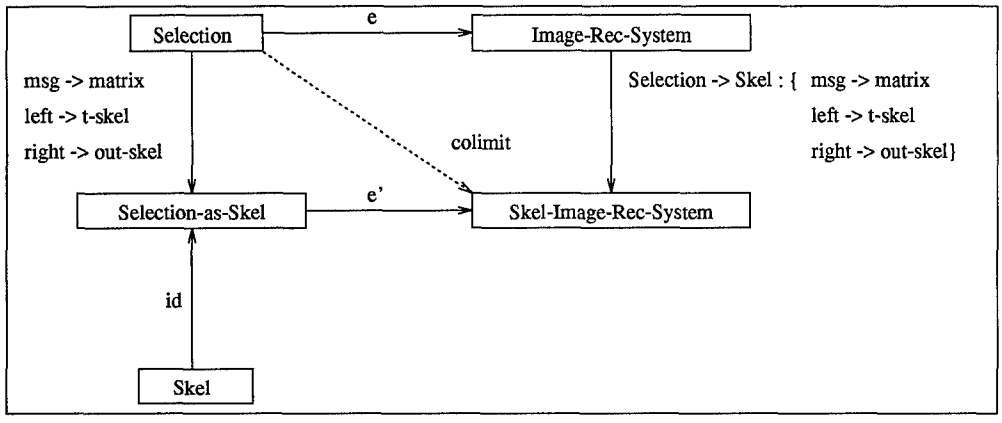


Figure 8.34 Selection as Skeleton in Image Recognition

der specification morphism to *Selection-as-Skel*, *Image-Rec-System* can be mapped under specification morphism to the specification *Skel-Image-Rec-System.* Specifically, the morphism *Image-Rec* $\rightarrow$ *Skel-Image-Rec-System* is defined by the map $\{msg \rightarrow matrix,\ left \mapsto t\text{-}skel,\ right \mapsto out\text{-}skel,\ Selection \mapsto Skel\}$ where the rest of the symbols in *Image-Rec* are mapped under the identity morphism.

The specification *Skel-Image-Rec-System* defines a piped-batch sequential design in which one of the sequentially composed filter processes, *Selection*, is implemented by the specification *Skel.*

8.5 Summary

The construction of a process based specification for the feature selection portion of an image recognition system was carried out in this chapter, where smaller process based specifications were defined and combined through process specification morphism to define an application-level specification. This specification development effort resulted in the formation of several reusable, domain independent designs. These reusable designs were specialized under specification morphism to define specifications for various subproblems. The definition, development, and use of these reusable designs provides at least some empirical validation for the notion of a library of reusable design elements described in Chapter II. In addition, the utility of architectural structuring specifications in the development of application specifications was demonstrated through their use in the con-

struction of the top level image recognition specification, *Image-Rec-System*, and through their use in defining subproblem structure.

Although not investigated as part of the feasibility demonstration, it should be possible to use architecture theories to guide the development of application level specifications. That is, rather than use architecture theories to compose application level specifications from the bottom up, as was done in this chapter, it should be possible to use architecture theories to guide the decomposition of a problem into smaller, more manageable "mind-sized" chunks. Additional research on this and other application level specification issues can be supported by the theoretical foundations established in the preceding chapters.

IX. Conclusions and Recommendations

The purpose of this investigation was to establish a formal foundation for software architecture which allows for the specification of large, non-trivial software systems using well founded, consistency preserving construction techniques. Based on this, two fundamental problems were addressed. First, how to define and express architectures formally using the concept of theories, and second, how architecture theories could be practically applied in specification construction.

The initial stages of this investigation sought to establish a formal, mathematical relationship between functional specifications of behavior and specifications defining system structure. Two experiments were defined and executed, and their results lead to the conclusions that an architecture defining the structure of functional operations could be defined within a functional logic, but more complex architectures, such as those involving collections of communicating processes, require a separate process logic. Based on these experimental results, a process logic based on Hoare's Communicating Sequential Processes (CSP) was presented and used in the definition of a process-based specification development system. Specifically, CSP was used in the definition of a category of process-based specifications and specification morphisms. CSP structures were introduced and defined to provide a trace semantic for process expressions within this category. Architecture theories expressed in terms of both functional specifications and process-based specifications were then defined, and relationships between these architecture theories were investigated. A feasibility analysis demonstrated that process-based specifications and architecture theories could be used to develop specifications for large, non-trivial applications.

A category of process specifications and process specification morphisms exists, and process-based architecture theories can be defined within this category. The category-based operations of importation, translation, product, coproduct, and colimit, among others, can be used to define consistency preserving process-based specification construction operations. In addition, the semantics of sorts and functional operators referenced within process expressions of process specifications can be given definition through associated functional specifications. Components are used to make this association. Finally, it was shown that components and component morphisms form a category, and it was shown how

component-based architecture theories could be defined and used in the construction of large, non-trivial specifications.

9.1 Conclusions and Results

In addition to the broad conclusions stated in the above paragraphs, several specific conclusions and results can be stated.

1. The category **PSpec** of process specifications and process specification morphisms proved to be effective in the construction of process specifications. In addition, definition of the satisfaction relation $\models$ could be extended to include a semantic more powerful than the trace semantic used in the definition of **PSpec**. The major benefit of defining a category of process specifications is that process specifications can be used to define state, communication, and processes, and consistency preserving specification construction operations such as colimits can be defined with the category.

2. The relationship defined between process-based specifications and functional specifications provides a means to develop system level specifications using logic-appropriate techniques. Sorts and functional operators can be defined using functional specifications, while these same sorts and functional operators can be referenced in a process specification defining communication, state, and process. Furthermore, a category **App** of related functional and process-based specifications was defined. Such a category permits application level specifications to be grown using consistency preserving construction techniques.

3. Functional architecture theories, process-based architecture theories, and component-based architecture theories were defined. Several process-based architecture theories, including layered, pipelined, and repository architectures were defined. Furthermore, it was shown how architecture theories can either be used in a bottom-up manner to define structure in terms of other, simpler structures, or used in a top-down manner to decompose an element into a structured collection of simpler elements.

4. A semantic for comparing process-based architecture theories was defined and used to establish a hierarchy of process-based architecture theories. It was shown, for

example, that any well-formed pipeline design can be translated under specification morphism to a layered design. The semantic developed is weaker than the trace semantic used to define process specification morphisms in that it is only concerned with sequences of communication events between a process and its environment; all internal communication and all non-communication events are ignored.

5. The utility of architecture theories was demonstrated through the development of a process-based specification for a segment of an image processing application, where pipeline, batch-sequential, piped-batch-sequential, and general process-based architecture theories were used during the development.

9.2 Future Work

The mathematical foundations established as part of this investigation permit exploration of issues associated with the specification of software architecture. However, the results are not complete and should be extended through further analysis and definition of the mathematical framework. Several areas requiring further work have been identified; a summary of these areas is presented below.

1. A formal definition of a grammar and domain model for the ISlang specification language should be developed. After defining the domain model and grammar for the language, the Composition Mechanism (CM) can be implemented. Implementing the CM requires elaboration of a deductive system for process expressions. Some work has been done in this area, e.g., (8) and (51).

2. Safety and liveness issues and their decidability within a process logic should be further defined. Some work on this issue, including identification of CSP constructs leading to finite state automata,(36) has been done.

3. Constraint representation and use needs further elaboration. It should be possible to use constraint information to complete partially defined morphisms. For example, sort compatibility can be used to complete some simple morphisms. It should be possible to extend this concept to use derived antecedents, for example, to complete more complex morphisms such as those defining architecture interpretations.

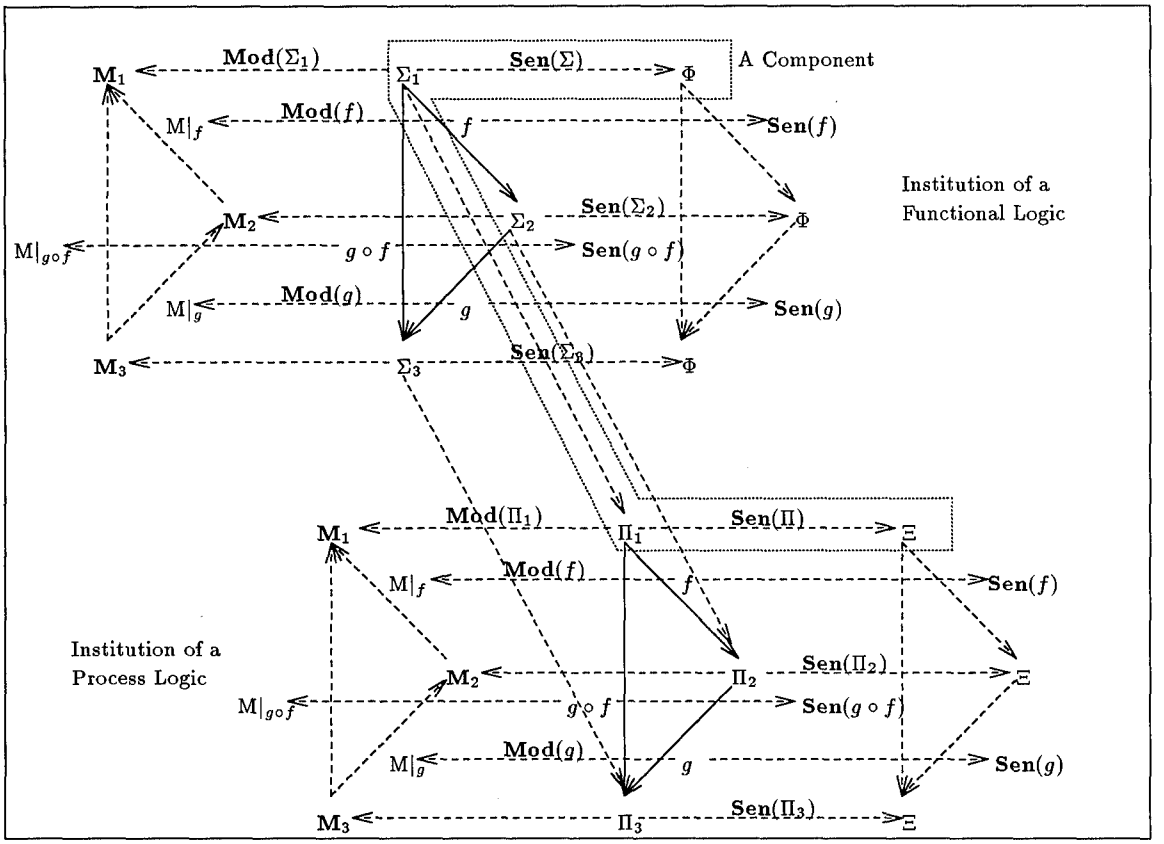


Figure 9.1 Formalizing the Relationship Between Functional and Process Institutions

Goguen suggests that constraints could be represented as a well-formed expression in some logic where the values of variables referenced in the expression are represented as morphisms to the constrained elements of the affected specifications. Another Ph.D. candidate at the Air Force Institute of Technology, Frank Young, is actively researching this topic.(129)

4. The mechanism used to map process specifications to models needs to be defined. Before this can be accomplished, a more rigorous definition and characterization of models of process specifications is required. Some work has been done in this area. See, for example, (50).

5. A generalization effort could be undertaken to provide an algebraic unification of processes and functional operators. Specifically, it may be possible to associate a process with a functional operator so that operators such as $f : u, v \rightarrow w$ can be defined as a collection of cooperating processes.

6. The relationship between functional specifications and process specifications needs to be further formalized. Figure 9.1 depicts the relationships between process specifications and functional specifications developed during this investigation. Specifically, a component is defined by the dotted box in the figure, where the dashed arrow from Σ-signatures to Π-signatures represents the functor i of a component. Missing from the figure is the relationship between models of the two institutions. Elaboration of this relationship would first require the elaboration of the process logic institution. Although the figure depicts an institution of process logic, neither the category **Mod[pSP]** of process specification models nor the functor *Mod* from Π signatures to Π-models were formally defined as part of this investigation.

7. Finally, an appropriate generalization effort could be undertaken to partition specification construction using architecture theories into two distinct aspects: a problem specific aspect of classification, and a problem independent aspect of solution/synthesis. The problem dependent aspect of specification development is concerned with defining interpretations from successively more refined architectures to the specified problem. The problem independent aspect of specification construction is concerned with defining interpretations from the architecture of the target platform to the architecture theories used to structure the problem.

