A Framework for Component Specification

Fred Loney
B.A., Oakland University, 1975
B.S., University of Washington, 1977
M.B.A., University of Washington, 1984

A thesis submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

April 1995

© Copyright 1995 by Fred Loney
All Rights Reserved

The thesis "A Framework for Component Specification" by Fred Loney has been examined and approved by the following Examination Committee:

T. Lougenia Anderson
Adjunct Faculty
Sequent Corporation
Thesis Research Adviser

Richard Kieburtz
Professor

David Maier
Professor

Dedication

I dedicate this thesis to the memory of Kathy Gervais, for her example of dedication, perseverance and grace under pressure.

Acknowledgements

I would like to thank my principal thesis adviser, Lougie Anderson, for editing, paring and refining the initial drafts into a publishable form; my thesis readers, Dick Kieburtz and Dave Maier, for their helpful suggestions; my parents, for their positive example and unflagging support; and especially Melinda, for her indulgence, patience and understanding.

Contents

Dedication iv

Acknowledgements v

Abstract xiii

1 Introduction 1

2 Object Specification 3

2.1 Information Models 3

2.2 The Role of a Framework in Object Specification 6

2.3 Comparison to Previous Work 9

3 Component Definition 12

3.1 Component Schemes 12

3.2 Component Structure 15

3.3 Behavioral Specification 21

3.4 Use and Identity 24

3.5 Change and Identity 27

4 **Behavioral Dynamics** 29

4.1 Method specification 29

4.2 Method activation 34

4.3 Monitoring actions 43

4.4 Agents: frames with an attitude 45

5 **Object Management** 49

5.1 Dimensions of Software Object Management 49

5.2 Functional Partitioning 51

5.3 Task Partitioning 52

5.4 Containment Hierarchy 55

5.5 Managing change 59

5.6 Reference Resolution 66

6 **A Component Specification Framework Prototype** 71

6.1 A specification example 72

6.2 Scheme representation 84

7 **Conclusions** 88

7.1 The Role of a Specification Framework Reconsidered 88

7.2 Framework Implementation Lessons 90

7.3 Future Work 91

7.4 Summary 93

Bibliography 94

List of Tables

2.1 Information modeling perspectives 4

2.2 Application characteristics 6

3.1 Associations between Frames 14

4.1 Specification-Computation analogues 34

6.1 Script for the Add Waitlist Feature use scenario 72

6.2 Meros classes by category 87

List of Figures

3.1 Examples of frames 13

3.2 Manager frame 14

3.3 Manager and Employee views 16

3.4 Hotel frame example 19

3.5 Component Structure 21

3.6 Reserve rule 22

3.7 Hold constraint 23

3.8 Manager delegation to Employee 23

3.9 Employee delegation to a source code interface 24

3.10 Fregean distinction 25

3.11 Denotation and change relationship 28

4.1 Reserve rule 30

4.2 Accept rule 30

4.3 Reservation status state variable domain values 31

4.4 Room occupancy status domain 31

4.5 Hold rule 32

4.6 Availability rule 32

4.7 Deny reservation action 33

4.8 Rule for binding a value to a slot 35

4.9 Resolution of a bound value 36

4.10 Reserve subactions 39

4.11 Overbook rule 40

4.12 Confirmation constraint 42

4.13 Reservation add exception rule 42

4.14 Hotel Full monitor on reserve 44

4.15 Hotel Full action 44

4.16 Hotel Full monitor on Occupancy 45

4.17 Overbooking Authorization agent 46

4.18 Overbooking Authorization rule 47

4.19 Rescind compensatory action 48

5.1 Hotel component in two libraries 52

5.2 Add Waitlist Feature activity 53

5.3 Example of projects 54

5.4 Membership references for the Add Waitlist Feature activity 56

5.5 Effect of membership properties on change visibility 58

5.6 Workspace structure 60

5.7 Workspace version graph visibility 62

5.8 Environment example 64

5.9 Configurable reference example 67

5.10 Reference resolution through a context map 69

6.1 Open Activity Editor 73

6.2 Add New Activity 73

6.3 Select Hotel Component 74

6.4 Browse Hotel History 75

6.5 Evolve Hotel History 76

6.6 Jay's active version 77

6.7 Open Hotel component 78

6.8 Create waitlist method 79

6.9 Open Reservation scheme 80

6.10 Copy Reservation frame 81

6.11 Create waitlist slot 82

6.12 Visibility of promoted change 83

6.13 Meros tool architecture 84

6.14 Scheme representation in Meros 85

6.15 Configurable frame 86

Abstract

A Framework for Component Specification

Fred Loney, M.S.
Oregon Graduate Institute of Science & Technology, 1995

Supervising Professor: T. Lougenia Anderson

This thesis demonstrates a practical approach to software specification that addresses data abstraction, object interaction, design evolution, incremental refinement, system partitioning and traceability. A framework is presented for representing and managing the specification of reusable components. A *component* encapsulates software artifacts that manifest an application concept. The structure and behavior of a component is defined in behavioral templates, or *schemes*, for use in specifications. The dynamic aspects of behavior are modeled by action rules and constraints, resulting in a computational model for specification executability. Facilities are provided for controlling the evolution and selection of components. Meros, a prototype framework implementing these concepts, is presented with examples.

Chapter 1

Introduction

Information systems model application domains in terms of programming languages. Interesting application domains are subtle, diverse and changeable; programming languages are prosaic, uniform and inflexible. This difference accounts for both the success and failure of software. Software is successful at performing structured, repeatable tasks quickly and reliably. Software is less successful at handling situations that do not fit comfortably within a simplistic behavioral model.

Design is the process of transforming a conceptual model to an implementable specification. The software designer reduces the tangled web of interactions among complex real-world entities to a set of discrete operations on a handful of stable datatypes. A *useful* design carves out an appropriate context from the problem space. A *faithful* design preserves essential semantics of the problem domain. A *full* design captures all of the relevant features within the system boundary. The goal of design is to transform the conceptual model to an implementation specification by a useful, faithful and full mapping of the problem domain to a solution space.

Unfortunately, the means of design is inherently at odds with its end. The design process is exploratory, tentative and iterative. The dilemma of design is that it typically deals with a "wicked problem", a problem that is not well-defined until its solution is

understood. Restated, if the solution is not understood, then the problem is not completely defined. Thus, design entails problem solution as well as problem definition. The task of the hapless designer is to achieve the end of a well-behaved design transformation by a process of circuitous tinkering.

This thesis describes an approach to specifying information systems and a lightweight tool for doing so. The focus is on the design of object-oriented applications, although the results generalize to the complete software development life-cycle of any information system application. The design transformation is viewed as a gradual refinement of specifications from concept to implementation. A specification is organized as a graph structure with connections between object templates. The dynamics of object interactions are modeled by actions. Meros, a tool for browsing specifications, is introduced as a non-intrusive means of visualizing the specification that respects the exploratory nature of the design process.

The major contribution of this thesis is an open, extensible representation for object specifications that addresses data abstraction, object interaction, design evolution, incremental refinement, system partitioning and traceability. The desire for rigor and conceptual integrity is balanced by the need for a concrete, straightforward, non-intrusive approach that is compatible with existing object-oriented programming languages and development techniques.

The thesis is organized as follows: Chapter 2 describes the role of a specification framework in the software development process. Chapter 3 presents the concepts pertinent to modeling information systems by incremental refinement of component specifications. The dynamic aspects of object interaction are discussed in Chapter 4. Chapter 5 places object specifications in the context of cooperative development. Chapter 6 illustrates these concepts in Meros, a proof-of-concept prototype for managing object specifications. Finally, Chapter 7 summarizes the accomplishments of the thesis and identifies areas for future research.

Chapter 2

Object Specification

2.1 Information Models

An *information model* is an abstract representation of relevant concepts from an application domain or a software system that applies to that domain. It provides a plan or reference point for realizing the desired system behavior. Models take different forms with varying degrees of specificity and realizability. Traditionally, three types of models are distinguished by their correspondence with three distinct phases of software development: analysis, design and implementation. An analysis model describes logical concepts intended to specify *what* the system is to accomplish. The analysis model serves as a formal problem description that is independent of the implementation architecture. A design model changes the focus to *how* to implement the desired system behavior. The design model factors implementation constraints and available components into a realizable model to guide further development. The system architecture is described in abstract terms with a close correspondence to implementation constructs. Finally, an implementation model prescribes the solution in source code. Programming languages are a highly codified but imminently realizable implementation modeling language, from which a compiler or interpreter can generate a concrete realization of the information system.

Modeling traditionally addresses three perspectives: data, function and control. Techniques and conventions have evolved for each perspective (Table 2.1). For example,

	Analysis	Design	Implementation
Data	Entity, Relationship	Database Schema	Record Structure
Function	Data Transform	Module Architecture	Procedure
Control	Control Transform	State Transition	Task, Semaphore

Table 2.1: Information modeling perspectives

a semantic data model, typified by the Entity-Relationship data model [Che76, TYF86], captures structural information about concepts as a bi-partite undirected graph of entities and relationships. A database schema factors implementation considerations into the semantic data model. The schema is manifested as sets of records, or fixed tuples of labelled, typed, single-valued fields.

Structural data relationships are complemented with functional models in each phase of the development process. The conceptual flow of data is analyzed by processes or data transforms [Mar78]. Behavior is allocated to modules in an architecture diagram or structure chart [PJ88], and realized in the procedure definition of a programming language. Control transforms complement data transforms to describe control flow [WM85, HP87]. State-transition diagrams or matrices serve as a general-purpose mechanism to elaborate system event-response behavior [Har88]. Control behavior is implemented using language-dependent communication and synchronization features [GJ82].

These techniques are directed toward applications characterized by processes acting on stable, passive, homogeneous tabular data structures. Object-oriented models, on the other hand, are intended to address the needs of applications with heterogeneous

entities and complex relationships such as computer-aided design and manufacturing (CAD/CAM), computer-aided software engineering (CASE), knowledge-based systems, and mutimedia document management [Kat87, Ber87, SB86, GT83]. Object orientation encapsulates both data and functions in class objects responsible for performing operations according to a well-defined interface [Cox86, Weg90]. The class provides a public interface and hides a private implementation. This separation of concerns promotes information hiding and ease of maintenance by localizing data-function dependencies in the subject class. Inheritance provides a compatibility relationship that is used to validate semantic constraints and share behavior and properties. Inheritance facilitates component reuse and economy of representation.

Object-oriented applications model conceptual entities of diverse size, format and behavior. Data manipulation is typically graph-oriented rather than set-oriented. Composition hierarchies are constructed of components within composite objects. Changes occur over long sessions and often have a cascading effect. Table 2.2 summarizes the differences between object-oriented and record-oriented applications. These characteristics will increasingly characterize conventional record-oriented applications as well, as these applications become more complex and flexible in the services they provide and activities they support.

A goal of object orientation is to reduce the semantic gap between a concept in the application domain and its realization in a programming language, as befits its origins in simulation [DMN68]. This blurs the analysis-design-implementation distinction. The analysis-design distinction is already on shaky ground ontologically[1], serving largely as

[1] The problem lies in the inadequacy of language–any language–for formulating statements about the content of what we perceive:

> Even in describing things we simply go on saying more about *how* they are and not even in this case get to *what*. The *what* always *turns into the how* because the *content* of the world does not in any circumstances get into the propositions.
>
> — H. L. Finch on Wittgenstein [Fin71].

Record-oriented applications	Object-oriented applications
Few, fixed types	Many, extensible types
Many small records	Few large records
Range and size limitations	"Long fields" of indefinite size, format
Tabular organization	Hierarchical organization
Set processing	Graph-walking
Homogeneous sets	Heterogeneous structures
Atomic entities	Composite objects
Register simple events	Relate complex interconnections
Update in place	Versioning
Short-lived locks	Check out/check in
Flat, serializable transactions	Nested transactions
Abort/restart conflict resolution	Cooperative reconciliation
Discrete events	Cascading effect of changes
Definite actions	Iterative, tentative session
User-assigned key identifiers	System-assigned, persistent OOPs
Single-type classification	Subtyping
Passive data store	Data encapsulation
Process-based invocation	Data-centered tool invocation
View definition	Behavior refinement

Table 2.2: Application characteristics

an artificial heuristic to defer premature design decisions. This thesis eschews these semantic tarpits by presenting a uniform representation of information models which may be refined in several stages, individually or as a group. This more accurately reflects practice [Dav88], especially with evolutionary software development approaches [Boe88]. (For a contrary argument, see [HG93].)

2.2 The Role of a Framework in Object Specification

The development of object-oriented applications reflects the characteristics of these applications. A large number of software artifacts are highly interrelated in a dynamic fashion. This complexity places demands on the development infrastructure. A *framework* helps manage system complexity by providing a set of interrelated services used

to build applications [ECM90]. It is the scaffolding for constructing and deploying integrated toolsets. A specification framework is used to manage specifications; it helps organize the application environment by providing the following services:

1. assembling specifications in reuse libraries
2. correlating specifications with software development artifacts
3. managing composite versions and configurations
4. controlling change of an evolving application
5. binding abstract descriptions to prototypical instances
6. activating specifications to drive workflow activities

One may distinguish two popular notions of a framework, a framework-in-the-large and a framework-in-the-small. A framework-in-the-large is a set of subsystems and applications that serves as the basis for enterprise-wide software development environments [Mas87, Ste87]. It integrates large-grained data and tools; an object typically corresponds to a file or set of files; a tool typically runs as an independent process. Integration may range from loose encapsulation to tight integration. Loose encapsulation entails placing a wrapper around tools to direct the flow of input and output information. Artifacts of the tool execution, typically files, are described and tracked by the framework. By contrast, a tightly integrated tool relies on framework services for fundamental user interface, data management and inter-tool communication operations [Was90, TN92].

A framework-in-the-small is a federation of objects and related methods used to perform a distinguished set of services [Deu89]. It represents a well-defined functional component with clear responsibility and integrated actions. The exemplar for this notion of framework is the Smalltalk-80[2] Model-View-Controller paradigm [Gol84]. The model,

[2]Smalltalk-80 is a trademark of Xerox Corporation.

view and controller represent the semantic, presentation and input-handling aspects, respectively, of an application object. A view is registered with the model as an interested party. The controller signals important input events to the model, which interprets the input event and broadcasts a message to interested views.

This thesis attempts to bridge the gap between a framework-in-the-large and a framework-in-the-small. The framework we present is neither as comprehensive as a framework-in-the-large nor as specialized as a framework-in-the-small. Rather, it serves to organize software objects developed using these frameworks into reusable abstractions, or *components*, that can be assembled into applications and applied to business tasks. This focus is consistent with our intent of exploring the largely uncharted territory between programming-in-the-small and programming-in-the-large. The goal is to enable the construction of applications from reusable components with a well-defined interface. The specification framework describes the component interface and behavior, guides the application design and tracks the evolution of a system from textual requirements to executable programs. It serves as the basis for developing advanced control applications, such as workflow automation, that rely on the flexible activation of autonomous, heterogenous components.

The framework provides the capability to faithfully model the system under construction as an integral part of incremental, evolutionary development. *Specifications* describe the architecture and dynamics of object representations. The process is directed toward developing libraries of reusable software components for ongoing medium-to-large-scale software projects. A component has one or more *views*; each view has an interface specification, acting as a behavioral template, and can encapsulate an artifact, such as a requirement statement, analysis model, source code, object module, test suite or test result. Specifications are developed in the context of a *workspace* that delimits the behavioral templates that can be used within the framework. The workspace defines a context for change scoped by a given activity. The evolution of components is tracked in *configurations* that relate consistent versions of components and views. For example,

adding a feature to a Reservation System application might define a context with an associated workspace and a configuration that includes a new version of the `csource` view of a `Hotel` component with modifications to support the new feature.

The framework does not dictate a symbolism, methodology or technique to use in design. Rather, it integrates the representations of an object produced by tools and provides a common representation of shared concepts, to the extent that this information is exposed in views. The information content is specified in the object schema. This schema and additional information about the object, such as records of design activity, annotations and the location of design representations, constitutes the *metadata*, or information about data, associated with the object. The metadata is stored in a metadatabase managed by the framework.

A component-specification framework is merely one facet of a complete software development environment. It does not supplant programming languages or application toolkits. It does not provide user interface, data management, tool encapsulation or inter-tool communication services. Its value lies in playing a non-intrusive, supporting role in the design of components and their use in applications.

2.3 Comparison to Previous Work

This work draws on previous work in semantic data models, abstract specification and configuration management. Semantic data models [PM88] form the basis for the static representation of component structure. The Semantic Database Model [HM81] pioneered the categorization of abstract application entities reflecting relativistic perspectives of an underlying range of record-based database values. The functional data model, typified by DAPLEX [Shi81], represents semantic properties of an application as functions between sets of entities. Function names may be overloaded and disambiguated by entity roles, and derived functions specify conceptual abstractions based on the values of other

properties.

This thesis extends these semantic data models with behavioral descriptions of objects. Similarly, ADAPLEX [SFL81] and Galileo [ACO85] extend the programming languages Ada and ML [GMW79], respectively, with persistent conceptual modeling capability. This thesis differs from these efforts in that it addresses the representation and management of abstract language-independent object specifications, rather than an extension of a particular programming language.

Semantic data models were prefigured by research on frames [Min75] and semantic nets [Sow92]. The scheme representation used in this thesis resembles frame instantiation [BKN+77] restricted to the domain of software components. The notion of extensible schemata to represent propositions is suggested by conceptual graphs [Sow84], which link concepts to primitive classifiers to produce structures that can be mapped into first-order logic. Touretzky [Tou86] develops a theory of inheritance hierarchies with exceptions. This thesis uses a similar hierarchical ordering to implicitly infer and override information, but adapted to a delegation mechanism constrained to preclude certain inappropriate inferences. In general, the focus of this inquiry is of more modest scope than the relevant artificial intelligence research (software development vis-a-vis human language and behavior) and coarser granularity (software artifacts vis-a-vis words and phrases), and hence hopefully addresses a more tractable problem.

There is a large and growing literature on abstract specification using object-oriented analysis and design (OOA/D) techniques (e.g. [WBJ90]). Early efforts [SM88, CY90] are largely a pastiche of traditional techniques recast in object-oriented terminology. Their most significant contribution is an attempt to relate these techniques by designating objects as a unifying concept. Structure and behavior are collated in class descriptions, and behavioral dynamics are modeled by finite state machines. Refinements include verifiable interaction models [SM91], use cases [JCJO92] and hierarchical state-transition specifications [Feh93]. Popular approaches may be roughly categorized as entity-based [SM90,

CY90, Boo91, RBP+91] or behavioral [WBWW90, SM91, JCJO92, RG92], reflecting the relative emphasis on static structure or dynamic interaction, respectively.

Those efforts are largely directed toward notation and methodology; by contrast, this thesis focuses on the underlying representation rather than the presentation of object specifications, and is independent of any particular methodology. A deficiency of current OOA/D techniques is the loose correspondence between analysis, design and implementation models. This thesis complements OOA/D by addressing the management of analysis and design models as development artifacts in their own right, in the context of a shared, distributed, evolving, interconnected development process.

Formal specification [LG86, GH93, FJ93] offers a rigorous basis for describing object behavior. That effort is largely directed toward the exploration of programming language structures conducive to the production of provably correct software. There has regrettably been little interaction with object-oriented methodologists, although heuristics have been suggested for translating informal requirements into formal specifications [Fra91, MH91]. This thesis attempts to partially bridge this gap by describing object specifications with practical application to OOA/D with the potential for a well-founded semantics.

The controlled evolution of systems by the component-specification framework builds on well-accepted configuration management concepts [KCB86, BM88]. These concepts are extended to support tasks and activity workspaces in the versioning model of Chiueh, et al. [CKK90]. Batory and O'Malley have described an approach to composing systems from software components [BO92]. This thesis is the first effort to our knowledge to characterize a software component as a unifying configuration of abstract views depicting a concrete manifestation.

Chapter 3

Component Definition

3.1 Component Schemes

System specification is based on the notion of a *component* as the entity that integrates an application concept with its manifestation in software. The component is the unit of abstraction suitable for specification, evolution and reuse. An application is then constructed by assembling components in a manner consistent with their specifications. The role of the component-specification framework is to manage the component and application specifications.

A component interface is presented in a basic *scheme* that defines the component's structure. A scheme is an abstract representation of a component's composition, or *attributes*, and functionality, or *methods*. The interface defines the structural basis for its use by other components. A specification is built from schemes to show how components actually plug together to accomplish the capabilities defined in their interfaces. A component is represented in a specification by a *frame*; the frame depicts a scheme that is a subset of the interface sufficient for its use within the specification.

Definition(*Frame*). A *frame* is a member of the set FRM indexed by sorts $COMPONENT$ and $METHOD$.

Interpretation. FRM is the universe of all frames in all specifications. Frames are sorted into two types, component frames and method frames.

The frame represents the potential for use of an object in a particular operational context. The context delimits the role the object plays, and thereby its applicable behavior. The individual `Donald` in Figure 3.1, for example, may be represented by the

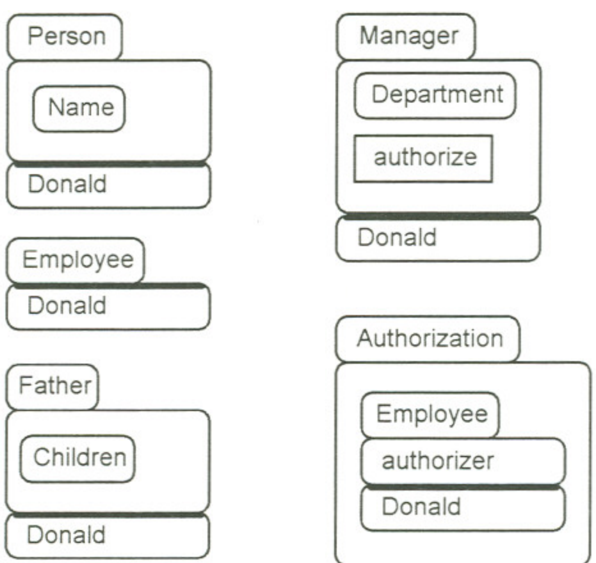

Figure 3.1: Examples of frames

distinct frames `Person`, `Employee`, `Manager` and `Father`, each with differing behavior according to his various roles.

Both an attribute and a method are represented in the component specification as frames contained within the component frame. The `Manager` component in Figure 3.2, for example, has a `Department` attribute and an `authorize` method. Attributes and method frames are *slots* that can be bound to a *value*: an attribute is bound to another component frame, a method is bound to a method scheme that characterizes an *action* performed when the method is invoked. The method may have *input slots* that are bound prior to the action, and a *result slot* that is bound to a frame upon completion

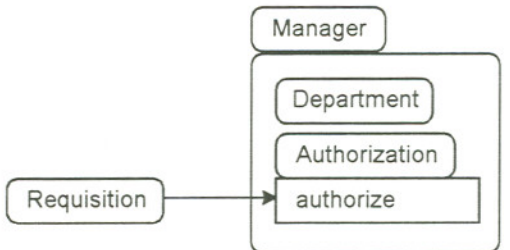

Figure 3.2: Manager frame

of the action. The values of the frame's attribute, input and result slots constitute the *acquaintances* of that frame; these are the components necessary to fully specify the frame interface. A *connection* binds the input slots with compatible frames, activating the frame to perform an action. The `authorize` method in Figure 3.2 has an input slot `Requisition` and a result slot `Authorization`.

