Integrating Informal and Formal Approaches to Object-Oriented Analysis and Design

By

Yile Enoch Wang

A Dissertation

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

Department of Computer Science

May 10, 1998

Advisor: Professor Betty H. C. Cheng

INTEGR

C

It is clearly

ingly, the need

where its corre

of software, for

to rigorously

notations are

cluding verifi

construct a f

document ca

syntactic de

the developer

specifications

as the Object

Abstract

Integrating Informal and Formal Approaches to Object-Oriented Analysis and Design

By

Yile Enoch Wang

It is clearly evident that the impact of software is significantly increasing. Accordingly, the need to have high assurance in software's correctness increases for systems where its correct operation is imperative. As a means to facilitate the development of software, formal software specifications are gaining increasing attention as a means to rigorously document requirements and design information since the well-defined notations are amenable to automated processing for numerous analysis tasks, including verification of the correctness of resulting systems. However, attempting to construct a formal specification directly from an informal, high-level requirements document can be challenging. Formal descriptions potentially involve considerable syntactic details and may require careful planning and organization on the part of the developer in order to develop modular (easily-decomposed and amenable to reuse) specifications. In contrast, object-oriented analysis and development techniques, such as the *Object Modeling Technique* (OMT), comprising diagramming techniques that

make use of in

ed today. How

of well-defined

process, particul

research is to in

ing a formal de

stepwise formal

development pro

specifications th

between levels o

make use of intuitive and easy to understand graphical notations, are extensively used today. However, the informal nature of the diagramming notations and the lack of well-defined semantics pose the potential to introduce errors in the development process, particularly as the systems become more complicated. The objective of this research is to introduce formal semantics to the graphical notations of OMT, including a formal definition of their integration, and to propose a process to conduct a stepwise formal, rigorous refinement of the diagrams during the design phase. The development process should enable a semi-automated generation of executable formal specifications that can be used to simulate the behavior and check the consistency between levels of specification refinements.

To my dea

to my

to my gra

To my dear wife, whose companionship is a blessing from God;

to my parents, who raised me to be a faithful Christian;

to my grandparents, who shed their unconditional love on me.

I would sincerely l
encouragement an
wish to thank Dr.
C. Pijanowski for
committee.

I would also l
Chen, Gretel V. C
in the Software E
kindness in the p

Finally, I wou
and most of all
looking forward t

ACKNOWLEDGMENTS

I would sincerely like to thank my advisor, Dr. Betty H.C. Cheng, without whose encouragement and wisdom, this dissertation would not have been possible. I also wish to thank Dr. Philip K. McKinley, Dr. Anthony S. Wojcik, and Dr. Bryan C. Pijanowski for their constructive suggestions and commitment for being on my committee.

I would also like to thank Gerald C. Gannod, William E. McUmber, Yonghao Chen, Gretel V. Coombs, Laura A. Campbell, Andrew S. Chen, and other members in the Software Engineering Research Group for all their friendship, support, and kindness in the past five years.

Finally, I would like to thank my wife, Tong, for all her support, understanding, and most of all confidence in me. I appreciate the sacrifices she has made and am looking forward to more time together with her.

LIST OF TABLE

LIST OF FIGUR

Table of Contents

H Complete LOT

5.1 The models u

List of Tables

List of Figures

8.10 Using *acc*
8.11 The refined ob
8.12 The ACT ON
8.13 The object fu
8.14 The object dy
8.15 The specific a
8.16 The specific a
8.17 The object fu
8.18 The object d
8.19 The specific a
8.20 An example
8.21 The service r
8.22 The service r
8.23 The refined
8.24 The LSL spe
8.25 The proof o
8.26 An example
8.27 An example
8.28 An example
8.29 The refined
8.30 The revised
8.31 The refined
8.32 The refined
dynamic
8.33 The compo
8.34 Using acce
Disk M
8.35 Using acce
Disk M

9.1 A high-lev
9.2 The desig
9.3 The forma
9.4 The forma
9.5 The trans
check i
9.6 The trans
for sim
9.7 The form
condit
9.8 A test pro
9.9 The trans
ENFO
9.10 The trans
ENFO

Chapter

Introduc

It is clearly evide

cordingly, the ne

systems where its

ing increasing att

information since

for numerous ana

systems. Howeve

informal, high-le

potentially involv

organization on

Chapter 1

Introduction

It is clearly evident that the impact of software is significantly increasing [1]. Accordingly, the need to have high assurance in software's correctness increases for systems where its correct operation is imperative. Formal specifications are gaining increasing attention as a means to rigorously document requirements and design information since the well-defined notations are amenable to automated processing for numerous analysis tasks [2], including verification of the correctness of resulting systems. However, attempting to construct a formal specification directly from an informal, high-level requirements document can be challenging. Formal descriptions potentially involve considerable syntactic details and may require careful planning and organization on the part of the specifier in order to develop modular specifications.

1.1 Proble

A complementary a
ing notations. For
used as a popular
taught as an objec
largely due to its si
tural, behavioral, a
and functional mod
similar to entity-rel
of a state diagram.
order to effectively
defined method to i
of three separate, in
that may help to c

One approach t
specifications and
construct a well-d
an existing specific
notation such as
requirements, desi
formalization rul
design tasks throu

1.1 Problem Description and Motivation

A complementary approach to describing requirements is the use of graphical modeling notations. For example, the Object Modeling Technique (OMT) [3] is extensively used as a popular object-oriented modeling technique in industry and is commonly taught as an object-oriented methodology in academic settings. Its popularity is largely due to its simple notation and the notational support for describing structural, behavioral, and functional aspects of a system through the object, dynamic, and functional models, respectively. The object model is represented in a notation similar to entity-relation (ER) diagrams [4]. The dynamic model is described in terms of a state diagram. And the functional model is captured by a data flow diagram. In order to effectively use OMT, particularly for design purposes, there must be a well-defined method to integrate the three models. Otherwise, OMT is only a combination of three separate, independent models that provide little more than intuitive diagrams that may help to clarify some ideas within the corresponding, separate models.

One approach that takes advantage of the automated processing afforded by formal specifications and the ease of use provided by graphical modeling techniques is to construct a well-defined syntax and semantics for the graphical models in terms of an existing specification language. A significant advantage to formalizing a modeling notation such as OMT is that it has a uniform notational support for modeling requirements, design, and detailed design information. Therefore, the corresponding formalization rules for diagrammatic models can be used to perform analysis and design tasks throughout the development process, rather than be limited to a specific

phase (e.g., requir

modeling notations

However, some

gebraic specificatio

fashion. Thus ful

Furthermore, the

resulting models, b

tion of the formal

models, from whic

hoc approach. Th

and refinement of

for graphical not

Thesis Stateme

velop formal syn

definition for the

for refining analy

fine a new object-

graphical models t

refinement.

phase (e.g., requirements), as is the case with numerous specification languages and/or modeling notations.

However, some features of the formal specifications, such as the equations in algebraic specifications, cannot be represented by graphical notations in an intuitive fashion. Thus fully automated generation of formal specifications is not realistic. Furthermore, the formalization rules can only derive formal specifications from the resulting models, but cannot help to conduct the development process. The application of the formal specifications may be limited by the fact that their corresponding models, from which the specifications are generated, are derived in an informal, *ad hoc* approach. Therefore, a process that supports a formal and rigorous development and refinement of analysis and design information with the use of formal semantics for graphical notations is necessary.

Thesis Statement: *The objective of the proposed research is threefold: (1) develop formal syntax and semantics for all three OMT models; (2) develop a formal definition for the integration of all three models; and (3) develop a refinement process for refining analysis information into design information. These three tasks will define a new object-oriented design paradigm that supports rigorous analysis of OMT's graphical models through specification execution, consistency verification, and stepwise refinement.*

1.2 Summa

This section gives h

Formalization of

dynamic models in t

guage of Temporal

an individual objec

the state diagrams

The state diagram

synchronized LOT

specification of the

executable specifi

Formalization o

into object-orient

functional model

tional Model (SR

object. An SRF

the services prov

lines for deriving

given. Based upo

describe the req

of services enabl

1.2 Summary of Research Contributions

This section gives highlights of the four major contributions of this research.

Formalization of Dynamic Model. New rules are proposed to formalize the dynamic models in terms of process algebra specifications expressed in LOTOS (Language of Temporal Ordering Specification) specifications. In OMT, the behavior of an individual object is modeled as a state diagram. The proposed rules formalize the state diagrams for individual objects in terms of LOTOS behavior specifications. The state diagrams of concurrent, communicating objects are formalized in terms of synchronized LOTOS behavior specifications. The formalization enables the precise specification of the behavior of objects and the simulation of system behavior through executable specifications.

Formalization of Functional Model. In order to integrate the functional models into object-oriented technology, the functional model of OMT has been modified. Two functional models, *Object Functional Model* (OFM) and *Service Refinement Functional Model* (SRFM), are introduced. An OFM captures the services provided by an object. An SRFM describes how a service of an object is implemented in terms of the services provided by the object at a lower level of abstraction. In addition, guidelines for deriving algebraic specifications from the object and functional models are given. Based upon algebraic specifications, pre- and postconditions are introduced to describe the requirements and constraints for services. The pre- and postconditions of services enable symbolic execution of the high-level design model thus providing a

means to perform s

software developme

Integration of th

three complementa

are derived in the

a dynamic model

the models with r

integration is achi

malization rules.

derived from the

grated and repres

is achieved by co

according to the

of an object in t

The integrati

complementary a

formal specifica

a design proces

successive refine

Process for M

on the investiga

means to perform simulation, verification, and validation during the early phases of software development.

Integration of the Three Complementary Models. The integration of the three complementary models is three-fold. First, the functional and dynamic models are derived in the context of object models. By associating a functional model and a dynamic model to every individual object, the conflicts can be resolved between the models with respect to their respective development philosophies. Second, the integration is achieved through the underlying formal semantics imposed by the formalization rules. By sharing common language constructs among the specifications derived from the object, functional, and dynamic models, the three models are integrated and represented in terms of a single formal specification. Third, the integration is achieved by composing: (1) the dynamic models of concurrent objects hierarchically according to the system structure specified in the object model; and (2) the SRFMs of an object in terms of the services provided by the aggregate objects.

The integration and formalization of the three models that describe a system from complementary aspects enable designers to perform analysis tasks by using the derived formal specifications. In addition, based upon the formalization and integration, a design process that incorporates formal specifications in a transitional, parallel successive refinement approach [5] is also possible.

Process for Model Construction, Specification, and Refinement. Based on the investigations into the formalization and integration of the models of OMT, a

design process is pr

in parallel with the

of the rigorous ma

and designers can l

ambiguities. In ad

understand the de

even with the cust

model refinement

earlier stages of so

1.3 Defini

Since our researc

cess, it is importa

sistency, and com

on various perspe

definitions apply.

- **Correctne**
  can still be
- **Consisten**
  have syntac
  rent, comm
  design spec
  refinement
- **Completer**
  ties as well

design process is proposed to facilitate the development of formal design specifications in parallel with the development of OMT's semi-formal, graphical models. Because of the rigorous mathematical foundation of formal specifications, both customers and designers can have a more precise means to describe the design thus avoiding ambiguities. In addition, symbolic simulation of the design can help designers better understand the design as well as facilitate the communication among designers and even with the customers. And finally, specification analysis can be performed during model refinement of the design process to detect and eliminate design flaws during earlier stages of software development.

1.3 Definitions

Since our research focuses on the application of formal methods and the design process, it is important for us to clarify what we mean by the terms correctness, consistency, and completeness. These terms have been given different meanings based on various perspectives [6, 7], but for the remainder of the dissertation, the following definitions apply.

- **Correctness:** after a refinement of a service, the postcondition of the services can still be satisfied.
- **Consistency:** the formal specifications derived from the models (1) do not have syntactical and semantic errors, (2) do not have deadlocks among concurrent, communicating objects, and (3) the test cases that are satisfied by the design specification of a higher level abstraction can still be satisfied after one refinement iteration.
- **Completeness:** the formal specifications describe all the required functionalities as well as the behavior of the system under all circumstances.