   Figure 9.2 conceptually represents specification construction using this approach. Abstractly shown in the figure is a *Basic System Specification* from which a definition of a specific problem such as *Skeleton* can be defined. Basic System Specification is also used to characterize the target platform, in this case an n-cube. A morphism from Basic Problem Theory to *Partition-Solve-Compose* (PSC) can be used to define an interpretation from PSC to *Skeleton* and from n-cube to PSC. Interpretations are shown in Figure 9.2 as a bold arrow. The specification *n-cube-as-PSC* defines how an n-cube can be used for the PSC architecture. Note that the construction of an interpretation from n-cube to PSC is problem independent. The specification *PSC-as-Skeleton* defines how PSC can be used to define solutions to the problem *Skeleton.*

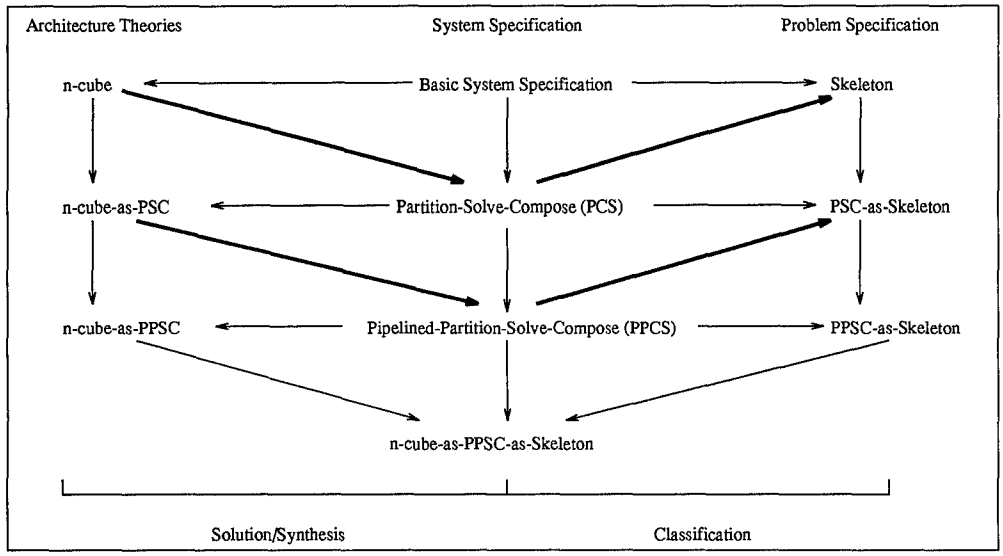


Figure 9.2 Using Architecture Theories

An additional level of refinement is shown in the figure. Specifically, a pipeline architecture theory (not shown as part of the figure) has been used to define *Pipelined-Partition-Solve-Compose* (PPSC). The colimit of the resulting diagram, the specification *n-cube-as-PPSC-as-Skeleton* defines how solutions to the problem *Skeleton* can be found using PPSC on an n-cube.

The architecture of target platforms could be represented using appropriate architecture theories. Additional process-based architecture theories such as wavefront arrays, systolic arrays, pyramids, or cubes (27) could be defined and used to characterize target platforms. Interpretations from the structures defined in the center column of Figure 9.2, such as PSC, to the architecture of the target platform are problem independent. Once such an interpretation has been defined, it becomes a piece of reusable knowledge that can be stored in the library of architecture specifications shown in Figure 2.1.

The center column of Figure 9.2 could be defined using the relationships between process-based architecture theories developed in Chapter VII. For example, Chapter VII demonstrated how pipeline designs could be refined as layered designs. The existence of specification morphisms involving designs of other architecture theories was demonstrated, but these morphisms were not defined. Development of the center

column of the figure will require further elaboration of the relationships between the various process-based architecture theories. These relationships could then be used exploit the characteristics of the problem to facilitate the definition of an interpretation from the architecture of the target platform to the architecture inherent in the given problem.

9.3 Summary

Based on the results and conclusions reported in this and previous chapters, it can be concluded that this investigation successfully met its objectives. Solid mathematical foundations for software architectures were established. Specifically, several categories of specifications and specification morphisms were defined wherein software architectures could be formally defined and used in the construction of specifications for large, non-trivial software systems, and relationships between various process-based architecture theories were investigated and formalized. An additional benefit of this investigation was its identification of additional areas of research and analysis in several other areas of specification development and implementation.

Appendix A. Category Theory

A.1 Initial and Terminal Objects

An interesting aspect of colimit objects is their initiality.

Definition A.1 Initial. *A C-object c is initial in a category C if and only if there exists a unique arrow from c to any other C-object in C.* □

A dual to initiality is the following:

Definition A.2 Terminal. *A c-object c is terminal in a category C if and only if there exists a unique arrow from any other C-object in C to c.* □

The definition of initial objects coupled with the definition of pushouts and colimits leads to the following theorem.

Theorem A.1 Initiality of Colimit Objects. *The colimit of a diagram D of a category is an initial object in a category whose C-objects extend D.*

Proof. *The proof follows from the definition of colimit and initial object.*

Denote by L the colimit of a diagram D. Let c be any other C-object extending D. Then there exists a cone from D to c. By the definition of colimit, then there exists a unique arrow from L to c. Since c was an arbitrary object extending D, we get that L is initial in the category of objects that extend D. ■

This theorem can be extended to pushouts as well:

Corollary A.1 Initiality of pushout objects. *Given a diagram D consisting of a pair of C-arrows* $a \xleftarrow{f} c \xrightarrow{g} b$ *with a common domain, the pushout of D is initial in a category whose C-objects extend D.*

Proof. *Follows from Theorem A.1.* ■

A.2 Homomorphisms

Figure A.1 depicts the concept of a homomorphism. The following paragraphs highlight this concept.

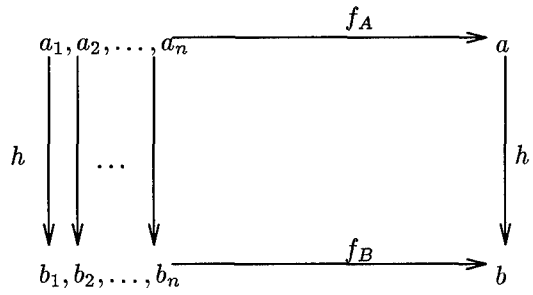


Figure A.1 Homomorphism.

The equality in the above definition can be used to determine if a proposed function between two Σ-algebras is a homomorphism. For example, consider the following signature Σ:

```
sign Σ is
      sorts N
      op    plus : N, N − > N
      op    succ : N    − > N
      op    pred : N    − > N
      op    zero :      − > N
end-sign
```

Then $\langle N, \Sigma_N \rangle$ is a Σ-algebra where

- N is the set of natural numbers,
- Σ_N is given by
  - $zero_N$ is the number 0;
  - $succ_N$ $N \rightarrow N$ is defined by $succ_N(n) = n + 1$;
  - $pred_N$ $N \rightarrow N$ is defined by $pred_N(0) = 0$ and $pred_N(n+1) = n$;
  - $plus_N$ $N \times N \rightarrow N$ is defined by $plus_N(n,m) = n + m$ where + denotes the usual arithmetic plus.

Denote by N_2 the set of even natural numbers. Then $\langle N_2, \Sigma_N \rangle$ is also a Σ-algebra where

- N_2 is the set of even natural numbers (i.e., $N_2 = \{m \mid m = n + n, n \in N\}$);
- Σ_N is given by

- $zero_{N_2}$ is the number 0;
- $succ_{N_2}$ $N \rightarrow N$ is defined by $succ_{N_2}(n) = n + 2$;
- $pred_{N_2}$ $N \rightarrow N$ is defined by $pred_{N_2}(0) = 0$ and $pred_{N_2}(n+2) = n$;
- $plus_{N_2}$ $N \times N \rightarrow N$ is defined by $plus_{N_2}(n,m) = n + m$ where + denotes the usual arithmetic plus.

Then $in : N \rightarrow N_2$ defined by $in(n) = 2n$ is a homomorphism:

- $in(succ_N(n)) = in(n+1) = 2n+2$ and $succ_{N_2}(in(n)) = succ_{N_2}(2n) = 2n+2$;
- $in(plus_N(n,m)) = in(n+m) = 2(n+m) = 2n+2m$, and $plus_{N_2}$ $(in(n), in(m)) = plus_{N_2}$ $(2n, 2m) = 2n+2m$;
- for $n \neq 0$, $in(pred_N(n)) = in(n-1) = 2n-2$ and $pred_{N_2}(in(n)) = pred_{N_2}(2n) = 2n-2$; for $n = 0$, $pred_N(in(0)) = 0$ and $pred_{N_2}(in(0)) = pred_{n_2}(0) = 0$.

A.3 Types of Morphism

Definition A.3 Types of Morphisms. *A Σ-homomorphism h is called*

- *an isomorphism if h is a bijection;*
- *an epimorphism if h is a surjection; and*
- *a monomorphism if h is an injection.*

If h takes an algebra back to itself, then h is called an endomorphism, and if it's bijective, an automorpism. □

We will use the symbol $\cong_\Sigma$ to denote isomorphism. That is, $A \cong B$ if and only if there exists an isomorphism between the Σ-algebras A and B. If the signature Σ is clear based on the context of the expression, the Σ subscript will be dropped.

Lemma A.1 *Identity function. The identity function* id *taking each sort and operation symbol onto itself is an isomorphism.*

Proof. *The proof is straight-forward. We first show that* id *is a homomorphism. Let A be a Σ-algebra. Then for all operation symbols $f : s_1 \times s_2 \times \cdots \times s_n \rightarrow s$ in Σ and for all $a_1 \in A_{s_1}, a_2 \in A_{s_2}, \ldots, a_n \in A_{s_n}$, we have*

$id(f_A(a_1, a_2, \ldots, a_n)) = f_A(a_1, a_2, \ldots, a_n)$ *and*

$f_B(id_{s_1}(a_1), id_{s_2}(a_2), \ldots, id_{s_n}(a_n)) = f_A(a_1, a_2, \ldots, a_n)$.

It is trivial to show that id is bijective:

- $\forall(a)(id(a) = a)$ *by definition of id,*
- $\forall(a, b)(id(a) = id(b) \Rightarrow a = b)$ *by definition of id.*

Therefore id *is a bijective homomorphism.* ■

Theorem A.2 Equivalence relation. *The relation $\cong$ between Σ algebras is an equivalence relation.*

Proof. We'll prove the theorem by showing $\cong$ is reflexive, symmetric, and transitive. In the following, let A, B, and C be Σ-algebras.

- *Reflexive. $A \cong A$ follows from Lemma A.1.*
- *Symmetric. Let $A \cong B$. Then there exists a bijective homomorphism h from A onto B. Since h is bijective, it has an inverse h^{-1} from B onto A such that $B \cong A$.*
- *Transitive. Let $A \cong B$ and $B \cong C$. Then there exists an isomorphism h between A and B and an isomorphism j between B and C. Let k be defined as the composition $j \circ h$. We claim that k defines an isomorphism between A and C. Proof of claim: For all operation symbols $f : s_1 \times s_2 \times \cdots \times s_n \rightarrow s$ in Σ and for all $a_1 \in A_{s_1}, a_2 \in A_{s_2}, \ldots, a_n \in A_{s_n}$, and for all $b_1 \in B_{s_1}, b_2 \in B_{s_2}, \ldots, b_n \in B_{s_n}$, and for all $c_1 \in C_{s_1}, c_2 \in C_{s_2}, \ldots, c_n \in C_{s_n}$ we have*

$$\begin{aligned}
k(f_A(a_1, a_2, \ldots, a_n)) &= (j \circ h) f_A(a_1, a_2, \ldots, a_n) \\
&= j(h(f_A(a_1, a_2, \ldots, a_n))) \\
&= j(f_B(b_1, b_2, \ldots, b_n)) \text{ by definition of h} \\
&= f_C(c_1, c_2, \ldots, c_n) \text{ by definition of j} \\
\text{similarly } k^{-1}(f_C(c_1, \ldots, c_n)) &= (j \circ h)^{-1}(f_C(c_1, \ldots, c_n)) \\
&= j^{-1}(h^{-1}(f_C(c_1, \ldots, c_n)))
\end{aligned}$$

$$= j^{-1}(f_B(b_1, \ldots, b_n))$$
$$= f_A(a_1, \ldots, a_n)$$

Therefore, $A \cong B \wedge B \cong C \Rightarrow A \cong C$. ■

Appendix B. Refinement of a Global Search Algorithm Theory

This appendix contains an in-depth discussion of the specialization of a global search algorithm theory for the problem of searching an ordered sequence of integers. The problem specification for this problem, shown in Figure B.1, incorporates architectural information in that *SORT1* is a call to an operation defined by another problem specification. A description of this problem can be found in Section 4.2.2.

The information in this appendix supplements the information contained in Section 4.2. This appendix is written with the assumption that the reader has some familiarity with the Kestrel Interactive Development System (KIDS).

B.1 Derivation of a Specialized Algorithm Theory

There are a couple of global search algorithm theory instances in the KIDS theory library that could potentially be specialized for *Find-Location*. One of these global search theory instances, *gs-binary-split-of-integer-subrange*, is shown in Figure B.2. Because the specifications *Find-Location* and *Key-Search* are so similar, and because the global search algorithm theory *gs-binary-split-of-integer-subrange* was used for the derivation of a specification for *Key-Search*, the algorithm theory *gs-binary-split-of-integer-subrange* was selected for specialization for the problem defined by *Find-Location*. The following paragraphs describe the specialization of this global search algorithm theory for the specification *Find-Location*.