Table 3.1 summarizes the associations between frames. The composition and method

Association	Source	Target
composition	Component	Attribute
method	Component	Method
input	Method	Input Slot
result	Method	Result Slot

Table 3.1: Associations between Frames

associations are visually depicted in a specification by containment of the attribute or method frame within the component frame. The input association is depicted by an arrow from the slot to the method. A method's result slot is situated on top of the method frame. A specification's underlying structure is that of a digraph with frames as nodes and these associations as edges.

Definition(*Specification*). A *specification* SP is a graph $\langle \nu, \eta, \partial_1, \partial_2 \rangle$ with nodes $\nu \subset FRM$, edges η, a source function $\partial_1 : \eta \rightarrow FRM$ and a target

function $\partial_2 : \eta \rightarrow FRM$, such that there is a graph embedding of SP into the graph $\mathcal{G}_{FRM}$ induced by the frame associations. $\mathcal{G}_{FRM}$ consists of frames as nodes and, for any frames F_1 and F_2 of $\mathcal{G}_{FRM}$, an edge from F_1 to F_2 if and only if there is a frame association from F_1 to F_2.
Interpretation. A specification diagrams selected frames and frame associations in a graph.

A scheme is a simple specification consisting of the component frame as a root node and edges to its attributes and methods.

Definition(*Scheme*). A *scheme* S is a specification graph with a single root of sort $COMPONENT$. The distinguished root node is called the *point* of the scheme, identified as $pt(S)$.
Interpretation. A scheme diagrams the interface of a distinguished component frame. The scheme point singles out this component frame; pointed schemes help relate interfaces consistently, as we will see in the definitions of the next section.

3.2 Component Structure

A component interface is related to the software artifact(s) that realize it by a *view*. An artifact is a persistent representation of a component interpretable by software tools; it is the tool's unit of access. The view encapsulates these artifacts in a form suitable for use independently of the tool that created the artifact. A distinguished *interface frame* serves as the canonical definition of component structure for the view.

Definition(*View*). A *view* V is a pair $\langle S, A \rangle$ consisting of a scheme S and a (possibly empty) set A of artifact handles. Each artifact handle is an uninterpreted value that serves to identify the location of an artifact for an

associated tool. The unique frame $pt(S)$ of V is the *interface frame* for the view.

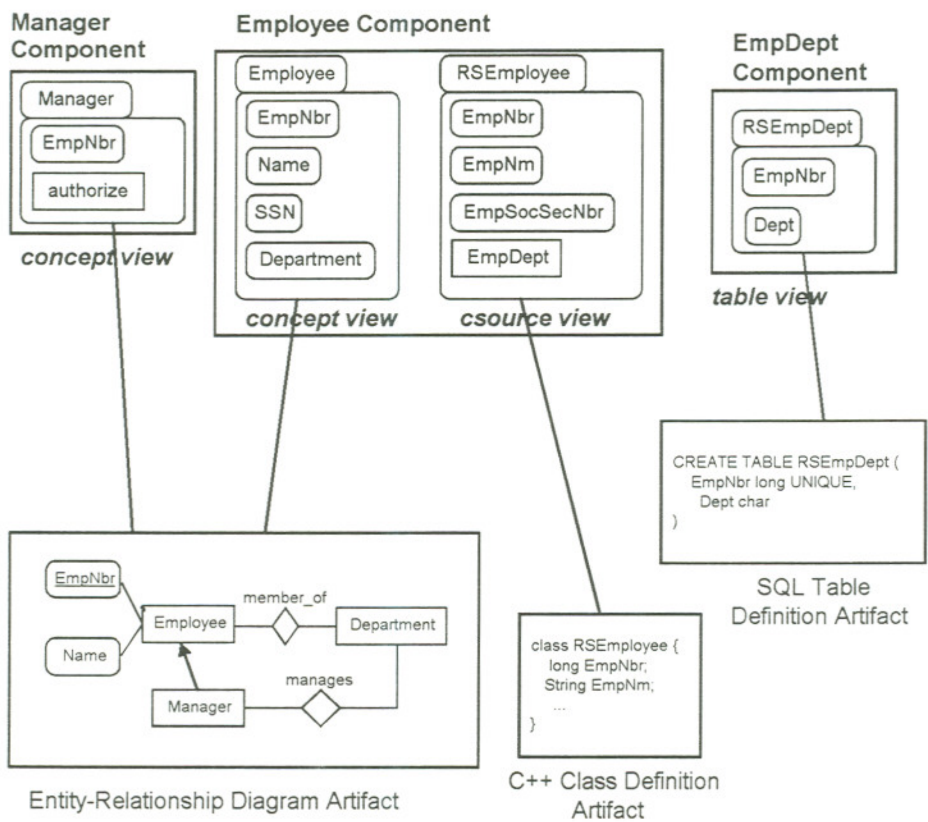

Figure 3.3: Manager and Employee views

Figure 3.3 presents `Manager` and `Employee` components sharing a `concept` view related to a common Entity-Relationship Diagram. The `Employee` conceptual specification is implemented in a `csource` view that encapsulates the **R**eservation **S**ystem class `RSEmployee`. The `Department` attribute in the conceptual scheme is implemented as the `EmpDept` method in the `csource` scheme. The `EmpDept` component encapsulates a database table that associates an employee with her department. This component has no conceptual scheme in this specification, but merely encapsulates a useful implementation artifact.

A *component* collects a consistent set of views for a given conceptual entity. In this respect, a component is analogous to a cell in the cell-view structure of the OCT CAD management system [HMSN86]. Every view belongs to at least one component, and every component has at least one view.

Definition(*Component*). Let $\equiv_{INT}$ be an equivalence relation on the domain FRM of frames. Two frames F_1 and F_2 are *intensionally equivalent* if $F_1 \equiv_{INT} F_2$. Two views are intensionally equivalent if their interface frames are intensionally equivalent. A *component* C is a set of intensionally equivalent views. The set of components covers the set of views, that is, $\forall$ view V, $\exists C \mid V \in C$.

Interpretation. A component relates consistent views of a common concept. This "consistency" is formalized as the intensional equivalence relation. Every component frame in a specification is derived from some particular view's interface frame. This interface frame is, in turn, related to the interface frames of other views by the component. Note that there may be more than one component per view. This allows several components to relate to a single concept. However, a component can only relate views that are intensionally equivalent. Furthermore, every view defines an interface for some component.

The view's interface frame represents the *abstraction facet* of the view, while the artifacts represent the view's *content facet* [CK91]. The abstraction facet of a view contains the view *metadata*, the data about the content facet. The metadata describes external characteristics and how the view relates to other views, abstracting the interface and composition from the concrete representation. External characteristics, or *annotations*, contain relevant information about the view regarded as an artifact—who created it, when it was created, release and qualification status, remarks on its use, etc.

The framework can incorporate any relevant artifacts of a component; examples of artifacts include a requirements document, structured specification, entity-relationship diagram, object diagram, state-transition diagram, spreadsheet, source code, object code, test suite or manual page. Tools generally fall into three categories: editors, translators and analyzers. An editor creates, browses and modifies a software artifact. The Meros framework, for example, provides a simple graphical specification editor for creating, browsing and executing specifications. A translator changes a representation to a form more suitable for another tool. A compiler is a translator that transforms a set of source code views into an object module. A test driver is an example of an analyzer, a tool that infers information about an existing artifact. The test driver accepts a test suite as input and produces a test result as a dependent artifact.

A tool may be associated with an artifact as part of a component's encapsulation. If the encapsulation supports tool invocation on method activation, then the tool may be directly invoked by the framework when the method is activated. For example, an implementation view of the `Manager` component with an associated tool with the capability of performing an `authorize` action enables the framework to initiate authorization by invoking the tool directly. This provides a higher level of abstraction for interaction with components. A component supplier or framework integrator defines the interface for a component. An application builder assembles the components into a form suitable to accomplish a discrete task. The application user enables a component by defining values for input slots and activating a method of the interface (cf. §4.2).

The view interface frame defines a component's structure for use in specifications. These specification frames may restrict the interface to a subset of functionality sufficient for their use within a particular specification context. Figure 3.4 on page 19 shows an interface frame for the `Hotel` component and its use in three specifications. The `Hotel` frames in the specifications derive their scheme from the base scheme defined in the interface frame. A derived frame is related to its base frame, or *prototype*, by a *delegation* rule, which defers the binding of an attribute or method frame through the derived frame

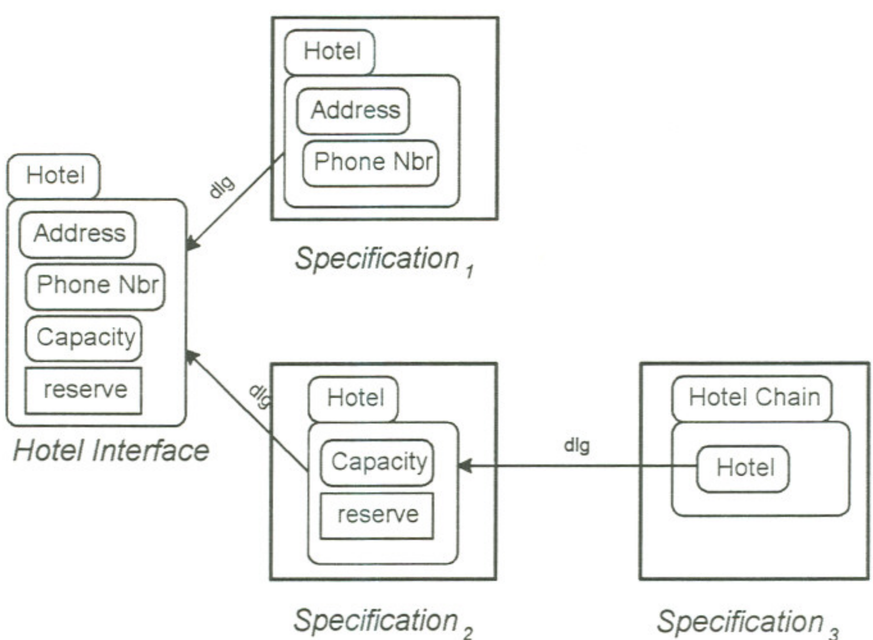

Figure 3.4: Hotel frame example

to a base frame (shown as `dlg` edges).

Definition(*Delegation*). A graph homomorphism $f : G_1 \rightarrow G_2$ is a mapping (f_ν, f_η) of nodes $f_\nu : \nu_1 \rightarrow \nu_2$ and edges $f_\eta : \eta_1 \rightarrow \eta_2$ such that

$$\forall e \in \eta_1, \partial_i(f_\eta(e)) = f_\nu(\partial_i(e)), i = 1, 2.$$

A pointed graph homomorphism $f = (f_\nu, f_\eta)$ is a homomorphism on pointed graphs that preserves points, i.e., $f_\nu(pt(G)) = pt(f(G))$. *Delegation* is a partial pointed graph monomorphism $dlg : SCHEME \rightarrow SCHEME$, i.e., a one-to-one mapping of schemes that relates the roots.

Interpretation. A scheme is a pointed specification graph, where pt singles out a component frame as the root of the graph. Delegation is *partial*, since it may only map selected attributes and methods of a frame, *pointed*, since it always maps a scheme's root to the root of the prototype scheme, and

monomorphic, since a given attribute or method can serve as the prototype for at most one frame of the delegated scheme.

The delegation relationship can recurse, i.e., a derived frame can serve as the target of another frame's delegation. The `Hotel` frame in $\texttt{Specification}_3$ delegates to the `Hotel` frame in $\texttt{Specification}_2$. A frame may also be derived from more than one base frame, with distinct attributes and methods delegated to each of the base frames. For example, a frame that shared characteristics of both the $\texttt{Specification}_1$ and $\texttt{Specification}_2$ `Hotel` might delegate to both frames. Delegations thus form a directed graphic with frames as nodes connected by delegation edges.

A component's specific functional characteristics are elaborated in independent specifications, as described in the next section §3.3. The frames representing the component in these specifications serve to *refine* the interface. The refinement frame, like all specification frames, delegates to the prototypical interface frame for its basic functionality. However, since the refinement specializes certain aspects of the base functionality, the interface frame in turn delegates the specific specialized attributes and methods to the refinement frame. Figure 3.5 depicts a component's structure with complementary delegation edges between the `Hotel` interface and one of its refinements. In the figure, the `Hotel` refinement in $\texttt{Specification}_1$ characterizes the behavior of a particular `Hotel` method. The `Hotel` refinement delegates to the base interface in general, whereas the interface frame delegates to the refinement for a detailed specification of the refined method.

The refinements can be specialized by other refinements with complementary delegation edges between the frames. The interface frame and its reachable refinements form a subgraph of the delegation graph. There is a path from any frame in this subgraph to any other frame in the subgraph, and the subgraph is maximal in this respect. The interface frame and its reachable refinements thus constitute (by a fortuitous terminological coincidence) a strongly connected component of the delegation graph. Though

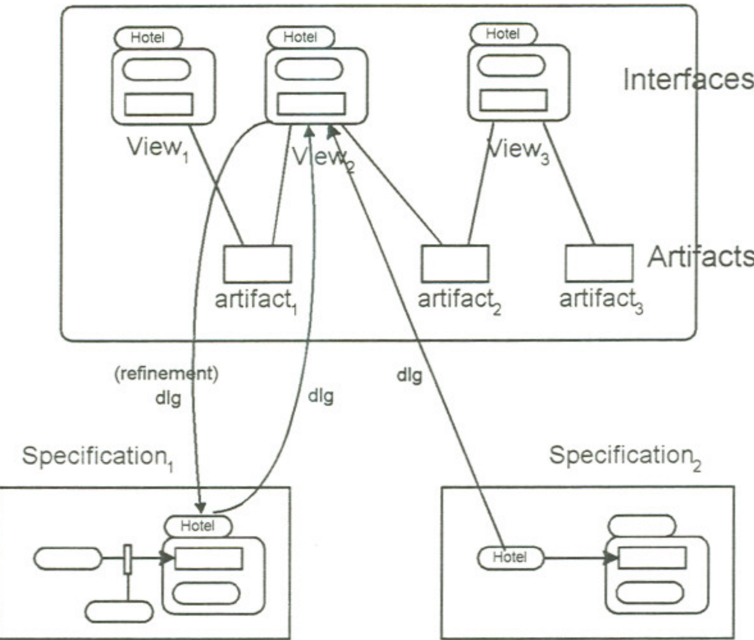

Figure 3.5: Component Structure

the full delegation graph has cycles, each path in the graph terminates in a connected component that includes a component interface frame.

3.3 Behavioral Specification

The component interface scheme defines the structural capability of a specification frame. The frame can be further characterized by a *rule* for transformation of inputs to outputs or a *constraint* to express conditions that must obtain in its use. A rule consists of a $\langle$*Condition*, *Action*$\rangle$ pair: the condition is a template that must be matched before the rule is enabled, the action describes a resultant binding of values to frames matching the condition. Figure 3.6 shows a rule for the `reserve` method, indicating that a hotel handling a reservation request with a room available for the requested period enables that room's `hold` action to be successfully activated. That the two `Room` frames in the

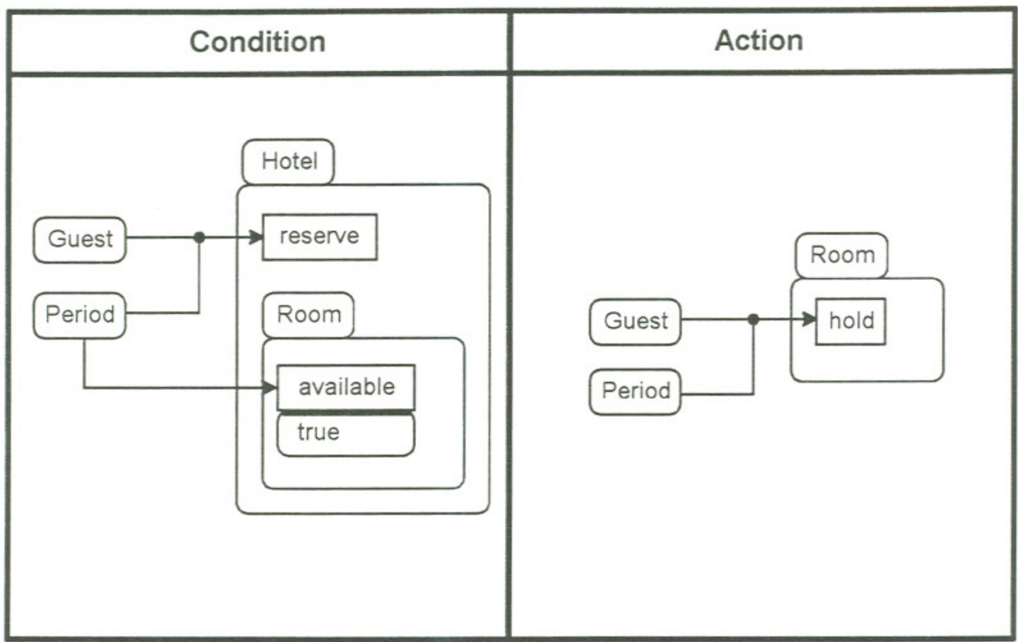

Figure 3.6: Reserve rule

specification denote the same `Room` component is established by a visual convention, for example by having the same color.

A constraint depicts a situation that must necessarily hold for a method or component. A method constraint consists of a method and at least one pre- or post-condition. The room in the `hold` method in Figure 3.7 is constrained to be available for the given period prior to being held and unavailable afterwords. The pre-condition (post-condition) is indicated by a ▯ connector on the method input (output) connection. In the figure, the pre-condition and post-condition both apply to the same `Period`. Note that the `reserve` rule of Figure 3.6 explicitly satisfies this pre-condition and does not implictly violate the post-condition. A constraint, expressing a <u>necessary</u> state of affairs, has a stronger effect than a rule, which provides a <u>possible</u> transformation. No action can reliably occur that violates a constraint. A rule indicates an action that may occur if the condition is matched and no constraint precludes this choice.

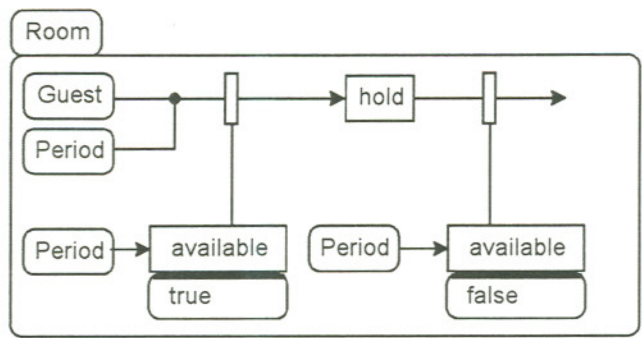

Figure 3.7: Hold constraint

A view may be implemented in terms of other views by specifying a delegation rule as in Figure 3.8. This example elaborates on the views defined in Figure 3.3. Here,

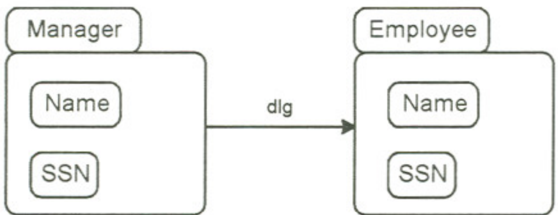

Figure 3.8: Manager delegation to Employee

the `Manager` frame delegates to an `Employee` frame for the `Name` and `SSN` attributes. Only those attributes amd methods occuring in the source frame of the delegation rule are subject to delegation. Thus, the `authorize` method of the `Manager` component is not implemented in terms of `Employee`, since it does not occur in the delegation rule of Figure 3.8. In Figure 3.9, the `Employee` frame in the `concept` view in turn delegates to the `RSEmployee` frame in the `csource` view. The `csource` view has a one-to-one correspondence with a `RSEmployee` class defined in a source code artifact of the Reservation System library.

The associations and rules defined in a specification are *bona fide* assignments rather

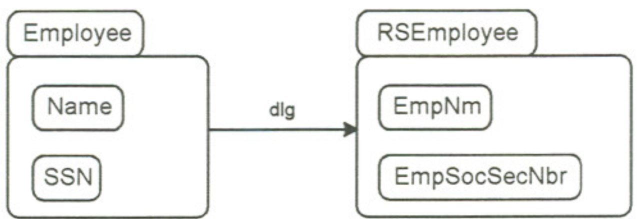

Figure 3.9: Employee delegation to a source code interface

than *de facto* regulations; that is, the framework does not guarantee that a view's concrete aspect is a complete and faithful reflection of its interface, nor that views are actually related in a rigorous fashion by specification rules. In particular, there is no assurance that views integrated into the framework are correct by construction. This is a weak notion of specification, but befits the non-intrusive nature of a framework targeted to an open development environment. The multiplicity of view representations within a component necessitates a loose coupling of interface to artifact and view to view. The view interface exposes just enough of the artifact to facilitate its use in the framework.

3.4 Use and Identity

A scheme serves as a template for modeling a collection of real-world entities that share some features in common. A frame may define how a feature is used within the context of the specification in which it occurs, or it may delegate an aspect of its scheme to a prototype, typically an interface frame. In addition, a frame may be *bound* to a proxy for a particular individual entity of the application domain. The prototypical component frame is an *intensional* entity, while the bound value represents the frame's *extension*. The intension shows the meaning of an extensional object, that is, how it is used within a particular context. Specifications are typically of unbound intensional objects[1], although

[1]Of course, the intensional objects of the specification are managed as extensional objects by the framework. A hotel component is an intensional object from the perspective of an application dealing

these are often explicated by their extensional effects on a prototypical frame. Use of prototypes encourages specification by example, unfolding the meaning of a software component by its action on typical instances. In this respect, the frame differs from a class concept and resembles a prototype or exemplar [Lie86, LT86, Bor86, US87, DMC92]. It is more appropriate to consider a frame as a specification for a class or an instance.

The intension-extension distinction has a long history in philosophy, dating back to the Aristotlean separation between an object's meaning, or *logos*, and our experiences of it. Frege [Fre92] considered every name as expressing some sense according to the relation in Figure 3.10. A name, or symbol, for an object has a sense (*Sinn*) and a reference or denotation (*Bedeutung*). The referent is the independent existent for which the name stands. The sense is an aspect of the word's use that determines it to have that referent. Different uses of the same object represent different senses with the same reference. "Names with different senses but having the same reference correspond to different routes leading to the same denotation." [Dum73].

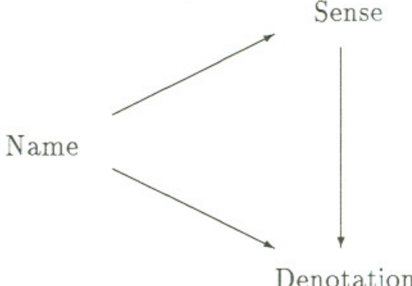

Figure 3.10: Fregean distinction

This distinction is real and important; it is also not easily accomodated in most programming languages, where object identity is either fragmented among its various uses (Donald has one identity as a father, another as a manager) or case logic proliferates (Donald is an employee, and employee behavior handles the specialize cases of father and manager).

with hotels, but is an extensional object from the perspective of a framework dealing with a component as an object to be managed.

This distinction is made in scheme specification by the following steps:

1. Decouple identity and behavior.

2. Define behavior by use within a context.

3. Represent separate uses of an object in separate frames.

4. Relate the separate uses to the single unique extension.

Each frame has an intensional identity relative to its parent composite frame. In addition, a frame may have a *(bound) value* given by another frame representing the extension. All frames representing a given object have a common extension; the recursive extension relation between frames forms a directed acyclic graph terminating in a *domain value* that directly represents the object in its application domain. A domain value can be bound to different frames of different types at the same or different times. For example, the domain value for `Donald` in Figure 3.1 may be bound to a `Person`, `Employee`, `Manager` or `Father` frame as the need arises.