1.4 Organi

The remainder of th
of the Object Mod
languages that will
the object model o
ter 4 revisits the
ization rule. Chap
in terms of the LO
model into object-
functional models.
els are integrated
the correspondin
and to perform s
ducts a stepwise
specifications. C
process to a larg
discusses future i
ules and formal

1.4 Organization of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 gives an overview of the Object Modeling Technique (OMT), formal methods, the formal specification languages that will be used, and preliminary investigations into the formalization of the object model of OMT. Chapter 3 discusses work related to this research. Chapter 4 revisits the object model formalization and introduces an additional formalization rule. Chapter 5 describes the formalization of the dynamic model of OMT in terms of the LOTOS specification language. Chapter 6 integrates the functional model into object-oriented technology and discusses the formalization rules for the functional models. Chapter 7 discusses how the object, functional, and dynamic models are integrated in terms of their underlying formal semantics. It also discusses how the corresponding specifications can be used to simulate the behavior of the system and to perform specification analysis. Chapter 8 proposes design process that conducts a stepwise refinement of design by using the diagrammatic models and formal specifications. Chapter 9 describes a case study that applies the proposed design process to a large-scale project, ENFORMS. Chapter 10 gives the conclusions and discusses future investigations. Appendices A through H give complete formalization rules and formal specifications generated from OMT models.

Chapter

Backgro

This chapter prese

is overviewed, incl

tively. Next, the

Finally, the forma

2.1 Objec

The *Object Mod*

et al [3], to facilita

three diagrammin

2.1.1 Comp

The essence of th

domain during th

Chapter 2

Background

This chapter presents background material relevant to the dissertation research. OMT is overviewed, including how it is used during the analysis and design phases, respectively. Next, the specification languages Larch and LOTOS are briefly introduced. Finally, the formalization of the object model is overviewed.

2.1 Object Modeling Technique Overview

The *Object Modeling Technique* (OMT) is a methodology developed by Rumbaugh, *et al* [3], to facilitate *object-oriented analysis* and *design (OOA and OOD)*. It includes three diagramming techniques to describe different aspects of a system.

2.1.1 Complementary diagramming techniques

The essence of the *Object Modeling Technique* is to build a model of an application domain during the analysis of a system that can be augmented with implementation

details during the

object model, dyn

of a system. Each

implementation de

the object diagram.

Object Model.

world that is imp

three diagrams. It

their relationships

it is straightforwa

The object diagra

models are based.

that can be valua

since the diagram

sent *classes*: lines

of association lin

relationships: tria

Dynamic Mode

describe the beha

Each state diagra

class of objects.

details during the design phase. The modeling consists of three orthogonal models, *object model*, *dynamic model*, and *functional model*, each depicting different features of a system. Each model is applicable during all stages of development and acquires implementation detail as the development progresses. The models are represented as the object diagram, state diagram, and data flow diagram, respectively.

Object Model. An object diagram is used to capture information about the real world that is important to an application. Thus it is the most important of the three diagrams. It describes the static objects in a system by showing their identity, their relationships to other objects, their attributes, and their operations. Therefore, it is straightforward to derive *abstract data types* (ADT) from the object diagrams. The object diagram forms a basic framework upon which the dynamic and functional models are based. The diagram provides an intuitive visual representation of a system that can be valuable in the communication between the customers and the developers since the diagrams serve to document the structure of a system. In OMT, boxes represent *classes*; lines between boxes represent associations; empty, solid circles at the end of association lines represent different multiplicities; diamonds represent *aggregation* relationships; triangles represent *inheritance* relationships.

Dynamic Model. A state diagram graphically represents the dynamic models that describe the behavioral aspects of a system concerned with events, time, and changes. Each state diagram depicts the state and event sequences allowed in a system for one class of objects. The notation used for dynamic models is a variation of Harel's

Statechart notation

transitions. In OMT

events that trigger

diagrams. Function

diagram: operation

diagram.

Functional Mod

grams, is the third

Data flow diagram

and data flows. r

the operations in

constraints for va

2.1.2 Requ

In OMT, the ob

derstandable, an

three aspects of

dynamic model

The object m

other, and it sho

the static structu

details, more st

Statechart notation [8], where ovals represent states; arcs with arrows represent state transitions. In OMT, rounded rectangles represent states; labels on arrows represent events that trigger state transitions. State diagrams also reference the other OMT diagrams. Functions in a data flow diagram correspond to the actions from the state diagram; operations on objects in the object diagram are modeled as events in a state diagram.

Functional Model. The functional model, depicted in the form of data flow diagrams, is the third dimension of the three orthogonal modeling techniques of OMT. Data flow diagrams (DFD) consist of nodes and arcs, which correspond to processes and data flows, respectively. The data flow diagram also specifies the meaning of the operations in the object model and the actions in the dynamic model, as well as constraints for values within an object model.

2.1.2 Requirements Analysis

In OMT, the objective of requirements analysis is to *devise a precise, concise, understandable, and correct model of the real-world* [3]. The analysis models describe three aspects of objects: static structure (object model), sequencing of interactions (dynamic model), and data transformation (functional model).

The object model describes real-world object classes and their relationships to each other, and it should be the first model to be derived from a problem statement because the static structure of a system is usually better defined, less dependent on application details, more stable as the solution evolves, and easier for humans to understand.

During the analysis

and operations, ar

is prepared: inherit

object model at the

and refinements.

The dynamic m

objects. It is the s

or more scenarios

external display fo

include all signals.

based upon the s

diagram, which is

event flow diagra

diagram for each

events the object

and event flow di

The functiona

without consider

be derived. In

output values of

is developed acc

is recursively ex

function. A des

During the analysis phase, modeling entities, such as classes, associations, attributes, and operations, are identified; a data dictionary that contains all modeling entities is prepared; inheritance relationships between object classes are identified. The final object model at the end of the analysis phase is obtained through repetitive iterations and refinements.

The dynamic model shows the time-dependent behavior of the system and the objects. It is the second model to be derived during the analysis stage. Initially, one or more scenarios (a scenario is a sequence of events) that show the major iterations, external display formats, and information exchanges are prepared. The events, which include all signals, inputs, decisions, interrupts, transitions, and actions, are identified based upon the scenarios. And each scenario is described in terms of an event trace diagram, which is an ordered list of events between different objects. In addition, event flow diagrams that depict events between classes are derived. Finally, a state diagram for each object class with nontrivial dynamic behavior that describes the events the object receives and sends is obtained based upon the event trace diagrams and event flow diagrams.

The functional model that shows data dependency and how values are computed without considering sequencing, decisions, or object structure is the last model to be derived. In OMT, similar to the process used in structured analysis, input and output values of the system are first identified. A top level data flow (DFD) diagram is developed according to the input and output values. Then each nontrivial process is recursively expanded into a low-level DFD until every process corresponds to a function. A description for each function, which can be in natural language, math-

ematical equation

once the DFD has

2.1.3 Desig

OMT decomposes

System Design.

guidelines for the

this stage, object

is to provide, co

tasks, decisions

are made. The i

from the analys

Object Desig

going to be im

made. This st

for operations

2.2 Fo

A formal me

governing t

language p

ematical equations, pseudo-code, or some other appropriate form, should be written once the DFD has been refined to a sufficient level of detail.

2.1.3 Design

OMT decomposes the design phase into system design and object design stages [3].

System Design. The objective of the system design stage is to establish high-level guidelines for the object design and to make global decisions concerning the design. In this stage, objects are grouped into *subsystems* according to *services* that the system is to provide, concurrency is identified, *subsystems* are allocated to processors and tasks, decisions about data store management and software control implementation are made. The implicit presumption of all these activities is that the models obtained from the analysis phase are sufficiently detailed to contribute to this phase.

Object Design. During the object design, decisions about how the classes are going to be implemented according to the strategy chosen during system design are made. This stage includes determining the data structure of attributes, algorithms for operations, association design, and addition of internal objects, and so on.

2.2 Formal Methods

A *formal method* is characterized by a formal specification language and a set of rules governing the manipulation of expressions in that language. A formal specification language provides [9]:

- a *syntactic d*
- a *semantic d*
- a *satisfaction* (implement)

Formal specifica
of analysis results a
execution [11], exec
steps (refinements

There exists ma
can be partitioned
fications are comm
denotational.

2.2.1 Axiom

The *axiomatic* ap
gramming langua
the proof of prope
fied input/outpu
a mathematical
ions are rules of
of mathematical
to provide a form
uninterpreted spe

- a *syntactic domain*: the notation in which the specifications are written.
- a *semantic domain*: a universe of elements that may be specified, and
- a *satisfaction relation*: indicates which elements in the semantic domain satisfy (implement) which specifications in the syntactic domain.

Formal specifications can be checked by tools that help explore the consequences of analysis results and design decisions [2], detect logical inconsistencies [10], simulate execution [11], execute symbolically [12], and prove the correctness of implementations steps (refinements) [13].

There exists many types of formal specifications. Formal specification languages can be partitioned according to a high-level classification [14]. The formal specifications are commonly categorized into three classes: axiomatic, operational, and denotational.

2.2.1 Axiomatic specifications:

The *axiomatic* approach to specification implicitly defines the semantics of a programming language by a collection of axioms and rules of inference, which enable the proof of properties of programs, such as program correctness, in terms of specified input/output relations. The assertions about programs can be proven by either a mathematical or an operational definition and mathematical reasoning. The axioms are rules of inferences that can be regarded as theorems within the framework of mathematical semantics. The objective of the axiomatic formal specification is to provide a formal system that enables a proof to be constructed using only the uninterpreted specification text.

Axiomatic spec

and algebraic spec

Predicate specifi

A predicate specifi

that a given imple

required functiona

constructivity. Th

facilitates specifica

and other behavio

and space efficien

dure that, given a

its square root.

Precondit
Precondit
Postcondi

Fig

Algebraic spe

An algebraic sp

equations, con

Axiomatic specifications can be further classified into predicate specifications [15] and algebraic specifications [16].

Predicate specifications:

A predicate specification explicitly describes properties of the behavior of a system that a given implementation must satisfy. The specifications describe the system's required functionality. Predicate specifications are not bound by the constraint of constructivity. The properties can be stated separately and then combined, which facilitates specification modularity. The properties include input/output constraints and other behavior conditions, such as fault tolerance, safety, security, response time, and space efficiency. Figure 2.1 contains a predicate specification describing a procedure that, given an appropriate argument, x, computes an integer approximation to its square root.

Precondition S **Postcondition**
Precondition: $x \geq 0 \wedge \text{integer}(x)$;
Postcondition: $\forall i : 0 \leq i \leq x : abs(x - result \times result) \leq abs(x - i \times i)$

Figure 2.1: A predicate specification of integer square root

Algebraic specifications:

An algebraic specification is a mathematical description language, based largely on equations, commonly used to specify *abstract data types (ADT)*. An ADT is a well-

defined data struc

services. The prop

An algebraic sp

- *Sort(s)*: the
- *Operations*
  syntactically
- *Axioms* or *t*
  specificatio

In Figure 2.2,

given in the Lar

2.2.2 Oper

An operational

ties, instead of d

cation is similar

defined data structure described by the available services and properties of these services. The properties of the data type are specified in terms of *equations*.

An algebraic specification typically consists of

- *Sort(s)*: the names of the abstract data types being described.
- *Operation(s)*: the services applicable to instances of the abstract data type and syntactically describe how they have to be invoked (*signatures*).
- *Axioms* or *theorems*: formally describe the semantic properties of the algebraic specification.

In Figure 2.2, an algebraic specification describing the properties of the stack is given in the Larch specification language.

```
Stack: trait
    introduces
      new: → S
      push: E, S → S
      pop: S → S
    asserts ∀ s: S, e: E
      pop (new) == new
      pop (push(e, s)) == s
```

Figure 2.2: A generic algebraic specification of *stack*

2.2.2 Operational specifications:

An operational specification [17] gives one solution that satisfies the required properties, instead of describing the required behaviors. Commonly, the operational specification is similar in format to a program. This approach has an advantage in that the

operational specifications can be executed directly as a rapid prototype of the system being specified. Thus the specifiers and their clients can obtain feedback about the software system quickly. The disadvantages are that it is difficult to extract the essential properties that the system must fulfill and the specifications tend to be relatively longer than behavioral specifications. The operational specification in Figure 2.3 gives a simple implementation algorithm to compute the square root of an integer.

```
int sqrt (int x)
    requires x ≥ 0;
    effects
      i = 0;
      while i × i < x
        i = i + 1 end
      if abs(i × i - x) > abs((i - 1) × (i - 1) - x)
        then return i - 1
        else return i
```

Figure 2.3: An operational specification of integer square root

2.2.3 Denotational specifications:

The denotational specification [18] maps a specification directly to its meaning, called its denotation. The denotation is usually a mathematical value, such as a number or a function. No interpreters are used; a valuation function maps a specification directly to its meaning.

A denotational definition is more abstract than an operational definition, since it does not specify computation steps. Its high-level, modular structure makes it espe-

cially useful to lan

can be studied wit

the implementor o

must be represent

be implemented as

Therefore, den

tion and less abstra

it can be stated in

designers.