Specializing a global search theory instance $\Pi_G = \langle D_G, R_G, I_G, O_G \rangle$ so that it satisfies a problem specification $\Pi_F = \langle D_F, R_F, I_F, O_F \rangle$ is accomplished in part by deriving an

```
function FIND-LOCATION (A : seq(integer), key1 : integer | le-ordered(SORT1(A)) )
    returns (index : integer |
                index in [1 .. size(A)]
            &   SORT1(A)(index) = key1
            &   le-ordered(SORT1(A)) )
```

Figure B.1 Problem Specification for Find-Location

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                              %
%                Binary Split of Integer Subrange                              %
%                                                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



form index-gs-binary-split-of-integer-subrange
   k=make-binding('k) &
   m=make-binding('m) &
   n=make-binding('n) &
   i=make-binding('i) &
   j=make-binding('j) &
   new-i=make-binding('new-i)  &
   new-j=make-binding('new-j)
--> '(Global-Search-Theory gs-binary-split-of-integer-subrange
        input-types  integer, integer                 %  D
        output-types  integer                         %  R
        input-vars  m,    n
        output-vars k
        input-condition true                          % I(x)
        output-condition m<=k & k<=n                  % k in  [m..n]
        subspace-types integer, integer               % R_hat
        subspace-vars i,j                             % r_hat
        subspace-split-vars new-i, new-j              % s_hat
        subspace-vars-constraint m<=i & i<=j & j<=n % I_hat
                                                      % [i..j]subset[m..n]?
        satisfies  i<=k &  k<=j                       % k in  [i..j] ?
        initial-space (<m,  n>)                       % r0
        split ((new-i = i  & new-j = ((i+j) div 2))
               or  (new-i = (1+(i+j)  div 2)  &  new-j = j))
        extract i=j & k=i
        Feasibility-filter true
        Simplified-Feasibility-filter true
        Splittable    i<j
        Extractable i=j  )' in  gs-theories-prop(find-global('integer))
```

Figure B.2 Global Search Algorithm Theory Morphism

```
spec Filtered-Global-Search is
Sorts D, R, R̂
      op I : D → Boolean
      op O : D,R → Boolean
      op Î: D,R̂ → Boolean
      op r̂0 : D → R̂
      op Satisfies: R,R̂ → Boolean
      op Split: D,R̂, R̂ → Boolean
      op Extract: R,R̂ → Boolean
      op Filter : D,R̂ → Boolean
      axiom GS0 is
            (fa x (implies (I x) (Î x (r̂0 x))))
      axiom GS1 is
            (fa x (fa r̂ (fa ŝ (implies (and (and (I x) (Î x r̂)) (Split x r̂ ŝ))
                                       (Î x ŝ)))))
      axiom GS2 is
            (fa x (fa z (implies (and (I x)(O x z))(Satisfies z (r̂0 x)))))
      axiom GS3 is
            (fa x (fa r̂ (fa z (implies (and (I x)(Î x r̂))
                                       (iff (Satisfies z r̂)
                                            (ex ŝ (and (Split* x r̂ ŝ)(Extract z ŝ))))))))
      axiom filter is
            (fa x (fa r̂ (ex z (implies (and (and (satisfies z r̂) (O x z))
                                            (Î x r̂))
                                       (filter x r̂)) )))
end-spec
```

Figure B.3 Global Search Theory

antecedent to the soundness axiom of the global search theory. Global search theory is shown in Figure B.3. The soundness axiom given below contains an existentially quantified variable $y : D_G$; the antecedent derived by KIDS will be over the sort D_G. This antecedent helps define a substitution — denoted Θ — that defines how to specialize Π_G for Π_F. The Θ substitutions for the sort R_G, variables over R_G, and the operations I_G and O_G can be directly extracted from Π_F.

```
axiom soundness is
(and (equal R_F R_G)
     (fa x (ex y (fa z (implies (and (I_F x) (O_F x z))
     (O_G y z))))))
```

The instantiated soundness axiom for the algorithm theory instance *gs-binary- split-of-integer-subrange* and problem theory *Find-Location* is shown below. An antecedent for Equation B.1 will define a substitution $\Theta_{D_G} : D_G \mapsto D_F$ which, when applied to the symbols of the sort D_G, satisfy the axiom. The substitution Θ_{D_G} completes the definition of Θ and thus completes the definition of a specialization of the algorithm theory instance for this problem.

$$\underbrace{\underbrace{\text{le}-\text{ordered}(\text{SORT1}(\text{A}))}_{I_F} \;\&\; \underbrace{\text{index} \in [1..\text{size}(\text{A})]\&\text{SORT1}(\text{A})(\text{index}) = \text{key1}\&\text{le}-\text{ordered}(\text{SORT1}(\text{A}))}_{O_F}}_{\text{antecedent}} \Rightarrow \underbrace{\overbrace{\text{m} \leq \text{index}}^{C_1} \& \overbrace{\text{index} \leq \text{n}}^{C_2}}_{O_G} \qquad \text{(B.1)}$$

Before attempting to discover Θ, KIDS first collects additional terms by applying several levels of forward inferencing to the antecedent. Additional terms are generated from the antecedent through the application of domain independent and domain dependent rewrite rules. Domain dependent rules are defined in the domain theories from which problem theories are defined. The domain theories from which *Find-Location* is defined are *Ordered-Search* and *Sorting-Theory.*

The additional terms generated via forward inferencing over the antecedent of Equation B.1 are shown in Table B.1. The source of a derived term is shown to the right of the term. This list of terms is used to derive an antecedent for the soundness axiom. In the case of *Find-Location*, conjunct C_1 of Equation B.1 and generated term 1.1 yield a witness $\text{m} \mapsto 1$, while conjunct C_2 of Equation B.1 and generated term 1.2 yield a witness $\text{n} \mapsto \text{size(A)}$. These witnesses define — in conjunction with the substitutions directly extracted from *Find-Location* — the morphism (or Θ substitution) required to specialize *gs-binary-split-of-integer-subrange* so that it satisfies the problem specification *Find-Location.* Figure B.4 shows the specialized global search theory.

Table B.1 Derivation of Additional Terms

Depth	Number	Term	Source
0	1	le-ordered(sort1(A))	I(x)
	2	index in [1..size(A)]	O(x,z)
	3	sort1(A)(index) = key1	O(x,z)
1	1	$1 \leq$ index	0.2
	2	index $\leq$ size(A)	0.2
2	1	$1 \leq$ size(A)	1.1, 1.2

```
New global search theory incorporating the substitution:
   (global-search-theory FIND-LOCATION  number-of-solutions ONE
      input-types seq(integer), integer
      output-types  integer
      input-vars A, KEY1
      output-vars    INDEX
      input-condition   LE-ORDERED(SORT1(A))
      output-condition  INDEX in [1 .. size(A)]
                      & SORT1(A)(INDEX) = KEY1
                      & LE-ORDERED(SORT1(A))
      subspace-types  integer, integer
      subspace-vars I, J
      subspace-split-vars  NEW-I, NEW-J
      subspace-vars-constraint   J <= size(A) & I <= J & 1 <= I
      satisfies  INDEX  <= J  &  I <=  INDEX
      initial-space <1, size(A)>
      split NEW-J = J & NEW-I = (J + I) div 2 + 1
            or NEW-J =  (J + I) div  2 &  NEW-I  = I
      extract INDEX  = I &  I = J
      feasibility-filter true
      simplified-feasibility-filter true
      splittable   I < J
      extractable  I = J)
```

Figure B.4 Specialized Global Search Algorithm Theory

Up to this point the attempt to specialize an algorithm theory for the problem defined by the specification *Find-Location* has been successful. However, the specialized global search algorithm theory does not yet include an efficient feasibility filter. As described in the following section, inclusion of architectural information in the specification *Find-Location* slightly complicates the derivation of a feasibility filter ψ.