The domain value is itself a typed frame occuring as a member of a collection. However, the domain value's characteristic type and parent are accidental, rather than essential, features of the domain value. The identity of the domain value is invariant through changes in its bound intensions, characteristic type and collection membership. Thus, the domain value for `Donald` may have begun as a frame of type `Employee` in a `Hotel Employees` collection, been assigned a `Manager` type in a `Hotel Managers` collection, and eventually end up as frame of type `Person` in an `Ex-employee` collection. Frames with a bound value of `Donald` continue to have identical extension despite these changes.

3.5 Change and Identity

Objects have unique, immutable identity, but changeable state. Identity is not a visible characteristic of the object. All visible characteristics, including names, are subject to change. The object identifier is a *surrogate* [Cod79], indepenendent of description and location, thereby avoiding the anomalies associated with a weaker notion of identity [KC86]. Every manifestation or use of a domain value refers back to the single occurence of the entity it represents in the application domain. An object has no extensional identity apart from its denotation, and two frames with a different denotation denote different (extensional) entities. Similarly, every specification has a unique identity that persists through changes to the specification, and every frame in the specification has a unique identity relative to the specification. All frames derived from a prototypical frame refer to the canonical identity of the prototype relative to its defining specification.

Consider the `Manager` attribute of the hotel object in Figure 3.11 on page 28. This attribute originally has a denotation of `Donald`. When the `Manager` attribute changes, it denotes a different domain value `Ivana`. The change to the hotel is reflected in the edge labelled `chg`. The hotel object's state changes, but it's identity remains the same. Similarly, the intensional identity of the `Manager` attribute, as a feature of the `Hotel` scheme, remains the same, although its extensional identity has changed from `Donald` to `Ivana`. Stated differently, there is a notion of `Manager` for a hotel, and the essence of that characteristic has not changed, although it is bound to different values over time. Likewise, `Donald` plays many roles at the same or different times, but each role (frame) has a consistent domain value (denotation) over time and use. Equivalent denotation establishes identical extension—the two `Hotel` frames in Figure 3.11 are identical by virtue of the equivalent denotations as reflected in the `dnt` edges. Denotation equivalence may be conveyed in a specification by other visual conventions as well, for example, by using the same color for denotationally equivalent frames.

Historicity is persistent, that is, the past is preserved. Thus `Donald` may no longer be

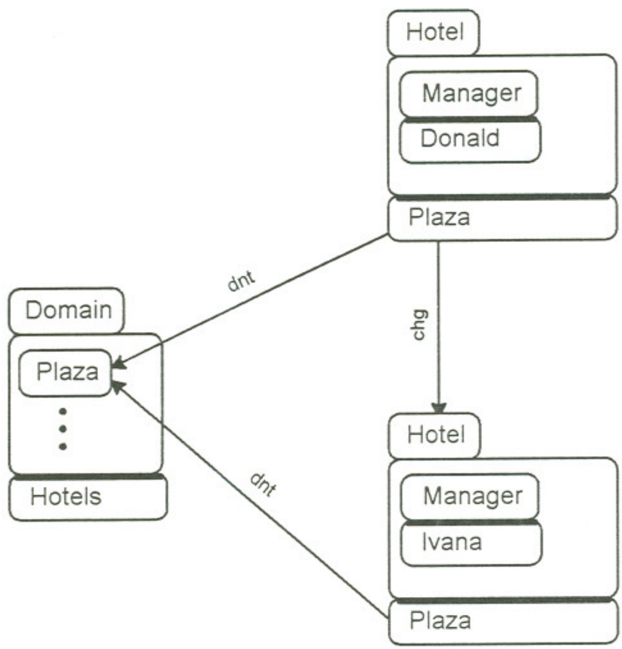

Figure 3.11: Denotation and change relationship

a manager of any hotel (the value `Managers.Donald` is no longer reachable from any current context), but the fact that `Donald` *was* a manager of the `Plaza` hotel at a particular time is recorded (the value `Donald` is reachable from a superseded context). The version history in this example is a linear sequence. However, versions may evolve independently from the same source, for example to explore alternative designs or exercise different scenarios and compare the results. In general, the version history forms a directed acyclic graph of frames connected by change edges; this DAG constitutes the *history* of the object (cf. §5.5). All changes represented in a specification are non-destructive actions, indicated by a new version of a frame. Object state at any point in time is given by the denotations of all of its attribute frames at that moment.

Chapter 4

Behavioral Dynamics

The constructs presented so far apply to a static specification—there has been little discussion of how objects interact to carry out the behavior specified in the method interface. As described in Chapter 3, the component interface expresses the latent capability of an object. A rule describes how a method uses its inputs to produce a result subject to constraints. This chapter elaborates these behavioral descriptions by introducing the dynamic capabilities of states, probes and agents. These capabilities result in a computational model for an *executable* specification. Executability serves a two-fold purpose: for the component developer, prototyping and validation of individual components; for the application builder, assembly of components into useful functional configurations.

4.1 Method specification

We motivate the discussion by reconsidering the `reserve` specification of §3.3, reproduced in Figure 4.1 with an `accept` action added to record the acceptance of the reservation. The `Hotel`'s `Room` attribute contains an `available` method, conveyed by the nesting of the frames. The same `Period` serves an input value to both the `reserve` and `available` methods, as shown by the arrows. The specification depicts the rule "IF an attempt is made to reserve a room and the hotel has an available room, THEN accept

the reservation and hold the room for the given period". Figure 4.2 shows a specification for the `accept` rule with the inclusion of state information for the `Reservation` component. State information is representing in a *state variable*, a single-valued, un-

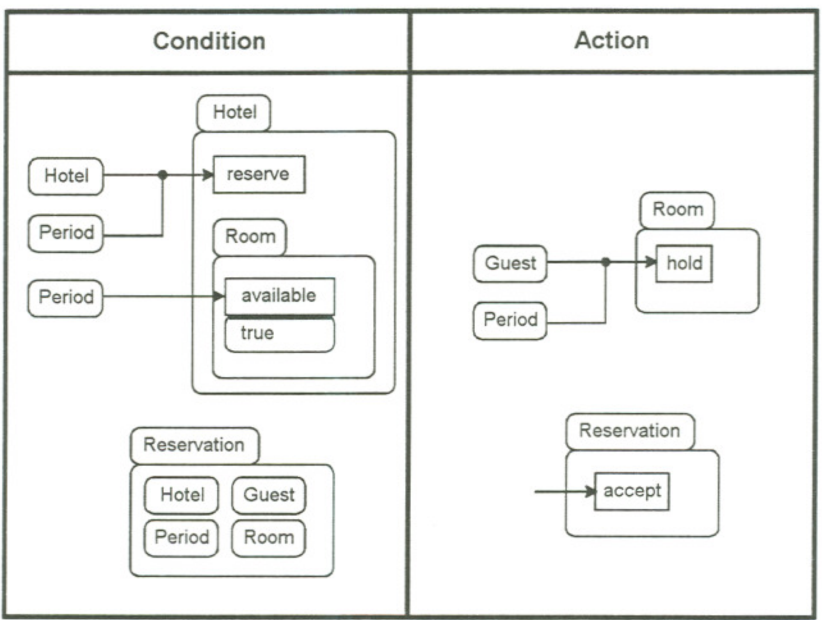

Figure 4.1: Reserve rule

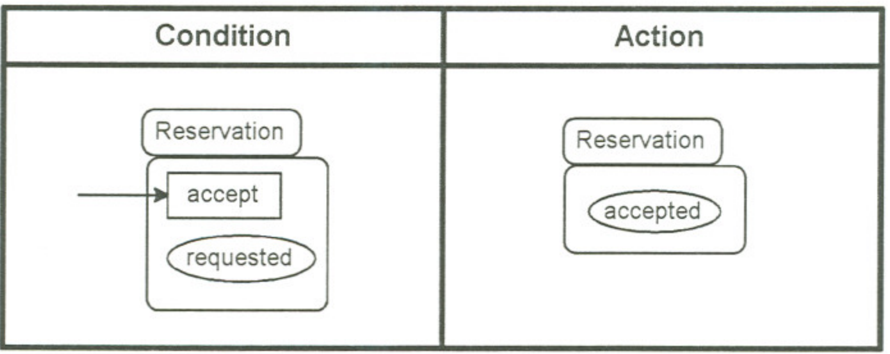

Figure 4.2: Accept rule

interpreted attribute used to contol a component's behavior and activate methods of dependent components. Figure 4.3 on page 31 shows the `Reservation Status` state

variable ranging over a domain of five discrete values, with inital value `requested`. An

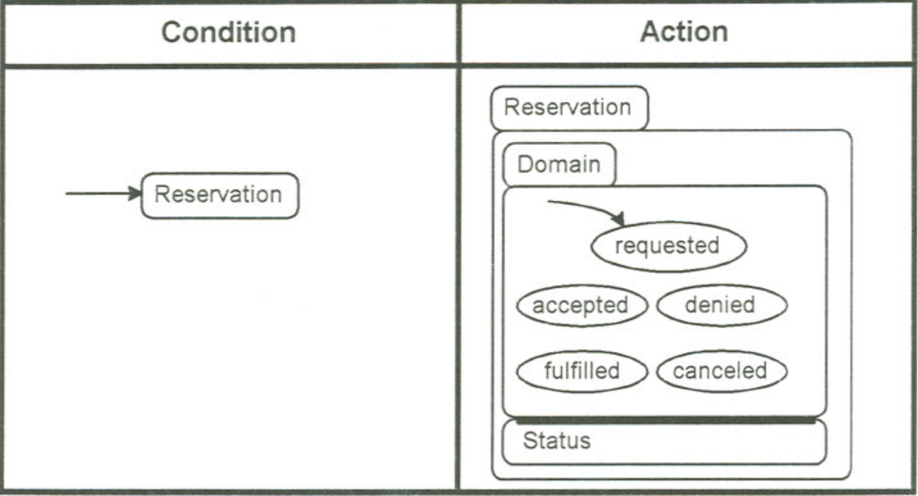

Figure 4.3: Reservation status state variable domain values

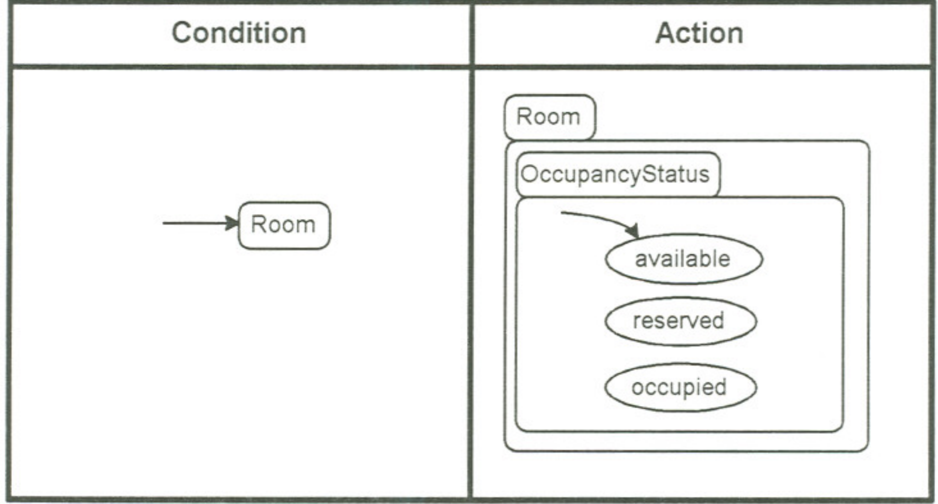

Figure 4.4: Room occupancy status domain

arrow without a source frame and directed to a component frame corresponds to a construction event—the rule applies whenever a `Reservation` frame is created, for example as a side-effect of the `reserve` action.

As an attribute, a state is set as a result of an action and can be used in rules and constraints. In the example, the `accept` action transforms the value of the `Status` state variable from `requested` to `accepted`. Similarly, the `Occupancy Status` state defined in Figure 4.4 is transformed by the `hold` rule of Figure 4.5 from `available` to `reserved`. Figure 4.6 translates the room's `isAvailable` action into a query on the `Occupancy` state variable.

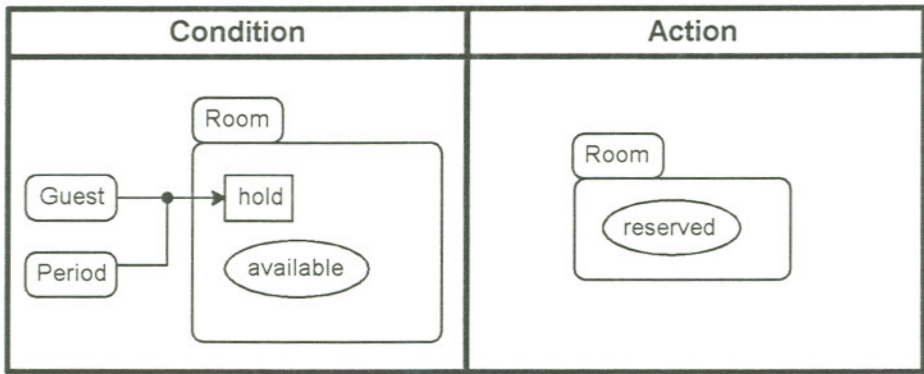

Figure 4.5: Hold rule

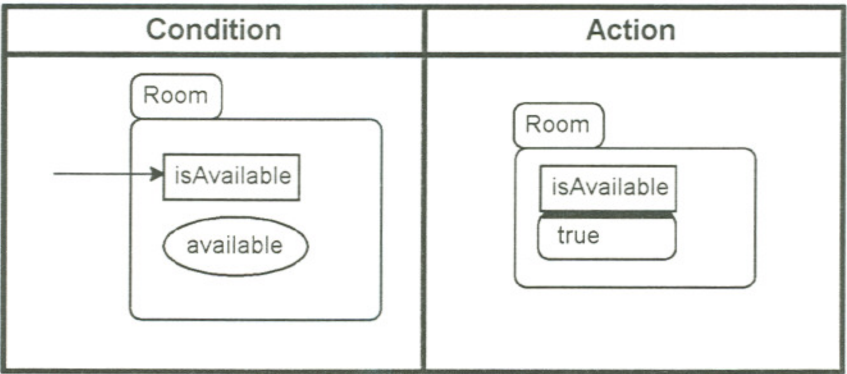

Figure 4.6: Availability rule

Finally, Figure 4.7 shows a negative condition: the grayed-out `Room` frame represents the absence of an available room for the given period, resulting in a denial of the reservation request[1].

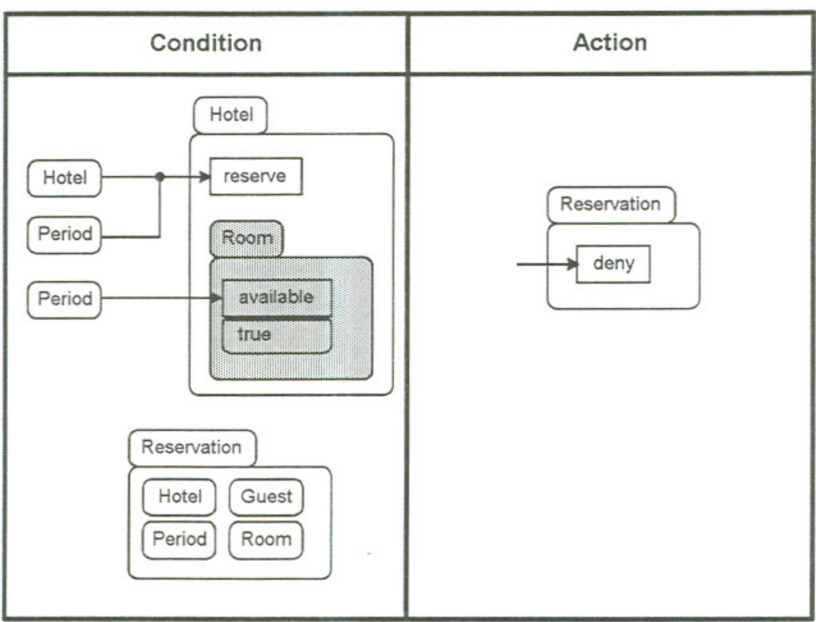

Figure 4.7: Deny reservation action

[1]Note that Room available true is not equivalent to Room available false. The former represents the absence of an available room in the subject hotel, whereas the latter represents the presence of an unavailable room in the subject hotel. Negation implies the absence of a value satisfying the given frame, a condition that cannot be asserted with positive frames alone.

The common intent of these actions is to transform a state variable from one value to another. An input state used in an action is no longer available for use in another action. State variable exclusivity differs from the use of input values in general, in that a component bound to an input slot is available for subsequent use after the action completes. The use of a frame as an input to an action is, however, consistent: an input frame is treated as a resource consumed by the action to produce a result. A frame bound to an input slot spawns a cloned frame upon activation for the exclusive use of the action. The lifetime of this cloned frame is the duration of the action. A state, on the other hand, is not bound to an input slot as a formal argument, but rather consumed directly by the action.

4.2 Method activation

As indicated in §3.1, a method serves as a specification for an action. The action is actually carried out by an *actor* [Agh85], the computational analogue of a frame (Table 4.1). An actor realizes the use of an object that a frame potentiates[2]. An actor is responsible for handling requests to perform actions described by the methods of a frame's

Specification	Computation
Frame	Actor
Method	Action
Connection	Channel

Table 4.1: Specification-Computation analogues

[2]The frame-actor distinction is similar in this respect to the class-instance distinction in object-oriented programming languages. The class (frame) defines a template for the computational capability of the instance (actor). In other respects the notions are independent: in the terms of the specification framework, an actor is an activated frame, whereas an instance is a class bound to a specific domain value. It is more appropriate to consider a frame as a specification of a class or instance within a particular context (cf. §3.4), and an actor as a multiplexed instantiation, with each actor handling just those requests appropriate to the corresponding frame. A frame may specify class or instance-specific behavior; an actor enables the specified class or instance behavior.

scheme. Requests are initiated by opening a *channel* for information transfer between the requestor and the method handler. The channel corresponds to a connection from bound input slots to the recipient frame's method. An input value at the channel source must be compatible with the input slot frame. Compatibility is determined by a rule that maps the value to the slot. For example, Figure 4.8 shows a rule for mapping a

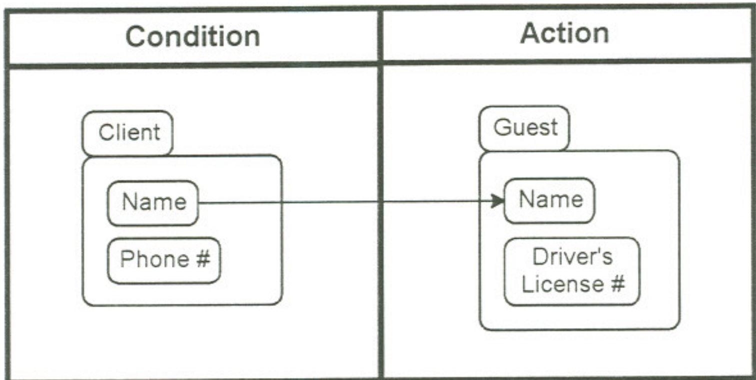

Figure 4.8: Rule for binding a value to a slot

`Client` of the Travel Agent application to a `Guest` of the Hotel Reservation application. This rule creates a new `Guest` frame bound to the value of the `Client` frame. The `Guest` frame's `Name` attribute is given by the bound `Client` frame's `Name` attribute. The arrow between the corresponding `Name` frames establishes a procedure for binding an existing `Guest` frame with a `Client` value. The `Client` value is compatible with the `Guest` frame by virtue of this mapping.

Attributes and methods peculiar to the `Client` frame are unused in the context of operations on the `Guest` slot. Attributes and methods peculiar to the `Guest` frame are used while the bound frame is used in the role of a `Client`. The `Client` frame's `Phone #` attribute is unused in the scope of the `Guest` binding, since it is undefined in the `Guest` interface. By the same token, the `Guest` frame's `License #` attribute is not accessible in the context of a specification using a `Client` frame.

When an input value is bound to an input slot, there must be some way to map

the input value's scheme to the input slot's scheme. A delegation rule such as that of Figure 4.8 serves this purpose. A delegation rule from a prototype of the input value to a prototype of the input slot may also serve this purpose, since recursive application of delegation will yield a map of the input value scheme to the input slot scheme. Figure 4.9 depicts the binding of the input value `Dave` to a `Guest` input slot of the `Hotel`'s `reserve`

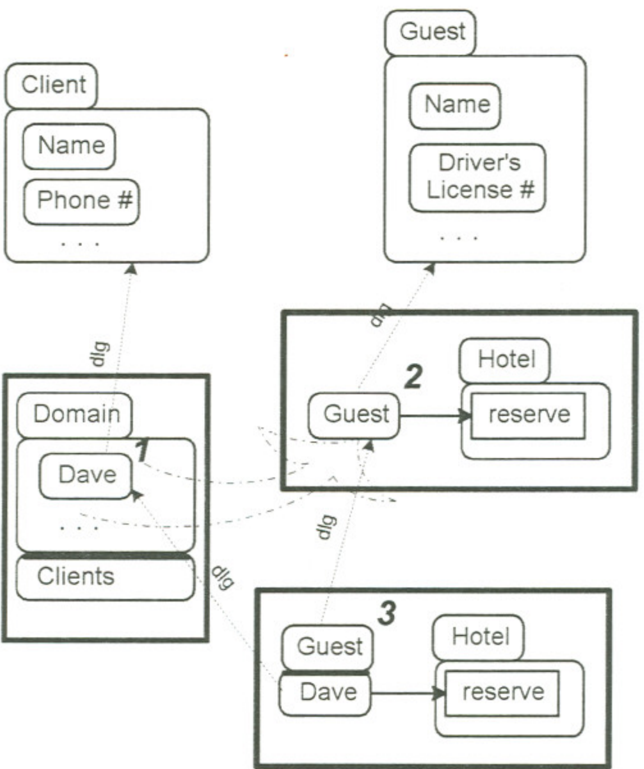

Figure 4.9: Resolution of a bound value

method. There is no direct delegation map of `Dave` to the `Guest` slot, but `Dave` is a `Client`, and there is a delegation map from a `Client` frame to a `Guest` frame. Therefore, this delegation rule may be used to map the attributes and methods of `Dave` to a `Guest`.

The effective binding rule is the "most specific" rule for delegating the bound input value to a prototype frame. Specificity is measured by the length of the path formed from delegation edges to prototypes of the input slot. Thus, if there were also delegation

rules from both the `Client` and the `Guest` frames to the same `Person` frame, this would serve as a candidate binding rule, but would be superseded by the effective binding rule from `Client` to `Guest`, since the delegation path to the `Guest` frame is shorter than that to the `Person` frame. To recapitulate, the procedure used to bind the `Guest` slot (2) of the `reserve` method to the `Client` frame `Dave` (1) in Figure 4.9 is to follow the delegation edges back from the candidate `Guest` frame until the first mapping rule is encountered from a prototype of the `Client` frame. The attribute(s) in common, `Name` in this case, are assumed by the bound frame (3).

All object interaction occurs by communication along channels between cooperating actors. A *dispatcher* at the channel destination monitors a *mailbox* of incoming requests. The dispatcher schedules the request and activates a *handler*, a dormant actor designated by the dispatcher to process the request. An actor is dormant until activated by a dispatcher, active during the course of its action, and passivated when the action completes. The action thus occurs in the time interval $[activation, passivation]$. Action intervals are disjoint for a given actor. Thus, activation is single-threaded for the actor, i.e. an actor is responsible for at most one action at a time. However, activation may be multi-threaded for a given object, since the object may have several actors active at the same time.