Figure 2.4 sho

denoted by an ex

maps an expressi

expression, e, E

a set of values. S

to Value. A pa

Therefore, a spe

where σ gives th

and $\sigma[y]$ = false

cially useful to language designers and users, since the individual parts of a language can be studied without having to examine the entire definition. On the other hand, the implementor of a language is left with more work. The numbers and functions must be represented as objects in a physical machine, and the valuation function must be implemented as the processor.

Therefore, denotational semantics is more abstract than an operational specification and less abstract than an axiomatic specification. Like an algebraic specification, it can be stated in modules, which makes it especially useful to system analysts and designers.

Figure 2.4 shows a denotational specification of boolean expressions. The value denoted by an expression depends on the state because it may contain variables. $\underline{\mathbf{E}}$ maps an expression onto a function from states to boolean values. For a particular expression, ε, $\underline{\mathbf{E}}[\![\varepsilon]\!]$: $\underline{\mathbf{S}} \rightarrow \underline{\mathbf{Bool}}$ is a function from $\underline{\mathbf{S}}$ to $\underline{\mathbf{Bool}}$, which corresponds to a set of values. States, $\underline{\mathbf{S}}$, is the set or data-type of functions from identifiers, $\underline{\mathbf{Ide}}$, to $\underline{\mathbf{Value}}$. A particular state, σ, is a particular function from variables to values. Therefore, a specific value is obtained by evaluating the expression $\underline{\mathbf{E}}[\![\varepsilon]\!]\sigma$: $\underline{\mathbf{Bool}}$, where σ gives the state in terms of identifiers and their values. Suppose $\sigma[\![x]\!] = \underline{\mathbf{true}}$ and $\sigma[\![y]\!] = \underline{\mathbf{false}}$ then the boolean expression x **and not** y can be evaluated as:

$$
\begin{aligned}
&\underline{\mathbf{E}}[\![x \ \underline{\mathbf{and}} \ \underline{\mathbf{not}} \ y]\!]\sigma \\
&= \underline{\mathbf{E}}[\![x]\!]\sigma \wedge \underline{\mathbf{E}}[\![\ \underline{\mathbf{not}} \ y]\!]\sigma \\
&= \underline{\mathbf{E}}[\![x]\!]\sigma \wedge \neg\underline{\mathbf{E}}[\![y]\!]\sigma \\
&= \sigma[\![x]\!] \wedge \neg\sigma[\![y]\!] \\
&= \underline{\mathbf{true}} \wedge \neg\underline{\mathbf{false}} \\
&= \underline{\mathbf{true}}
\end{aligned}
$$

Exp:
ε ::= **and** ε |
ε **or** ε |
not ε |
true |
false |
ξ

Ide:
ξ ::= syntax for identifiers

E: **Exp** $\rightarrow$ **S** $\rightarrow$ **Bool**
$\mathbf{E}[\![\varepsilon \textbf{ and } \varepsilon']\!]\sigma = \mathbf{E}[\![\varepsilon]\!]\sigma \wedge \mathbf{E}[\![\varepsilon']\!]\sigma$
$\mathbf{E}[\![\varepsilon \textbf{ or } \varepsilon']\!]\sigma = \mathbf{E}[\![\varepsilon]\!]\sigma \vee \mathbf{E}[\![\varepsilon']\!]\sigma$
$\mathbf{E}[\![\textbf{ not } \varepsilon]\!]\sigma = \neg\ \mathbf{E}[\![\varepsilon]\!]\sigma$
$\mathbf{E}[\![\textbf{ true }]\!]\sigma$ = true
$\mathbf{E}[\![\textbf{ false }]\!]\sigma$ = false
$\mathbf{E}[\![\varepsilon]\!]\sigma = \sigma[\![\varepsilon]\!]$

σ: **S** = **Ide** $\rightarrow$ **Value**

Figure 2.4: A denotational specification of boolean expression

2.3 Specification Languages Used in Project

This section overviews two specification languages that will be used in the formalization of OMT.

2.3.1 Larch

The Larch family of specification languages [19] uses a two-tiered approach to formal specifications in which one tier, the Larch Shared Language (LSL), is an algebraic specification language used to specify properties that are independent of a particular

programming language and paradigm. Algebraic specifications can be used to describe object-oriented software in a straightforward manner, using abstract data types as the basic unit in software specification. Accordingly, the basic unit of specification in LSL is the *trait*, which axiomatizes theories about functions and data types that are used in programs. A collection of general purpose traits that are designed for constructing application-specific traits is called a trait *handbook* [19]. The other tier, the Larch Interface Language (LIL), specifies program behavior and programming language interfaces. For example, LCL [20] is a LIL for the C language that specifies program behavior in terms of predicate logic and function and procedure interfaces in terms of C syntax. In Figure 2.5, an algebraic specification describing the properties of the *table* is given in the Larch specification language. There are three operators for the *table*: create a new table, add an integer, and test for membership. The properties stated in the *asserts* section indicate that there are no elements in a newly created table and an existing element can be found using a linear search.

```
Table: trait
    includes Integer
    introduces
        new: → Tab
        add: Tab, Ind, Val → Tab
        ∈: Ind, Tab → Bool
    asserts ∀ i i1: Ind, v: Val, t: Tab
        ¬ (i ∈ new);
        i ∈ add(t, i1, v) == i = i1 ∨ i ∈ t
```

Figure 2.5: A Larch algebraic specification of table

2.3.2 LO

LOTOS was
nication syst
specification
simple extern
consists of an
bra. Full LO
behaviors alo

The algeb
Shared Langu
specifications.
consisting of s
thus the entire

The genera
parameters. w
with "!") or a
follows: *g !3 ?*
natural numb

A *behavior*
are built from
operations: *act*

2.3.2 LOTOS

LOTOS was originally designed to specify the behavior of networking and communication systems [21]. Basic LOTOS, based upon *process algebras* (an operational specification) [22, 23, 24], contains the primitives that enable specifiers to model simple external, observable behaviors without data exchange; Full LOTOS [21, 25] consists of an algebraic specification language [26], ACT ONE, and a process algebra. Full LOTOS enables specifiers to specify both abstract data types and external behaviors along with process communication and value passing.

The algebraic specification language, ACT ONE, which is similar to the Larch Shared Language, is used to describe the abstract data types that are used in LOTOS specifications. A system, as a whole, can be specified as a single process, possibly consisting of several subprocesses. Each subprocess is considered to be a process, and thus the entire system may be viewed as a hierarchy of processes.

The general form of a Full LOTOS event consists of a *gate* and a list of interaction parameters, where each parameter can either be a value offered at the gate (labeled with "!") or a variable accepted at the gate (labeled with "?"). An example is as follows: *g !3 ?data: Nat;*. This event offers value *3* and accepts a value of sort Nat (natural number) for variable *data* at gate *g*.

A *behavior expression* models the activity of a process. Behavior expressions are built from actions and other behavior expressions by using a predefined set of operations: *action prefix*, *choice*, *parallel composition*, *enable*, and *disable*.

For e

this spe

s of typ

The cho

Once an

proc
b

a

endp

2.4

When us

models.

functiona

for compl

Bound

propriate

erations.

describes a

called ass

For example, Figure 2.6 shows a typical LOTOS process algebra specification. In this specification, process **state1** has two gates in the form of **[a,b]** and a parameter **s** of type **S**. Keyword **noexit** specifies that the process **state** does not terminate. The choice operator, [], captures the notion that two alternative choices are possible. Once an event occurs, then the choice for the path to take is determined.

```
process state1 [a,b](s: S): noexit:=
  b ?k:K; b !b(s,k);
      state1 [a,b](s)
  []
  a ?d:D;
      state1 [a,b](s)
endproc
```

Figure 2.6: A typical LOTOS process algebra specification

2.4 Object Model Formalization

When using OMT, the object model is central to the construction of the other two models. While this dissertation focused on the formalization of the dynamic and functional models, an overview of the formalization of the object model is included for completeness sake and to provide context for the dissertation.

Bourdeau and Cheng [27] identified a subset of the object model notation appropriate for describing requirements and added formal syntax and semantics to the notations. In object diagrams, rectangles enclose the names of *classes*, where a class describes a particular type of object in the system. Relationships between objects, called *associations* in OMT, are specified by connecting the classes involved in the

relationship by

models express

to represent a

a system. It c

In Bourdea

instance diagra

overview of th

form

In this figu

formal concept

arrow labeled

tween an algeb

literature [28].

as algebraic sp

result, the OM

in terms of an a

relationship by a line with the name of the association centered on the line. Object models expressed in the context of requirements analysis are referred to as A-schemata to represent *analysis object schemata.* An A-schema describes the static structure of a system. It consists of a set of classes and associations among those classes.

In Bourdeau and Cheng's formalization, the semantics of the A-schemata and instance diagram notations are described by an algebraic formalization. A graphical overview of this formalization process is given in Figure 2.7.

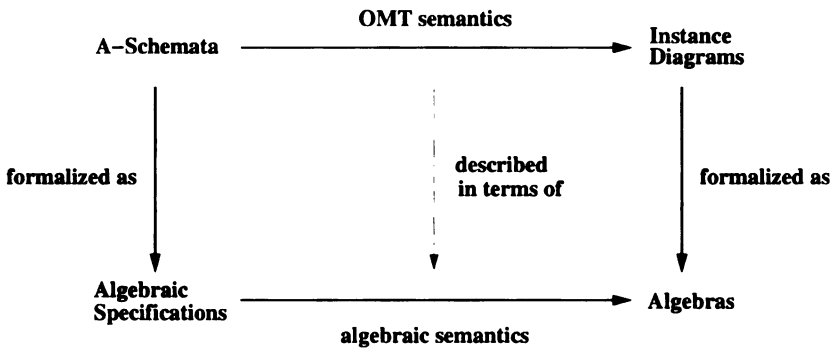

Figure 2.7: Basic approach to formalization.

In this figure, the arrow labeled "OMT semantics" represents the currently informal concept of consistency between an instance diagram and an A-schema. The arrow labeled "algebraic semantics" represents the formal concept of consistency between an algebra and an algebraic specification that has been well-developed in the literature [28]. In Bourdeau and Cheng's formalization, A-schemata are formalized as algebraic specifications, and instance diagrams are formalized as algebras. As a result, the OMT semantics, which were previously not well-defined, are now described in terms of an algebraic semantics.

First, the semantics of classes, objects, and object-states will be given. Next, the semantics of associations will be addressed, and finally, the combination of all of the formalizations will be presented in order to describe the semantics of A-schemata.

Bourdeau and Cheng used the *Larch Shared Language* (LSL) [19] to illustrate how these basic formalisms are incorporated into a structured, algebraic specification. In Figure 2.8, *SPECNAME* is the name of the specification module. The sorts that are to be considered as the parameters of the module are given in the parameter list following the name of the trait. *Includes* indicates other traits upon which the given trait is built. The *introduces* section itemizes *function signatures*, each of which gives the number and types of input arguments and result type of a function. *Asserts* defines the constraints for the specification. When using LSL, one assumes that a basic axiomatization of Boolean algebra is a part of every trait. This axiomatization includes the sort *BOOL*, the Boolean constants *true* and *false*, the connectives '$\wedge$' and '$\vee$', implication '$\Rightarrow$', and negation '$\neg$'.

SPECNAME (*parameters*) : **trait**
includes
list of pre-existing specification modules to be used
introduces
syntax declarations for functions are listed here
asserts
axioms are listed here

Figure 2.8: High-level structure of LSL specifications

2.4.1

Let S be

S. Forma

of a sort

character

state s o

for input

2.4.1 Semantics for Classes and Object States

Let $\mathcal{S}$ be an A-schema, and let $\mathcal{C} = \{C_1, \ldots, C_n\}$ be the set of class names given in $\mathcal{S}$. Formally, each class name $C_i \in \mathcal{C}$, where $1 \leq i \leq n$, is considered to be the name of a *sort* (type). For each class name C_i, a sort C_i-*STATES* is introduced, which characterizes the set of states that are possible for any C_i-object. For each object-state s of class C_i, s is specified with the signature (syntax and type specifications for input arguments and output value)

$$s: \quad \rightarrow C_i\text{-}STATES \quad .$$

States are nullary functions with no input arguments, therefore they are considered to be constants. For every class C_i, the set of possible states defined by C_i-*STATES* must include a state $undef_{C_i}$, in which case the state of C_i is *undefined.* The corresponding signature is

$$undef_{C_i}: \quad \rightarrow C_i\text{-}STATES \quad .$$

For every pair of object-states s_1 and s_2 of C_i (including $undef_{C_i}$), $s_1 \neq s_2$. In order to bind a C_i-object to one of its possible states, a *valuation function*, \$, is introduced with the signature

$$\$: C_i \rightarrow C_i\text{-}STATES \quad ,$$

for each class name $C_i \in \mathcal{C}$. Figure 2.9 contains a simple object model that has an object class C and its states s_1 and s_2 (the target of double-headed arrows).