```
function KEY-SEARCH (A : seq(integer), key1 : integer | lt-ordered(A) )
      returns (index : integer |
                    index in [1 .. size(A)]
            &       A(index) = key1 )
```

Figure B.5 Key-Search Problem Specification

B.2 Derivation of a Feasibility Filter

A feasibility filter is used to prune the search space. The type of filter used in KIDS is a necessary filter: necessary filters prune only those portions of the search space that contain no solutions. In the previous section, the global search algorithm theory specialization did not attempt to strengthen the default feasibility filter *true*. In this section, the development of a stronger filter based on the filter equation presented in Figure B.3 is explored.

A feasibility filter is derived in KIDS by performing forward inference over the antecedent of the filter equation. Any conjunction of terms generated from the antecedent which contain references to the subspace descriptors (including the term *true*) can be used for the filter. As a basis of comparison, the filter derived for the similar problem defined by the specification *Key-Search* is described. The specification *Key-Search* is presented in Figure B.5.

B.2.1 Key-Search Feasibility Filter. The instantiated feasibility filter for *Key-Search* is presented below. Note that KIDS will assume $I(x)$ during the forward inference process.

$$\underbrace{1 \leq \mathrm{I} \wedge \mathrm{I} \leq \mathrm{J} \wedge \mathrm{J} \leq \mathrm{size(A)}}_{\hat{I}(x,\hat{r})} \wedge \underbrace{\mathrm{I} \leq \mathrm{INDEX} \wedge \mathrm{INDEX} \leq \mathrm{J}}_{\mathrm{Satisfies(z,\hat{r})}} \wedge \underbrace{\mathrm{A(INDEX)} = \mathrm{KEY1} \wedge \mathrm{INDEX} \in [1..\mathrm{size(A)}]}_{\mathrm{O(x,z)}} \Rightarrow \psi(x, \hat{r}) \tag{B.2}$$

r_1: assert MONOTONICITY-OF-LE-ORDERED
$\forall$ (S, A, B)
(LE-ORDERED(S) $\wedge$ 1 $\leq$ A $\wedge$ B $\leq$ size(S) $\Rightarrow$ A $\leq$ B = S(A) $\leq$ S(B))

r_2: assert MONOTONICITY-OF-LT-ORDERED-1
$\forall$ (S, A, B)
(LT-ORDERED(S) $\wedge$ 1 $\leq$ A $\wedge$ B $\leq$ size(S) $\Rightarrow$ A < B = S(A) < S(B))

r_3: assert MONOTONICITY-OF-LT-ORDERED-2
$\forall$ (S, A, B)
(LT-ORDERED(S) $\wedge$ 1 $\leq$ A $\wedge$ B $\leq$ size(S) $\Rightarrow$ A $\leq$ B = S(A) $\leq$ S(B))

Figure B.6 Domain Specific Rules

Using domain rules such as those shown in Figure B.6, KIDS generates the list of terms shown in Table B.2 from the antecedent of Equation B.2. A conjunction of the terms *A(I)* $\leq$ *KEY1* and *KEY1* $\leq$ *A(J)* forms a useful filter for *Key-Search.* Because *Find-Location* so closely parallels *Key-Search*, and is in fact built from the same domain theories, a natural expectation would be that a similar set of filter terms would be generated for *Find-Location.* That is, the filter terms *Sort1(A)(I)* $\leq$ *KEY1* and *KEY1* $\leq$ *Sort1(A)(J)* should be generated for *Find-Location.*

B.2.2 Find-Location Feasibility Filter. The problem theories *Key-Search* and *Find-Location* are almost identical; the only significant difference between them concerns references to the input variable A.

Key-Search includes *le-ordered(A)* as a precondition or input assumption (i.e., as $I(x)$). Like *Key-Search*, *Find-Location* also searches only an ordered sequence, but instead depends on a call to *Sort1* to order the sequence. Therefore all references to A in *Find-Location* are prefixed with a call to *Sort1*, hence both $I(x)$ and $O(x, z)$ of *Find-Location* include the term *le-ordered(SORT1(A)).* However, this term no longer matches the form of the monotonicity laws of Figure B.6. These monotonicity laws are quantified over sequences, not *functions* that return sequences. The sequence-sorted variable S in the term *le-ordered(S)* in the antecedents of the monotonicity laws of Figure B.6 does not unify

Table B.2 Generation of Terms for the Key-Search Feasibility Filter

Depth	Number	Term	Source
0	1	$1 \leq I$	$\hat{I}$
	2	$I \leq J$	$\hat{I}$
	3	$J \leq$ size(A)	$\hat{I}$
	4	INDEX $\leq$ J	Satisfies
	5	I $\leq$ INDEX	Satisfies
	6	A(INDEX) = KEY1	O(x,z)
	7	INDEX in [1 .. size(A)]	O(x, z)
1	1	$1 \leq$ J	0.1, 0.2
	2	I $\leq$ size(A)	0.2, 0.3
	3	A(I) $\leq$ A(J)	0.1, 0.3, 0.2, r_3
	4	$1 \leq$ INDEX	0.7
	5	INDEX $\leq$ size(A)	0.7
2	1	$1 \leq$ size(A)	1.2, 0.1
	2	A(1) $\leq$ A(J)	1$\leq$1, 0.3, 1.1, r_3
	3	A(1) $\leq$ A(I)	1$\leq$1, 1.2, 0.1, r_3
	4	A(J) $\leq$ A(size(A))	1.1, size(A)$\leq$size(A), 0.3, r_3
	5	A(1) $\leq$ A(INDEX)	1$\leq$1, 1.5, 0.5, r_3
	6	A(I) $\leq$ A(INDEX)	0.1, 1.5, 0.5, r_3
	7	A(INDEX) $\leq$ A(J)	1.4, 0.3, 0.4, r_3
3	1	A(1) $\leq$ KEY1	2.5, 0.6
	2	A(I) $\leq$ KEY1	2.6, 0.6
	3	KEY1 $\leq$ A(J)	2.7, 0.6
	4	A(1) $\leq$ A(size(A))	2.2, 2.4
	5	A(I) $\leq$ A(size(A))	1.3, 2.4
	6	A(INDEX) $\leq$ A(size(A))	2.4, 2.7

with expressions of the form *SORT1(A)* found in *Find-Location* because *Sort1(A)* is of sort *function*. As a result, the monotonicity laws will not be applied to the instantiated filter equation (Equation B.3 below), resulting in very weak filter terms. This is an artifact of the inference mechanism and does not in general invalidate the nested function call approach to defining structure.

$$\underbrace{\underbrace{1 \leq I \wedge I \leq J \wedge J \leq size(A)}_{\hat{I}(x,\hat{r})} \wedge \underbrace{\mathrm{I} \leq \mathrm{index} \wedge \mathrm{index} \leq \mathrm{J}}_{Satisfies(z,\hat{r})} \wedge}_{O(x,z)} \mathrm{sort1(A)(index)} = \mathrm{key1} \wedge \mathrm{index} \in [1..\mathrm{size(A)}] \wedge \mathrm{le-ordered(sort1(A))} \Rightarrow \psi(x,\hat{r}) \quad \text{(B.3)}$$

Table B.3 Generation of Terms for the Find-Location Feasibility Filter

Depth	Number	Term	Source
0	1	$1 \leq I$	$\hat{I}$
	2	$I \leq J$	$\hat{I}$
	3	$J \leq$ size(A)	$\hat{I}$
	4	INDEX $\leq$ J	Satisfies
	5	I $\leq$ INDEX	Satisfies
	6	SORT1(A)(INDEX) = KEY1	O(x,z)
	7	INDEX in [1 .. size(A)]	O(x, z)
1	1	$1 \leq J$	0.1, 0.2
	2	I $\leq$ size(A)	0.2, 0.3
	3	$1 \leq$ INDEX	0.7
	4	INDEX $\leq$ size(A)	0.7
2	1	$1 \leq$ size(A)	1.2, 0.1

The terms generated from Equation B.3 are shown in Table B.3. As can be seen in the table, none of the terms form very useful filters. To compensate for the weakness in the inference mechanism, the domain theory for *Find-Location* could be modified by including references to *Sort1* in the theory rules. Some of the monotonicity rules for *Find-Location* incorporating these references are shown in Figure B.7.

Modifying the domain theory rules for *Find-Location* yielded additional terms that could be used for a feasibility filter. The filter equation, Equation B.3, remains unchanged. As shown in Table B.4, the additional theory rules yielded the term $sort1(A)(I) \leq Key1$ that could be used for an effective filter. However, not all possible terms were generated. For example, at depth two we should have generated the term $sort1(A)(index) \leq sort1(J)$ from terms 1.3, 0.3, 0.4, and modified rule r_{1_m}. At depth three, we would've then generated the term $key1 \leq sort1(J)$ from terms 0.6 and $sort1(A)(index) \leq sort1(J)$. Again, this reflects a weakness in the inference mechanism and does not indicate any fundamental theoretical difficulties.

B.3 Summary

Although a relatively weak filter equation was generated, a global search algorithm theory for the problem specification *Find-Location* was successful specialized. Development

r_{1_m}: assert MONOTONICITY-OF-LE-ORDERING-OF-SORT1-A
$\forall$ (z, i, j)
(LE-ORDERED(SORT1(z)) $\wedge$ 1 $\leq$ i $\wedge$ j $\leq$ size(z) $\wedge$ i $\leq$ j
$\Rightarrow$ SORT1(z)(i) $\leq$ SORT1(z)(j))

r_{2_m}: assert MONOTONICITY-OF-LE-ORDERING-OF-SORT1-A-2
$\forall$ (z, i, j)
(LE-ORDERED(SORT1(z)) $\wedge$ 1 $\leq$ i $\wedge$ i $\leq$ size(z) $\wedge$ i $\leq$ j
$\Rightarrow$ SORT1(z)(i) $\leq$ SORT1(z)(size(z)))

r_{3_m}: assert MONOTONICITY-OF-LT-ORDERING-OF-SORT1-A
$\forall$ (z, i, j)
(LT-ORDERED(SORT1(z)) $\wedge$ 1 $\leq$ i $\wedge$ j $\leq$ size(z) $\wedge$ i $\leq$ j
$\Rightarrow$ SORT1(z)(i) $\leq$ SORT1(z)(j))

r_{4_m}: assert MONOTONICITY-OF-LT-ORDERING-OF-SORT1-A-2
$\forall$ (z, i, j)
(LT-ORDERED(SORT1(z)) $\wedge$ 1 $\leq$ i $\wedge$ i $\leq$ size(z) $\wedge$ i $\leq$ j
$\Rightarrow$ SORT1(z)(i) $\leq$ SORT1(z)(size(z)))

Figure B.7 Domain Specific Rules Incorporating *Sort1*

of a feasibility filter was somewhat less successful, and depended in part on an extended domain theory for *Find-Location.* In addition, the methodology used to extend the domain theory may not scale well to deeply nested functions, and results in domain theory rules that are application specific rather than domain specific.

Table B.4 Generation of Feasibility Filter Terms for Find-Location using the Enhanced Domain Theory

Depth	Number	Term	Source
0	1	$1 \leq I$	$\hat{I}$
	2	$I \leq J$	$\hat{I}$
	3	$J \leq$ size(A)	$\hat{I}$
	4	INDEX $\leq$ J	Satisfies
	5	I $\leq$ INDEX	Satisfies
	6	SORT1(A)(INDEX) = KEY1	O(x,z)
	7	INDEX in [1 .. size(A)]	O(x, z)
	8	LE-ORDERED(SORT1(A))	O(x,z)
1	1	$1 \leq J$	0.1, 0.2
	2	$I \leq$ size(A)	0.2, 0.3
	3	$1 \leq$ INDEX	0.7
	4	INDEX $\leq$ size(A)	0.7
2	1	$1 \leq$ size(A)	1.2, 0.1
	2	SORT1(A)(I) $\leq$ SORT1(A)(size(A))	0.1, size(A)$\leq$size(A), 1.2, r_{1_m}
	3	SORT1(A)(J) $\leq$ SORT1(A)(size(A))	1.1, size(A)$\leq$size(A), 0.3, r_{1_m}
	4	SORT1(A)(I) $\leq$ SORT1(A)(INDEX)	0.1, 1.4, 0.5, r_{1_m}
3	1	SORT1(A)(I) $\leq$ KEY1	2.4, 0.6

Appendix C. An Informal Introduction to Components and Connectors

This appendix contains an informal description of components and connectors. A formal treatment of these topics can be found in Chapter V.

C.1 Components

A component specification has two distinct parts:

1. a functional specification which introduces and defines the sorts and operations of the component; and
2. an interface specification which defines how the operations of a component may be accessed.

Each of these two specifications are discussed in the following subsections.

C.1.1 Component Functional Specification. The functional specification of a component is defined by a SLANG specification. The sorts and operations of a component C are defined in an associated functional specification. This section briefly describes the functional specifications of components.

There are essentially two ways in which operations may be defined:

1. Using axioms written in equational logic; and
2. Abstractly using a *problem theory.*

Each of these approaches are described in the following subsections.

C.1.1.1 Operator Definition using Axioms Written in Equational Logic. Specification of operators using equational logic is not new; many textbooks on programming languages or software engineering use equational logic to define the operators of abstract data types (ADTs). In SLANG, there is a small twist. Equational axioms in SLANG may take one of two forms: a simple axiom form or a *definition* form. The affect of these two forms is not equivalent.

Simple axioms such as *axiom (equal (size []) zero)* can be used to define a relationship between operators, in this case between the operator *size* and the constant *zero.* A series of such axioms could be provided to constrain the class of models to those in which the interpretation of *size* is initial (i.e., there is only one possible non-trivial interpretation of the operator). A more powerful axiomatization uses the definition form.

The definition form of specifying an operator induces an additional axiom that states that induction is a sound inference rule for the operator. For example, if the operator *size* was defined using a definitional form as given below, then SpecWare will add an axiom that states that induction is a sound with respect to the given operator.