The actor handling the request must spawn a replacement actor to handle subsequent incoming requests. The dispatcher may schedule another request once a replacement actor is available. In this way the single-threaded action of individual actors can be scheduled concurrently to the extent allowed by the application and execution environment. In the degenerate case of a single-threaded environment, the handler simply designates itself as the replacement actor upon completion of its action.

A handler may progressively enable its replacement, monotonically increasing the number of methods the replacement can service as the active handler completes. For example, a handler that enters a critical section may restrict its replacement to handle only

messages unaffected by the excluded resource until it completes. Similary, a versioned coarse-grained component responding to a long-lived update method may initially designate an immutable base version as its replacement restricted to read-only methods, then operate on a new mutable version in a private environment and enable the remaining methods when it completes (cf. Chapter 5).

Actions are specificied by directly describing the effects of actions on objects in the generation and use of frames. A requestor cannot rely on implicit assumptions about the order of action execution. Action scheduling is the responsibility of the dispatcher and activation completion is the responsibility of the request handler. Thus, request activation and completion is asynchronous from the standpoint of the requestor in the absence of explicit control dependencies. Control is externalized from actions by monitoring a channel for action completion (cf. §4.3). The requestor can serialize actions by waiting upon the handler to post a result to the open channel and signal completion prior to issuing a subsequent request.

Just as an actor can be considered an independent computational unit, an action can be considered a discrete process (conceptually, of course; the allocation of tasks to processes can differ considerably in the underlying system architecture). Parallelization occurs whenever more than one handler is active. Concurrent activation occurs whenever 1) an actor continues processing while a request it initiated is being actively handled by another actor on an open channel, or 2) a method handler's replacement policy allows designating a replacement actor prior to the handler's passivation. A perspicacious framework can infer parallelization opportunities from the action specification and induce multiple concurrent activations. Alternatively, the specification can include hints by, for example, distinguishing read-only and write actions that can be used to synchronize actions. A pessimistic replacement policy, on the other extreme, simply single-threads execution.

The `reserve` rule in Figure 4.10 entails a database update operation and a confirmation action. In this case, the fully-enabled replacement actor must include the database

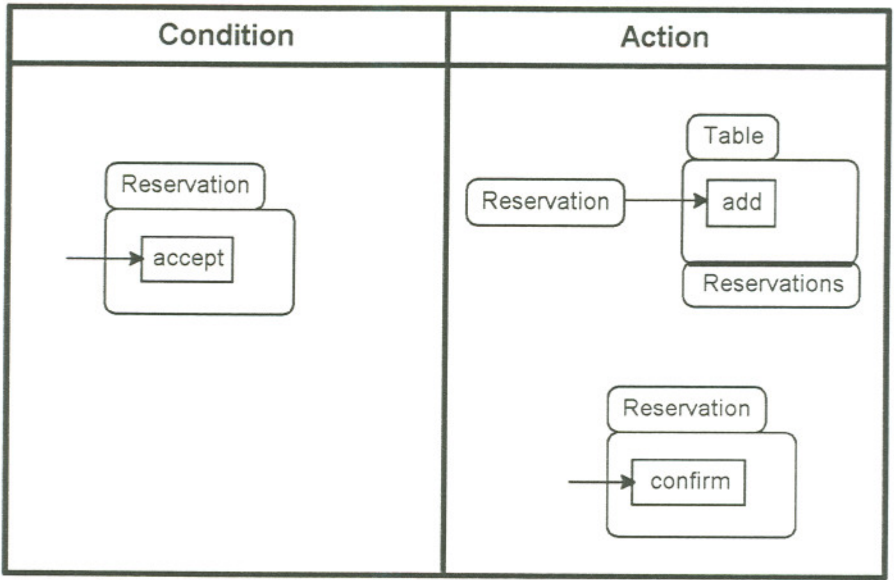

Figure 4.10: Reserve subactions

state change, but need not wait upon completion of the non-destructive and potentially lengthy confirmation action. A replacement can therefore be designated upon completion of `add` while the original `Reservation` actor remains engaged in the `confirm` action. The `Hotel` replacement actor references the newly-added `Reservation` frame and can activate a `Reservation` actor that is indistinguishable from the `Reservation` actor still engaged in the `confirm` action.

A method can have several associated constraints and rules. Hence, a connection may match the condition for several conditions and (potentially conflicting) rules. In general, a constraint or rule of an active frame is enabled whenever its conditions are satisfied. For example, the accept rule of Figure 4.10 could be broken out into two rules with identical `accept` condition and independent `add` and `confirm` actions. The two alternative specifications for the `accept` action would be semantically equivalent so long as the subactions were independent. For example, if the `confirm` action used the

Reservations table entry as input, as would be indicated by an arrow in the specification from Reservations to confirm, then splitting the rule into two rules would yield a semantically different specification, with no guarantee that the Reservations table included the new Reservation.

Alternatively, a default rule may be explicitly specialized to apply a more exacting condition. Consider the default rule for handling the reserve action in the absence of an available room presented in Figure 4.7 of §4.1. This rule is specialized to handle an overbook condition in Figure 4.11. The Hotel frame in Figure 4.11 refines the Hotel

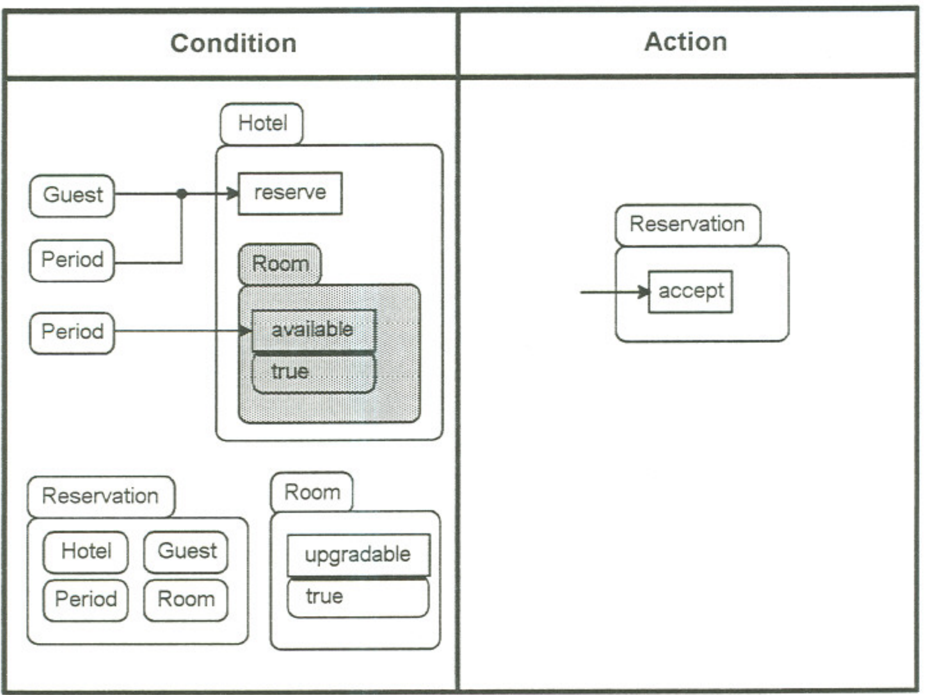

Figure 4.11: Overbook rule

frame in Figure 4.7, which itself is a refinement of the Hotel frame in the Hotel component interface. Both Figure 4.7 and Figure 4.11 match the Hotel fully booked condition. However, the specification in Figure 4.11 allows for accepting the reservation if there is some room matching the reservation request that, though not available, is upgradable

to an available room of similar or greater quality that could serve as a substitute during the given period. Each and only the most specific constraints and rules with a satisfiable condition are applied to a given active frame. Hence, Figure 4.11 applies in place of Figure 4.7 if and only if no room is available but some room is upgradable. Rule specialization implements a form of default reasoning with exceptions [ER83, Zad87].

An action specification is computational, since it is a rule for computing new values from existing values. However, the computation is descriptive, providing the essential semantics of object interactions, rather than prescriptive, indicating a candidate sequence of steps to implement the action. The subactions of an action are performed concurrently. Action atomicity is guaranteed only to the extent that a replacement actor is defered pending action completion. The `add` and `confirm` subactions of Figure 4.10 can be performed concurrently. The replacement `Hotel` actor is made available after completion of the `add` subaction.

The `reserve` action has no external effect on application state until this replacement actor becomes available. An external effect cannot be undone in the event of a partial failure. For example, a confirmation sent to a customer as a result of the `confirm` subaction is not withdrawn if the `add` subaction fails. A partial failure can be handled by either serializing the subactions or registering a compensatory action. Serialization is indicated by adding a constraint on the successor action with the predecessor action(s) as pre-condition(s), as in Figure 4.12. Actions should be serialized in the specification only when there is an explicit requirement to do so. The serialization mechanism is dependent on the underlying implementation environment. Preferably, action scheduling occurs at run time based on optimization criteria.

If there is an acceptable means of compensating for a failed subaction without impeding the progress of related subactions, then an alternative to serialization is a specification of a compensatory action in an *exception rule.* For example, the exception rule of Figure 4.13 specifies that a failed `add` action results in the sending of a note to the

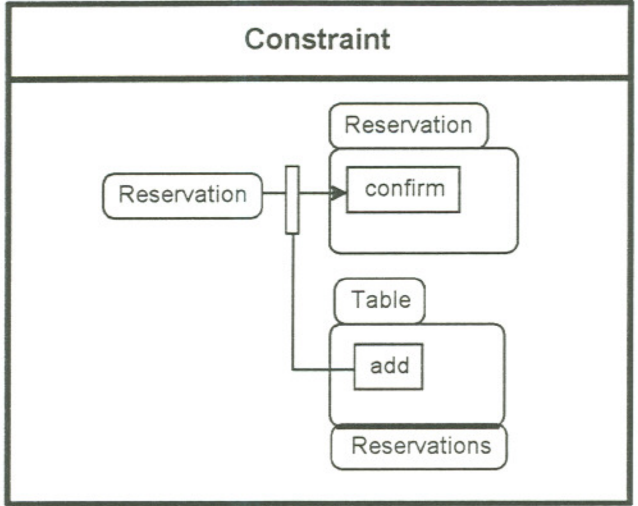

Figure 4.12: Confirmation constraint

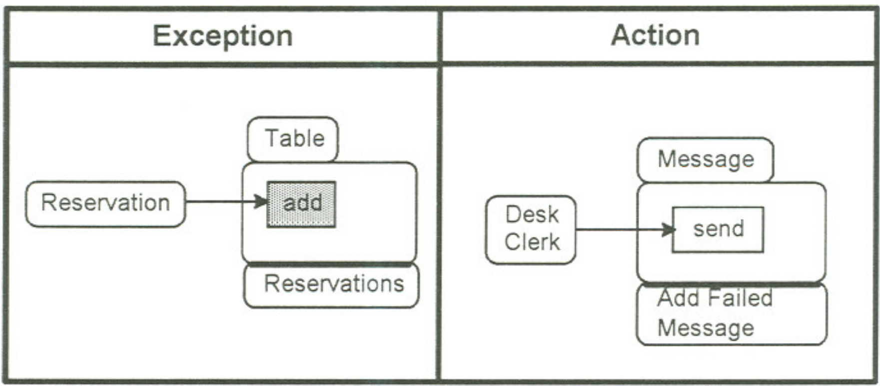

Figure 4.13: Reservation add exception rule

desk clerk, who presumably will correct the problem. The failure of the `add` action is indicated by the grayed `add` frame, representing the inability to complete the action. Graying out the method frame is analogous to the use of a grayed attribute frame, as in Figure 4.11, to represent the inability to match the shaded frame subject to the given condition. An exception rule is the mechanism to handle an exception whose detection is not possible or desirable prior to activation. If the action fails for any reason, every matching exception rule is enabled.

The rule may simply post an informational message or it can induce a corrective action of arbitrary complexity. Action atomicity is expressed by specifying an abort exception. In the example, if both the `confirm` and the `add` subactions had to complete before any change were committed, then an abort exception could be specified. An abort rolls back changes upon failure of any subaction to the extent possible in the underlying artifact implementation. The default compensatory action is to raise an abort exception for all actions initiated by the matched condition and raise an exception.

4.3 Monitoring actions

The activity of a component may be monitored by a specialized frame called a *monitor*. The monitor typically collects, filters and disseminates information or performs a discrete, ancillary action in response to a well-defined stimulus. An attribute) is monitored by attaching a *probe* to the relevant frame. The probe is stimulated whenever the attribute's value changes. If a value is supplied for the attribute frame, then the probe is stimulated whenever the attribute acquires this value. A probe may also be attached to a connection or method. A probe attached to a connection is stimulated whenever an action result is posted to a connection channel. The connection input and output frames may be supplied a value to qualify the conditions under which the probe is stimulated. A probe attached to a method is stimulated upon completion of the method's action.

The `Hotel Full` monitor in Figure 4.14 continuously monitors the reserve action for a fully booked condition. The probe is stimulated when the occupancy rate acquires the value 100%. Upon activation, a `Hotel Full` message is sent to the manager frame (Figure 4.15). Figure 4.16 shows two equivalent forms for the hotel occupancy probe. In the first case, the probe is stimulated whenever the `Occupancy` attribute changes value. In the second case, the probe is stimulated whenever the `Occupancy` attribute attains 100%. The extra condition on the `Occupancy` attribute redundantly restricts the `Occupancy` attribute to the value specified in the `Hotel Full` monitor's condition

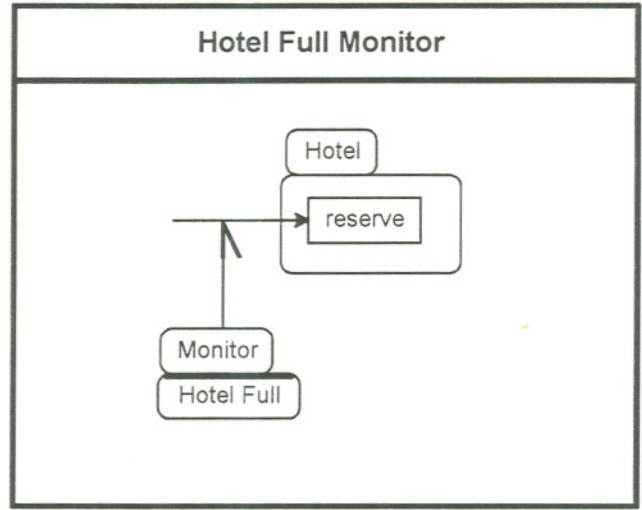

Figure 4.14: Hotel Full monitor on reserve

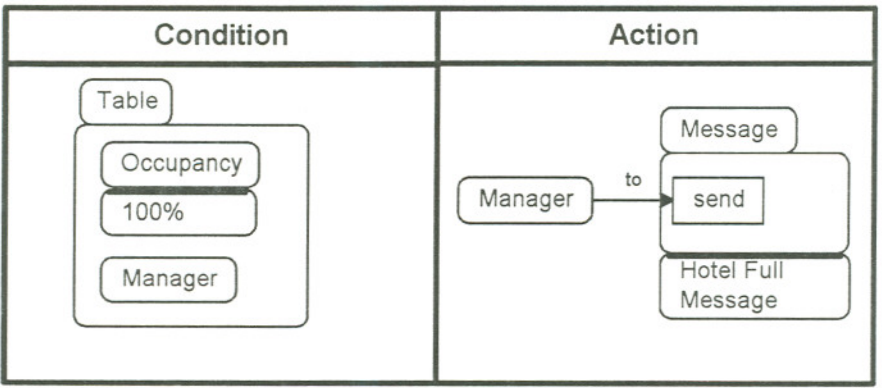

Figure 4.15: Hotel Full action

of Figure 4.15. The advantage of this redundant condition is to directly indicate the monitor's condition in the probe attachment, at the cost of redundancy and the attendant update consistency problems.

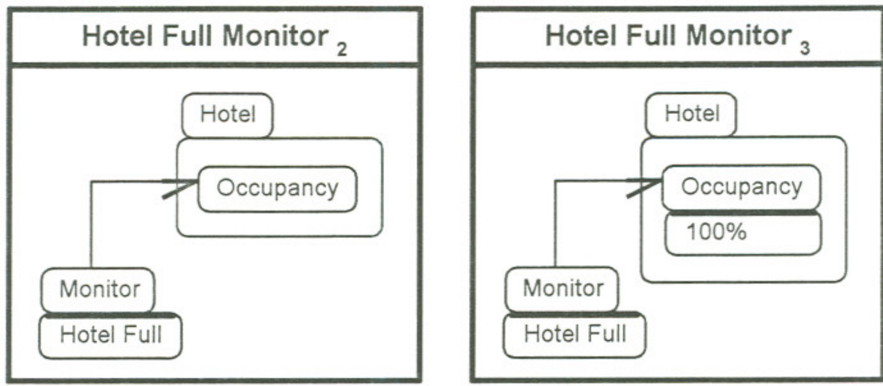

Figure 4.16: Hotel Full monitor on Occupancy

4.4 Agents: frames with an attitude

The static and dynamic specifications of object schemes we have described provide a natural scaffolding for overlaying goal-directed computation. The static component interface scheme specification describes the potential for frame activation; the action specification shows control flow. The activation heuristic in each case is simple—a static specification propagates activation through delegation, whereas an action specification activates an action in a rule whose condition is satisfied. Action propagation is opportunistic, or forward-chaining, in that every action with a satisfied condition in activated.

The simple activation heuristic of static and dynamic specification suffices to describe reactive systems under the control of an external intelligent agent, for example a user interacting with a system to satisfy an unknown (to the system) set of goals, beliefs and plans. The value of a system description is fully realized when this description is used to advantage in activities that engage the productive participation of purposeful agents. This motivates the ability to model such agents directly in the object specification. Representing agents in the specification is useful in development as a means of understanding goal-directed behavior in system interactions, e.g. usage scenarios in a workplace context.

Perhaps the greatest potential for a component specification framework, however, lies in *workflow automation*, the flexible activation of discrete component operations according to a plan to accomplish a set of goals. Workflow automation requires at least the ability to flexibly define an activity as a prescribed sequence of component operations. A more powerful approach is to represent the plans, goals and beliefs of the agent directly in the framework. An activity is then defined and initiated dynamically based on the agent's perception of the current situation in relation to the agent's goals.

An *agent* is a frame with a *propositional attitude* [LGP+90]. A propositional attitude may be construed as a second-order binary proposition whose arguments are a holder of the attitude and another proposition, e.g. `believes(Manager, HotelIsFull)`. The fundamental propositional attitudes of interest to reasoning systems are *belief* and *desire*. We deanthropomorphisize these terms for our purpose by recognizing the propositional attitudes of *assertion* and *goal*. An assertion posits a defeasible state of affairs that serves as adequate justification for provisional actions. A goal represents a prefered state that motivates the actions of an agent. The agent filters information to arrive at assertions, and acts on those assertions to realize its goals.

The `Overbook Authorizer` agent in Figure 4.17 holds as a goal the maximization of

Figure 4.17: Overbooking Authorization agent

hotel occupancy. To that end, the assertions that a class of rooms is full but upgradable leads to the `authorize` action to permit overbooking specified in Figure 4.18.

Assertions are characterized by a *local truth value* and *defeasibility*; they represent

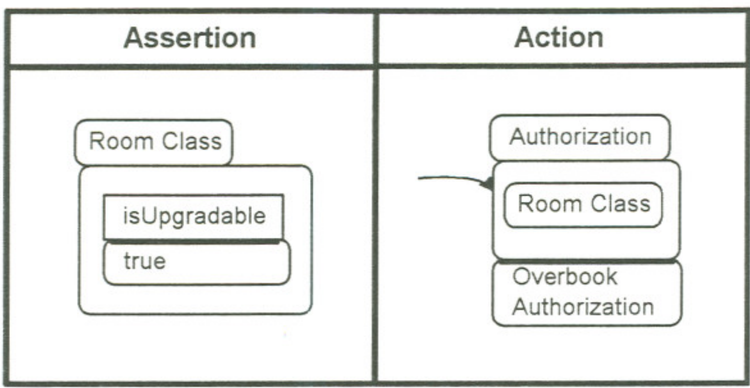

Figure 4.18: Overbooking Authorization rule

a state of affairs to the best of an agent's knowledge. For example, a travel agent may hold that a hotel room is available if and only if the room may be reserved, whereas the hotel manager may withhold certain available rooms from consideration for reservation or, conversely, overbook a room if the occupancy can be upgraded to an available suite. The difference may be due to temporal vagaries in information accessibility or, as is the case in the example, an intentional difference that represents the different perspectives of different roles. Thus, $assert(A, P) \supset P$ is true for a proposition P for a scope local to the agent A, but $assert(A, P) \not\supset P$ in general.

Furthermore, an assertion is *defeasible*, or subject to revision as the state of the world and an agent's perception of it changes. It is thus necessary to monitor assertion dependencies and detect changes that affect dependent assertions and goals. If a previously held belief is withdrawn based on new information, then it is necessary to invoke a compensatory action to rectify any actions based on the now discredited belief, and recursively analyze the belief dependency structure for new revisions. Figure 4.19 describes the `rescind` compensatory action activated whenever it is necessary to withdraw a previously approved authorization based on a change in one of the assertions that led to the authorization. The burden of truth maintenance [Doy80, dK86] is alleviated by specifying explicit compensatory actions.

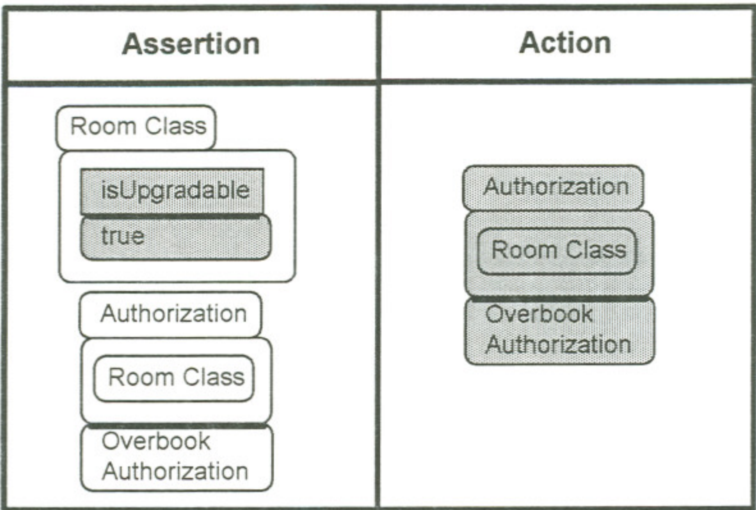

Figure 4.19: Rescind compensatory action

We have seen in this chapter the beginnings of a programming model for the component specification framework. An action specification serves as a visual script for enabling actor behavior. Actor scripting has three uses:

1. Experiment with the specification by simulating component interaction.
2. Validate an artifact against its component specification by observing its behavior under situations described in the specification script.
3. Build applications by scripting components and deploying agents to accomplish goal-directed tasks.

These uses represent a progressively increasing level of involvement of a component specification with its underlying artifact and associated tools. It is thus critical to the full employment of a robust component-specification framework in a shared, distributed environment that the software objects related to a component are managed in a flexible, consistent manner. Software object management appropriate to the framework is the topic of the next chapter.

Chapter 5

Object Management

5.1 Dimensions of Software Object Management

Software design and construction generates a large number of artifacts, including requirements documents, analysis models, design models, module interfaces, source code, documentation, test cases and verification results. The object management subsystem of a software development framework has the responsibility of organizing these artifacts in meaningful, consistent structures that support the development process. This responsibility is pronounced for design structures, since a design demonstrates purpose only when related back to requirements and guides construction by its relation forward to implementation.