Figure 2.

Figures 2

Fig

compos

stores

the St

Figure 2.10 shows its corresponding A-schema $\mathcal{S}_3$, containing object-states s_1 and s_2. Figures 2.11 and 2.12 give a summary of the formalization of the object models.

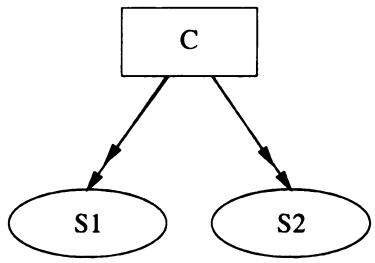

Figure 2.9: A simple object model

```
SCHEMA_diagram: trait
    includes CLASS_C

CLASS_C: trait
    introduces
      err_C   :  -> C
      s2      :  -> C_STATES
      s1      :  -> C_STATES
      state_C :  C -> C_STATES
    asserts
       forall x, y: C_STATES
      (x=s2 / y=s1) => (x~=y);
```

Figure 2.10: The corresponding algebraic specification of Figure 2.9

Figure 2.13 shows an object diagram of a simplified disk manager [27, 29] that is composed of (represented by a diamond) *Storage* and *Compressor*. The disk manager stores compressed data in the *Storage* and decompresses data that is retrieved from the *Storage*. We will use this example throughout this dissertation.

Definitio
Let C be
D_2. The

(OM1)

(OM2)

(OM3)
co
pa

(OM4)

Th

(OM5)
C t

is a

(OM6)
sp

is a
the

Definition (Semantics of object models) :
Let $\mathcal{O}$ be an object model. Let R be a binary association in $\mathcal{O}$ relating objects from classes D_1 and D_2. The semantics of $\mathcal{O}$ is an algebraic specification satisfying the following data.

(OM1) Each class C in the object model $\mathcal{O}$ is denoted by a sort of the same name.

(OM2) For each class C, a sort *C-STATES* is introduced as well as two nullary functions given by

$$undef_C : \quad \rightarrow C\text{-}STATES \ , \quad err_C : \quad \rightarrow C \ .$$

(OM3) Each object-state s, for which a double-headed arrow leads from a class C to the oval containing s, is denoted by a function with signature $s : \quad \rightarrow C\text{-}STATES$, and for every pair of object-states s_1 and s_2, the axiom $s_1 \neq s_2$ is included.

(OM4) For each class C, a valuation function '$' is introduced with the signature

$$\$: C \rightarrow C\text{-}STATES \quad .$$

The valuation of the error object is added as an axiom:

$$\$(err_C) = undef_C \quad .$$

(OM5) If there is a double-headed arrow labeled a (to indicate an attribute), leading from a class C to a class D (which depicts an attribute a of C), then the function signature

$$a : C \rightarrow D \quad ,$$

is added to specification for class C.

(OM6) If the class D in rule (OM5) is an external class, then the trait for D is **included** by the specification for C. If D has no parameters, then the clause

```
includes
     CLASS-D
```

is added. If D has parameters $p_1, \ldots, p_k$, and there is a line connecting each p_i to a class q_i, then the following clause is added:

```
includes
     CLASS-p1, ..., CLASS-pk,
     CLASS-D ( q1 for p1, ..., qk for pk)
```

Figure 2.11: Summary of Object Model Semantics

(OM7) Association R is denoted by the predicate

$$R : D_1, D_k \rightarrow BOOL \ .$$

(OM8) The endpoints of association R determine a set of axioms. Suppose the D_1-endpoint depicts a multiplicity of m and the D_2-endpoint depicts a multiplicity of n. Then the axioms are derived by the following steps:

1. Decompose the m-to-n association R into an m-to-1 and 1-to-n binary association,
2. Determine the second-order specifications, P_1 and P_2, of each of these associations using the basis schemata,
3. Calculate the "intersection", P, of the specifications P_1 and P_2,
4. Unfold and skolemize P, yielding a set of first-order axioms that are included in the trait for R.

(OM9) Error object constraints are introduced:

$$(\forall d_1 : D_1, d_2 : D_2 \bullet R(err_{D_1}, d_2) \wedge d_1 \neq err_{D_1} \Rightarrow \neg R(d_1, d_2)) \ ,$$

$$(\forall d_1 : D_1, d_2 : D_2 \bullet R(d_1, err_{D_2}) \wedge d_2 \neq err_{D_2} \Rightarrow \neg R(d_1, d_2)) \ .$$

Figure 2.12: Summary of Object Model Semantics (continued)

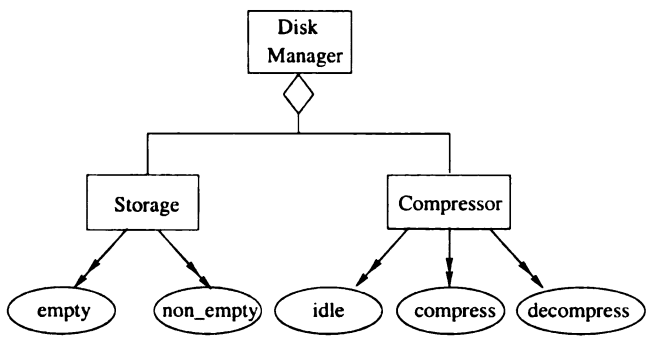

Figure 2.13: A simplified disk manager

Applying the object model formalization, Figure 2.14 contains the Larch specification for the *Storage* object class. **Empty** and **non_empty** are two states for the *Storage* class objects. The axiom in the **asserts** part specifies that the two states are mutually exclusive.

```
CLASS_Storage : trait
  introduces
     err_Storage : ->Storage
     empty :       -> Storage_STATES
     non_empty :   -> Storage_STATES
     State_Storage : Storage -> Storage_STATES
  asserts
     forall x, y : Storage_STATES
       (x=empty /\ y=non_empty) => (x ~= y);
```

Figure 2.14: A high-level *Storage* class specified in Larch

Given the similarity between LSL and ACT ONE, both of which are algebraic specificationsthe previous discussion and definitions are also applicable to the ACT ONE syntax. As a means to facilitate the integration between the object and dynamic models, we will use ACT ONE syntax instead of LSL to specify the object model in the remainder of this dissertation. Figure 2.15 contains the ACT ONE specification for the *Storage* object class. Respectively, the **sorts** and **opns** parts declare the *sorts* and specify the operators (constants are nullary operators) along with the signature for the type. The **eqns** section lists the equational axioms that the operators must satisfy. **Empty** and **non_empty** are two states for the *Storage* class objects. The axiom in the **eqns** part specifies that the two states are mutually exclusive.

```
type CLASS_Storage  is Boolean
  sorts
    Storage, Storage_STATES
  opns
    err_Storage : ->Storage
    empty :       -> Storage_STATES
    non_empty :   -> Storage_STATES
    State_Storage : Storage -> Storage_STATES
  eqns
    forall x, y : Storage_STATES
       ofsort Bool
         (x=empty /\ y=non_empty) => (x ~= y);
```

Figure 2.15: A high-level *Storage* class specified in ACT ONE

Ch

Re

In thi

catego

non-ob

3.1

This s

ROOA

metho

Chapter 3

Related Work

In this chapter, we overview related work. The related work is grouped into three categories: formalization of object-oriented development techniques, formalization of non-object-oriented approaches, and systematic refinement approaches.

3.1 Formalization of Object-Oriented Modeling Approaches

This section introduces object-oriented modeling approaches. TROLL [30] and ROOA [31] propose formal semantics for the models. ROOA and the Fusion method [7] provide detailed guidelines that facilitate the derivation of models.

3.1.1

Overvi

Jungla
the rep
systems
is a ter
integra
tion [34
As a co
propert

Forma

The pr
declara
and on
object
and an
class.

TR
scribe a
subclass

3.1.1 TROLL–A Language for Object-Oriented Specification of Information Systems

Overview

Jungclaus *et al.* developed a formal specification language, TROLL [30], to support the *requirement specification* phase in the early stages of object-oriented information systems development, including OMT. The underlying formal semantics of TROLL is a temporal logic [32]. Using object-oriented structuring mechanisms, TROLL integrates elements of algebraic specifications of data types [33], process specification [34, 35], the specification of reactive systems [36], and conceptual modeling [37]. As a consequence of the integration, TROLL is able to specify both the structural properties of objects via attributes and the behavior of objects through events.

Formalization Approach

The primary specification construct of TROLL is the *class template* that includes declarations for data types, attributes, and events, constraints on the observable states and on the evolution of attribute values, effects of events, behavior specification, and object activities. A class specification is composed of a class template specification and an identification mechanism that is used to distinguish individual objects of the class.

TROLL also provides the abstraction mechanisms, *specialization* and *role*, to describe an object from different aspects. Specialization describes objects belonging to subclasses that depend on observable properties (attributes) whereas roles describe

objects b

life occu

Comp

TROLL

compone

actions

For

tertaces

relation

among

tion pa

instanc

ject in

encaps

Comp

Simila

orienta

orienta

ours i

event

mecha

that

objects belonging to subclasses that depend on certain situations during an object's life (occurrence of events).

Composite objects that are composed of other objects are also supported by TROLL. In TROLL, a composite object is specified by including specifications of component objects in the composite object specification and synchronizing the interactions between component objects.

For a system comprising interaction objects, TROLL uses *relationships* and *interfaces* to define the interconnections among relatively independent objects. The *relationship* construct supports the specification of communication and constraints among objects. Using *relationship* constructs, the interconnection and communication patterns among separately defined objects can be defined; the constraints among instances of classes can be described. The *interface* construct is used to define object interfaces that are accessible to other objects, thus providing an information encapsulation mechanism.

Comparison with our approach

Similar to our approach, TROLL also proposes a set of formalization rules for object-oriented models. Its formalization rules cover all three basic aspects of an object-oriented system (static, dynamic, and functional). The formalization is similar to ours in that objects are formalized as ADTs and dynamic behavior is specified as event sequencing patterns. However, TROLL was proposed for use in the requirements analysis phase but not for design. TROLL has a new specification language that currently lacks tool support offered by other existing specification languages.

The advant

constructs c

to formaliza

derivation o

3.1.2 R

Overview

The Rigor

oriented an

a set of rul

specificatio

system. Ti

cation aga

omissions.

Analysis

ROOA ad

includes t

specificat

At fir

object-or

OMT [3

The advantage of designing a new specification language is that the structure and constructs of the language may fit the formalization objective very well. In addition to formalization rules, specific guidelines are also needed to facilitate analysts in the derivation of the formal specifications during analysis.

3.1.2 Rigorous Object-Oriented Analysis

Overview

The Rigorous Object-Oriented Analysis (ROOA) [38, 31, 39] formalizes object-oriented analysis notations in terms of the LOTOS specification language and includes a set of rules to conduct object-oriented analysis. The resulting formal, executable specifications of ROOA integrate the static, dynamic, and functional properties of a system. The specifications are amenable to checking the conformance of the specification against the original requirements and may be used to detect inconsistencies, omissions, and ambiguities early in the development process.

Analysis process

ROOA advocates a development process for object-oriented analysis. The process includes three stages: object model derivation, object model refinement, and formal specification derivation.

At first, a high-level, informal object model is built by using any of the usual object-oriented analysis methods, such as the methods of Coad and Yourdon [40] and OMT [3]. During the refinement stage, the object model is *completed* by adding *in-*

terface objects, static relationships, and attributes; the dynamic behavior is identified by defining interface scenarios, event trace diagrams (ETDs), object communication table (OCT), and adding message connections; the object model is structured into subsystems or aggregates to achieve modularity. Finally, formal specifications are derived to specify the static, dynamic, and functional properties of the system. This stage consists of the following steps: creating the object communication diagram (OCD); specifying the class templates as LOTOS processes and ADTs; composing objects; prototyping the object model by executing the LOTOS specifications; and refining the specifications according to the results of the rapid prototyping.

Formalization approach

ROOA formalizes objects in terms of LOTOS processes and abstract data types, where the LOTOS processes are used to specify the dynamic behavior of the objects, and the abstract data types (ADTs), given as parameters of the processes, are used to specify the state information of the objects. An object that merely plays the role of an attribute of another object is specified as a single abstract data type in ROOA. In order to avoid design issues, ROOA, proposed for use in the analysis phase, defines ADTs that contain only the necessary information to allow the specification to be prototyped with state information and values to be passed during the inter-object communication.