```
definition defn-of-size is
      axiom (equal (size []) zero)
      axiom (equal (size (concat x y)) (plus (size x)(size y)))
      axiom (equal (size [a]) one)
end-defn
```

In either case, equational axioms such as those listed above will usually be used to define well understood, simple operations such as *size* for sequences, sets, and bags, or *neighbor* for graphs. More abstract problems, those requiring search strategies or decomposition methods, will usually be defined abstractly using problem theories.

C.1.1.2 Abstract Specification of Operators Using Problem Theories. An operation of a component may be abstractly defined by a problem specification $\mathcal{B}_F$ which is written using the sorts and operations defined in a *domain theory.* A domain theory defines the sorts, the operations, and the axioms of the problem domain. A problem specification is a sentence in the domain theory.

For example, consider the problem of boolean satisfiability. The domain theory for this problem might contain operations such as *clause-satisfied* which could be used to determine if a given assignment of truth values to the literals of some clause satisfies the clause. The domain theory in this case would contain definitions of literals and clauses, and would contain axioms defining relationships between the operation *clause-satisfied* and operations used to construct its arguments. For example, if the rank of *clause-satisfied* is

map, clause, then one possible set of axioms in the domain theory could define how *clause-satisfied* distributes over the map constructors.

A problem theory can be represented in a program-like format.(105) Shown below is the format for a problem theory and its corresponding program-like representation:

spec	*ProblemTheory* **is**	**function**	*F(x:D) : set(R)*
sorts	D, R	**where**	*I(x)*
op	I : D → Boolean	**returns**	$\{z \mid O(x,z)\}$
op	O : D × R → Boolean	= *Body*	
axiom			
	$\forall(x \in D)(I(x) \Rightarrow \exists z \in R \mid O(x,z))$		
end-spec			

I(x) constrains the input domain **D**; the output condition *O(x,z)* describes the conditions under which the output domain value $z \in \mathbf{R}$ is a feasible solution with respect to input $x \in \mathbf{D}$; *Body*, if present, is the code that can be executed to compute F (105, 100). The single axiom of *ProblemTheory* above indicates that if the input assumption I is satisfied, then there is a value z satisfying the output condition. This is the one solution form of *ProblemTheory*. All solutions satisfying the output condition O for an input x satisfying the input condition can be obtained by introducing an additional operator $f : D \rightarrow R$ and replacing the above axiom with the axiom $\forall(x \in D)(I(x) \Rightarrow f(x) = \{z \mid O(x,z)\}$. Thus the simplest form of *ProblemTheory* is one which contains no axioms. Extensions to this simple problem theory can be made to define problem theories that return one solution or all solutions depending on the axiom used.

A problem specification can be referred to by the tuple $< D, R, I, O >$. Regardless of the format, a problem specification $\mathcal{B}_F$ for a particular problem is defined by a specification morphism which maps

- the sort D to the sort(s) of the problem space;
- the sort R to the sort(s) of the solution space;
- I to a boolean function over D; and which maps
- O to a boolean function over $D \times R$.

In other words, $\mathcal{B}_F$ is created by instantiating the generic sorts and operations of $\mathcal{B}$. A problem specification for a four sum adder is shown in Figure C.1. The specification shown in the figure uses some domain specific operations, such as *complex-add*, as well as some domain independent operations such as *image* and *size*. The domain specific sort *complex* and the operation *complex-add* are imported into the specification *Complex-Adder* via an imports clause. In this case, the definition of these domain specific sorts and operations are located in the specification *Complex*, which may in turn import other specifications. The relationship between *Problem-Theory* and *Complex-Adder* is made explicit by the morphism *Problem-Theory -¿ Complex-Adder* defined by the mapping $D \mapsto$ *seq*, $R \mapsto$ *comp-nat*, $I \mapsto$ *input-cond*, and $O \mapsto$ *output-cond*. The sort *comp-nat* is a product sort consisting of a complex value and a natural value. The sort of *add4* is *comp-nat*, where the complex portion represents the summation of the complex values in the input sequence, and the natural portion represents the number of elements summed. Using (x_1, x_2) to represent a complex value with real part x_1 and complex part x_2, *(add4 '((1.0, 1.0), (3.5, -2.0), (3.2, 1.2)))* evaluates to the tuple $\langle(7.7, 0.2), 3\rangle$. Note that problem theories are not generally used to represent simple problems such as *complex-add*. Problem theories are typically used for problems that involving searching or problem decomposition.

The current version of SLANG includes only *Boolean* as an integral part of every specification. Other common data types such as sets, sequences, and integers must explicitly defined.

A functional specification only partially defines a component. The other part of a component, an interface specification, defines how the component interacts with its environment. Interface specification is described next.

C.1.2 Component Interface Specification. An interface specification of a component defines how it interacts with its environment. Interface specifications for a component are defined using CSP. For example, an interface specification for a component whose functional specification defines exactly two operations f and g may have process descriptors P_f and P_g defining the interfaces of the operation f and g respectively.

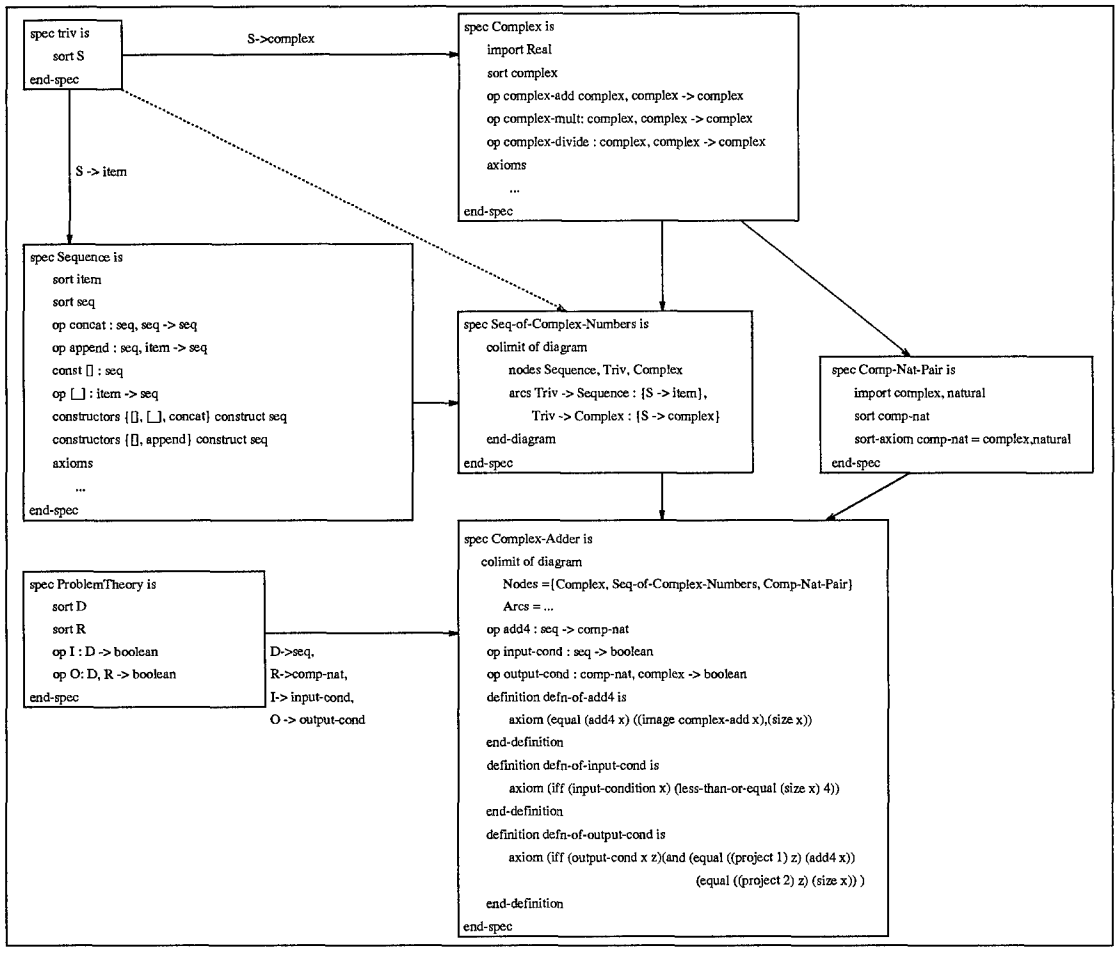


Figure C.1 Problem Specification for Complex Adder

Each operation defined in the component's functional specification can have a unique pair of *ports* associated with it, one for inputs and the other for outputs. All communication with a component takes place over its ports. Any operation defined in a component's functional specification that does not have a pair of ports associated with it is not accessible outside of the component and can only be accessed indirectly.

A port is one-half of a CSP channel. Because CSP channels are unidirectional, ports are unidirectional. Ports are also strongly typed. For example, if P is a process descriptor, then the process expression $P = p_1?x \xrightarrow{CSP} (p_2!f(x) \xrightarrow{CSP} P)$ defines a process that reads a value x, communicates over the port p_2 the value $f(x)$, and repeats. The sort of port p_1 is defined by the rank of f while the sort of p_2 is defined by the sort of f. A more formal treatment of this topic can be found in Chapter V.

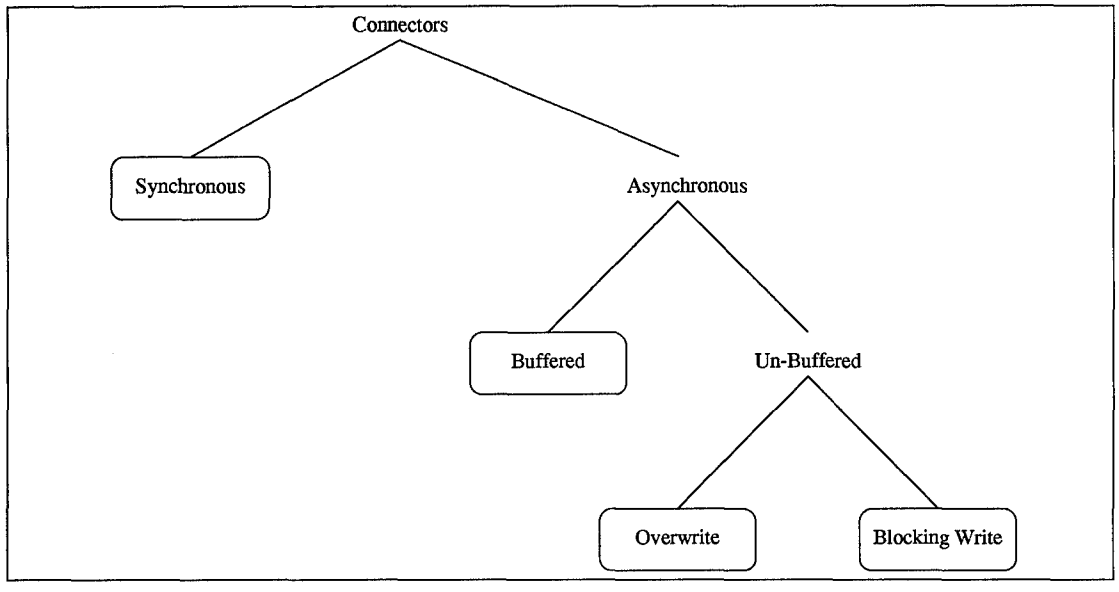


Figure C.2 Communication Taxonomy (based on (112))

An interface specification of a component has visibility into the syntax of the operations defined in the functional specification of the component and may reference the operation symbols found there. For example, a problem theory could be used to develop the following interface specification for a component encapsulating a single problem theory: $P = (c_1?x \xrightarrow{CSP} ((c_2!f(x) \xrightarrow{CSP} Skip) \not\leftarrow I(x) \not\rightarrow Skip) \mid \surd$. The process P initially engages in a communication event over port c_1, obtaining a value stored in the variable x. If $I(x)$ is *true*, then the value $f(x)$ is communicated over port c_2; if $I(x)$ is *false*, then the input is ignored.

C.1.3 Summary of Components. In summary, a component specification has two parts: an interface specification which defines how the component interacts with its environment and a functional specification that introduces and defines the sorts and operations of the component. Components interact with each other by sending messages over CSP channels.

A specific type of component, *connectors*, are described in the next subsection.

C.2 Connectors

The primary purpose of a connector is to define inter-component handshaking conventions. Communication over a CSP channel is defined by the semantics of CSP to be a synchronous event. Both the recipient and the sender of the data over a channel must synchronize and simultaneously engage in the data transfer. However, the use of a connector between components allows for the definition of a variety of handshaking conventions as depicted in Figure C.2 and described below.

- Synchronous. The sending component C_S transmits its data d to the connector C over a channel shared between C_S and C. The connector C then relays the data d just read to another component C_R over a channel shared by the connector C and C_R. The connector C then transmits a an acknowledgment to C_S; on receipt of the acknowledge signal, C_S may proceed with other computation. C_S remains blocked until it engages in an acknowledge communication event. The general form of this interaction is:

  $$C_S = \ldots c!d?ACKNOWLEDGE \ldots$$
  $$C = \ldots c.left?d \rightarrow p!d \rightarrow c.right!ACKNOWLEDGE \ldots$$
  $$C_R = \ldots p?x \ldots$$

  where $c!x?y$ is defined to be the atomic process defined by $c.left!x \rightarrow c.right?y$, and where C_S, C, and C_R are process definitions found in the sending component, the connector, and the receiving component respectively.

- Asynchronous Buffered. This communication paradigm is slightly more complex than the synchronous case. In asynchronous buffered communication, the sending component transmits its data to the connector which then enqueues the data. The receiving component obtains its data by via a dequeuing operation. Note that in this case the sender is free to continue with other processing after enqueuing its output data, and that the connector must have a component encapsulating a buffer abstract data type slaved to it playing the role of the queue. Any buffer abstract data type will work here, from a stack to a queue depending on the desired effects.

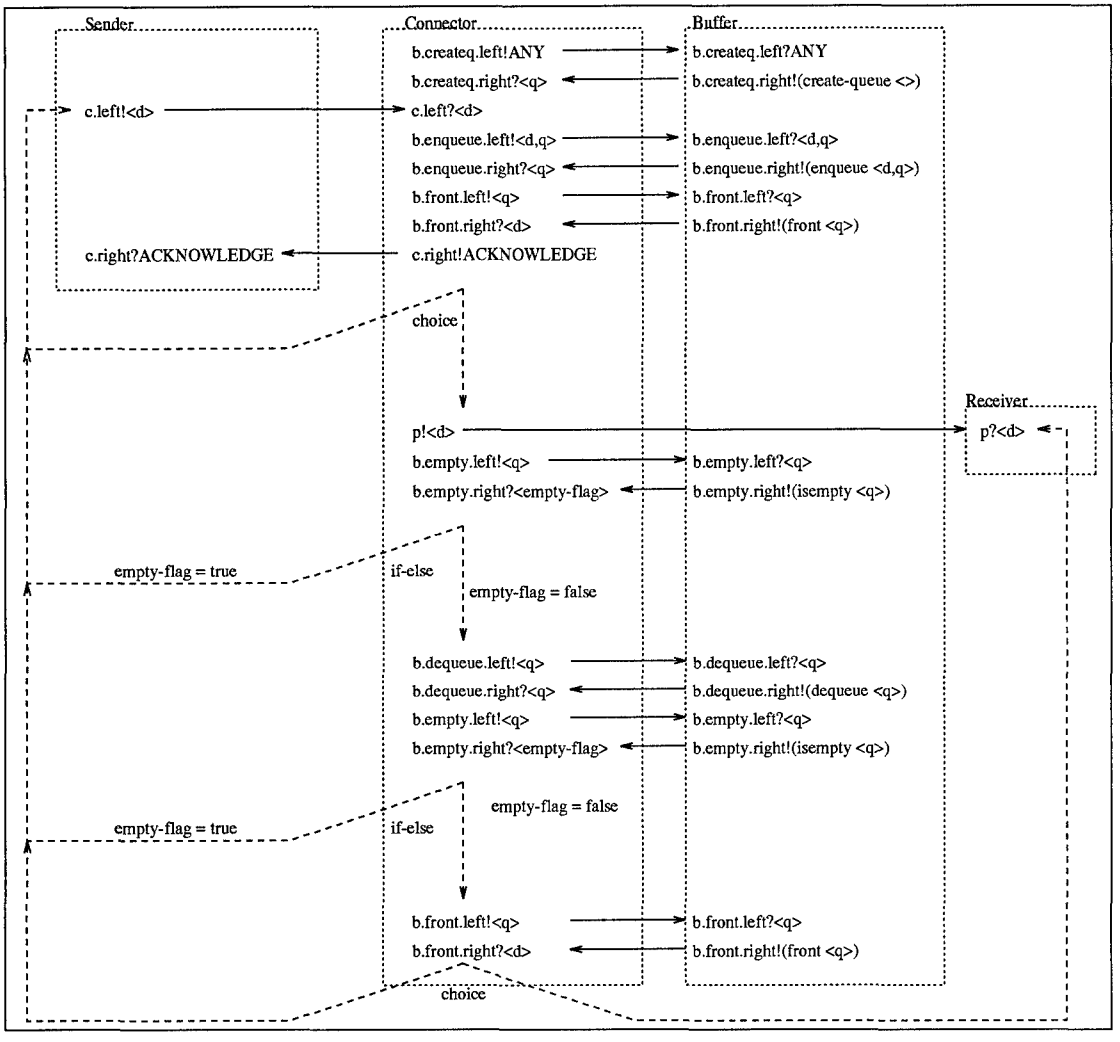


Figure C.3 Asynchronous Buffered Communication

The general form of this interaction is listed below and shown in Figure C.3. In the figure, dashed arrows represent flow of control, solid arrows denote communication, and dotted boxes have been drawn around each of the interface processes. The general form of this interaction is:

$$C_S = \ldots c!d?ACKNOWLEDGE \ldots$$

$$C = I\ ;\ E;\ A$$

$$I = b.CreateQueue!ANY?q$$

$$A = (E \mid D)\ ;\ A$$

$$E = c.left?d \rightarrow b.front!(eval\ b.enqueue!\langle\ d,\ q\ \rangle?q)?d$$

$$D = p!d \rightarrow b.dequeue!q?q \rightarrow$$
$$(E < (eval\ b.empty!q?empty\text{-}flag) > b.front!q?d)$$

$$C_R = \ldots p?x \ldots$$

where

- C_S is part of an interface specification for a component creating data values which will be enqueued;
- C is part of the connector's interface specification in which
  - q of is a state variable representing the state of the data queue;
  - d is a state variable guaranteed to be equal to the object at the head of the queue;
  - I is a process definition which requires the queue be created before it can be accessed;
  - A is a process definition consisting of two subordinate process definitions, E for enqueuing new objects, and D for dequeuing objects;
- C_R is part of an interface specification for a component which consumes the objects enqueued by C_S; and
- *ANY* and *ACKNOWLEDGE* are events shared between the components and connectors of the application.

Note that the above process definitions make use of the *eval* construct, where *(eval c!x?y)* accesses the value y. Also note that the protocols of the data object producer and the data object consumer are unchanged from those of the synchronous case. This is a theme that will carry over to the asynchronous unbuffered case as well. Only the connector interface specification is affected by the choice of handshaking mechanism.

- Asynchronous Unbuffered. This case is a slightly simplified version of asynchronous buffered communication. The component producing data objects engages in a communication event with a connector but unlike the synchronous case, the connector

first engages in a communication of an acknowledgment before relaying the data object. The connector then attempts to engage in a communication event with a component which will consume the object. The connector blocks until the consuming component is ready to receive the data. This implies that the producer may become blocked if the consumer has not yet consumed the previously communicated data object. If this happens, there are at least two possible courses of action:

1. Overwrite the old data with the new. In this case, the interface specifications will have the following form:

$C_S = \ldots c!x?ACKNOWLEDGE \ldots$

$C = \ldots (c.left?x \rightarrow c.right!ACKNOWLEDGE) \rightarrow D$

$D = ((c.left?x \rightarrow c.right!ACKNOWLEDGE) \mid p!x) \rightarrow D$

$C_R = \ldots p?d \ldots$

2. Block on the second write attempt. In this case, the interface specifications will have the form:

$C_S = \ldots c!x?ACKNOWLEDGE \ldots$

$C = c.left?x \rightarrow c.right!ACKNOWLEDGE \rightarrow p!x \rightarrow C$

$C_R = \ldots p?d \ldots$

C.3 Summary

This appendix has given an informal introduction to components and connectors. Components were informally defined as a combination of a functional specification written in SLANG and an interface specification written in CSP. Connectors were informally defined as a component whose primary purpose is to define handshaking conventions. Several handshaking conventions were introduced and defined, including synchronous, asynchronous buffered, and asynchronous unbuffered.

Appendix D. Constraints

D.1 Introduction

Although process specification morphisms require the preservation of models under a trace semantic equivalence, other constraints could be placed on the development of process specifications in an effort to prevent the specification of degenerate or unrealizable processes. These constraints could take one of the following two forms:

1. Constraints over the use of functional operations within a process specification. For example, Basic Problem Theory specifications of Chapter III have an explicit boolean operation used to characterize the range of acceptable values for which the operation characterized by the problem theory is guaranteed to find solutions. A constraint could be defined such that the output condition(s) of any operation supplying data to the operation satisfy the input condition of that operation. Any specification construction that violates this constraint could be flagged as an invalid construction.
2. Constraints over process expressions. For example, process expressions should be free of both deadlock and live-lock.

Each of these type of constraints are discussed in the following sections.

D.2 Constraints over operations

Process specifications define networks of communicating processes, where the output of one process may be consumed by another process. The values communicated over CSP channels may have been generated as the output of a functional operation, and may be used as an input of another operation. Constraints could be expressed over process specifications such that any data values communicated between processes are used in both a syntactically and semantically consistent manner. Syntactical consistency is ensured by the strong typing of CSP. One facet of semantic consistency, that the values of actual parameters be within acceptable ranges, is described in the following paragraphs.

Functional operators referenced within process expressions of a process specification are defined by functional specifications. The only operations defined using Slang speci-

fications that have explicit input value restrictions identified for them are those defined using Problem Theory specifications. For these operations, a constraint expressing input condition satisfaction may be defined, where input condition satisfaction for functional operations of process specifications is defined below.

Definition D.1 *Denote by pSP a process specification containing a process expression P where* $f : s_1, s_2, \ldots, s_n \rightarrow s$ *is an operation defined by a problem specification such that f is used as an argument in an output event* $c!f(x_1, x_2, \ldots, x_n)$ *in P. Let* $c_i?x_i, 1 \leq i \leq n$ *be a collection of input communication events in P preceding* $c!f(x_1, x_2, \ldots, x_n)$ *in any trace* $t \in traces(P)$*, and let* P_i *be a collection of process expressions in pSP such that* c_i *is a port symbol in* P_i *such that* $c_i!h_i(y_i)$ *is an output event in* traces $(\Xi(P_i))$.

Then pSP is consistent with respect to f in P if, for all occurrences of f in P,

$$\bigwedge_{i=1..n} O(y_i, h_i(y_i)) \Rightarrow I_f(x_1, x_2, \ldots, x_n) \tag{D.1}$$

where $h_i(y_i) = x_i$. *If* h_i *is not defined by a problem specification, then Equation D.1 is undefined.* □

For example, consider once again the sort-search problem described in Section 4.2, and suppose that a process specification defining an interface process for each of the operations *sort : bag → seq* and *search : seq, item → index* has been defined and contains the two process definitions below.

$$P_{sort} \quad \text{sat} \quad c_{in}?x \xrightarrow{CSP} (c!\text{sort}(x) \xrightarrow{CSP} P_{sort})$$

$$P_{search} \quad \text{sat} \quad p?el \xrightarrow{CSP} (c?y \xrightarrow{CSP} (c_{out}!\text{search}(el, y) \xrightarrow{CSP} P_{search}))$$

Based on the semantics of CSP communication, the value of x following communication over the channel c is *sort(w).* Recalling that $O_{sort}(x,z)$ = *ordered(z)* ∧ *bag-equal(elements-of(x), elements-of(x)* ∧ z = *sort(x)* and that I_{search} *(y) = le-ordered(y)* it can be proven, as shown in Table D.1, that the input condition of the operation *search* is satisfied by the output condition of *sort.* It can therefore be concluded that the process

Depth	Number	Term	Source
0	1	le-ordered(z)	O(x,z)
	2	bag-equal(elements-of(x), elements-of(z))	O(x,z)
	3	z = sort(x)	O(x,z)
	4	y = sort(x)	process expression
1	1	z = y	0.3, 0.4
2	1	le-ordered(y)	0.1, 1.1

Table D.1 Proof of Input Condition Satisfaction

specification containing P_{sort} and P_{search} is consistent with respect to the operation *search* in P_{search}.

Table D.1 shows the terms used in the proof and their source. Some of the terms, e.g., the term $y{=}z$, are generated through analysis of the process expressions. Note that this proof involves the axioms of functional specifications, and as such a *proof schema* must be used to relay the conjecture from the process logic of ISlang to the functional logic of SpecWare. Proof schemas are described in the following section.

Although this example is rather simplistic, it serves to illustrate the concept of proving process specifications consistent with respect to the operations they reference. Further elaboration of a proof mechanism for this purpose is left for future research.

If a proof of the property defined in Definition D.1 fails, a *derived antecedent* could be used to define the output condition for a conditioning operation which, when applied to incoming data, is guaranteed to satisfy the input condition of the consuming operation. This concept is formalized in the following paragraphs.

Definition D.2 Derived Antecedent. *Let* $P(x_1, x_2, \ldots, x_n) \xrightarrow{w} Q(y_1, y_2, \ldots, y_m)$ *be a well-formed boolean formula where* $P(x_1, x_2, \ldots, x_n)$ *is an expression over the variables* $\{x_1, x_2, \ldots, x_n\}$, $Q(y_1, y_2, \ldots, y_m)$ *is an expression defined over the variables* $\{y_1, y_2, \ldots, y_m\} \subseteq \{x_1, x_2, \ldots, x_n\}$, *and where* $\xrightarrow{w}$ *is a well ordering such as* $\Leftarrow$, $\Rightarrow$, *or* $\leq$. *Then a* derived antecedent *is an expression* $A(z_1, z_2, \ldots, z_p)$ *where* $\{z_1, z_2, \ldots, z_p\} \subseteq \{x_1, x_2, \ldots, x_n\}$ *such that*

$$A(z_1, z_2, \ldots, z_p) \Rightarrow (P(x_1, x_2, \ldots, x_n) \xrightarrow{w} Q(y_1, y_2, \ldots, y_m)) \quad \square$$

For example, if the input condition of the operation *search* mentioned above was *ge-ordered(y)* rather than *le-ordered(y)*, the proof shown in Table D.1 would fail. The derived antecedent would, in this case, be *le-ordered(y)*. This antecedent can be used to define a new operation, *condition : seq* $\rightarrow$ *seq*, constructed as follows:

1. The sort and rank of the operation are defined by the sort of the variable being conditioned. In this case, the sort and the rank of the operation are `seq`.
2. The input condition of the operation *condition* is defined by the output condition of the producer as follows. All references to producer input variables in the expression $O_{producer}(x, z)$ are eliminated, and the remaining variables are renamed to maintain consistency with variables of the process in which the conditioning operation is referenced. In the example above, elimination of non-output variables from the output condition of the producer *sort* results in the expression *le-ordered(z)*. However, z is not a variable defined in P_{search}. Based on the semantics of CSP communication, the variable y in P_{search} equals the value of z following a communication event on channel c. Thus references to z in the expression *le-ordered(z)* are replaced with references to y to obtain the expression *le-ordered(y)*. Thus $I_{condition}(y)$ = *le-ordered(y)*.
3. The output condition of the operation *condition* is defined by the derived antecedent. For this example, $O_{condition}(y,z)$ = *ge-ordered(z)*.

Note that the above definition of the operation *condition* permits a trivial terminal model. That is, the operation which returns the empty sequence is a valid model of the specification. Whether this is acceptable depends on the application.

Once a functional specification for the conditioning operation has been defined, the process specification of the consumer can be modified to include references to the condi-

tioning operation. For the above example, P_{search} and P_{sort} are modified as follows:

$$P_{sort} \quad \text{sat} \quad c_{in}?x \xrightarrow{CSP} (\text{c!sort}(x) \xrightarrow{CSP} P_{sort})$$
$$P_{search} \quad \text{sat} \quad p?el \xrightarrow{CSP} (c?y \xrightarrow{CSP} (c_{out}!\text{search}(el, \text{condition}(y)) \xrightarrow{CSP} P_{search}))$$

Note that the communication network (the functional model) defined by the process expressions remained intact; the only change to the process specification was the addition of the operation *condition* to the signature of P_{search} and its nesting within the call to *search*.

Although a process specification may be consistent with respect to the sorts and operations it contains, it still may be degenerate. For example, the modified set of process expressions shown below is consistent with respect to the sorts and operations it references, yet is degenerate in that processes P_1 and P_2 are live-locked.

$$P_1 \quad \text{sat} \quad (e \xrightarrow{CSP} P_1); (c_{in}?x \xrightarrow{CSP} (c!\text{sort}(x) \xrightarrow{CSP} P_{sort}))$$
$$P_2 \quad \text{sat} \quad (e \xrightarrow{CSP} P_2) \| (p?el \xrightarrow{CSP} (c?y \xrightarrow{CSP} (c_{out}!\text{search}(el, y)))) \xrightarrow{CSP} P_{search})$$

The above two processes, once engaged, endlessly engage in a series of events e, and never interact with any other processes. Constraints placed on process expressions aimed at eliminating live-lock and deadlock are discussed in the following subsection.

D.3 Constraints over process expressions

Live-lock and deadlock are are both *safety* properties, where a safety property is one which states that something must not happen.(10) For example, the statement $\langle r^\wedge e_1^\wedge s^\wedge e_2 \rangle \notin$ *traces(P)* for some process P defines a constraint that P must not engage in the event e_1 before engaging in the event e_2. This constraint is equivalent to requiring that the language $W = \{t \mid t = r^\wedge e_1^\wedge s^\wedge e_2, r, s \in (\alpha P)^*\}$ contain no words in common with the language $L(P)$. This concept is formalized by the following definition.

Definition D.3 *Given a process specification P, a safety property of P is a language L_S defined over the alphabet αP which constrains $L(P)$ such that $L_S \cap L(P) = \{\}$.* □

However, this question is undecidable in general. See, e.g, (67), for a discussion concerning the decidability of language issues such as this.

A *liveness* property is one which states that something must happen. Liveness properties can be viewed as defining words or sentences that must be accepted by the processes defined in a process specification. This notion is expressed in the following definition.

Definition D.4 *Given a process specification P, a liveness property of P is a language L_L defined over the alphabet αP which constrains $L(P)$ such that $L(P) \subseteq L_L$.* □

As is the case with safety properties, this question is undecidable in general. Again, the reader is referred to (67) for a discussion concerning the decidability of liveness properties.

D.4 Recommendations for Future Research

The approach of using derived antecedents to define additional operations could be generalized to include more than one level of process analysis. That is, the approach could be generalized to include a deeper analysis of the flow of data within a CSP process structure to identify invariant properties preserved by sequences of operations. These invariant properties could be used to help establish proofs concerning the satisfaction of input assumptions for those operations consuming data produced by the operation sequence.

The use of derived antecedents to define addition operators poses some interesting questions related to the operational paradigm of the framework described in Chapter II. Specifically, should the failure of proofs of the form defined in Definition D.1 be considered error conditions that the user of the framework must resolve, or should derived antecedents be used to automatically add additional operations to a developing application? The answer to this question, as well as an elaboration of an algorithm to perform automatic insertion of derived operations is left for future research.