We consider the following six dimensions of interest in software object management:

1. Function – organization of components within functional areas
2. Task – assembly of components to perform a work activity
3. Containment – collection within a containment hierarchy
4. Composition – aggregation of subcomponents
5. Time – version configuration of component content

6. Location – physical storage location of component information.

The points along these dimensions are, respectively, the library, activity, member, composite, version and handle for the object. For example, the `Hotel` component could be organized functionally in the `Reservation System` library, used in the `Add Waitlist Feature` task, a member of an `Exports` container, composed of Room subcomponents, currently at revision 6 and holding an artifact with location handle `//orion/project/rsv_sys/src/hotel.c`.

This chapter elucidates the role of a specification framework with respect to the function, task, containment and time dimensions. The composition of subcomponents as attributes within a composite component is described in Chapter 3. Location of specifications and artifacts is distinguished by its absence: the framework strives for the illusion of location independence, permitting free use of components in specifications without consideration for their location in a network of distributed applications and data. Achieving this illusion is a significant challenge, but one that relies on underlying object technology that is incidental to our purpose in this thesis.

Each of the dimensions are logically independent, in that a change in any one dimension need not necessitate a change in the other dimensions. Dimension independence is desirable in practice because it reduces interdependence of framework subsystems, facilitating a pluggable framework architecture and reducing the complexity of managing the application development process. Furthermore, component organization need not affect the contents of a view's tool-dependent representation. The framework encapsulates the view's tool-dependent representation and provides a consistent context for its use. References within the object to another object are represented as abstract component identifiers that resolve to the appropriate target within a given context. Contextual objects track the reference target's evolution, organization and location without affecting the content of the source object holding the reference.

It is the framework's responsibility to determine the context sufficient to resolve references and control evolution. The framework selects the appropriate component and view by filtering candidates based on this context. A view representation can then be materialized by mapping the abstract aspect of the view to the location of its concrete aspect and invoking the appropriate tool. The framework thereby provides assistance in locating view representations and maintaining component consistency. For example, resolving a reference to the `Hotel`'s `csource` artifact might use the information that the developer is involved in the `Add Waitlist Feature` as context for identifying the correct working version of the `csource` file.

5.2 Functional Partitioning

Any non-trivial system quickly faces the need to partition functionality into manageable, self-consistent units. This decomposition of large systems into semi-autonomous units is, in fact, the organizing principle of systems design. Object-oriented applications are particularly susceptible to this requirement because of the multiplicity, reactivity and specialization of objects. There are a large number of diverse objects that interact by responding to messages and may specialize the behavior defined for other objects. The fundamental functional unit identified by the framework is the component. Views are defined in order to permit operation on a component as the logical unit of abstraction.

A *library* serves to group components related to a broad functional area or subsystem. It is possible to share a single component among multiple libraries, or to represent a single conceptual entity by distinct components in different libraries. Sharing of a component among libraries and a view among components is based on development and deployment policy. For example, the hotel entity can correspond to the same or distinct components in the `Reservation System` and `Inventory System` libraries. If distinct, the two `Hotel` components can share views or evolve independently (Figure 5.1). Here, the `concept` views are shared, but relate to distinct `csource` views, `DCHotel` for the `Desk Clerk`

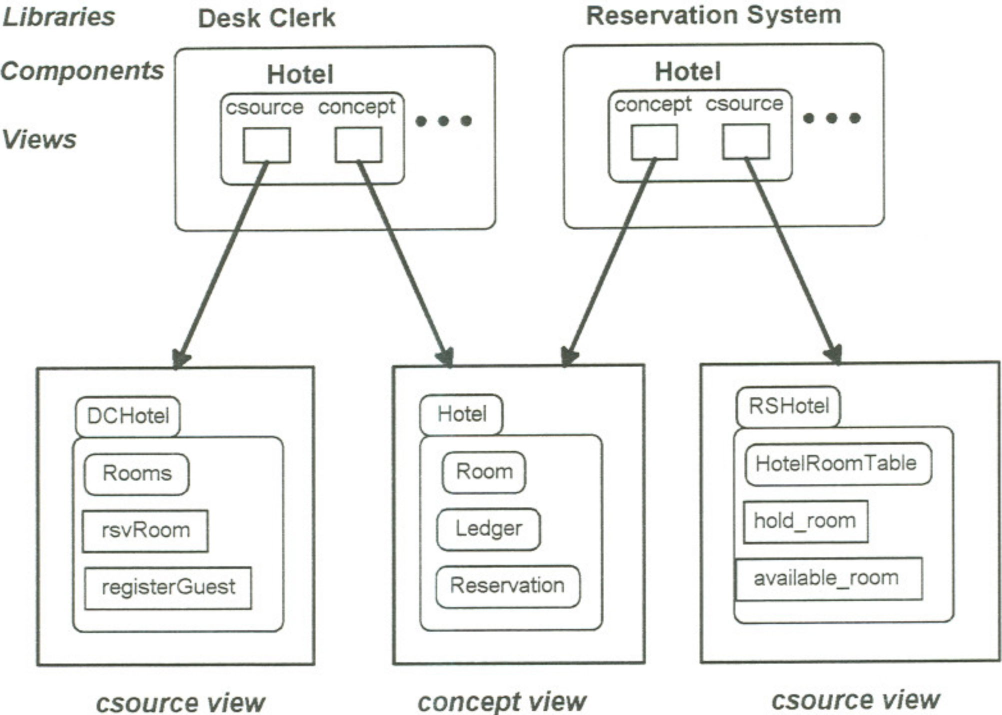

Figure 5.1: Hotel component in two libraries

library and `RSHotel` for the **`Reservation System`** library.

A library is a functional partitioning only, and does not carry an implication for the physical storage location of a view's concrete aspect. Location is set by a policy or as view metadata. In the example, the `concept` view's content facet might reside in a database, the `Desk Clerk` `csource` in a local file system, and the the `Reservation System` `csource` at a remote site.

5.3 Task Partitioning

The task dimension organizes components by use in accomplishing a discrete purpose, or `activity`. The activity sets the context for use and modification of individual components. The `Add Waitlist Feature` activity in Figure 5.2 uses the `Hotel`, `Room`,

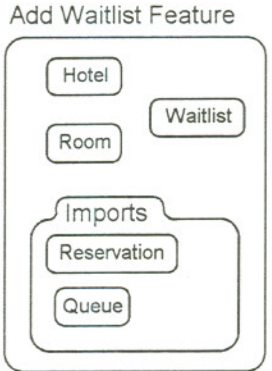

Figure 5.2: Add Waitlist Feature activity

`Waitlist`, `Reservation` and `Queue` components. The `Reservation` and `Queue` components are contained in a *folder*, or generic container, named `Imports`. An activity may encompass tasks for both an individual person or agent as well as a group of people or agents, and may be recurring (e.g. `Print Monthly Statement`) or transitory (e.g. `Add Waitlist Feature`).

The activity may model an application task (`Print Monthly Statement`) or a development task (`Add Waitlist Feature`). In either case, the framework is managing *meta*-objects of the application—specification objects that describe application objects—rather than the application objects themselves. An activity, as defined here, reflects operations *on* components rather than *of* components. The latter are described entirely by the action specifications of component methods. An activity, on the other hand, sets a context for selecting versions of components, views and artifacts.

Activities are grouped into a project or process to coordinate work effort. A *project* is intended to model one-time efforts, whereas a *process* models repeated efforts. (In the context of this discussion, the two are otherwise identical, and a description of one applies to both). A project (process) can contain subprojects (subprocesses), activities, folders and components. The `Desk Clerk` project in Figure 5.3 contains the `Registration` subproject, `Fix DB Post Bug` and `Add WaitList Feature` activities and `Imports` folder.

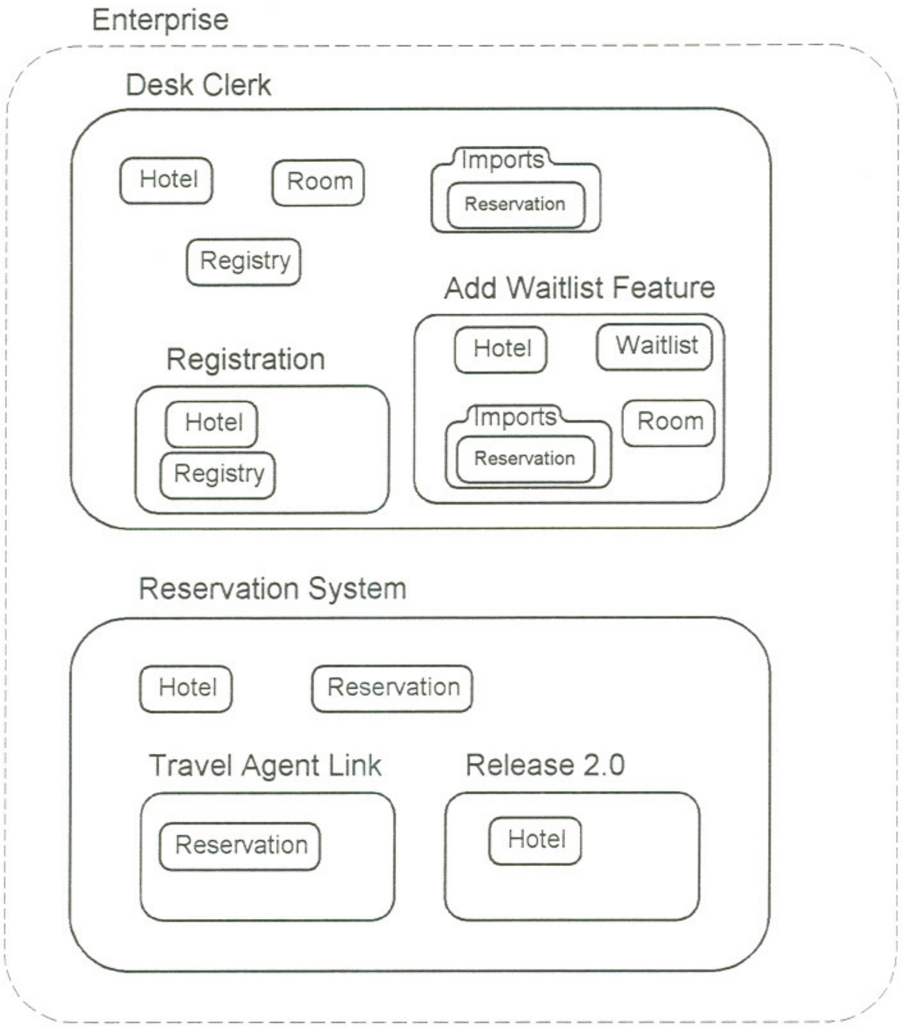

Figure 5.3: Example of projects

Another project, `Reservation System`, contains two subprojects. There is a distinguished top-level `Enterprise` psuedo-project that serves as the root of the activity hierarchy and contains all projects and processes accessible at a given site.

The activity hierarchy is chosen by its convenience for the task at hand, independent of physical location or functional organization. Projects are distinguished from libraries in that a library groups components according to function, whereas a project acts as a

locus of task responsibility. The two coincide when projects are defined functionally or libraries are organized by task responsibility. Identification of tasks with libraries is often the case for software development projects. A process, on the other hand, typically uses components drawn from assorted libraries for a particular business purpose. It is also possible to partition projects by physical location (for distributed development), project life-cycle (early phases separated from later phases), or any other convenient scheme.

5.4 Containment Hierarchy

The activity hierarchy is a special case of a *containment* hierarchy. There are five species of container objects: project, process, activity, library and folder. A folder acts as a general-purpose container to augment containment given by the activity hierarchy. Components are collected into a container by an *aggregation rule* that determines the criteria for inclusion. A common rule for an activity, for example, is *augmented selection*, selection of members of the parent project augmented by components peculiar to the activity. In the example of Figure 5.3, the `Add Waitlist Feature` activity contains the `Hotel`, `Room` and `Imports` members of the `Desk Clerk` parent project, as well as a new `Waitlist` component. More sophisticated aggregation rules are possible; the only requirement is that the rule can be evaluated and returns a set of components or other containers. For example, an `Approved` folder might have an aggregation rule that collects a set of approved components by filtering incoming e-mail for change approval notices.

Membership in containers is by independent, shared composite reference [KBG89]; that is, the existence of a component is independent of membership in any certain container, and a given component can be shared as a member of more than one container. A container is typically created by copying selected members of an existing container. Copying is a logical operation that creates a membership reference from the target container to the referent object. The `Imports` folder in the `Add Waitlist Feature` activity was, as the name suggests, copied from the `Imports` folder in the `Desk Clerk` project.

The Hotel and Room components were copied from the parent Desk Clerk project as well.

Change capability and visibility are conditioned by three membership reference properties: deferability, fixity and mutability. Deferability refers to whether the reference is *direct*, resolving directly to the referent or *defered*, defering resolution to another container. Membership can be further characterized by its fixity—a *fixed* reference always resolves to the same referent, whereas a *floating* reference may resolve to different objects over time. The mutability property determines the capability for changing the referent—a *mutable* member can be modified in place, an *immutable* member cannot.

Direct membership has the effect of isolating the container from changes in membership in other containers. Defered membership relies on another container to determine a member's contents. In Figure 5.4, Add Waitlist Feature holds a direct reference to

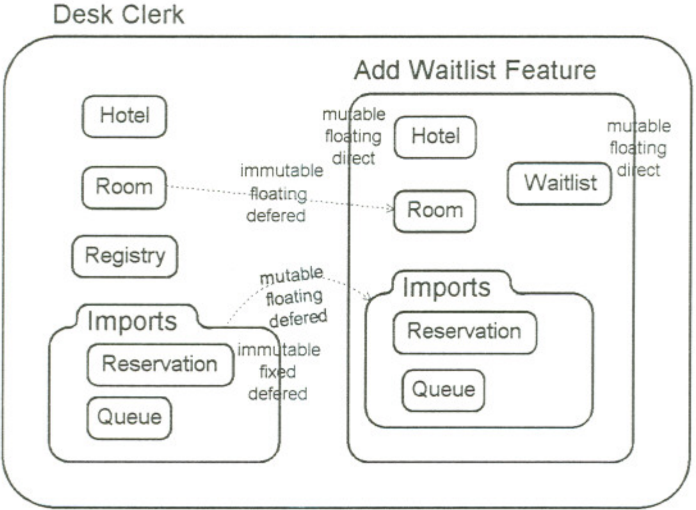

Figure 5.4: Membership references for the Add Waitlist Feature activity

Hotel and Waitlist, but defers to the Desk Clerk parent project to resolve the Room membership reference. Thus, a change applied to the Room member of the Desk Clerk project is visible to the Add Waitlist Feature activity. Hotel and Waitlist, on the

other hand, are unaffected by changes in the `Desk Clerk` project.

The `Imports` container is shared between the `Add Waitlist Feature` activity and the `Desk Clerk` project, and any change in membership made to either representation of `Imports` is reflected in both representations. Changes cannot be made to the `Imports` members themselves, however, since these are immutable references. They are defered through the source project from which the component was imported, but are fixed in order to preserve a stable copy of the imported components that is shared among all activities within the `Desk Clerk` project. The deferal retains the original project-component membership relationship but conceals it in the `Imports` folder. It is possible to recover the original membership by querying the original source container of the defered copy. Querying the source container enables navigation to an imported component's source project as it existed when the member was copied. The source is fixed at a stable, internally consistent snapshot of the project.

The effect of change given these three properties is portrayed in Figure 5.5. The first case *(i)* shows the effect of a change to a defered floating reference: the change in container `A`'s reference is reflected in the defered reference of container `B`. This is useful, for example, for an `Import` folder of one project which always reflects changes made to membership of another project's `Export` folder. In the second case *(ii)*, the change to container `B`'s direct reference does not affect the original container `A`'s reference. This is equivalent to a deep copy of the original container, for example an `Import` folder whose members come from an `Export` folder but thereafter lose all correspondence with the source. `B`'s defered fixed reference in *(iii)* is unaffected by a change in container `A`'s reference. The fixed reference has the effect of freezing the reference at a point in time. This applies whether the reference is direct or defered; if the reference is defered, then this has the effect of freezing a virtual copy of the original container holding a direct reference (`A` in this case). This is useful, for example, to fix the `Import` folder's membership at that of the `Export` folder at a point of time, isolated from subsequent changes to the `Export` folder. Finally, the referent in *(iv)* is shared by the original container `A` and

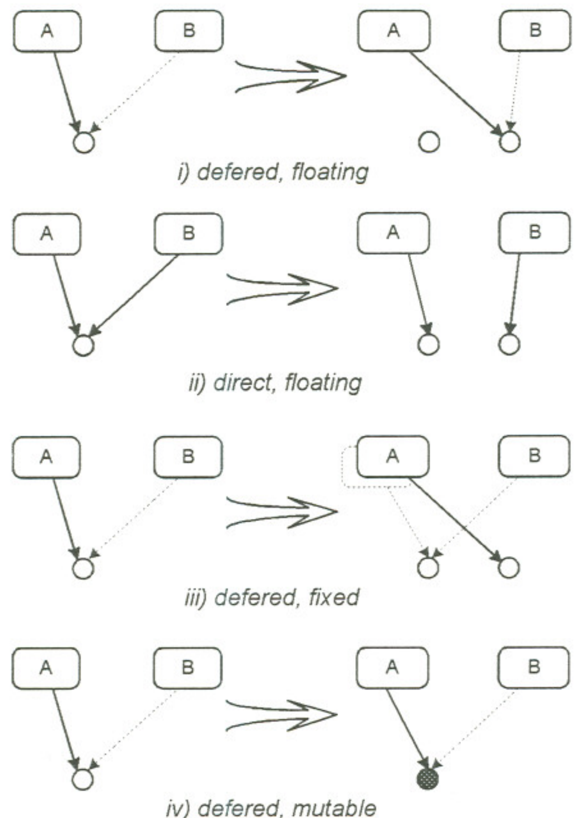

Figure 5.5: Effect of membership properties on change visibility

the container `B`, and changes to the referent through the defered, mutable reference are applied directly to the shared referent and visible to both `A` and `B`. This is similar to a *link* in that a change made to a member of **`Imports`** is reflected in the corresponding member of **`Export`** as well.

These three properties are orthogonal, distinguishing which reference is changed (deferability), and whether the change is applied to the reference (fixity) or the referent (mutability). However, some combinations of membership properties are more meaningful than others. Combinations of chief interest characterize susceptibility to change, whether by modifying a reference or its referent. These are subsumed in a composite *stability* property—a *stable* reference is fixed and immutable; a *volatile* reference is one

that is not stable.

In summary, the aggregration mechanism described in this section balances flexibility, economy and protection. Components are organized in libraries based on functional criteria. Folders can be arbitrarily assembled from components and other folders. A container can defer to other containers to resolve membership or be isolated from changes made in the context of other containers. The deferability, fixity and mutability properties, taken collectively, provide a mechanism for implementing access protection and change control policies by restricting the permissible combinations. For example, a policy that constrains accessiblity to its mutable members to direct, floating references only allows updates that do not affect the source containers. Conversely, a source container which only allowed a target container to establish defered, fixed, immutable references to its members prohibits changes to its members from any access path except that of the source container; furthermore, it isolates any changes made in the source container from affecting the members of existing target containers. Deferal also provides the mechanism for implementing versioning as a means of change control, considered in detail in the next section §5.5.

5.5 Managing change

Components are subject to concurrent development and use in evolving systems. The component specification framework is a natural foundation for tracking changes to components and ensuring the consistent use of compatible versions. The project and activity containers define a context for change and use. Associated with each such context is a *workspace* that controls the visibility and accessibility of software objects. Workspaces are used to promote integrity and concurrency by controlling access to design evolution, isolating change and permitting evaluation of local changes in a broader context. A workspace is an administrative mechanism whose control is at the discretion of the designer or system integrator.

A workspace acts as a filter on the universe of software objects in an enterprise. Components and views are filtered "horizontally" by a projection of a subset of relevant members and "vertically" by a selection of versions visible in the workspace. Figure 5.6 illustrates version derivation (thin line) and version selection (thick line) of components.

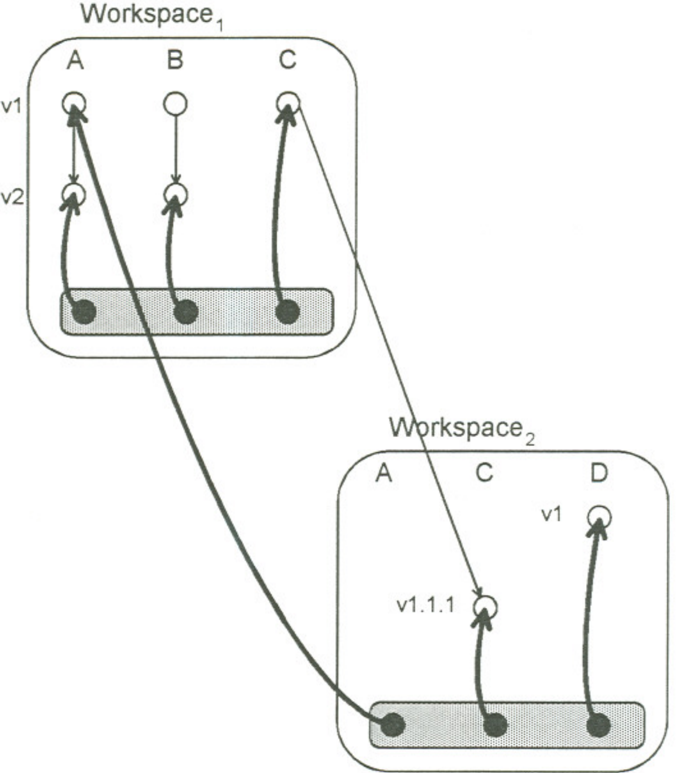

Figure 5.6: Workspace structure

$Workspace_2$ selects components `A` and `C` from $Workspace_1$ augmented by the new component `D`. The versions of `A`, `C` and `D` selected are 1, 1.1.1 and 1, respectively. Version 1.1.1 of `C` is a new version created in the context of $Workspace_2$ and is owned by that workspace. It is derived from version 1 owned by $Workspace_1$. The workspace maintains a version graph managed by a *history object* for each object within its scope.

Workspaces are organized into a hierarchy that mirrors the project-activity hierarchy. A parent workspace owns versions created in that workspace; a child workspace has

selective visibility to versions in the version graph of the parent workspace, as well as versions created in the child workspace. In the example, Workspace_1 is a parent of Workspace_2. A *configuration* distinguishes the current versions of the objects in the workspace. The (shaded) configuration in Workspace_2 of the example selects version 1 of `A`; the parent workspace's configuration selects version 2.

A child workspace evolves a software object from a version owned by its parent with the *check-out* operation. The check-out creates a new version owned by the child workspace and sets the currency of the history object to this version. Typically, the parent's current version is checked out; this establishes a local context for object evolution and evaluation that isolates the impact of changes from the shared global context of the parent workspace. However, the check-out can be of a version which is not the current version of the parent, in which case the current version of the child workspace is not a successor of the current version of the parent workspace. In this way, alternatives can be explored and evolved in a local context without affecting the global notion of currency of a software object.

A new version is promoted to its parent workspace by the *check-in* operation. This creates a successor reference from the source of the check-out to the subject of the check-in. The parent's version graph is amended to include this successor and, if the immediate predecessor is current in the parent, currency is moved forward to the new successor. No intermediate versions which may have been created during the object's evolution in the child workspace are promoted to the parent workspace. These intermediate versions are retained in the child workspace, and may be preserved for historical reasons or purged if no longer needed.