ROOA uses a *class template*, which is specified as a LOTOS process definition with ADTs as parameters, to represent common features of objects of the same kind. A *class* that represents a collection of objects consists of a *class template* and an

object id
uniquely
a class t
In F
is form
specific
that d
Th
cation
opera
cesse
may

Com

ROO
lang
mak
pha
as a
Ob
Ro
ra
Va

object identifier. An *object identifier* is an instance of a special LOTOS ADT used to uniquely identify objects. An object, a member of a class, is created by instantiating a class template and is assigned a unique identification value.

In ROOA, each object attribute is formalized as an ADT; each object service is formalized as an *operation* of an ADT. ADT operations that belong to an object specification are referenced as LOTOS value declarations in the corresponding process that describes the behavior of the object.

The communication between objects are formalized in terms of LOTOS communication constructs that include *full synchronization*, *interleaving*, and *synchronization* operators. Two objects that communicate and exchange data are specified as two processes that are synchronized through events on a *gate*. Complex object interactions may be built of simpler interactions by using the LOTOS composition operators.

Comparison with our approach

ROOA is similar to our approach in that both use the LOTOS formal specification language to formalize the object-oriented models. However, ROOA focuses on formal specification derivation during the analysis phase, while we focuses on the design phase. The formalization rules of ROOA do not treat the data properties of an object as an ADT but formalize every attribute as an ADT. The approach that ROOA uses to specify the dynamic features of objects is entirely different from what we proposed. ROOA defines states as sets of attribute values and formalizes states as process parameters. In contrast, we recognized that the date types and their corresponding values are not available neither during the analysis nor the high-level design stages.

Ther

gebra

aspe

ROO

that

as th

desig

may

3.1.3

Over

Colem

port t

menta

certai

Booch

from a

ever, t

comput

Instead

of soft

models

Therefore we symbolically formalized the states in terms of processes of process algebras and focused on the behavior in different states of an object. The functional aspect, which is another important feature of a system, is not fully addressed in ROOA. This may leave the specification incomplete for the design phase. We feel that there is too much design information included in the analysis information, such as the attributes and services provided by individual objects. This may interfere with design decisions. However, its introduction of the object identifier is interesting and may be worth investigating further.

3.1.3 The Fusion Method

Overview

Coleman *et al.* [7] proposed the Fusion object-oriented development method to support the entire software development life cycle, including analysis, design, and implementation. It can be used to develop sequential object-oriented systems as well as certain restricted types of concurrent systems. By integrating and extending OMT [3], Booch [41], CRC [42], and formal methods, Fusion attempts to provide a direct route from a requirements definition to a programming language implementation. However, the use of formal methods in the context of Fusion does not have the typical connotation. No specific formal specification language is used to describe the models. Instead, sets of informal check lists expressed in natural language for different phases of software development are given to check the *completeness* and *consistency* of the models.

The

Fusio

meta

of O

life cy

give t

when

Anal

varia

interf

of the

objec

T

in te

relati

cycle

to de

its te

opera

of the

the op

The approach

Fusion includes a set of models and a process for software development. The graphical notations of the models are based on the existing, well-known notations, such as those of OMT [3] and Booch [41] methods. The development process divides the software life cycle into phases and indicates what should be done in each phase. Guidelines give the order of the tasks to be completed within phases and the criteria that indicate when to move to the next phase in order to ensure that development makes progress.

Analysis. The analysis phase of Fusion is mainly based on OMT [3] with some variations. Unlike OMT, the individual objects during the analysis of Fusion have no interface and no dynamic behavior. Instead, Fusion focuses on the intended behavior of the *system* during the analysis phase. The analysis results in three models: system object model, life-cycle model, and operation model.

The system object model defines the static structure of the information of a system in terms of a set of objects to be built. It is derived by eliminating the classes and relationships that belong to the environment in the original object model. The life-cycle model characterizes the allowable sequencing of system operations and events to describe how the system communicates with its environment from creation until its termination. The operation model depicts the behavior (effect) of each system operation by defining their effect (by using preconditions and postconditions) in terms of the state change it causes and the output events it sends. The life-cycle model and the operation model together specify the behavior of a system.

The inf

consistency

models (sy

possible sy

to ensure

other.

Design.

methods.

definition

CRC. wh

produced

object in

model.

For

is deriv

An obje

collecti

model.

paths in

descript

model.

attribu

The informal checklist given by Fusion includes the completeness checking and consistency checking. The intent of completeness checking is to ensure that the three models (system object, life-cycle, and operation) cover all the static information, possible system scenarios, and system operations. The consistency checking is used to ensure that the information expressed by different models are consistent with each other.

Design. The design phase of Fusion is based upon the CRC [42] and Booch [41] methods. During design, Fusion introduces *software* structures to satisfy the abstract definitions generated during analysis. The systematic design process is derived from CRC, whose main goal is to explore object interactions from the operation model produced during analysis. During design, four models are developed and refined: object interaction model, visibility model, class description model, and inheritance model.

For each system operation in the operation model, an object interaction model is derived to show how functionality is distributed across the objects of a system. An object interaction model defines the sequences of messages that occur between a collection of objects to implement a certain operation. Based on the object interaction model, a visibility model is derived for each class to describe the object communication paths in the system. A visibility model shows how a class is referenced. A class description model for a particular class collates information from the system object model, object interaction model, and visibility model to derive the methods, data attributes, and object-valued attributes of the class, respectively. Given the functional

definition o

reference st

description

inheritanc

All th

analysis n

the desig

Compar

This app

developr

the cons

the deve

be used

els. thu

informa

specific

propose

fication

matter

ous co

paradi

may be

definition of operations in the object interaction models, the consequences to the class reference structures in the visibility models, and the class specifications in the class description models, commonalities and abstraction can be identified to introduce an inheritance model for the classes of the system.

All the models produced during the design phase can be checked against the analysis models. In addition, rules are also given to check the consistencies among the design models.

Comparison with our approach

This approach is an attempt to integrate the best aspects of different well-known development methods. Similar to our approach, Fusion proposes checklists to check the consistency between the models in order to eliminate errors introduced during the development process. Although formal methods are mentioned and claimed to be used, there is no use of formal specification languages for representing the models, thus preventing rigorous analysis of the models and leaving the checklists in an informal format. Our approach introduces formal specifications in terms of a formal specification language beginning with the analysis phase. Our development paradigm proposed for design focuses on a stepwise refinement that is based on formal specifications. Our approach has significant advantage over the Fusion approach in that mathematical reasoning about the properties of the target system and formal, rigorous consistency checking between models are possible. However, the development paradigm for analysis and design of Fusion is easier to follow than that of OMT and may be useful for our design paradigm development.

3.1.4

Overvi

The Ra

attemp

object-

metho

progre

Unifie

standa

Octob

for sp

tweer

devel

Unif

The

struc

mos

foll

3.1.4 The Unified Method

Overview

The Rational Corporation founded by Grady Booch and James Rumbaugh has been attempting to develop a *unified method* that can be used as a standard method for object-oriented software development. Joining forces with other software development methodologists (e.g., Ivar Jacobson, Bran Selic, etc.), Rational has made significant progress in the past year (1997). The promising products of Rational include the Unified Modeling Language (UML) [43, 44, 45, 46], which has been adopted as the standard object modeling language by the Object Management Group (OMG) in October 1997, the Object Constraint Language (OCL) [47] that uses simple logic for specifying invariant properties of systems comprising sets and relationships between sets, and a development process, Objectory [48], that covers the entire software development life cycle.

Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems [44]. The primary goals of UML are as follows:

- Provide users with a ready-to-use, expressive visual modeling language to be used to develop and exchange meaningful models.
- Provide extensibility and specialization mechanisms to extend the core concept.
- Be independent of particular programming languages and development processes.

• Pr

• E

• St
pa

• In

UM

differ

human

Th

The U

langua

Th

subset

langua

to ho

diagra

natur

prima

on th

In

Stan

•

•

- Provide a formal basis for understanding the modeling language.
- Encourage the growth of the OO tools market.
- Support higher-level development concepts such as collaborations, frameworks, patterns, and components.
- Integrate the best software engineering development practices.

UML focuses on providing (1) a common metamodel (which unifies semantics of different object modeling approaches), and (2) a common notation (which provides a human rendering of these semantics).

The semantics of UML diagrams are described in terms of the UML metamodel. The UML metamodel is described in a combination of graphic notation, natural language, and formal language.

The abstract syntax for UML diagrams is provided as a model described in a subset of UML notation, consisting of a UML class diagram and a supporting natural language description. (In this way, UML is bootstrapping itself in a manner similar to how a compiler is used to compile itself.) The well-formedness rules for UML diagrams are provided using a formal language, Object Constraint Language, and natural language (English). Finally, the semantics for UML diagrams are described primarily in natural language, but may include some additional notation, depending on the part of the model being described.

In terms of the views of a model, the UML defines the following graphical diagrams:

- **Use case diagram**: describes the relationship among users and use cases (scenarios that a system is used) within a system.
- **Class diagram**: describes the types of objects in the system and the various kinds of static relationships that exist among them.

- **Behavior diagrams**: describe different perspectives of the behavior of the objects.
  - **Statechart diagram**: shows the sequences of states that an object or an interaction goes through during its life in response to received stimuli, together with its responses and actions.
  - **Activity diagram**: is a variation of a state machine in which the states are activities representing the performance of operations and the transitions are triggered by the completion of the operations. (It represents a state machine of procedure itself; the procedure is the implementation of an operation on the owning class.)
  - **Interaction diagrams**: describe how groups of objects collaborate in some behavior.
    - **Sequence diagram**: shows an interaction arranged in time sequence.
    - **Collaboration diagram**: shows an interaction organized around the objects in the interaction and their links to each other.
- **Implementation diagrams**: show aspects of implementation, including source code structure and run-time implementation structure.
  - **Component diagram**: shows the dependencies among software components, including source code components, binary code components, and executable components.
  - **Deployment diagram**: shows the configuration of run-time processing elements and the software components, processes, and objects that live on them.

Object Constraint Language (OCL)

OCL is a formal language that can be used to express side-effect-free constraints. UML uses OCL to specify the well-formedness rules of the UML metamodel. In the context of UML modeling, OCL can be used for a number of different purposes [47]:

- To specify invariants on classes and types in the class model.
- To specify type in invariants for *Stereotypes* (concrete examples).
- To describe pre- and postconditions on *Operations* and *Methods*.
- To describe *Guard* on transitions in state diagrams.
- To specify constraints on operations.

Because OCL is a pure expression language (declarative language with no side effect), the state of the system will never change because of an OCL expression, even though an OCL expression can be used to specify a state change, e.g., in a postcondition.

As a typed language, OCL provides a set of predefined types, with associated operations, for modelers. The predefined types, including *Boolean*, *Integer*, *Real*, *String*, etc., are independent of any object model and part of the definition of OCL. In the context of UML, all types/classes from the UML model are types in OCL that are attached to the model.

OCL expressions can refer to types, classes, interfaces, associations and data types. All attributes, association-ends, methods, and operations without side-effects (change the state of the system) that are defined on these types can be used.

Objectory process

Objectory [48] is a process developed by Rational to support the full life cycle of software development. Objectory advocates a controlled iterative process, with strong focus on architecture. It is a use-case driven, object-oriented process, using UML as a notation for its models.

The Objectory process can be described in two dimensions: *time* and *process components*. In terms of time, the Objectory process divides a development cycle into four consecutive phases:

- **Inception phase: establishes the business case for the system and delimits the project scope.**

- Elaboration
  foundation
  the project
- Constructio
  is ready to
- Transition p

Similar to oth

ponents that Ob

- Requiremen
  velopers an
  supplement
- Analysis an
  phase.
- Implementa
- Test: verifi

Comparison w

Rational's prod

quite different i

UML origin

used in our app

the notation in

explicitly used i

dynamic model

an object are in

data flow refine

- Elaboration phase: analyzes the problem domain, establishes a sound architectural foundation, develops the project plan and eliminates the highest risk elements of the project.
- Construction phase: iteratively and incrementally develops a complete product that is ready to transition to its user community.
- Transition phase: transitions the software to the community.

Similar to other software development processes, the four engineering process components that Objectory has are:

- Requirements capture: describes *what* the system should do and allows the developers and the customers to agree on that description. A use-case model and supplementary requirements are the results of this component.
- Analysis and design: shows *how* the system will be realized in the implementation phase.
- Implementation: produces the sources that will result in an executable system.
- Test: verifies the entire system.

Comparison with our approach

Rational's products share a lot of similarities with our approach, yet they are still quite different in several aspects.