It may be possible to define and develop algorithms designed to ensure process specifications satisfy liveness conditions. However, an investigation of this issue is out of scope of this research effort.

Algorithms for ensuring safety properties exist. For example, operating systems make use of resource allocation graphs to ensure that allocation of resources such as line printers and disk drives to processes (applications) will not result in a deadlocked system state. Development and maintenance of resource allocation graphs is accomplished algorithmically. Such algorithms could potentially be adapted for use here. Specifically, events shared between processes could be viewed as shared resources. An allocation graph could be constructed where the nodes of the graph consist of the shared events, and an edge from node n_1 to node n_2 exists if event n_1 can occur only after event n_2. This set of edges could be built through analysis of the process expressions contained within a process specification. Unlike the resource allocation graphs of operating systems, the resource allocation graphs of process specifications must be dynamically analyzed based on the set of enabled events. That is, the occurrence of an event e in the set of enabled events will result in a new set of enabled events, and hence in a new set of edges for the resource allocation graph. Deadlock is possible if there exists a sequence of events in the alphabet of the process specification which results in the formation of a resource allocation graph containing a knot, where a knot is defined to be a subset S of the set of nodes of the graph such that S is closed under the relation defined by the set of edges. Elaboration of this algorithm is left for future research.

Bibliography

1. Agrawala, Ashok, et al. "Domain-Specific Software Architectures for Intelligent Guidance, Navigation, & Control." *The Domain Specific Software Architecture Program, Technical Report CMU/SEI-92-SR-9* 63–72, Carnegie Mellon University, Pittsburgh, PA 15213: Software Engineering Institute, June 1992.
2. Ajisaka, Tsuneo, et al. "An Automatic Program Generation System Based on Function Schema Base." *IEEE COMPSAC 87*. 214–220. 1987.
3. Albus, James, et al. *NSA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)*. Technical Report 1235, National Bureau of Standards, 1989.
4. Allen, Robert and David Garlan. *A Formal Approach to Software Architectures*. Technical Report, School of Computer Science, Carnegie Mellon University, 14 February 1992.
5. Allen, Robert and David Garlan. *Towards Formalized Software Architectures*. Technical Report CMU-CS-92-163, School of Computer Science, Carnegie Mellon University, July 1992.
6. Allen, Robert and David Garlan. "Formalizing Architectural Connection." Submitted to ICSE 16, 1 September 1993.
7. Ambriola, V., et al. "Software Process Enactment in OIKOS." *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*. 183–192. Irvine CA: SIGSOFT Engineering Notes, 1990.
8. Apt, Krzysztof R., et al. "A Proof System for Communicating Sequential Processes." *ACM Transactions on Programming Languages and Systems 2*, 359–385, ACM, July 1980.
9. Baiardi, F. *Debugging Environment*, chapter 8. World Scientific Publishers, 1992.
10. Bailor, Paul D. *Theory for Graph Based Language Specification, Analysis, and Mapping with Application to the Development of Parrallel Software*. PhD dissertation, Air Force Institute of Technology, Wright-Patterson AFB, OH 45433, 1989.
11. Barbacci, Mario R., et al. *Durra: A Task-Level Description Language Reference Manual (Version 3)*. Technical Report CMU/SEI-91-TR-18, ESD-91-TR-18, School of Computer Science, Carnegie Mellon University, December 1991.
12. Barstow, David. "A Perspecitive on Automatic Programming." *Readings in Artificial Intelligence and Software Engineering* edited by Charles Rich and Richard C. Waters, 537–559, Morgan Kaufmann Publishers, Inc., 1986.
13. Barstow, David R. "An Experiment in Knowledge-based Automatic Programming." *Readings in Artificial Intelligence and Software Engineering* edited by Bonnie Lynn Webber and Nils J. Nilsson, 289–312, Morgan Kaufmann Publishers, Inc., 1981.
14. Batory, Don. *A Process and Retrospection on Creating a Domain Model for Avionics Software*. Technical Report ADAGE-UT-93-04, Austin, Texas 78712: Department of Computer Science, The University of Texas, May 1993.

15. Belz, Frank C., et al. "Application of ProtoTech Technology to the DSSA Program." *The Domain Specific Software Architecture Program, Technical Report CMU/SEI-92-SR-9* 77–85, Carnegie Mellon University, Pittsburgh, PA 15213: Software Engineering Institute, June 1992.

16. Berstra, J. A. and J. V. Tucker. "Initial and final algebra semantics for data type specifications: Two characterization theorems," *SIAM J. Comput.*, *12*(2):366–387 (May 1983).

17. Bŏhm, Corrado and Giuseppe Jacopini. "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, *9*(5):366–371 (May 1966).

18. Blaine, Lee, et al. "Specware^{TM} SLANG Core 3 User Manual (Draft)." Kestrel Institute, 3260 Hillview Avenue, Palo Alto CA 94304, June 1994.

19. Blaine, Lee, et al. "Specware^{TM} SLANG Core 3 SLANG Language Manual (Draft)." Kestrel Institute, 3260 Hillview Avenue, Palo Alto CA 94304, June 1994.

20. Blaine, Lee, et al. "SLANG Language Manual, Specware^{TM} Version Core 4." Kestrel Institute, 3260 Hillview Avenue, Palo Alto CA 94304, October 1994.

21. Blaine, Lee and Allen Goldberg. *Verifyably Correct Data Type Refinement.* Technical Report KES.U.90.5, Palo Alto, CA: Kestrel Institute, December 1990.

22. Brachman, R. J. and J. Schmolze. "An Overview of the KL-ONE Knowledge Representation System," *Cognitive Science*, *9*(2) (1985).

23. Burstall, R. M. and J. A. Goguen. "Putting Theories Together to make Specifications." *Proceedings of the 5th International Joint Conference on Artificial Intelligence 2*. 1045–1058. 1977.

24. Cheng, Jingwen. "Parameterized Specifications for Software Reuse," *Software Engineering Notes (ACM SIGSOFT)*, *17*(4):53–59 (May 1992).

25. Coglianese, Louis H. and Raymond Szymanski. "DSSA-ADAGE: An Environment for Architecture-based Avionics Development." *DSSA-ADAGE Collected Papers of the Domain-Specific Software Architectures (DSSA) Avionics Domain Application Generation Environment (ADAGE)* MD 0210, Owego NY 13827: IBM Federated Sector Company, May 1993.

26. Cohen, Daniel I.A. *Introduction to Computer Theory.* John Wiley and Sons, 1991.

27. DeCegama, Angel L. *The Technology of Parallel Processing: Parallel Processing Architectures and VLSI Hardware.* Prentice Hall, 1989.

28. Devanbu, Premkumar, et al. "A Knowledge-Based Software Information System." *IJCAI-89 Proceedings of the 11th International Joint Conference on Artificial Intelligence*, edited by N. S. Sridharan. 110–115. Palo Alto, CA: Morgan Kaufman, 1987.

29. Devanbu, Premkumar, et al. "LaSSIE: A Knowledge-Based Software Information System," *Communications of the ACM*, *34*(5):34–49 (May 1991).

30. D'Ippolito, Richard S. "Using Models in Software Engineering." *Tri-Ada '89*. 256–265. 1989.

31. Doubleday, Dennis L., et al. *Building Distributed Ada Applications from Specifications and Functional Components*. Technical Report CMU/SEI-91-TR-22, Software Engineering Institute, Carnegie Mellon University, December 1991.

32. Doubleday, Dennis L., et al. *A Descrition of Cluster Code Generated by the Durra Compiler*. Technical Report CMU/SEI-91-TR-19, ESD-91-TR-19, School of Computer Science, Carnegie Mellon University, December 1991.

33. Dougherty, Edward R. and Charles R. Giardina. *Mathematical Methods for Artificial Intelligence and Autonomous Systems*. Prentice Hall, 1988.

34. Fiadeiro, J. and T. Maibaum. "Describing, Structuring and Implementing Objects." *REX School/Workshop on Foundations of Object-Oriented Languages489*. Lecture Notes in Computer Science. 274–310. Noordwijkerhout, The Netherlands: Springer-Verlag, May 1990.

35. Fiadeiro, José and Amílcar Sernadas. "Structuring Theories on Consequence." *Recent Trends in Data Type Specification, 5th Workshop on Specification of Abstract Data Types*. Lecture Notes in Computer Science, Volume 332. 44–72. Gullane, Scotland: Springer-Verlag, September 1987.

36. Fredlund, Lars-ake, et al. "An Implementation of a Translational Semantics for an Imperative Language," *Lecture Notes in Computer Science*, *458*:246–262 (1990).

37. Garlan, David and David Notkin. "Formalizing Design Spaces: Implicit Invocation Mechanisms." To appear in "Proceedings of VDM '91: Formal Software Development Methods", October 1991.

38. Garlan, David and Mary Shaw. "An Introduction to Software Architecture." To appear in *Advances in Software Engineering and Knowledge Engineering*, Volume I, World Scientific Publishing Company.

39. Garlan, David, et al. "Experiences with a Course on Architectures for Software Systems." *Proceedings of the Sixth SEI Conference on Software Engineering Education*. October 1992.

40. Goguen, J. A. and R. M. Burstall. "Some Fundamental Algebraic Tools for the Semantics of Computation, Part 1: Comma Categories, Colimits, Signatures and Theories," *Theoretical Computer Science*, *31*(2):175–209 (1884).

41. Goguen, J. A. and R. M. Burstall. "Introducing Institutions." *Logic of Programs, Lecture Notes in Computer Science, Volume 164* edited by E. Clarke and D. Kozen, 221–255, Springer, 1984.

42. Goguen, Joseph A. "Mathematical Representation of Hierarchically Organized Systems." *Global Systems Dynamics*. 112–128. 1969.

43. Goguen, Joseph A. "Reusing and Interconnecting Software Components," *IEEE Computer*, 16–28 (February 1986).

44. Goguen, Joseph A. and Timothy Winkler. *Introducing OBJ3*. Technical Report, 333 Ravenswood Ave, Menlo Park, CA: Computer Science Laboratory SRI International, August 1988.

45. Goguen, Jospeh. "Principles of Parameterized Programming." *Software Reusability, Volume I, Concepts and Models* chapter 7, 159–225, New York: ACM Press, 1989.

46. Goldberg, Allen T. "Knowledge-Based Programming: A Survey of Program Design and Construction Techniques," *IEEE Transactions on Software Engineering*, *SE-12*(7):752–768 (July 1986).

47. Group, Crossbow-S Architecture Technical Working. *System Concept Document for the Joint Modeling and Simulation System (JMASS) Program, Version 1.1*. Technical Report, ASD/RWWW, WPAFB OH 45433-6503, 15 November 1991.

48. Guttag, John V. and James J. Horning. *LARCH: Languages and Tools for Formal Specification*. New York: Springer-Verlag, 1993.

49. Hayes, Ian, editor. *Specification Case Studies*, chapter Specification of the UNIX Filing System, 91–140. Prentice-Hall International Series in Computer Science, 1987. Carroll Margan and Bernard Sufrin, Authors.

50. Hennessy, M. C. B. and W. Li. "Translating a Subset of Ada into CCS." *IFIP Working Conference on Formal Description of Programming Concepts II*, edited by Dines Bjorner. 227–249. North-Holland Publishing Company, 1982.

51. Hennessy, Matthew. *Algebraic theory of processes*. MIT Press, 1988.

52. Hoare, C. A. R. *Communicating Sequential Processes*. Prentice/Hall International, 1985.

53. Hoshino, Hiroshi, et al. "Software Development Method with Logic Language." *IEEE COMPSAC 87*. 613–619. 1987.

54. Jalote, Pankaj. "Functional Refinement and Nested Objects for Object Oriented Design," *IEEE Transactions on Software Engineering*, *15*(3):264–270 (March 1989).

55. Jüllig, Richard. "Applying Formal Software Synthesis." To appear in IEEE Software, 1993.

56. Jüllig, Richard and Yellamraju V. Srinivas. "Diagrams for Software Synthesis." *The Eighth Knowledge-Based Software Engineering Conference*. 10–19. Los Alamitos CA: IEEE Computer Society Press, September 1993.

57. Jüllig, Richard and Yellamraju V. Srinivas. "Describing Software Architectures Formally: A Case-Study Using KWIC." Kestrel Institute, Palo Alto, CA 94304, April 1994.

58. Kang, Kyo C. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

59. Lafontaine, Christine, et al. "An Experiment in Formal Software Development: Using the B Theorem Prover on a VDM Case Study," *Communications of the ACM*, *34*(5):62–87 (May 1991).

60. Lane, Thomas G. *Studying Software Architecture Through Design Spaces and Rules.* Technical Report CMU/SEI-90-TR-18, ESD-90-TR-219, Carnegie Mellon University, Pittsburgh, PA 15213: Software Engineering Institute, November 1990.

61. Lee, Kenneth J., et al. *Model-Based Software Development (DRAFT).* Technical Report CMU/SEI-92-SR-00, ESD-92-SR-00, Carnegie Mellon University, Pittsburgh, PA 15213: Software Engineering Institute, December 1991.

62. Levi, Shem-Tov and Ashok K. Agrawala. *Real Time System Design.* McGraw-Hill Publishing Company, 1990.

63. Lowry, Michael R. and Robert D. McCartney, editors. *Automating Software Design.* AAAI Press and the MIT Press, 1991.

64. Luckham, David C. and James Vera. "μ-Rapide: An Executable Architecture Definition Language." DARPA funded project under ONR contract N00014-90-J-1232 and AFOSR under Grant AFOSR91-0354, 7 April 1993.