Component `C` in Figure 5.7 was checked out to both of the sibling Workspaces 2 and 3. It is evolved in Workspace 2 to versions 1.1.1 and 1.1.2; version 1.1.2 is subsequently checked into the parent and relabeled version 2. Meanwhile, Workspace 3 has evolved version 1 to versions 1.2.1 and 1.2.2. Version 1.2.2 is then merged with version 2 in the

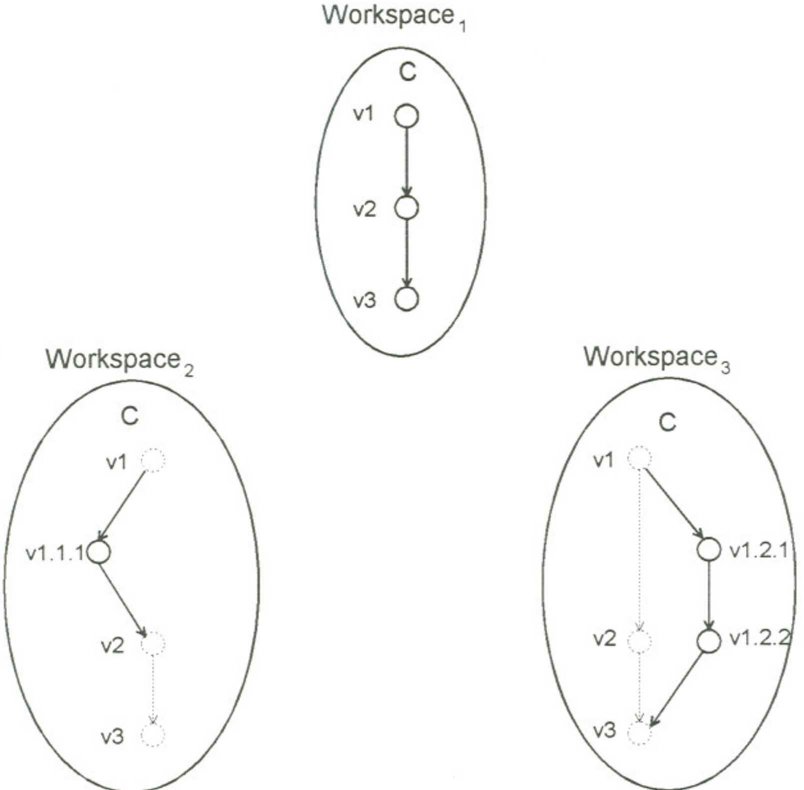

Figure 5.7: Workspace version graph visibility

parent workspace to form a new version 3 upon check-in. Intermediate versions are still visible in the child workspace, but these are not selectable in the other workspaces. The visible versions and change-derivation edges are either owned by the workspace (solid line in the example) or inherited from the parent (dashed line). Any redundant edge, such as that connecting versions 1 and 2 in Workspace 2, is removed.

A child workspace may simply designate a current version from the parent workspace without an intent to evolve it. In this case, the version history maintained by the child workspace is the same as the version history of the parent workspace, but the currency of the child workspace overrides that of the parent. Furthermore, any change to the currency in the parent workspace is not reflected in the child workspace. Fixing currency

in the child workspace effectively takes a snapshot of the object in the parent workspace and freezes the currency of that object while it remains in the child workspace. For example, Component `A` in Figure 5.6 is frozen to version 1.

The set of current versions in a workspace defines a default *configuration*, or set of consistent versions, for that workspace. The configuration relates entities to versions; this partial monomorphism determines version currency. A configuration is a software object and may evolve; a configuration object is a history object that manages configurations as versions. The default configuration of the workspace is then the current configuration in this configuration object's version history.

The default configuration is different from other software objects in the workspace in that it is not an immutable version; every check-in operation updates the default configuration in place. Configurations are versioned by the *snapshot* operation instead. A snapshot commits the default configuration as an immutable back version and creates a copy as the new, mutable default configuration. Frozen shared configurations are typically labelled with a meaningful version designator such as `rls1` or `v3.1`.

A developer may freeze a shared configuration by making a local copy of the default configuration in her individual workspace and using the local copy. Any change to the parent's default configuration does not affect the local copy. An entity in the local copy may be related to any version of that entity visible in the development workspace. The local copy can be pared down to include just objects of interest, included in another configuration and subsequently discarded, all without affecting the version binding of other developers.

Configurations are used to construct an *environment* that binds each referencable entity to a specific software object. The environment maintains a partial order on a collection of configurations. This partial order determines the version binding of the entities represented in its configurations. Each workspace contains a default environment, which defines a sufficient context for the evaluation of all references held by software

objects within that workspace. The default environment typically consists of the transitive closure of default configurations over the ancestors of the workspace, ordered by the parent relationship, as well as selected configurations of workspaces that own referenced software objects not included in the ancestry.

The example in Figure 5.8 shows public workspaces corresponding to the `Desk Clerk`,

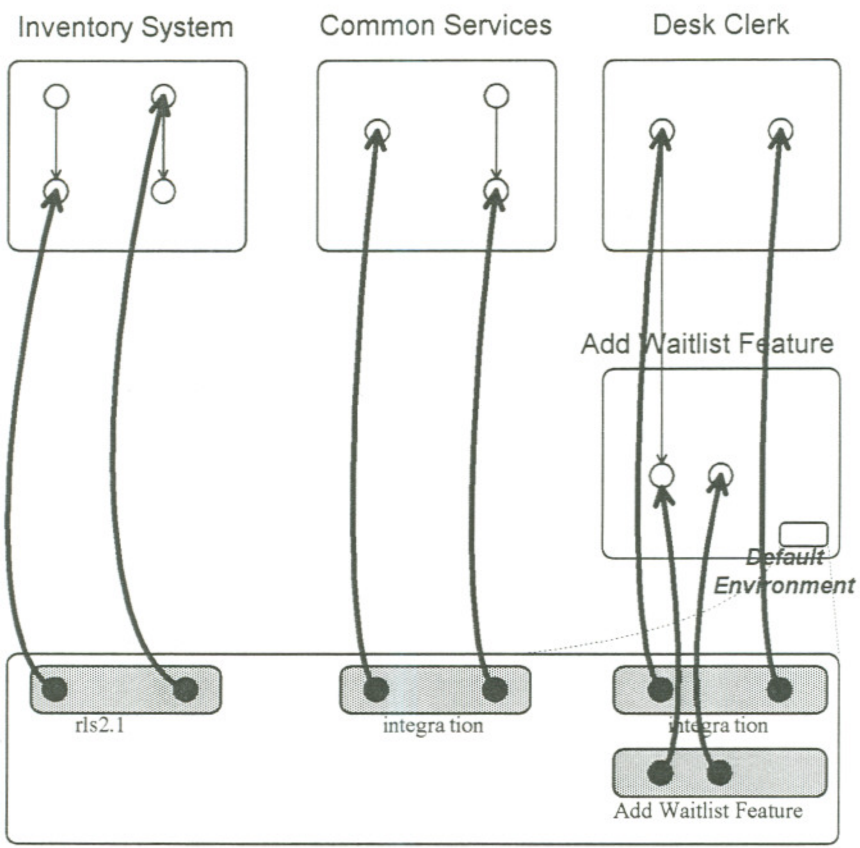

Figure 5.8: Environment example

`Common Services` and `Inventory System` projects, a private workspace for the `Add Waitlist Feature` activity, versions of entities (circles) within these workspaces, the default environment of the `Add Waitlist Feature` workspace (at bottom), and four configurations (shaded) corresponding to the four workspaces. The four configurations

include the default configuration for the `Add Waitlist Feature` activity, configurations labeled `integration` for the parent `Desk Clerk` and imported `Common Services` projects, and the configuration `rls2.1` for the `Inventory System` project. In this example, a `Desk Clerk` entity will bind preferentially to a software object checked out into the `Add Waitlist Feature` workspace. Otherwise a `Reservation System` entity will bind to the version in the `integration` configuration of the parent workspace. References to `Common Services` and `Inventory System` components always bind to the versions called out by the `integration` and `rls2.1` configurations, respectively.

Potential ambiguities arise when a single entity is represented in more than one sibling workspace. For example, if a released component is checked out into two sibling integration workspaces and a developer's workspace is a child of both of these integration areas, then the environment has insufficient information to select a prefered version of the component. Such ambiguities are resolved by a developer decision, e.g. by copying the prefered binding to a local configuration that takes preference over the conflicting parent configurations.

An environment, like a configuration, is a software object managed in a workspace. There may be several environments in a workspace; different environments represent different perspectives on the design space and can lead to different evaluation results. It is possible to create recursive relationships of configurations within environments within configurations, and is entirely reasonable to do so. For example, it may be desirable for traceability to capture the `Add Waitlist Feature` environment used to bind versions prior to commiting and archiving the changes made as a result of this activity. This can be accomplished by including the `Add Waitlist Feature` environment itself as an entity selected by the `Add Waitlist Feature` configuration.

5.6 Reference Resolution

A component was previously defined as a software object that collects views of a conceptual entity. A component may now be understood as a specialized configuration object—a version of a component is the configuration that collates a consistent set of versions of views. A workspace configuration contains component configurations, managed as composite objects. For example, the `Add Waitlist Feature` configuration may select version 2 of the `Hotel` component, which in turn selects versions 2, 2 and 3 of its constituent `concept`, `design` and `csource` views, respectively.

A configurable reference takes the form $\langle Component, View \rangle$, where *Component* and *View* are abstract entity surrogates. The binding of a configurable reference to a particular version of a view is a three-step process:

1. The active environment selects a context for evaluating the reference.
2. This context determines the binding of the *Component* entity to a component version.
3. The configuration represented by the selected component version binds the *View* entity to a version of that view.

The bound version of the view is then available for use, for example to construct a frame from the view's scheme. Continuing the `Add Waitlist Feature` environment example of Figure 5.8 in Figure 5.9, a reference to a `Hotel` component's view can now be understood as a configurable reference through the environment to a version of the `Hotel` component, then to the version of the view as configured by this component version.

The context used to evaluate a reference is typically the configuration managing the component with the highest precedence in the active environment. Configuration precedence ensures that every referenced component resolves to the same version when

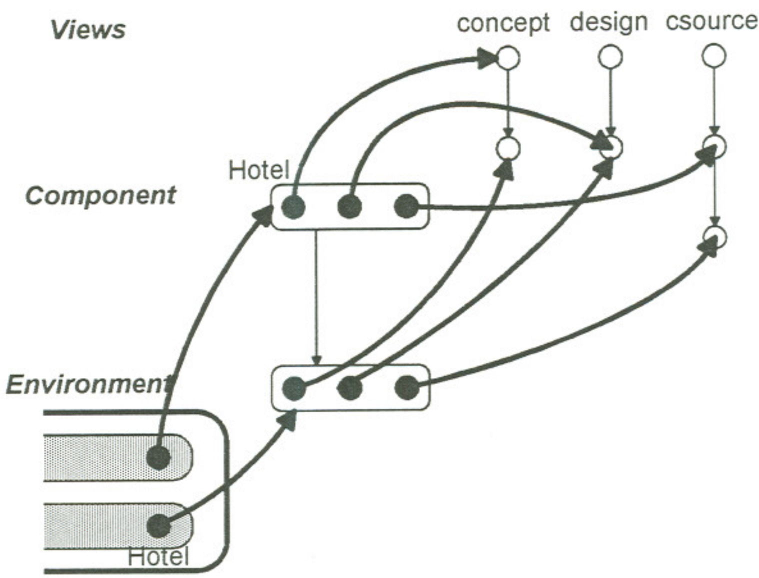

Figure 5.9: Configurable reference example

evaluated in that environment. However, it is occasionally desirable to treat a referenced component as a black box for evaluation purposes, so that any reference from that component is evaluated in a context different from that given by the active environment.

In the example, if a component imported from the `Inventory System` references `Hotel`, it may be desirable to resolve this reference using the environment in effect when the `Inventory System rls2.1` configuration was built. This alternative resolution reproduces the reference evaluations in effect at the point of release and ignores any changes that would otherwise be visible in the current active environment. The resulting evaluation treats the released component as a black box upon which the active environment has no effect. An imported `Inventory System` component is then treated as a self-contained entity unaffected by changes to components in the private workspace.

The first step of reference resolution is complicated somewhat by this flexibility, since the active environment must maintain a *context map* that relates a component to the

context used to evaluate references from views of that component. By default the context map is empty, signifying that all references resolve through the active environment. However, a component or container of components may be explicitly mapped to a different environment by a user action. In the presence of a context map, all references directly or indirectly from a mapped component will be evaluated in the new context. The second and third steps of reference resolution remain the same, modulo this context switch.

In the example of Figure 5.10, the `Inventory System` and `Desk Clerk` projects use different versions of a common `Hotel` component. References from an `Inventory System` specification are mapped through an `Inventory System` environment, whereas references from an `Desk Clerk` specification are mapped through the `Add Waitlist Feature` environment. Thus, a frame bound to the `Hotel` component slot in the `Inventory System` specification conforms to a prototype specified in version 1 of the `Hotel` component, whereas a `Desk Clerk` specification conforms to a version 1.2 prototype.

Configuring reference resolution through a context map yields a more controlled evaluation by isolating the scope of changes within the active environment, but precludes an understanding of the impact of local changes on a shared environment. A context map is appropriate when there may be multiple versions of an object active in an operating environment with dynamic communication between statically bound components. For example, the `Hotel` component specification could reflect a dynamic request from version 1.2 of a `Hotel` component to an `Inventory System` component, which in turn statically embeds version 2 view of the `Hotel` component. In this case, it is desirable to treat the imported `Inventory System` component as a black box unaffected by local changes to `Hotel` in the `Desk Clerk` project. Context-sensitive version selection faithfully models a situation involving several active versions of the same object. Activation of multiple component versions is potentially confusing, and the benefit of faithfully modeling such a situation must be balanced against the potential for propagating confusion in the specification.

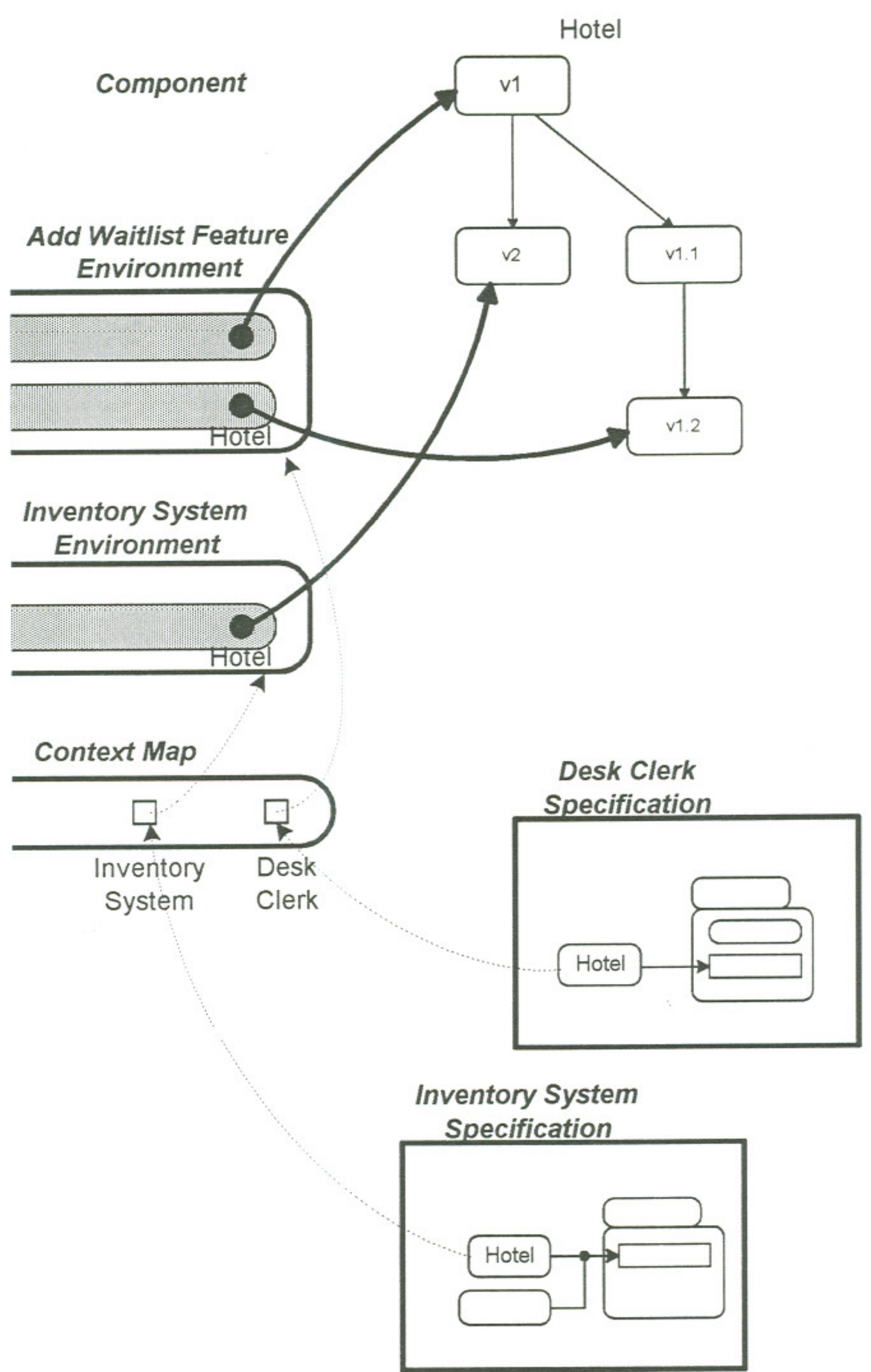

Figure 5.10: Reference resolution through a context map

The fidelity-complexity trade-off is a recurring theme in software object management. We will return to this topic in the conclusion. For now, let it be noted that the goal of the framework is to manage software complexity rather than mask it. The model of object management developed in this chapter attempts to relate components to artifacts in a flexible and consistent manner. The next chapter illustrates this model by describing a proof-of-concept prototype that implements many of the concepts.

Chapter 6

A Component Specification Framework Prototype

This chapter describes Meros[1], a tool suite for managing object specifications. Meros is a proof-of-concept prototype for demonstrating features presented in this thesis. The prototype consists of the following tools:

Project Browser	organizes activities within projects
Library Browser	collects components in libraries
Component Editor	browses and edits component specifications
History Browser	tracks the evolution of components

Tool usage will be described by way of example in §6.1, followed by a discussion of scheme representation in §6.2.

[1]Meros is from the Greek *μερος*, signifying "part" and connoting an essential aspect of an object as manifested in its interaction with other objects in a larger context:

> An Anaxagorean *μοῖρα* is a 'portion' in the sense of a 'share' rather than of a 'piece' or 'particle'. The essential characteristic of such a 'portion' seems to be that it is something which neither in theory nor in practice can ever be actually reached and separated out from that which contains it.
>
> — G.S. Kirk and J.E. Raven [KR57].

Initiator	Action	Participant	Service
Sue	open Desk Clerk Project	Activity Editor	open a project
Sue	create Waitlist Activity	Desk Clerk Project	add an activity
Sue	browse Hotel Mgmt Library	Hotel Mgmt Library	open a library
Sue	select Hotel Component	Hotel Component	select component
Sue	browse Hotel History	Hotel History	display history
Sue	evolve Hotel Version	Hotel History	create a version
Jay	browse Hotel History	Hotel History	display history
Sue	open Hotel Component	Hotel Component	open a component
Sue	create Waitlist Method	Hotel Component	add a method
Sue	open Reservation Spec	Reservation Comp	open a spec
Sue	clone Reservation Frame	Hotel Spec	clone a frame
Sue	create Reservation Slot	Waitlist Method	add a slot
Sue	save Hotel Spec	Hotel Component	save a spec
Sue	promote Waitlist changes	Waitlist Activity	promote activity
Jay	browse Hotel History	Hotel History	display history

Table 6.1: Script for the Add Waitlist Feature use scenario

6.1 A specification example

This example traces a use scenario [RG92] for adding a new feature to an existing application. A script for this scenario is presented in Table 6.1. The steps in this script follow the actions of a hypothetical developer, Sue, in adding a waitlist feature to the `reserve` action in the Desk Clerk tool of the Hotel Reservation application.

Sue begins by creating an **Add Waitlist Feature** activity within the Desk Clerk Project in Figures 6.1 and 6.2. The Activity Editor is invoked from the Meros Tool Dashboard in Figure 6.1. The activity hierarchy consists of the `Desk Clerk` and `Reservation System` projects, denoted by the icon, and several subprojects and the `Fix DB Post Bug` activity, denoted by the icon. The **Add Waitlist Feature** activity added in Figure 6.2 defines a new context for resolving component references by creating an associated workspace for controlling the evolution of objects modified while this activity is active. Subsequent changes to components are local to this activity's workspace until promoted to the parent Desk Clerk Project. A reference to a component checked out to the **Add Waitlist Feature** activity resolves to the version current in that workspace. Other references are resolved in the parent Desk Clerk Project context.

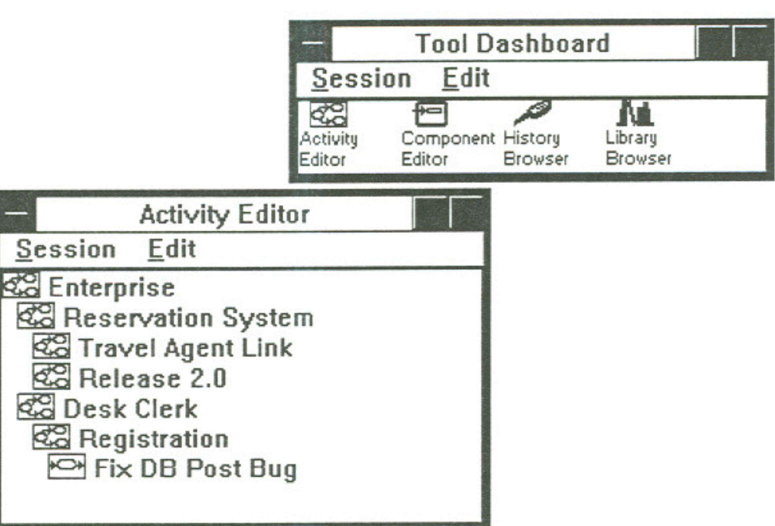

Figure 6.1: Open Activity Editor

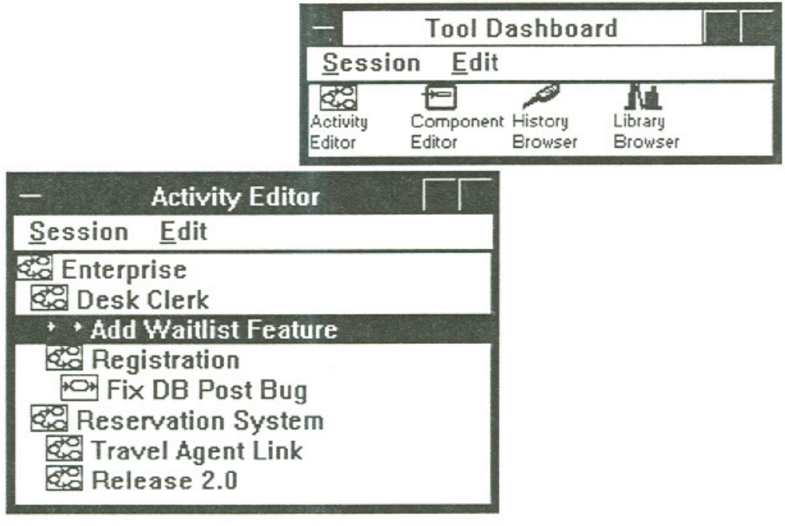

Figure 6.2: Add New Activity

A component view may be checked out to the active context explicitly by editing that object's version history or implicitly by editing the view as a generic object. In the example, Sue browses the `Hotel Management` library using the Library Browser and

selects the generic `Hotel` component (Figure 6.3).