UML originated from OMT [3], Booch's method [49], and OOSE [50]. OMT is used in our approach as the graphical front end. The OMT notation is a subset of the notation included in UML, with the exception that data flow diagrams are not explicitly used in UML. The *Activity* diagram in UML shares similar concepts with our dynamic model refinement. Both of them are intended to describe *how* the methods of an object are implemented. Our newly introduced SRFM that focuses on describing data flow refinement does not have a counterpart in UML. While UML notation is the

crucial pr

front-end

formal se

methods

operators

purpose.

defined i

to suppo

of OCL.

and utili

graphica

it is anti

software

In g

proach

object-

tation.

to *dem*

engine

langua

contras

formal

using fo

crucial product component from Rational, OMT notation only serves as the graphical front-end in our approach. The emphasis and focus of our approach are the underlying formal semantics. OCL in UML allows users to specify pre- and postconditions for methods and operations, and guarding conditions for state diagrams. In our approach, operators defined in the algebraic specifications (in ACT ONE) are used for the same purpose. Since OCL uses methods depicted in *Class* diagrams instead of operators defined in well-defined algebraic specifications, no rewriting mechanism is available to support a rigorous reasoning of the pre- and postconditions expressed in terms of OCL. The Objectory process covers the entire life cycle of software development and utilizes UML notation. Our design process focuses on utilizing formalized OMT graphical notations to conduct a stepwise refinement during the design phase, while it is anticipated that the technique can be extended to the implementation phase of software development.

In general, the underlying motivation and philosophy of Rational's and our approach are quite different. Rational attempts to introduce standardized notation for object-oriented modeling as well as a development process that utilizes the UML notation. A large amount of design effort of Rational's Objectory process is contributed to *demystify* the design process. Concerned with the lack of familiarity of the software engineering community with formal methods, only a light-weight formal specification language, OCL, is introduced to describe a part of the system during modeling. In contrast, the motivation of our research from the beginning has been to introduce formal methods into software development. In order to alleviate the difficulty of using formal methods and textual, abstract formal notation, we chose OMT as the

graphica

framewo

and serv

the forr

formal

ment o

formal

sis are

develo

of a fo

3.1.5

Over

Real-

for re

and (

bines

emp

Rabi

abi

els.

pro

graphical front-end and formalized its notation. A well-defined formal specification framework with sound mathematical foundation forms the basis of our formalization and serves as the back-end of the approach. Our design process is developed to utilize the formalized and integrated graphical notation in order to systematically employ formal methods during the design process. The process advocates a stepwise refinement of design. Because well-defined formal specifications constitute the underlying formal semantics of the graphical notation, rigorous analysis and specification analysis are amenable to designers. Therefore our research serves to provide a means for developers to move from an easy-to-use graphical modeling technique to the benefits of a formal description of a system amenable to automated analysis.

3.1.5 The Real-Time Object-Oriented Modeling

Overview

Real-Time Object-Oriented Modeling (ROOM) [51, 52] is an object-oriented method for real-time systems developed originally at Bell-Northern Research by Bran Selic and Garth Gullekson. ROOM is a method for developing real-time software that combines the object paradigm with advanced domain-specific modeling concepts. Special emphasis is placed on modeling the architectural levels of software that are key to reliability, understandability, and evolvability. The method is also distinguished by its ability to take advantage of computer-based automation (through executable models, reuse, and automatic code generation) for better product quality and greater productivity.

The approach

ROOM includes
nization framew

The modeling

time modeling l
principle of usi
the modeling la
discontinuities i
Actors are objec
models are com
along protocols
described by *R*
protocols, and

In terms of

- Actor cla
- ROOMch

ROOM als
ROOM specifi
This enables RO
the executable

The approach

ROOM includes a modeling language, a set of modeling heuristics, and a work organization framework.

The modeling language. The ROOM modeling language was developed as a real-time modeling language rather than as a general-purpose language. Based upon a principle of using the same set of models for all phases of the development process, the modeling language is developed to avoid introducing any arbitrary or unnecessary discontinuities into the development process. The key concept in ROOM is an *actor*. *Actors* are objects that are independent, concurrently active logical machines. ROOM models are composed of actors that communicate with each other by sending messages along protocols. Actors may be hierarchically decomposed, and may have behaviors described by *ROOMcharts*, a variant of Harel's state charts. Descriptions of actors, protocols, and behaviors can all be reused through inheritance.

In terms of the views of a model, ROOM defines the following graphical diagrams:

- Actor class diagram: captures all high-level structures of a system.
- ROOMchart: captures the high-level behavior of actors.

ROOM also has a standard *ROOM linear form* that can be used to represent ROOM specifications, described in terms of graphical diagrams, in textual format. This enables ROOM to subject its model specification to some type of analysis. Given the executable nature of ROOMcharts, formal analytical approaches to validation is

possible. Valid

are specified in

Modeling heu

of modeling heu

development pr

validation and

the basis for th

Work organi

volved with cr

experience wit

opment metho

Comparison

ROOM and c

proach attem

opment proce

are different.

the interactio

mantics for th

analysis. How

system. The

well-defined, s

possible. Validation of ROOM models is primarily by execution of scenarios, which are specified in terms of test cases.

Modeling heuristics. Based upon the modeling language, ROOM provides a set of modeling heuristics to conduct model development. The heuristics include a model development process that advocates an iterative approach to model construction and validation and heuristics that pertain to the architecture of a system, which forms the basis for the system's long-term evolution.

Work organization. Work organization refers to the top-level process issues involved with creating products and managing projects. Based upon actual industrial experience with large complex projects, ROOM advocates a *product-oriented* development methodology.

Comparison with our approach

ROOM and our approach share a number of similarities. Both ROOM are our approach attempt to use the same set of models for all phases of the software development process. Although the specific modeling techniques of the two approaches are different, both of them focus on modeling individual objects, their behavior, and the interaction between active, concurrent objects. The executable nature of the semantics for the behavior model enables both approaches to perform formal validation analysis. However, ROOM has special strength in describing static structures of a system. The key difference between the two approaches is that our approach has a well-defined, solid mathematical foundation in terms of process algebras and algebraic

specificati

more rigo

analysis c

3.2

This sect

approach

to forma

data flow

3.2.1

Overvi

The SA

Design

with th

pure q

Forma

There

analys

models

terms o

specifications, whereas ROOM relies on rigor definition of ROOMcharts. This enables more rigorous analysis methods amenable to our specifications other than validation analysis only.

3.2 Non-Object-Oriented Based Approaches

This section gives an overview of the formalization of three non-object-oriented based approaches. SAZ [53] and Semmen's method [54] both use the Z specification language to formalize structured development methods. France [55] formalized an extended data flow diagram in terms of an algebraic specification language.

3.2.1 SAZ Method

Overview

The SAZ Project [53] combines the benefits of the Structured Systems Analysis and Design Method (SSADM), comprising data flow and entity-relationship diagrams, with that offered by the Z specification language. In SAZ, Z can be used either as a pure quality assurance tool or as an integral systems analysis technique.

Formalization approach

There are two uses of Z in the SAZ project. The simpler one is its use for quality audits. A formal data model is prepared after requirements analysis and formal models of logical processing are derived at the end of requirements specification in terms of Z specifications by a third party in conjunction with the development process.

If any pr

to be ref

The

process

includes

physica

tion of

partic

compa

of the

inqui

T

If any problems or ambiguities are identified in the Z specifications, alterations have to be referred back to the development team.

The other approach is to integrate the use of Z into the SSADM development process to conduct a rigorous, formal software development. The SSADM method includes five modules: feasibility, requirements analysis and specification, logical and physical design. The system state of the logical data model (LDM), which is a variation of Entity-Relationship models, and parts of the functional requirements that are particularly complex or critical are then presented in the specification language Z. It comprises three main elements: the specification of the system state (or a sub-model of the state), the specification of critical processing, and the specification of selected inquiries. The specifications can also be refined with the development of the system.

The principal benefits of the integrated method are:

- the ability to express features of the functional system requirements that are not well documented or which have been omitted from the SSADM requirements specification;
- the expression of all functional requirements in a common, mechanically-checkable notation;
- the ability to provide (formal or informal) reasoning about, for instance, pre-conditions to operations;
- the facility for concise expression of error processing.

Compa

SAZ is a

in orde

Logical

terpart

approa

focuses

tions.

In ad

sophis

icate

emplo

tegra

easier

3.2.

Ove

For

syn

and

Comparison with our approach

SAZ is also an attempt to introduce formalisms into a software development process in order to eliminate inconsistency, ambiguity, and incompleteness. The formalized Logical Data Structure (LDS, the diagrammatic part of LDM) and DFD have counterparts in our approach. Its formalization mechanism is entirely different from our approach. Z is a specification language based on set theory. SAZ's specification focuses on obtaining a state specification of the system, which comprises type definitions, entity types, entity sets, and relationships that represent the state of a system. In addition, SAZ is currently still a proposed methodology and, so far, there is no sophisticated tool support. Our approach is based upon algebraic specification, predicate specification, and process algebras. We advocate an systematic approach to employ formal methods during software design process in a stepwise fashion. The integration of the OMT graphical models and formal methods also makes our approach easier for the developers to use.

3.2.2 Yourdon and Z

Overview

For the Yourdon modeling approach, Semmens and Allen [54] have developed a Z syntax for the *Entity Relationship* diagrams (ERDs), *Data Flow* diagrams (DFDs), and data dictionary to integrate formal methods with structured analysis.

Formaliz

Initially, 7
schema t
tities, re
are deriv
schemas
upon th
tion sch
operati
the dia

Th
specifi
can he
tected
used.

Com

Semr
deve
of LI
the Y
appr

Formalization approach

Initially, Z specifications are derived from ERDs. During this stage, basic types and schema types are defined corresponding to the attributes of each entity and the entities, respectively; state schemas for the instances of each entity and its subtypes are derived; relationships in the diagram are declared; the entity and relationship schemas are combined to provide a complete specification of the system state. Based upon the formal specifications derived from ERDs, DFDs are formalized as Z operation schemas. The DFD does not give enough information to specify completely the operations, but some parts of the operation schemas can be obtained directly from the diagram and the data dictionary.

The result of the analysis by using the proposed integrated method is a formal specification that describes both static and dynamic properties of the system. These can help to significantly reduce the chances of specification errors remaining undetected until later phases of the life cycle when conventional, informal methods are used.

Comparison with our approach

Semmen's approach is another attempt to formalize models of structured software development method. Its formalization of ERDs is similar to SAZ's formalization of LDS, which is a variation of ERD. Since the dynamic behavior of the system in the Yourdon modeling approach refers to DFDs instead of state diagrams, Semmen's approach, unlike ours, formalizes DFDs as a means to formally describe the dynamic

behavior c

and objec

of applica

better se

formaliz

duce for

formaliz

3.2.3

Overv

France

extend

contro

forma

applic

Form

Ext:D

C-DF

enti

inve

behavior of the system. The difference that lies in the nature of structured methods and object-oriented methods puts Semmen's and our approaches in distinct fields of application. By proposing a stepwise development framework, our approach may better serve the software development process rather than only providing a set of formalization rules. Although Semmen's method is proposed as a means to introduce formalisms into analysis for structured methods, the information captured and formalized from DFDs is also applicable to our approach.

3.2.3 Semantically Extended Data Flow Diagrams

Overview

France defined a *semantically Extended DFD* (ExtDFD)[55] based on a control-extended DFD (C-DFD), which is a DFD supplemented with notation for describing control dependencies among its elements, associated with formal semantics. By giving formal semantics to the C-DFD, ExtDFD allows the specifiers to investigate desired application properties and verify semantic decompositions of data transformations.

Formalization approach

ExtDFD formalizes C-DFD in terms of algebraic specifications. The components of C-DFD, including the data and state transformation, asynchronous flows, external entities, and data stores, are formalized as *processes*; and a C-DFD is semantically interpreted as a system of communicating processes. *Processes*, specified algebraically

by *algebraic st*

and events and

The semanti

up fashion fro

lowing approa

1. Derive A

supplied

called th

2. Derive a

among

the *Sy*

DFD, is

3. Derive

the sy

resultir

Given th

and I/O co

verify the c

Comparis

Although A

specificatio

by *algebraic state transition systems* (ASTSs) [56, 57], are composed of sets of states and events and classes of behaviors.

The semantics of a C-DFD are specified in terms of ASTSs generated in a bottom-up fashion from ASTSs that describe the individual C-DFD components in the following approach:

1. Derive ASTSs characterizing the behavior of each C-DFD component from specifier-supplied descriptions. The resulting set of ASTSs, together with the C-DFD, is called the *Basic Interpreted C-DFD.*

2. Derive an ASTS characterizing the synchronous interactions that can take place among C-DFD components from the Basic Interpreted C-DFD. This ASTS is called the *Synchronous Interaction Specification* (SIS). The SIS, together with the C-DFD, is called *Basic ExtDFD.*

3. Derive an ASTS characterizing the permissible time-dependent relationships among the synchronous interactions specified in the SIS from the Basic ExtDFD. The resulting ASTS is called the *Behavioral Specification* (BS).