65. MacLane, S. *Categories for the Working Mathematician.* New York: Springer-Verlag, 1971.

66. MacLane, Saunders and Garrett Birkhoff. *Algebra.* Chelsea Publishing Company, 1993.

67. Manna, Zohar. *Mathematical Theory of Computation.* McGraw-Hill Computer Science Series, 1974.

68. Mash, Frederick C., editor. *Webster's Ninth New Collegiate Dictionary.* Merriam-Webster Inc., 1985.

69. Mattala, LtCol Erik and Marc H. Graham. "The Domain-Specific Software Architecture Program," *Cross Talk: The Journal of Defense Software Engineering*, 19–21 (October 1992).

70. McArthur, D. "ENCORES: an environment for constructing or reasoning with engineered software." *Software Engineering Environments* edited by Pearl Brereton, chapter 6, 69–78, Ellis Horwood Limited, 1988.

71. McIlroy, M. D. "Mass Produced Software Components," *Software Engineering Concepts and Techniques*, 88–98 (1969). Paper presented at the 1969 NATO Conference on Software Engineering.

72. Mettala, LtCol Erick and Marc H. Graham. *The Domain Specific Software Architecture Program.* Technical Report CMU/SEI-92-SR-9, Carnegie Mellon University, Pittsburgh, PA 15213: Software Engineering Institute, June 1992.

73. Neighbors, James M. "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering*, *10*(5):564–574 (September 1984).

74. Nii, H. P. "Blackboard Systems Parts 1 & 2," *AI Magazine*, 7(3 & 4):38–53, 62–69 (1986).

75. Paige, Robert and Shaye Koenig. "Finite Differencing of Computable Expressions," *ACM Transactions on Programming Languages and Systems*, *4*(3):402–454 (July 1982).

76. Perry, Dewayne E. and Alexander L. Wolf. "Foundations for the Study of Software Architecture," *Software Engineering Notes (ACM SIGSOFT)*, *17*(4):40–52 (October 1992).

77. Place, Patrick R., et al. *Survey of Formal Specification Techniques for Reactive Systems*. Technical Report CMU/SEI-90-TR-5, ESD-TR-90-206, Carnegie Mellon University, Pittsburgh, PA 15213: Software Engineering Institute, May 1990.

78. Platek, Richard and James H. Taylor. "Domain-Specific Software Architectures for Hybrid Control." *The Domain Specific Software Architecture Program, Technical Report CMU/SEI-92-SR-9* 73–76, Carnegie Mellon University, Pittsburgh, PA 15213: Software Engineering Institute, June 1992.

79. Polak, Wolfgang, "CIDL." Course notes and slides.

80. Polak, Wolfgang and Henson Graves. *The CIDL Language User Manual, Version 0.9*, May 1992.

81. Pressburger, Thomas T., et al. *KIDS Manual, Kestrel Interactive Development System, Version 1.0*. Kestrel Institure, 3260 Hillview Avenue, Palo Alto CA, September 1991.

82. Prieto-Díaz, Rubén. "Domain Analysis for Reusability." *Proceedings of COMPSAC '87*. 23–29. 1987.

83. Prieto-Díaz, Rubén. "Classification of Reusable Modules." *Software Reusability, Volume I, Concepts and Models* chapter 4, 99–123, ACM Press, New York, 1989.

84. Prieto-Díaz, Rubén. "Implementing Faceted Classification for Software Reuse," *Communications of the ACM*, *34*(5):88–97 (May 1991).

85. Prieto-Díaz, Rubén. "Making Software Reuse Work: An Implementation Model," *Software Engineering Notes*, *16*(3):61–73 (July 1991).

86. Prieto-Díaz, Rubén and James M. Neighbors. "Module Interconnection Languages," *Journal of Systems and Software*, *6*(4):307–334 (1986).

87. Purtilo, James M. *The Polylith Software Bus*. Technical Report UMIACS-TR-90-65, CS-TR-2469, University of Maryland: Institute for Advanced Computer Studies and Department of Computer Science, May 1990.

88. Reasoning Systems Inc., 3260 Hillview Avenue, Palo Alto CA, 94304. *Refine User's Guide*.

89. Rich, Charles and Yishai Feldman. "Seven Layers of Knowledge Representation and Reasoning in Support of Software Development," *IEEE Transactions on Software Engineering*, *18*(6):451–469 (June 1992).

90. Rumbaugh, James, et al. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

91. Sannella, Donald and Andrzej Tarlecki. "Toward Formal Development of Programs from Algebraic Specifications: Implementations Revisited," *Acta Informatica*, *25*:233–281 (1988).

92. Setliff, Dorothy and Rob Rutenbar. "Knowledge Representation and Reasoning in a Software Synthesis Architecture," *IEEE Transactions on Software Engineering*, *18*(6):523–533 (June 1992).

93. Shaw, Mary. "Larger Scale Systems Require Higher-Level Abstractions." *Fifth International Workshop on Software Specification and Design*. 143–146. 1989.

94. Shaw, Mary. "Heterogeneous Design Idioms for Software Architecture." *6th International Workshop on Software Specification and Design*. 158–165. 1991.

95. Shaw, Mary. *Software Architecture for Shared Information Systems*. Technical Report SMU/SEI-93-TR-3, ESC-TR-93, Software Engineering Institute, March 1993.

96. Shoham, Yoav. *Reasoing about Change*. MIT Press, 1988.

97. Silberschatz, A., et al. *Operating System Concepts* (Third Edition). Addison Wesley, 1991.

98. Smith, Douglas R. "Top-Down Synthesis of Divide-and-Conquer Algorithms." *Readings in Artificial Intelligence and Software Engineering* edited by Charles Rich and Richard Waters, 35–61, Morgan Kaufmann Publishers, Inc., 1986.

99. Smith, Douglas R. "Applications of a Strategy for Designing Divide-and-Conquer Algorithms," *Science of Computer Programming*, *8*:213–229 (1987).

100. Smith, Douglas R. "KIDS: A Semiautomatic Program Development System," *IEEE Transactions on Software Engineering*, *16*(9):1024–1043 (September 1990).

101. Smith, Douglas R. "Structure and Design of Problem Reduction Generators." *Constructing Programs from Specifications* 91–124, Elsevier Science Publishers B.V. (North-Holland), 1991.

102. Smith, Douglas R. *Constructing Specification Morphisms*. Technical Report KES.U.92.1, 3260 Hillview Avenue, Palo Alto, CA, 94304: Kestrel Institute, April 1992.

103. Smith, Douglas R. *Structure and Design of Global Search Algorithms*. Technical Report KES.U.87.11, Palo Alto CA: Kestrel Institute, 1992.

104. Smith, Douglas R. "Transformational Approach to Transportation Scheduling." *KBSE '93: The Eighth Knowledge Based Software Engineering Conference*. 60–68. IEEE Computer Society Press, 1993.

105. Smith, Douglas R. and Michael R. Lowry. "Algorithm Theories and Design Tactics," *Science of Computer Programming*, *14*:305–321 (1990).

106. Srinivas, Yellamraju V. *Category Theory Definitions and Examples*. Technical Report 90-14, Department of Information and Computer Science, Irvine, CA: University of California, 18 February 1990.

107. Srinivas, Yellamraju V. *Algebraic Specification: Syntax, Semantics, Structure*. Technical Report 90-15, Department of Information and Computer Science, Irvine, CA: University of California, 19 June 1990.

108. Srinivas, Yellamraju V. "Algebraic Specification for Domains." *Domain Analysis and Software Systems Modeling* 90–124, IEEE Computer Society Press, 1991.

109. Srinivas, Yellamraju V., "Personal conversation with Yellamraju V. Srinivas of Kestrel Institute, Palo Alto CA 94304," March 1993.

110. Stanat, Donald F. *Discrete Mathematics in Computer Science*. Prentice Hall, 1977.

111. Thatcher, J. W., et al. "Data-type specification: Parameterization and the power of specification techniques," *ACM Transactions on Programming Languages and Systems*, *4*(4):711–732 (October 1982).

112. Thomas, Donald E., et al. "A Model and Methodology for Hardware-Software Codesign," *IEEE Design & Test of Computers*, *10*(3):6–15 (September 1993).

113. Thomas, J. W. *Module Interconnection in Programming Systems Supporting Abstraction*. PhD dissertation, Brown University, June 1976.

114. Tracz, William and Lou Coglianese. "An Avionics Domain-Specific Software Architecture," *Cross Talk: The Journal of Defense Software Engineering*, 22–24+ (October 1992).

115. Tracz, William J. *Formal Specification of Parameterized Programs in LILEANNA, Sixth Draft*. PhD dissertation, Stanford University, May 1993.

116. Tsang, Edward. *Foundations of Constraint Satisfaction*. San Diego, CA: Academic Press Inc., 1993.

117. Turski, Wladyslaw M. and Thomas S. E. Maibaum. *The Specification of Computer Programs*. Addison-Wesley Publishing Company, 1987.

118. van Glabbeek, R. J. *The Linear Time-Branching Time Spectrum*. Technical Report CS-R9029, Centre for Mathematics and Computer Science, Amsterdam, 1990.

119. Van Horebeek, Ivo and Johan Lewi. *Algebraic Specifications in Software Engineering*. Springer-Verlag, 1989.

120. Vestal, Steve. "A Cursory Overview and Comparison of Four Architecture Description Languages." Honeywell Sytems and Research Center, (612) 951-7049, 18 February 1993.

121. Vestal, Steve. *Software Programmer's Manual for the Honeywell Aerospace Compiled Kernel (MetaH Language Reference Manual), Multiprocessor Version α.5, Draft*. Honeywell Systems and Research Center, 3660 Technology Drive, Minneapolis, MN 55418, 18 June 1993.

122. Wallnau, Kurt C., "The CARDS Program and Research Opportunities for Architecture-Centered, Reuse Library Assisted Software Reuse." Briefing slides, 6 May 1993. Paramax Systems Corporation.

123. Wallnau, Kurt C., "CARDS Reuse Technology Principles and Applications of Model-Based Reuse." Briefing, 10 August 1993. Paramax Systems Corporation.

124. Wang, T. C. *Reacto-Verifier 1.0 User's Guide*. Kestrel Institute, 3260 Hillview Avenue, Palo Alto CA, November 1990.

125. Waters, Richard C. "The Programmer's Apprentice: A Session with KBEmacs." *Readings in Artificial Intelligence and Software Engineering* edited by Charles Rich and Richard C. Waters, 351–375, Morgan Kaufmann Publishers, Inc., 1986.

126. Wenban, Alan S., et al. "Codeign of Communication Protocols," *Computer*, 46–52 (December 1993).

127. Wileden, Jack C., et al. "Specification-Level Interoperability," *Communications of the ACM*, *34*(5):88–97 (May 1991).

128. Wirsing, Martin. "Algebraic Specification." *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics* edited by J. van Leeuwen, chapter 13, 675–788, Elsevier Science Publishers B.V., 1990.

129. Young, Frank C. D. "Constraint Directed Hardware Software Codesign." Dissertation Prospectus, August 1994.

Vita

Captain Mark Gerken [redacted] In February 1985 he married Debora Sue Mallory. He received a Bachelor of Science degree in Computer Engineering from Iowa State University in May of the following year. After receiving his commission through ROTC, he was assigned to 82nd Student Squadron at Williams AFB, Arizona, where he attended flight training. He was later assigned to the Directorate of Engineering Reliability within the Acquisition Logistics Division at Wright-Patterson AFB Ohio, and later transferred to the C-17 System Program Office at Wright-Patterson AFB. In May of 1990, Mark entered the Air Force Institute of Technology at Wright-Patterson AFB in pursuit of a Master of Science degree in Computer Engineering and continued into the Ph.D. program. Captain Gerken has three children: [redacted]

Permanent address: [redacted]

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1995	Ph.D. Dissertation

4. TITLE AND SUBTITLE

FORMAL FOUNDATIONS
FOR THE SPECIFICATION OF
SOFTWARE ARCHITECTURE

5. FUNDING NUMBERS

6. AUTHOR(S)

Mark J. Gerken, Captain, USAF

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

Air Force Institute of Technology
WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/DSG/ENG/95M-01

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

AFOSR/NM
110 Duncan Ave, Suite B115
Bolling AFB, DC 20332

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This investigation establishes a formal foundation for software architecture that allows for the specification of non-trivial software systems using well-founded, consistency preserving techniques. Two fundamental problems were addressed: how to define architectures formally, and how architecture theories can be applied in specification construction. The initial stages of this investigation sought to establish a formal relationship between functional specifications of behavior and specifications defining system structure. Experimental results lead to the conclusion that architectures defining the structure of functional operations can be defined using functional logic, but more complex architectures require a separate process logic. A process logic based on Hoare's Communicating Sequential Processes (CSP) was selected for representing and reasoning about system structure and was used in the definition of a process-based specification development system. Specifically, CSP was used to define a category of process-based specifications, allowing well-founded specification construction techniques such as colimits, and interpretations to be applied to the construction of consistent software architecture. Architecture theories expressed in terms of functional and process-based specifications were defined, and translations between these architecture theories were investigated. A feasibility analysis demonstrated that architecture theories can be used to develop specifications for non-trivial applications.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software Architecture, Algebraic Specification, Formal Methods	327
	16. PRICE CODE

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