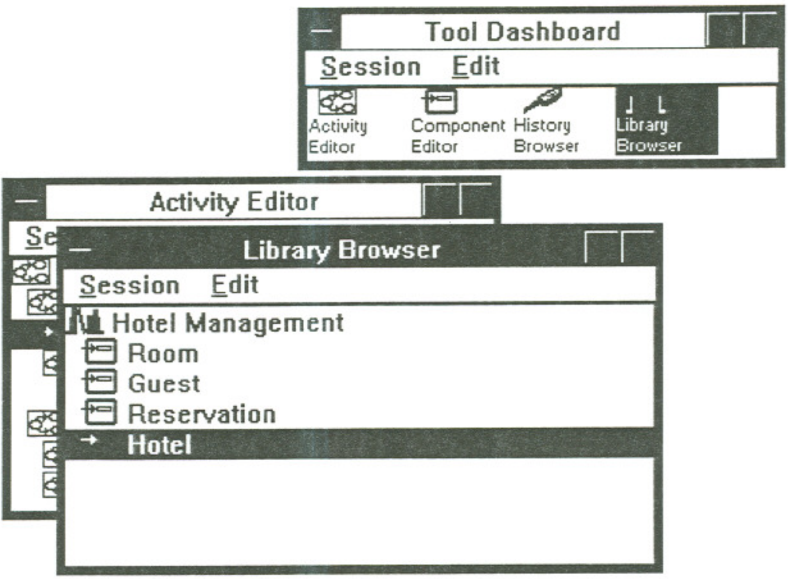

Figure 6.3: Select Hotel Component

At this point, Sue could check out the version of the object current in the parent workspace implicitly by directly opening the generic `Hotel` component presented in the Library Browser. She chooses to check it out explicitly by opening the object's history (Figure 6.4) and evolving the version marked as current (Figure 6.5, page 76). This action

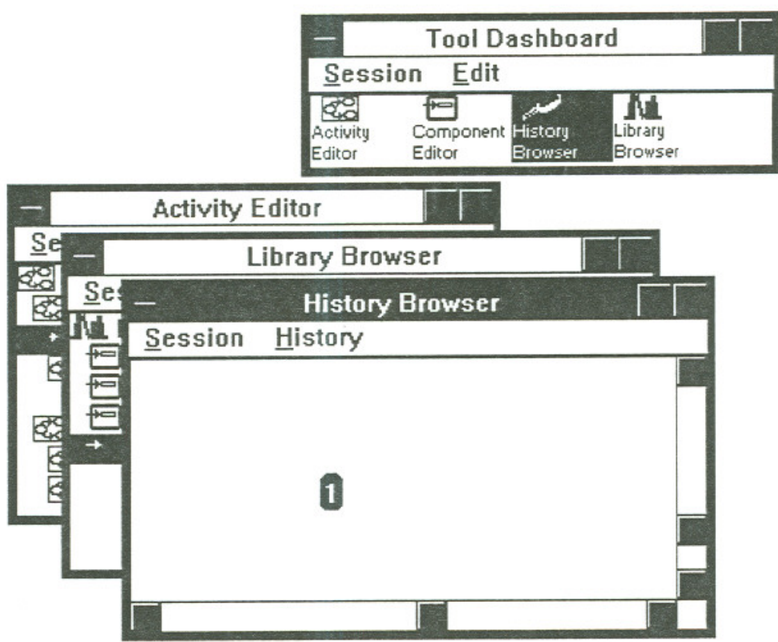

Figure 6.4: Browse Hotel History

creates a new editable version of the Hotel view that is isolated from other development activities. Thus, another developer, Jay, operating in the parent workspace retains version 1 of `Hotel` as the active version (Figure 6.6, page 77).

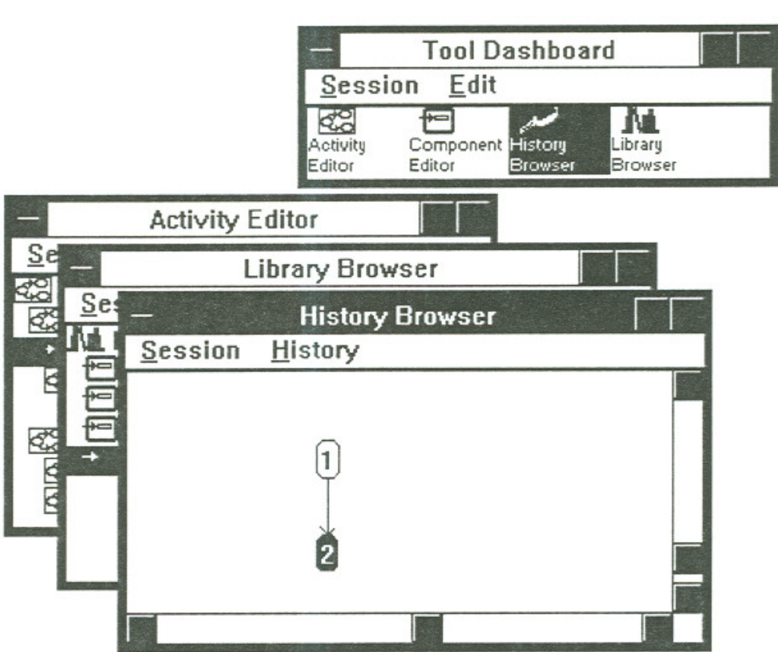

Figure 6.5: Evolve Hotel History

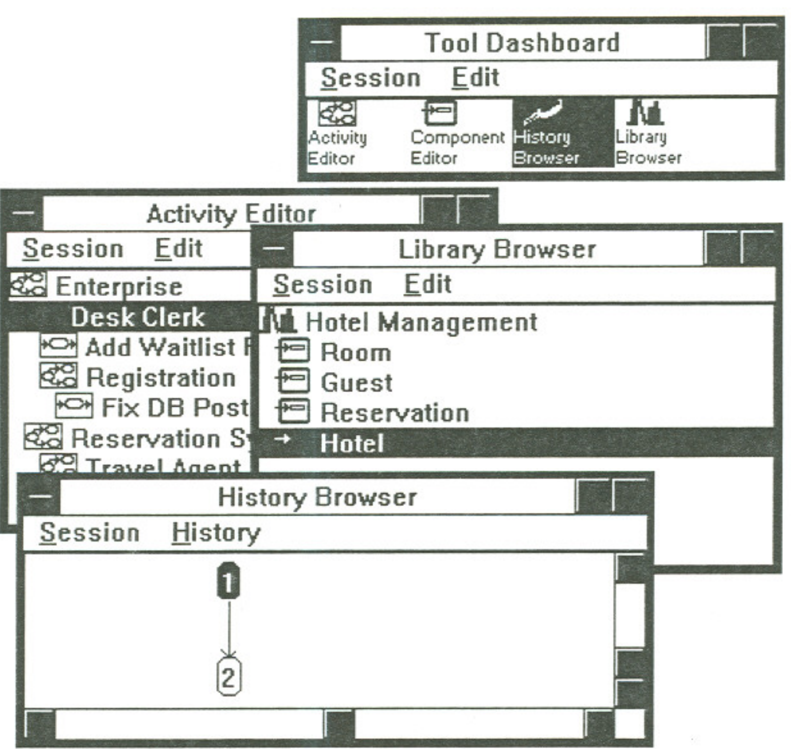

Figure 6.6: Jay's active version

Sue, meanwhile, opens the `Hotel` interface scheme (Figure 6.7) and creates a new

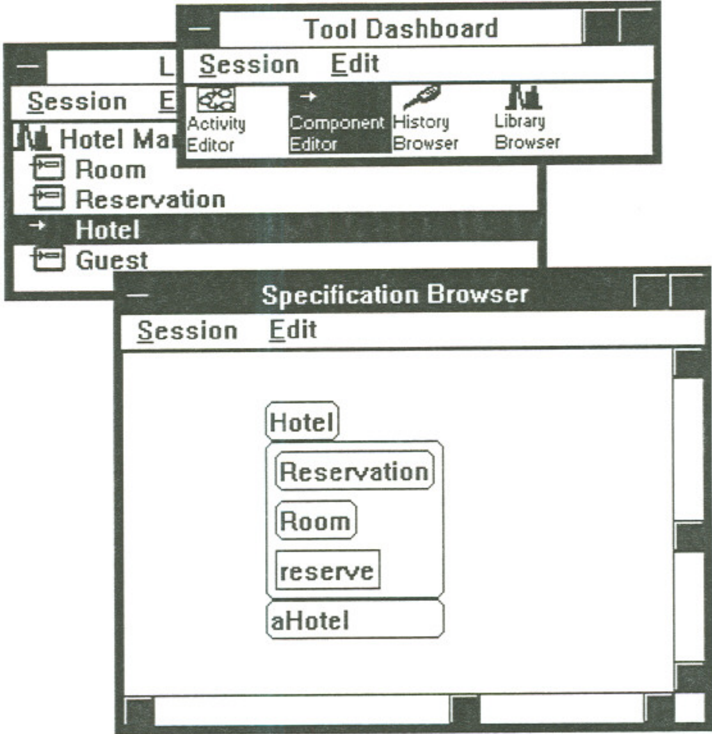

Figure 6.7: Open Hotel component

`waitlist` method (Figure 6.8). Sue must add a `Reservation` slot to the `waitlist` method. Adding the slot is accomplished by selecting the appropriate `Reservation` component interface scheme (Figure 6.9), creating a clone from this prototype in the `waitlist` specification (Figure 6.10), and attaching this frame to a new slot labelled `rsvn` (Figure 6.11). Sue then saves the specification and promotes this change to the parent project workspace. This action makes the change visible in other developers' contexts (Figure 6.12).

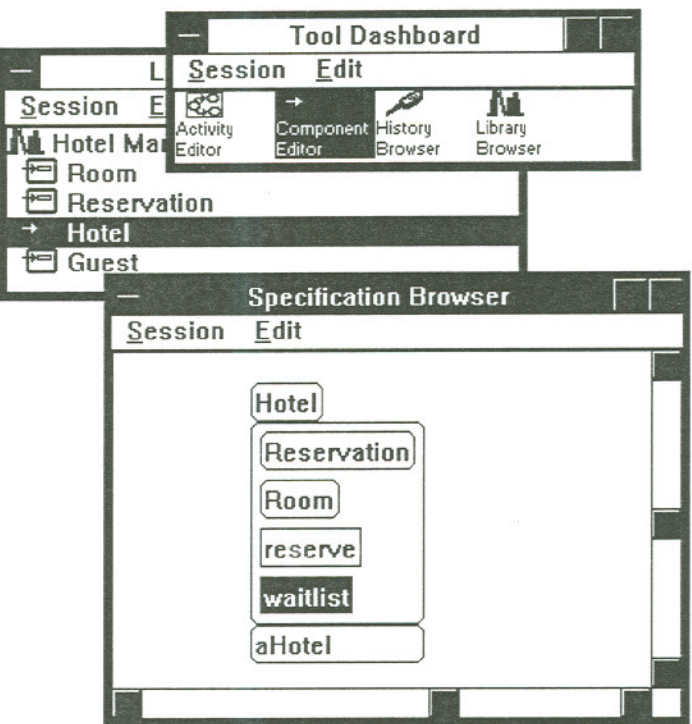

Figure 6.8: Create waitlist method

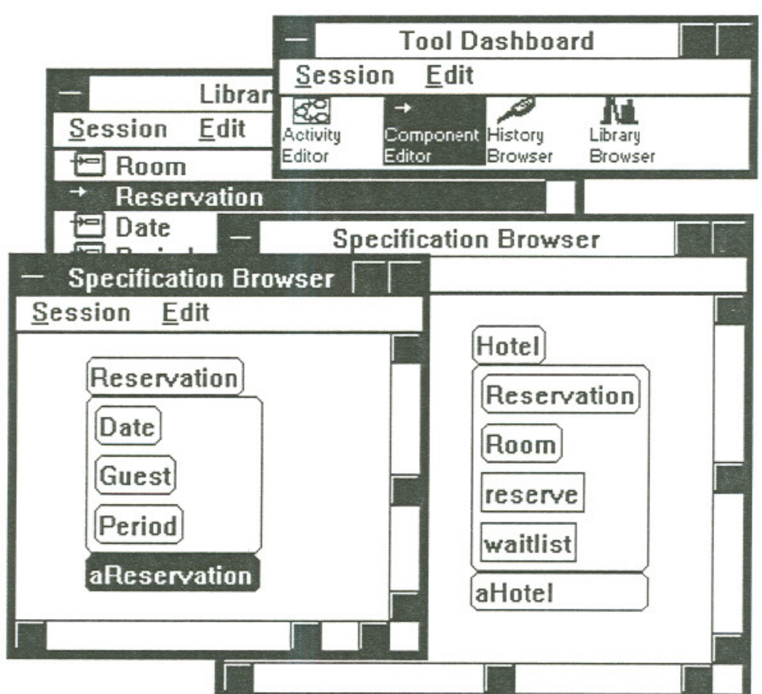

Figure 6.9: Open Reservation scheme

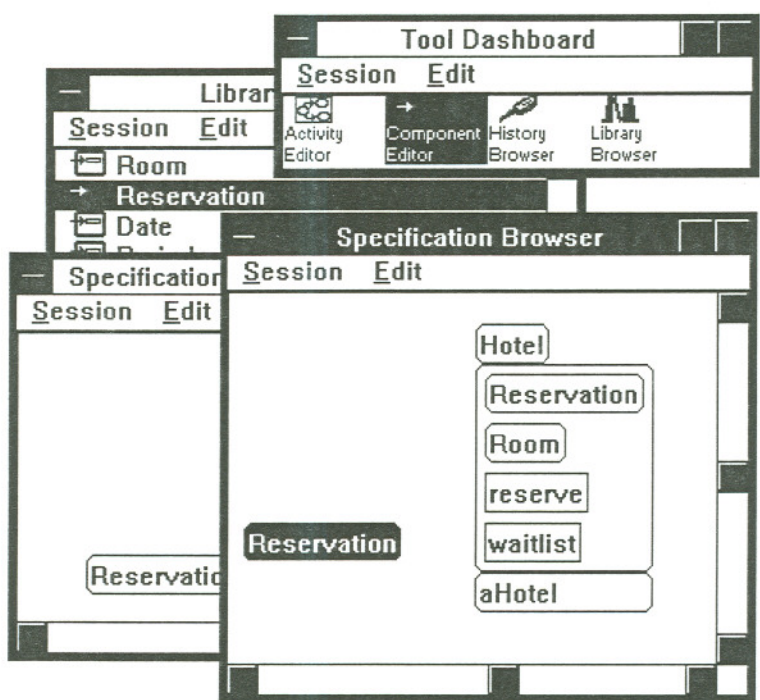

Figure 6.10: Copy Reservation frame

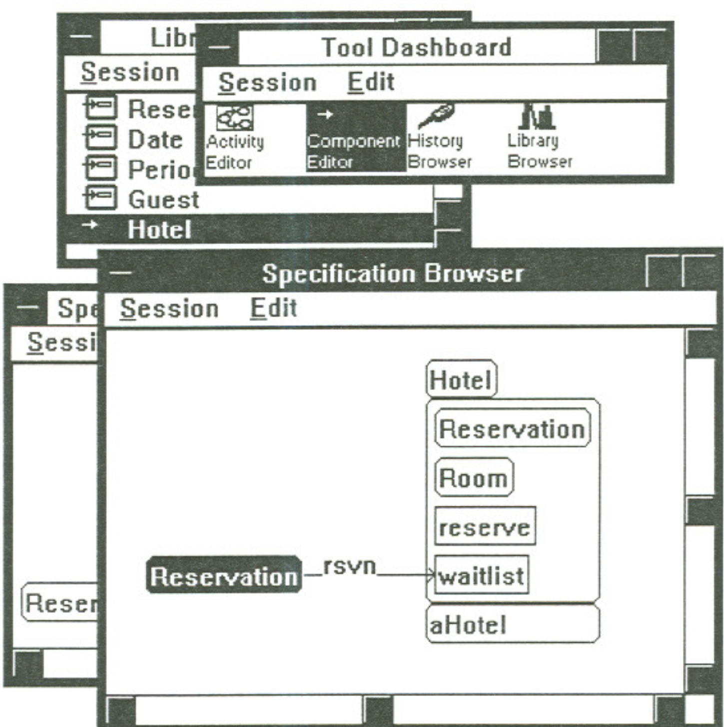

Figure 6.11: Create waitlist slot

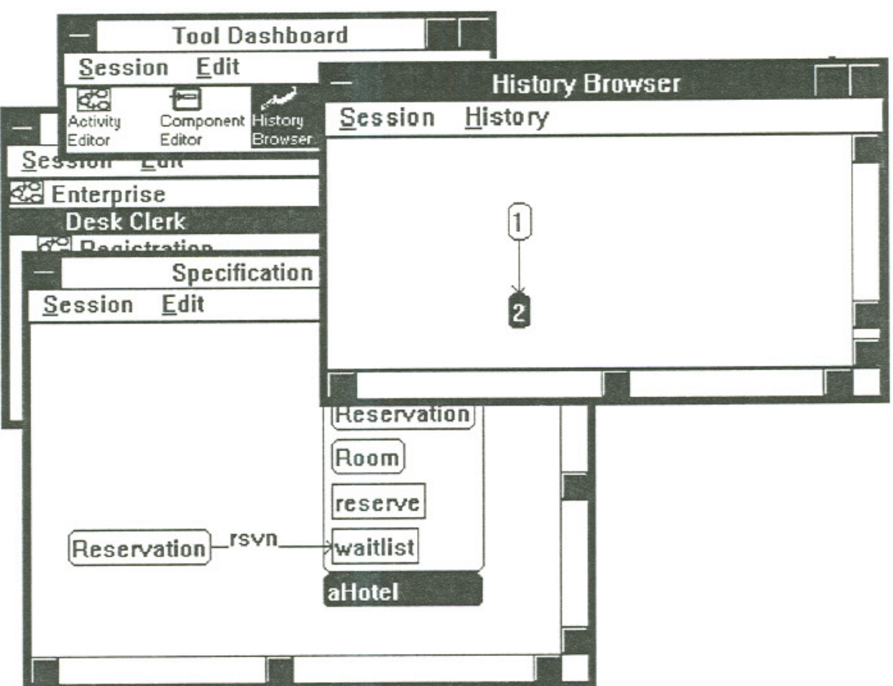

Figure 6.12: Visibility of promoted change

6.2 Scheme representation

Figure 6.13, page 84, is a block architecture diagram of the major subsystems of the Meros

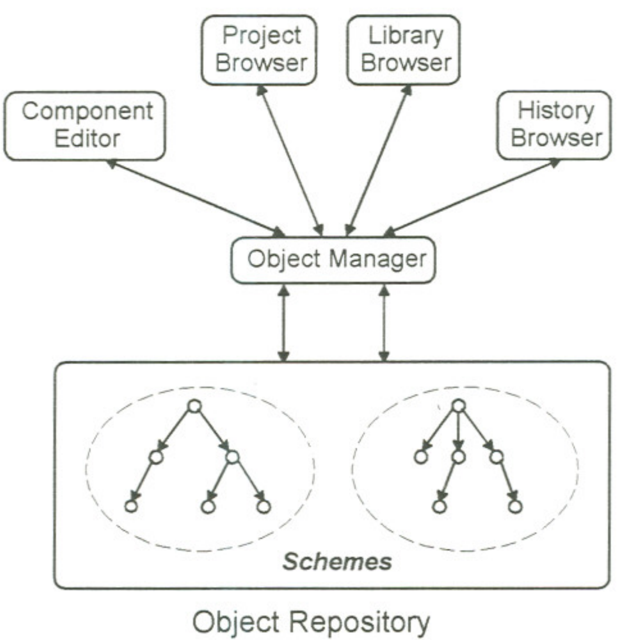

Figure 6.13: Meros tool architecture

environment. Interactive tools access a common object repository, or metadatabase. Objects and their associations are represented in a graph data model with *system schemes* and *component schemes*. The system schemes include the following:

Project Scheme	project/activity hierarchy
Workspace Scheme	workspace contents and parentage
Library Scheme	library component containment
Tool Scheme	definition of tools associated with views

The component scheme represents the essential, non-visual, shared aspects of a component interface.

Each component interface is represented in a separate component scheme. The component scheme, as described in §3.1, is a graph of frame nodes connected by typed edges.

The root of the interface scheme is a frame for the subject component. This frame has edges to its attribute and method frames. The method frame in turn has an edge to each of its slot frames. The component's attribute and slot frames comprise its acquaintances. Each acquaintance frame holds a configurable reference to another component. The reference identifies a generic component surrogate that may be resolved to a specific frame for that component (Figure 6.14). Typically, acquaintance references resolve to

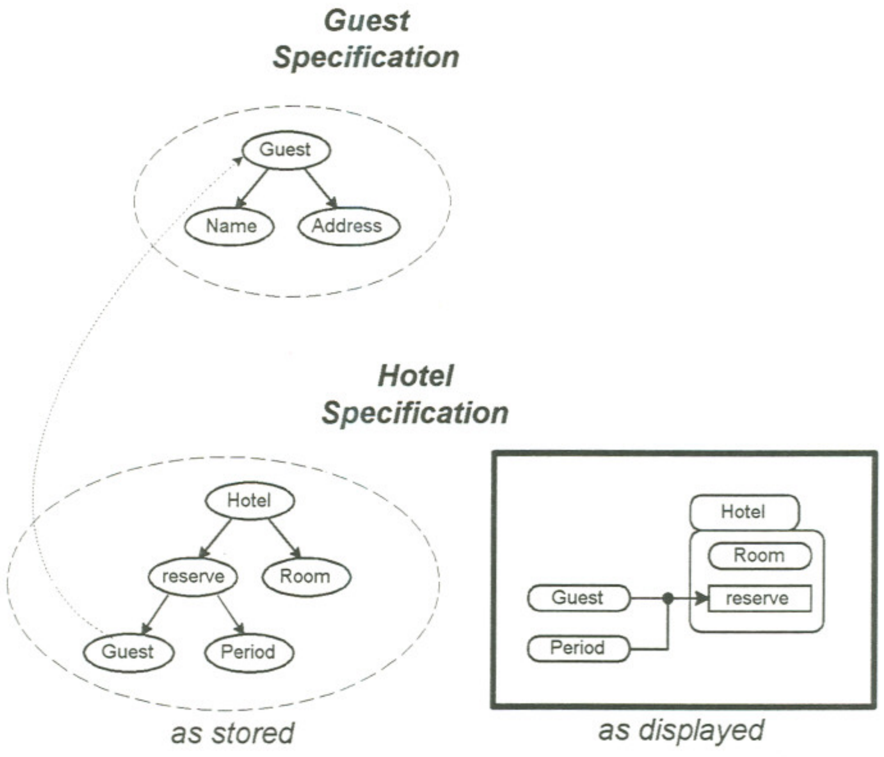

Figure 6.14: Scheme representation in Meros

the root of the interface scheme for the target component.

The Component Editor presents an interface scheme. The scheme is represented in the Object Repository as a simple tree structure rooted at the component frame. The Component Editor maintains a graph in its persistent specification artifact that reflects this base tree, augmented with graphics attributes governing font, color, selection and display. Each frame may be selectively shown or hidden. A display attribute determines whether the frame is displayed as simple or composite—a composite frame recursively

displays its visible contents.