Given the formal semantics of C-DFDs in terms of ASTSs, *syntactic consistency* and *I/O consistency* are defined for the ExtDFD in order to formally and rigorously verify the consistency between levels of refinements.

Comparison with our approach

Although ASTS, the underlying formal specification language, is also an algebraic specification language similar to LSL and ACT ONE, the ExtDFD is significantly

different from our approach. The ExtDFD is proposed for structured methods; our approach is intended for object-oriented methods. The ExtDFD does not contain any model that is similar to the object model. The integration of the data flow information and control information in ExtDFD is achieved from extending the DFDs by including state and control information, whereas the integration of the two models in our approach is achieved in terms of the underlying formal semantics of the models. In ExtDFD, the formalization rules only formalize C-DFDs in terms of ASTSs but do not contribute to model integration. In our approach, the three OMT models are formalized in terms of algebraic specifications, predicate specifications, and process algebras. The integration among the models are achieved in terms of the underlying formal semantics of the models. In addition, our approach also includes a design process to take advantage of the formalization and integration techniques.

3.3 Systematic Approaches to Refinement of Analysis Information for Design

In this section, we describe two approaches that advocate rigorous, systematic refinement for software development.

3.3.1

Overvie

VDM [5

piler des

process

to its in

develop

Forma

VDM

velopr

with a

specif

and c

refine

T

defin

defin

rely

Post

relat

to sa

3.3.1 Vienna Development Method (VDM)

Overview

VDM [58, 59] was originally developed to be used for language definition and compiler design in the 1960's. The original objective was to develop a systematic design process that progresses from a rigorous definition of a given programming language to its implementation. Later, the objective was broadened to include the systematic development and refinement of designs.

Formalization approach

VDM includes both a formal description language (META-IV) and a systematic development process. The VDM design is a series of specifications in META-IV, starting with an abstract specification of the functions that the system will perform. Each specification in the series is a refinement of the previous one that is more concrete and closer to the implementation. Proof obligations that ensure the correctness of the refinement are identified and must be fulfilled by further development and refinement.

The VDM [59] specification language constructs include: modules, interfaces, type definitions, state definitions, value definitions, function definitions, and operation definitions. A VDM specification defines a set of states and a set of operations that rely on and transform the states. A single operation is specified in terms of pre- and postconditions. Given a specification, the design phase realizes data objects by data reification (refinement) in high-level design stages and develops control constructs to satisfy postconditions by operation decomposition in low-level design stages. In

addition to th

obligations ar

operation dec

Comparison

VDM also focu

process. It is

The formal s

stepwise fash

for data reifi

that ensure t

benefit our f

3.3.2 C

Overview

Based on D

that deals w

program tra

total correct

by reasonin

of axioms a

addition to the rules that conduct data reification and operation decomposition, proof obligations are also given by VDM to guarantee the correctness of data reification and operation decomposition.

Comparison with our approach

VDM also focuses on stepwise design and correctness preservation in the refinement process. It is function-oriented and is suitable for structured development methods. The formal specifications are derived directly from requirements and refined in a stepwise fashion without assistance of intuitive graphical models. However, its rules for data reification and operation decomposition, particularly, the proof obligations that ensure the correctness of refinement are worthy of further investigation and may benefit our formalization of the design paradigm.

3.3.2 Correctness preserving program refinements

Overview

Based on Dijkstra's work [60], Back [61] proposed a program development approach that deals with refinements concerned with data representations, control structures, program transformation rules, and implementation of procedure. In addition, the total correctness of a program can be syntactically proved (not semantically argued) by reasoning about the correctness of refinement steps in a formal system with a set of axioms and proof rules.

The app

The unde

logic. L

and conj

over fini

infinite

Bac

the lan

langua

selecti

Ha

precc

tions

refin

Cor

Bo

de

lev

sp

n

n

The approach

The underlying logic that carries out proofs of program properties is an infinitary logic, $L_{\omega_1\omega}$, which is an extension of ordinary first-order logic, that allows disjunctions and conjunctions over a countably infinite number of formulas and quantifications only over finite sequences of variables. There are also inference rules that have a countably infinite number of premises.

Back used the language of *descriptions* to describe state transformations, that is, the language is used to specify a program. The primitive statement constructs of the language includes: a nondeterministic assignment, control structures of composition, selection, iteration, and nondeterministic binary choice.

Having a set of theorems for proving refinement correctness in terms of weakest preconditions, Back further proposed a top-down stepwise refinement using *descriptions.* The proposed formal development approach allows the use of proof rules for refinement to establish the correctness of the refinement steps.

Comparison with our approach

Both Back's and our approaches advocate a stepwise refinement during the software development process. However, Back's approach deals with the refinement of low-level program development and algorithm design, whereas ours focuses on high-level specification refinement during the design process. The application areas of Back's method and ours are totally different. Back's approach mainly treats the development of procedures and assumes that the specification of program termination can be

acquire

serve a

to the

further

acquired and directly represented in terms of $L_{\omega_1\omega}$. Our specification derivation may serve as the preceding stage for Back's method. This method is directly applicable to the implementation of the detail design specification and may be valuable for our further research when we move from design to the implementation phase.

Chapter 4

Object Model Formalization Revisited

In this chapter, we re-examine the object model formalization and introduce a new formalization rule for the object model.

4.1 The Derivation of Algebraic Specifications

Although the algebraic specification shown in Figure 2.15 can be used for requirements analysis purposes, it does not provide sufficient design information for implementation such as attributes and operations. The original intent of the object diagram formalization [27] was to define formal semantics for the high-level object diagram used in the analysis phase, where these specifications should be amenable to refinement to include design details. However, the acquisition of such detailed specifications is not easy to obtain and requires more information than can be directly determined from

the formalized object diagrams. Since the design stage typically introduces more details about operations that describe the system behavior, we use information from the functional and dynamic models to contribute to the design specifications.

The functional model represents the operations and data transformations of the system in terms of data flow diagrams. At the most detailed level, each of the operations, or processes, are operators of objects depicted in the object model. The process bubbles in a data flow diagram determine the services of the system, thereby defining the operators for the objects and the events, actions, and activities in the dynamic model. From the formalization perspective (see Chapter 6), the signature of a given operator for an object is determined by the input and the output data flows to/from the corresponding process bubble. This signature information is used in both the object model definition during the design stage and in the dynamic model specification.

4.2 Distinguished Sort

In a specific problem domain, by sharing some commonality, a set of objects can be abstracted into a class. In the world of software engineering, the commonality is represented as attributes that are expressed in terms of specific data values. In the corresponding algebraic specification for a class of objects, each attribute is represented by a data sort that denotes a range of values for the attribute. In order to refer to the set of sorts for attributes of an object class, a sort that represents the object class and a *tuple* operator can be introduced into the algebraic specification.

The sort

the typ

attribu

sort to

struct

of an

tinguis

its att

rigoro

Fo

SSN

for th

with

In th

4.3

In a

rul

dis

Op

att

alg

The sort of the object class is defined as *distinguished sort*, which is sometimes called the type of interest or data sort [19]. The *tuple* operator is proposed to contain the attributes of an object to form a value of the distinguished sort. The distinguished sort together with the *tuple* operator are like a typical record type declaration construct provided by most programming languages. If, for some reason, the attributes of an object class cannot be identified, an abstract sort can be declared as the distinguished sort instead. The abstract distinguished sort may be refined into sorts of its attributes later during the design process when it is necessary. In order to achieve rigorous specification, we insist that every object class must have a distinguished sort.

For instance, every person in our society has a gender and a social security number (SSN). In the world of computer science, this commonality is represented as attributes for the *person* class. In terms of algebraic specifications, the *Person* distinguished sort with *Gender* and *SSN* as attributes results in the LOTOS specification in Figure 4.1. In this specification, the *tuple* operator is also named *Person*.

4.3 An added formalization rule

In addition to the object model formalization rules given in Section 2.4, another rule, **OM10**, is added to formalize the attributes of an object class in terms of a distinguished sort. The attributes of an object are a list of variable-type pairs in OMT's object model. The motivation for this rule is to enable access to individual attribute values. The object class and types of attributes are formalized as sorts of algebraic specifications. A tuple that maps from the sorts of attributes to the sort

```
type Person is
    sorts
      Person, Gender, Nat
    opns
      male      : -> Gender
      female    : -> Gender
      Person    : Gender, Nat -> Person
      getgender : Person -> Gender
      getssn    : Person -> Nat
    eqns
      forall p: Person, s: Nat, g: Gender
      ofsort Person
         Person (getgender(p), getssn(p)) = p
      ofsort Gender
         getgender(Person(g,s)) = g
      ofsort Nat
         getssn(Person(g,s)) = s
endtype
```

Figure 4.1: The ACT ONE specification for *person*

of their corresponding object class is also introduced. The variables are formalized in terms of operations that map from the sort of the object class to the sorts of the variables.

OM10 The attributes, in the form of variable-type pairs, of an object class are formalized as sorts and a tuple that maps attribute types to the distinguished sort that represents the object class.
For object O, given attributes $a_1 : A_1, a_2 : A_2, ..., a_n : A_n$ (where $a_1, a_2, ..., a_n$ are variables and $A_1, A_2, ..., A_n$ are types), create the following expression in LOTOS specification

```
typedef O is
    sorts
       O, A_1, A_2, ..., A_n
    opns
       O : A_1, A_2, ..., A_n  ⟶  O
       get_a_1 : O  ⟶  A_1
       get_a_2 : O  ⟶  A_2
       ......
       get_a_n : O  ⟶  A_n
    eqns
       forall a_1 : A_1, a_2 : A_2, ..., a_n : A_n
       ofsort A_1
          get_a_1(O(a_1, a_2, ..., a_n)) = a_1
       ofsort A_2
          get_a_2(O(a_1, a_2, ..., a_n)) = a_2
          ......
       ofsort A_n
          get_a_n(O(a_1, a_2, ..., a_n)) = a_n
endtype
```

Regardless of whether attributes are given to object O, sort O is treated as the distinguished sort. Figure 4.2 gives a sample object diagram. Based upon formalization rule **OM10**, the automatically generated formal specification is shown in Figure 4.3.

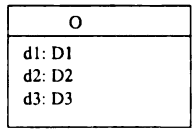

Figure 4.2: A sample object model that contains attributes for object O

typedef
sor

opr

eq

endtyp

Figur
mode

4.4

An a

set o

o 'S

gene

of th

whos

incl

are s

```
typedef O is
    sorts
        O, D1, D2, D3
    opns
        O : D1, D2, D3 -> O
        get_d1 : O -> D1
        get_d2 : O -> D2
        get_d3 : O -> D3
    eqns
        forall d1: D1, d2: D2, d3: D3
        ofsort D1
            get_d1 (O(d1, d2, d3)) = d1;
        ofsort D2
            get_d2 (O(d1, d2, d3)) = d2;
        ofsort D3
            get_d3 (O(d1, d2, d3)) = d3;
endtype
```

Figure 4.3: The automatically generated formal specification from the sample object model

4.4 Formal Representation of Algebraic Specification

An algebraic specification can be represented as a tuple $\langle \Phi, \Gamma, \Psi, \phi \rangle$, where Φ is a set of sorts, Γ is a set of operators with their signatures, Ψ is a set of axioms, and ϕ ($\in \Phi$) is a distinguished sort. The operators Γ can be classified into four categories: *generators*, *observers*, *extensions*, and *auxiliaries*. The *generators* create all the values of the distinguished sort. The *extensions* are the remaining operators, such as **delete**, whose range is the distinguished sort. The *observers* are the operators whose domain includes the distinguished sort and whose range is some other sort. The *auxiliaries* are similar to *observers* but the domain does not include the distinguished sort.

In order to facilitate the discussion of the dynamic model formalization, we orthogonally classify the operators as *modifiers* and *non-modifiers*. The *modifiers* include *generators* and *extensions*, whose range is the distinguished sort; the *non-modifiers* consist of *observers* and *auxiliaries*, whose range is some other sort. Thus Γ is partitioned into two mutually exclusive sets, Ω and Θ, where Ω is the set of modifiers and Θ is the set of non-modifiers. According to this classification, operator `0` in Figure 4.3 is a modifier; operators `get_d1`, `get_d2`, and `get_d3` are non-modifiers.