An acquaintance frame may be opened, viewed and embedded within a specification artifact. The specification must then reflect the acquaintance's interface scheme as resolved by the component reference attached to the acquaintance frame. Furthermore, this information must be stored persistently in the specification artifact and updated dynamically as the acquaintance's referent interface changes. Dynamic update is accomplished by the use of a *configurable frame* for the acquaintance. Figure 6.15 shows how a

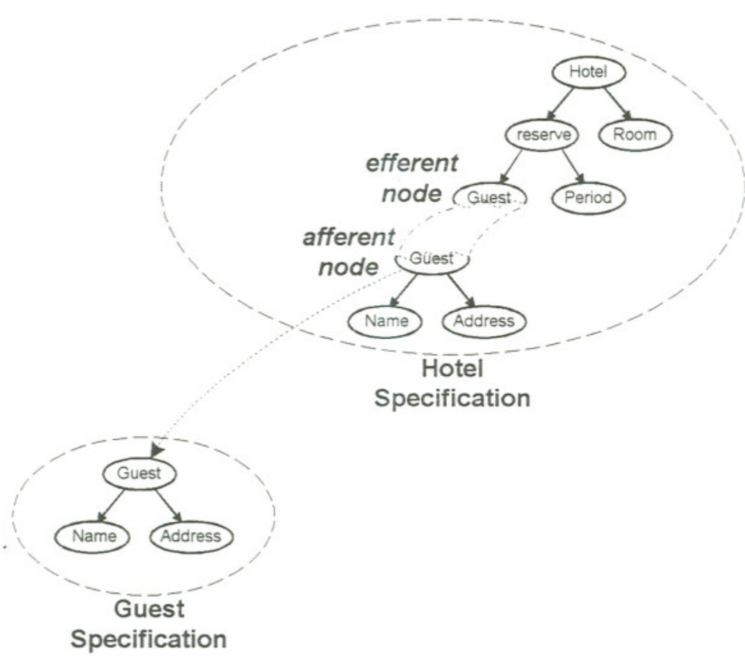

Figure 6.15: Configurable frame

`Guest` frame's scheme is dynamically determined in the context of a `Hotel` specification. The `Hotel` specification has a `Guest` frame that is the input to the `reserve` method. The `Guest`'s scheme must be dynamically determined based on a shared `Guest` specification that is imported into the `Hotel` specification.

The `Guest` frame is represented as a configurable reference in the `Hotel` specification graph. A configurable reference consists of two nodes, an *efferent node* and an *afferent node*. The efferent (afferent) node only holds inbound (outbound) arcs. The afferent node

is the root of a component interface scheme. A *configuration map* relates each afferent node in the specification to a configurable reference to the interface scheme containing the efferent node. Upon opening the frame within the specification, the Component Editor consults the configuration map, resolves the reference to the efferent node's interface scheme, grafts the referenced interface scheme into the base specification graph at the afferent node, and displays the result. The embedded interface scheme is cached within the new virtual graph and updated if the original interface scheme is modified.

The implementation of Meros is heavily reliant on a consistent scheme representation. Meta-schemes represent the information of the framework itself, such as library contents, the activity hierarchy and workspace parentage. The prototype is implemented in Smalltalk/V[2] for a single-user Windows[3] environment. The prototype extends the base Smalltalk environment with 52 classes containing approximately 820 methods (Table 6.2). The base Smalltalk environment collection classes and graphics classes were

Category	**Classes**
Specification Objects	13
Graphs and Schemes	14
Graphics	11
Applications	6
Auxiliary	8

Table 6.2: Meros classes by category

heavily used. The Smalltalk change notification mechanism assisted in maintaining consistency of schemes with their display as connected frames. Of the classes introduced in the prototype, the scheme classes saw the greatest reuse, since nearly every aspect of specification management was represented by some form of specification graph built from schemes. The integrated tool suite was inextricably based on underlying meta-schemes. The development and use of the prototype contributed to an understanding of component specification technology. The next chapter draws on this understanding to reconsider the role of a framework, discuss implementation lessons, and speculate on future work.

[2]Smalltalk/V is a registered trademark of Digitalk Corporation.
[3]Windows is a trademark of Microsoft Corporation.

Chapter 7

Conclusions

7.1 The Role of a Specification Framework Reconsidered

The claim was advanced in §2.2 that a component specification framework is motivated by the need to manage software development complexity. Development is facilitated, it was argued, by abstracting interface schemes from concrete artifacts, relating the resulting views to a component as the unit of reuse, elaborating and refining these components in specifications, and assembling appropriate components into applications. A component specification framework provides tools to enable and administer this process. The value of the framework is its capacity for *abstraction* of software into a form suitable for use.

This thesis describes an approach to managing component specifications that is intended to be relatively unobtrusive. It is nevertheless apparent that what is gained in dealing with software at a higher level of abstraction comes at a cost of devising the appropriate abstractions and keeping track of how they relate to underlying software artifacts. Given the complexity inherent in both the framework itself and its use, it is reasonable to reconsider whether this model of software development reduces system complexity or merely shifts the source of complexity.

The net effect of component-based development will depend largely on the effectiveness of the underlying component technology. This thesis refrains from the methodological judgement of what constitutes a reasonable component and how it is constructed

and deployed. In its most rudimentary form, a component can be identified with a corresponding programming language construct, such as a class in C++ [Str76] or a module in Modula-2 [Wir82]. The framework then offers a way to specify and keep track of these language constructs and their translation into execution units.

However, as alluded to in Chapter 4, perhaps the greatest potential for component specification is workflow automation, providing the scaffolding for the flexible construction of task-related applications and the deployment of autonomous, goal-directed agents. Conventional object-oriented programming languages are not well-suited for this purpose, since they lack dynamic expressivity and reconfigurablity without detailed knowledge of the language and component representations.

A component framework reduces this hurdle, but is constrained by the underlying component implementation technology. If incremental extensibility requires static compilation or grafting onto a self-contained programming environment, the ability to gracefully compose task-specific applications is limited. The framework's value increases as the ease of dynamic component query, configuration and extensibility increases. The importance of this enabling technology has recently spawned significant commercial efforts, including OLE2 [Cor94], CORBA [OMG93] and SOM/DSOM [IBM93], that promise the capability required of a component framework.

The framework serves as a vehicle for transition from language-based to component-based construction, and from application-based to task-based use. The organizational impact is to transform the role of the application developer and create two new classes of software developers, those of component builder and framework integrator. Application development then relies on components provided by component builders and introduced into a local framework by a framework integrator. The components are assembled into vertical applications suited to a particular task. Components are refined and configured to suit the task at hand, independent of the component construction language or environment.

The significant gains in managing complexity will come from the ability to compose novel applications from an evolving set of components in an environment that filters out unnecessary details, partitions components in useful ways and sets a context for

change isolation and control. This capability will not have the effect of reducing overall system complexity so much as shifting the focus of development—and, with it, the complexity—from self-contained applications to component construction, assembly and interoperability. The role of the component specification framework is to help manage the complexity resulting from this transition to component-based development.

7.2 Framework Implementation Lessons

Development of the Meros tool yielded useful lessons regarding the nature of components and how specifications can be effectively managed. This effort was begun before commercial component technology of the sort described above was announced or available. Consequently, much of the intuition about what form components should take arose out of the attempt to build the framework for managing them. Existing programming language environments proved an inadequate basis for realizing the full potential of a framework. The framework could at best capture class developer's intent and execute specifications to validate requirements against use scenarios. Building workflow applications requires components that are well-defined, semi-autonomous execution units. The necessary encapsulation technology is beyond the scope of a specification framework; fortunately, the commercial potential of this technology has become apparent and products are rapidly becoming available.

The primary lesson in managing specifications is the importance of a consistent scheme representation. A framework is integrative technology, keeping track of how independently developed objects interrelate. As such, the conceptual model is dominated by connections between objects rather than semantics of the objects themselves as independent entities. Consequently, the implementation must rely heavily on graph-based constructs.

Originally, a variety of specialized graph-like structures were developed for the internal implementation of framework objects, including the activity hierarchy, component library, views within components, specifications for views, interface schemes within specifications, methods in interfaces, slots within methods, bindings to slots, *et al.* Furthermore, common information necessitated data sharing in some form. The demands

of maintaining data consistency overwhelmed any approach relying on data replication and caching.

It quickly became apparent that a common form of graph implementation construct was needed that could be used within the framework's class implementation and provided for controlled structure sharing. This evolved into the representation of schemes as a directed graph structure, extendible through the graph join mechanism described in §6.2, to capture much of the framework information. Framework administrative information was represented in the system schemes for component, activity and workspace hierarchies.

Support for versioning and change control led to a series of design decisions (for example, whether components and views are versioned independently) that resulted in the proposed change model. The key role of configurations and the representation of a component as itself a configuration of versions of views permitted a change model that mediated reference resolution without imposing extraneous requirements on otherwise independent concepts, such as aggregation and frame relationships. Actions occured within the context of an active workspace that controlled scheme evolution. Setting the active workspace and relating specification frames by indirect component reference rather than direct object pointers was then the only instrumentation required to enable the change model.

7.3 Future Work

The most pressing task in furthering the understanding of component specification is to provide a well-founded conceptual basis. Although beyond the scope of this thesis, many of the concepts were developed with such a foundation in mind. The most attractive possibility for such a foundation lies in category theory [Mac71, BW90, AL91], which provides a mathematical workspace for relating algebras. Categories are graphs with identity and composition; "categorical thinking" involves abstracting the underlying structure of algebras and understanding their essential characteristics by how they map into other structures rather than how they operate internally. For example, a set object is understood by how it relates to other sets, particularly special sets like the empty

set and singletons, rather than how it operates on the members of the set. This capacity for understanding abstractions makes categories attractive for developing a sound conceptual foundation for component specification semantics.

The other area of foundational work needed is formalization of component interaction dynamics. The activation model elaborated in §4.2 can be translated into an appropriate process logic along the lines of Girard's linear logic [Gir87]. This logic captures the essential characteristic of resources as consumable objects. In conventional logic, a proposition can be used repeatedly in a proof. Linear logic, by contrast, can constrain a proposition to a single use in a proof. This constraint is a good match for a logical treatment of agents and actions. Until specifications are understood as formulae in a well-defined logic, agent activation is at best an *ad hoc* device.

Besides providing a conceptual foundation for component frameworks, there are several areas that would benefit from further effort. One such area is framework distribution and cooperative development. Meros was developed with a distributed, shared implementation in mind, with provision for multiple users, transactions and distributed resources. For example, a `User` object is registered with a workspace manager, provided local resources and a change context of nested `Transaction` objects that are committed with the save operation. However, the implementation currently merely provides hooks for a persistent implementation. Similarly, the current implementation provides only the raw material for the dynamic aspects of the framework, in the form of frame bindings and connections. The implementation needs to be extended to support agent activation and coordination.

There is also potential for understanding how to refine and relate views. Is a specification framework suitable for domain analysis? Can views be progressively refined and transformed to carry through from initial requirements to final deployment? Are views parameterizable? There are opportunities for reverse engineering tools that can infer a specification from an implementation artifact. Reverse engineering might take a form as simple as translating a class definition into an interface scheme, or as sophisticated as infering constraints and action rules from call paths and execution traces.

Similarly, it would be useful to investigate the expressive power of forward engineering tools to generate workable artifacts from the specification. It is not clear how complete

and faithful a specification is in mapping to a programming language to express, for example, a critical timing constraint or to support frame activation. A more important task is relating the component framework to emerging component technologies such as OLE2 and CORBA. A robust framework should accomodate these standards. Conversely, research in enabling component technology would benefit from an appreciation of component specification framework requirements.

7.4 Summary

A component specification framework provides one piece of the support necessary for enabling an adaptive software development model. Such a model recognizes that our interpretation of the world is in a constant state of flux and requires continual, incremental adjustment. Schema change in such a model is frequent, wide-spread, incremental and non-intrusive. Component definition is flexible, extensible and customizable. Any available frame in a specification can be used as a prototype to build new specifications. Software is freely adapted to accomodate the task at hand, rather than adjusting the task to the available software. Information is represented at several levels of abstraction, with well-defined transformations from requirements to implementation. This thesis describes one approach that points in the direction of an adaptive development model by facilitating the restructuring of brittle, monolithic applications into extensible, configurable components.

Bibliography

[ACO85] A. Albano, L. Cardelli, and R. Orsini. A strongly typed interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–236, 1985.

[Agh85] G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1985.

[AL91] A. Asperti and G. Longo. *Categories, Types and Structures: An Introduction to Category Theory for the Working Computer Scientist.* MIT Press, 1991.

[Ber87] P.A. Bernstein. Database systems support for software engineering. In *Proc. Ninth Int'l. Conf. on Software Engineering*, pages 166–177, 1987.

[BKN+77] D. G. Bobrow, R. M. Kaplan, D. A. Norman, H. Thompson, and T. Winograd. GUS: a frame-driven dialog system. *Artificial Intelligence*, 8:155–173, 1977.

[BM88] D. Beech and B. Mahod. Generalized version control in an object-oriented database. In *Proc. 1988 IEEE Data Engineering Conference*, pages 14–22, February 1988.

[BO92] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[Boe88] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[Boo91] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings, 1991.

[Bor86] A. Borning. Classes versus prototypes in object-oriented languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, pages 36–40, 1986.

[BW90] M. Barr and C. Wells. *Category Theory for Computer Science.* Prentice Hall, 1990.

[Che76] P. Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[CK91] T. C. Chiueh and R. H. Katz. Trait: an attribute management system for VLSI design objects. In J. Clifford and R. King, editors, *Proc. 1991 ACM SIGMOD Int'l. Conf. on the Management of Data*, pages 228–237, June 1991.

[CKK90] T. C. Chiueh, R. H. Katz, and V. King. A history model for managing VLSI design process. In *Proc. 1990 Int'l. Computer Conference on Computer Aided Design*, November 1990.

[Cod79] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 13(6):377–387, 1979.

[Cor94] Microsoft Corporation. *OLE2 Programmer's Reference.* Microsoft Press, 1994.

[Cox86] B. J. Cox. *Object Oriented Programming - an Evolutionary Approach.* Addison-Wesley, 1986.

[CY90] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice-Hall, 1990.

[Dav88] A. M. Davis et al. A strategy for comparing alternative software development life cycle models. *IEEE Trans. on Software Engineering*, 14(8):1453–1461, October 1988.

[Deu89] L. P. Deutsch. Design reuse and frameworks in the Smalltalk-80 system. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability, Vol. II*, pages 55–71. ACM Press, 1989.

[dK86] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, 1986.

[DMC92] C. Dony, J. Malefant, and P. Cointe. Prototype-based languages: from a new taxonomy to constructive proposals and their validation. In *ACM Proc. of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 201–217, October 1992. ACM SIGPLAN Notices 27(10).

[DMN68] O. J. Dahl, B. Myrhaag, and K. Nygaard. *Simula 67 Common Base Language*. Norwegian Computing Center, 1968.

[Doy80] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1980.

[Dum73] D. Dummet. *Frege: Philosophy of Language*, page 96. Duckworth, 1973.

[ECM90] ECMA. A reference model for frameworks of computer-assisted software engineering environments. Research Report TR/55, ECMA, 1990.

[ER83] D. W. Etherington and R. Reiter. On inheritance hierarchies with exceptions. In *Proceedings of the National Conference on Artificial Intelligence*, pages 104–108, Menlo Park, Ca., 1983.

[Feh93] J. Fehling. A concept of hierarchical Petri nets with building blocks. In G. Rozenberg, editor, *Advances in Petri Nets '93*, volume 674 of *Lecture Notes in Computer Science*, pages 148–168. Springer-Verlag, 1993.

[Fin71] H. L. Finch. *Wittgenstein - the Early Philosophy: an Exposition of the Tractatus*, page 35. Humanities Press, 1971.

[FJ93] L. M. G. Feijs and H. B. M. Jonkers. *Formal Specification and Design*. Cambridge University Press, 1993.

[Fra91] M. D. Frasier et al. Informal and formal requirements specification languages: Bridging the gap. *IEEE Trans. on Software Engineering*, 17(5):454–466, May 1991.

[Fre92] G. Frege. Über Sinn und Bedeutung. *Zeitschrift für Philosophie und philosophische Kritik*, 100:22–50, 1892. Translated as 'On Sense and Reference' in [GB52]. See also [Dum73].

[GH93] J. V. Guttag and J. J. Horning. *LARCH: Languages and Tools for Formal Specification.* Cambridge University Press, 1993.

[Gir87] J. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GJ82] C. Ghezzi and M. Jazayeri. *Programming Language Concepts.* John Wiley, 1982.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

[Gol84] A. Goldberg. *Smalltalk-80: the Language and Its Implementation.* Addison-Wesley, 1984.

[GT83] S. Gibbs and D. Tshiritzis. A data modeling approach for OIS. *ACM Transactions on Office Information Systems*, 1(4):299–314, October 1983.

[Har88] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[HG93] G. M. Hoydalsvik and S. Guttorm. On the purpose of object-oriented analysis. In A. Paepke, editor, *ACM Proc. of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 240–255, October 1993. ACM SIGPLAN Notices 28(10).

[HM81] M. Hammer and D. McCleod. Database descriptions with SDM: A semantic database model. *ACM Transactions on Database Systems*, 6(3):351–386, 1981.

[HMSN86] D. Harrison, P. Moore, R. Spickelmier, and R. Newton. Data management and graphics editing in the Berkeley design environment. In *Proc. Int'l. Computer Conf. on Computer Aided Design*, pages 20–24, 1986.

[HP87] D. J. Hatley and I. A. Pirbhai. *Strategies for Real-Time System Specification.* Dorset House, 1987.

[IBM93] IBM Corporation, Armonk, N.Y. *SOMobjects Developers Toolkit – User's Guide and Reference Manual*, 1993.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering - a Use Case Drive Approach.* Addison-Wesley, 1992.

[Kat87] R. H. Katz. *Information Management for Engineering Design.* Springer-Verlag, New York, 1987.

[KBG89] W. Kim, E. Bertino, and J. F. Garza. Composite objects revisited. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proc. 1989 ACM SIGMOD Int'l. Conf. on the Management of Data*, pages 337–347, June 1989.

[KC86] S. Khoshafian and G. P. Copeland. Object identity. In N. Meyrowitz, editor, *ACM Proc. of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 406–416, November 1986. ACM SIGPLAN Notices 21(11).

[KCB86] R. H. Katz, E. Chang, and R. Bhateja. Version modeling concepts for computer-aided design databases. In C. Zaniolo, editor, *Proc. 1986 ACM SIGMOD Int'l. Conf. on the Management of Data*, pages 379–385, June 1986.

[KR57] G. S. Kirk and J. E. Raven. *The Presocratic Philosophers*, page 377. Cambridge University Press, 1957.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* The MIT Press, 1986.

[LGP+90] D. B. Lenat, R. V. Guha, K. Pittman, D. Pratt, and M. Shepherd. Cyc: Toward programs with common sense. *Communications of the ACM*, 33(8):30–49, August 1990.

[Lie86] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In N. Meyrowitz, editor, *ACM Proc. of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 214–223, November 1986. ACM SIGPLAN Notices 21(11).

[LT86] W. R. LaLonde and D. A. Thomas. An exemplar-based Smalltalk. In N. Meyrowitz, editor, *ACM Proc. of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 30–37, November 1986. ACM SIGPLAN Notices 21(11).

[Mac71] S. MacLane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1971.

[Mar78] T. De Marco. *Structured Analysis and System Specification*. Yourdon Press, 1978.

[Mas87] V. Masurka. Requirements for a practical software engineering environment. In *Proc. 24th ACM/IEEE Design Automation Conference*, pages 67–73, June 1987.

[MH91] K. Miriyala and M. T. Harandi. Automatic derivation of formal software specifications from informal descriptions. *IEEE Trans. on Software Engineering*, 17(10):1126–1142, October 1991.

[Min75] M. Minsky. A framework for representing knowledge. In P. W. Winston, editor, *The Psychology of Computer Vision*. McGraw-Hill, 1975.

[OMG93] Object Management Group. OMG common request broker architecture and specification (CORBA). Technical report, Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, December 1993. OMG TC Document 93.12.43.

[PJ88] M. Page-Jones. *The Practical Guide to Structured Systems Design*. Prentice-Hall, 1988.

[PM88] J. Peckham and F. Maryansli. Semantic data models. *ACM Computing Surveys*, 20(3):153–189, 1988.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[RG92] K. S. Rubin and A. Goldberg. Object behavior analysis. *Communications of the ACM*, 35(9):48–62, September 1992.

[SB86] M. Stefik and D. C. Bobrow. Object-oriented programming: themes and variations. *AI Magazine*, pages 40–62, 1986.

[SFL81] J. M. Smith, S. Fox, and T. Lancers. Reference manual for ADAPLEX. Technical Report CCA-81-02, Computer Corporation of America, January 1981.

[Shi81] D. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981.

[SM88] S. Shlaer and S. J. Mellor. *Object-Oriented Systems Analysis.* Prentice-Hall, 1988.

[SM90] S. Shlaer and S. J. Mellor. *Object-Oriented Analysis: Modeling the World in Data.* Yourdon Press, 1990.

[SM91] S. Shlaer and S. J. Mellor. *Object Lifecycles: Modeling the World in States.* Yourdon Press, 1991.

[Sow84] J. Sowa, editor. *Conceptual Structures.* Addison-Wesley, 1984.

[Sow92] J. Sowa, editor. *Principles of Semantic Networks.* Addison-Wesley, 1992.

[Ste87] V. Stenning. On the role of an environment. In *Proc. Ninth Int'l. Conf. on Software Engineering*, pages 30–35, March 1987.

[Str76] B. Stroustrop. *The C++ Programming Language.* Addison-Wesley, 1976.

[TN92] I. Thomas and B. Nejmeh. Definitions of tool integration for environments. *IEEE Software*, pages 29–35, March 1992.

[Tou86] D. S. Touretzky. *The Mathematics of Inheritance Systems.* Morgan Kaufmann Publishers, 1986.

[TYF86] T. J. Teory, D. Yang, and J. P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, 18(2):40–62, 1986.

[US87] D. Ungar and R. B. Smith. Self: the power of simplicity. In *ACM Proc. of the Conference on Object-Oriented Programming, Systems, Languages*

and Applications, pages 227–241, December 1987. ACM SIGPLAN Notices 22(12).

[Was90] A. J. Wasserman. Tool integration in software engineering environments. In *Software Engineering Environments: Proc. Int'l. Workshop on Environments*, pages 30–34, 1990.

[WBJ90] R. J. Wirfs-Brock and R. E. Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, September 1990.

[WBWW90] R. J. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

[Weg90] P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, 1990.

[Wir82] N. Wirth. *Programming in Modula-2*. Springer-Verlag, fourth edition, 1982.

[WM85] P. Ward and S. J. Mellor. *Essential Modeling Techniques*, volume 2 of *Structured Development Techniques for Real-Time Systems*. Prentice-Hall, 1985.

[Zad87] W. Zadrozny. A theory of default reasoning. In *Proceedings of the National Conference on Artificial Intelligence*, 1987.

Biographical Note

The author received a B.A. degree in Mathematics from Oakland University in 1975 and a B.S. degree in Forestry from the University of Washington in 1977. He gravitated to computer programming as an occupation less tedious than the available alternatives and eventually pursued a career in information technology. He acquired an M.B.A. from the University of Washington in 1984, with a concentration in Information Systems and a research project in dynamic hashing. He was employed as a Database Analyst and Supervisor at U.S. Bancorp from 1984 to 1988. He joined Mentor Graphics Corporation as a Software Engineer in 1988, where he developed object management software for CASE and EDA applications. He enjoys hiking, bicycling and skiing with his wife, Melinda.