Ch

Dy

The

form

nota

num

and

pro

be

mo

for

for

La

Sp

Chapter 5

Dynamic Model Formalization

The dynamic model in OMT [3] is based on David Harel's *statecharts* [8]. In our formalization [62, 63], we retain the hierarchical structure and most of the syntactic notations of *statecharts*. The OMT approach for creating state diagrams is to create numerous scenarios, generate event trace and event flow diagrams from the scenarios, and then create state diagrams from the event diagrams. Although this *ad hoc* approach may be useful in determining necessary operations of objects and interactions between objects, the lack of explicit formal semantics makes the derived dynamic models inherently exempt from formal, automated analysis. Therefore we define a formal semantics for the dynamic model so that formal specifications can be derived for the reasoning and analysis at the intra-model and inter-model levels.

We have chosen LOTOS as the formal specification language to describe the dynamic models of OMT. The major motivations for selecting LOTOS as the target specification language are as follows.

- It has a sound mathematical foundation defined in terms of process algebras and algebraic specifications so that formal analysis can be conducted.
- It provides support for integrating algebraic specifications with process algebras.
- It supports concurrency.
- There are numerous support and analysis tools.
- The behavior specifications are executable thus enabling behavior simulation and requirements validation.
- It has been accepted as an ISO standard and has been successfully used for numerous industrial projects [64, 65, 66].

The remainder of this chapter is organized as follows. Section 5.1 discusses a preliminary formalization of the state diagrams. Section 5.2 gives the formalization of the dynamic models for individual objects. Section 5.3 shows how concurrent state diagrams are formalized. Section 5.4 discusses the integration of the dynamic model with two other models. Section 5.5 summaries the chapter.

5.1 Preliminary Formalization

In OMT, a state diagram is used to describe the aspects of a system that are concerned with time and change, flow of control, interactions, and sequencing of operations in a system of concurrently active objects. In general, this objective is ambitious. Accordingly, we restrict the dynamic model to describe only the (observable) behavior of a system. (Unless otherwise noted, behavior refers to observable behavior.)

Observable behavior. The observable behavior of an object, as the name implies, refers to its interactions with the environment and the sequences of interactions that

can take place. The interactions can be observed outside of the object; the environment can either be other objects or the system environment.

Definition-1: Observable Behavior: *Observable behavior is defined to be interactions (inter-object communication and object operations) between the object and its environment and the interaction pattern.*

Dynamic model. In most cases, interaction is a synonym for communication. Here, the interaction includes the inter-object communication and the action initiation and reaction of the objects. The inter-object communication is the communication between objects. Therefore, the behavior of an object consists of all its possible communication and operation sequences with the environment.

Definition-2: Dynamic Model: *The dynamic model formally specifies all the possible interaction patterns of an object; inter-object communication and object operations are considered interactions.*

Communication model. The communication mechanism is not explicitly addressed in OMT [3]. The state diagrams in the text imply a broadcasting communication mechanism. Figure 5.1 shows a typical Harel statechart (with a broadcasting communication mechanism [67]), whose notation is used in OMT and other process-control modeling systems, including RSML [68]. The condition, *in(G)*, under which event f triggers the state transition from B to C is valid in statechart notation. However, when the *statechart* is introduced into an object-oriented methodology without any modification, potential problems arise. Since A and D are two concurrent states,

they are actually the dynamic models of two independent objects in OMT. Because the class objects are encapsulated information, information is passed only by methods, the only inter-object communication mechanism. Obviously, condition "*D is in G*" is not visible to *A*, and therefore it is invalid to use dynamic models of OMT in exactly the same way as the original *statechart* is used [67].

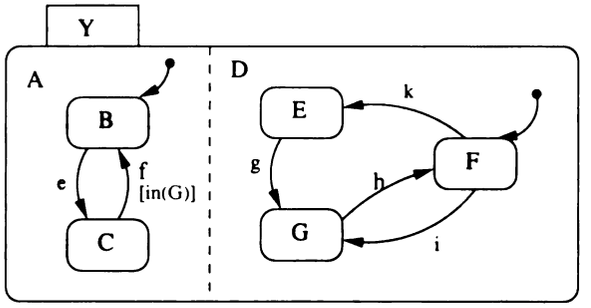

Figure 5.1: A typical statechart

With respect to information encapsulation, one key characteristic of any object-oriented methodology is that a change in an object is not visible to external objects. The only means that a change can be reflected to other objects is through inter-object communication. Accordingly, the inter-object communication is the only communication mechanism in the formalized model; the visible events of an object are carried by the inter-object communication external to the object.

Definition-3: Communication Model: *Inter-object communication is the only communication mechanism in the formal model.*

5.2 Formalizingstate diagrams of individual objects

In this section, we briefly introduce the formalization of the dynamic model in terms of a LOTOS specification. In order to facilitate our discussion and the presentation of the formalization, a state diagram is represented as a tuple $D = (S, \Sigma, \Lambda, \Delta, \Phi, C, \Xi, \psi, \lambda, \delta, \varphi, \zeta, s_0, c_0)$, where

- S is a finite set of states; s_0 is the initial state in D
- Σ is a finite set of events
- Λ is a set of variables
- Δ is a set of actions and activities, and $\Delta \subseteq \Gamma$ (Γ are operators defined in the corresponding algebraic specifications derived from object and functional models)
- Φ is a set of data sorts defined in the corresponding algebraic specification
- C is a set of guarding conditions; c_0 represents an empty condition that always holds (true)
- Ξ is a set of external events to be triggered
- $\psi : S \times \Sigma \times C \mapsto S$; ψ is a transition function mapping $S \times \Sigma \times C$ to S (ψ can be a partial function)
- $\lambda : S \mapsto \Delta$; λ is a function that associates a state with an activity (λ can be a partial function)
- $\delta : S \times \Sigma \times C \mapsto \Delta$; δ is a function that associates a transition with an action
- $\varphi : S \times \Sigma \mapsto 2^{\Lambda \times \Phi}$; φ is a function that maps a state and an event to a set of variable-data-sort pairs that are associated with the event
- $\zeta : S \times \Sigma \times C \mapsto \Xi$; ζ is a function that maps a transition to an external event to trigger

In the process of bringing formalisms into the dynamic model, we introduced some new notations in order to enhance the formality of the state diagrams (The

most recent release of UML [69, 70] indicates that its state diagram is also using a similar set of notations.)

Figure 5.2 gives a typical state diagram with an initial state and three state transitions. A label on an arc in the format of $e(d_1 : D_1, d_2 : D_2, ..., d_n : D_n)[c]/a(d_{x_1}, d_{x_2}, ..., d_{x_m})\hat{}O.e'(d_{y_1}, d_{y_2}, ..., d_{y_l})$ denotes an event e, data items $(d_1, d_2, ..., d_n)$ associated with the event e, a guarding condition c, the corresponding action a, and the event e' of object O to be triggered.

The remainder of this section introduces formalization rules for each component of a state diagram. The application of these formalization rules yields a portion of the formal specification of the corresponding state diagram.

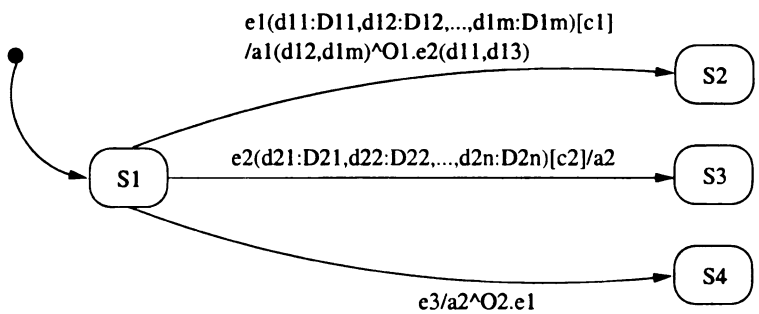

Figure 5.2: A typical state diagram

5.2.1 State

The derivation and justification of object states is one of the most difficult issues for dynamic modeling. The root of this difficulty is that the role of the states is not clearly defined within OMT. With the formal definition of the dynamic model given

in **Definition-2**, it is straightforward to ascertain that the basic function of the states is to describe the different interaction sequences.

Since the formalized object model requires that every leaf class contain a distinguished sort (type of interest), we model the behavior of a leaf class as an individual state diagram, whose states are determined by partitioning the values of the distinguished sort into classes.

DFR-1 The object states S are formalized as LOTOS processes.
For every state s, $s \in S$, create the following LOTOS expression

```
process s : noexit :=
endproc
```

Given that most transitions involve attribute value changes, the processes that formally specify the states are associated with a parameter of the distinguished sort.

DFR-2 Every process that formally specifies a state s is associated with a parameter x of the distinguished sort ϕ.
For every state s, $s \in S$, and distinguished sort $\phi \in \Phi$, create the following LOTOS expression

```
process s (x: φ): noexit :=
endproc
```

5.2.2 Communication model and event

An event is defined as inter-process communication because only the observable external behavior is well-defined and thus can be modeled. While this convention may seem restrictive, it is well-defined and exactly models the communication in the real world; it also achieves information encapsulation in the event definition.

For a give

observable be

Therefore, th

The gate

world and is

events of the

communicati

that correspo

DFR-3 The eve
commun
events, Σ
$[\Sigma \cup \Xi]$
For ever
$\Phi, \Sigma \subseteq$
pr
en

For those

formalize the

---

[1]The notat
and its type de

For a given object, the use of an operator determines whether it corresponds to observable behavior (that is, inter-object communication) or is internal to the object. Therefore, the inter-object communication is a subset of the object operators, Γ.

The gate construct in LOTOS is used to describe the interface to the external world and is able to accept and deliver data. It is straightforward to specify the events of the dynamic model as gates in LOTOS specifications. The inter-object communication (message passing) is modeled as an event arriving at a certain gate that corresponds to a specific operator.

DFR-3 The events of an object together with the events to be triggered are formalized as the inter-object communication external to the object, which is a subset of Γ (set of the object operators). The events, Σ, and events to be triggered, Ξ, in a state diagram D are specified as a formal gate list $[\Sigma \cup \Xi]$ for the processes in S.
For every state s, $s \in S$, distinguished sort ϕ, events Σ, and events to be triggered Ξ ($\phi \in \Phi, \Sigma \subseteq \Gamma$), create the following expression

```
process s [Σ ∪ Ξ](x: φ): noexit :=
endproc
```

For those events that are associated with arguments, the following rule is used to formalize the event definition.[1]

[1]The notations $x_i : a_i$ and (x_i, a_i) are used interchangeably, both of which represent a variable and its type declaration.

DFR-4

In

objec

the c

DFR-5

5.2

A c

Star

exp

Pos

DFR-4 Attribute pairs $x_i : a_i$ (data items) associated with an event e are formalized as *variable declarations* $?x_i : a_i$ associated with a LOTOS gate e.
For every state s and event e ($s \in S, e \in \Sigma$), distinguished sort ϕ, events Σ, and events to be triggered Ξ ($\phi \in \Phi, \Sigma \subseteq \Gamma$), if the partial function φ maps (s,e) to pairs $x_1 : a_1, x_2 : a_2, ..., x_n : a_n$, a non-empty subset of $\Lambda \times \Phi$ ($\{(x_1,a_1),(x_2,a_2),...,(x_n,a_n)\} \in 2^{\Lambda \times \Phi}$), create the following expression

```
process s [Σ ∪ Ξ](x: φ): noexit :=
  e ?x1 : a1, ?x2 : a2, ..., ?xn : an;
endproc
```

If the partial function φ maps (s,e) to an empty set (no associated attribute for event e), create the following expression

```
process s [Σ ∪ Ξ](x: φ): noexit :=
  e;
endproc
```

In some cases, a transition may be associated with a triggerable event of another object. This case is formalized as a value declaration of the data to be transferred at the corresponding gate.

DFR-5 An event to be triggered, e', is formalized as a value declaration ($!y_i$) at a LOTOS gate.
For every state s, event e, and condition c ($s \in S, e \in \Sigma, c \in C$), distinguished sort ϕ, events Σ, and events to be triggered Ξ ($\phi \in \Phi, \Sigma \subseteq \Gamma$), if the partial function δ maps (s,e,c) to a triggerable event e', create the following expression

```
process s [Σ ∪ Ξ](x: φ): noexit :=
  e' !y1, !y2, ..., !ym;
endproc,
```

where $\{y_1, y_2, ..., y_m\}$, a subset of $\{x_1, x_2, ..., x_n, x\}$, is specified in the corresponding state diagrams.

5.2.3 Condition

A condition, in the form of predicates, is used to describe circumstances under which a state transition can take place. We formalize guarding conditions as LOTOS guarded expressions in form of a predicate preceding a *behavior expression*, which becomes possible when the predicate holds.

DFR-6 A guard

For ever

Σ. and e

$a_1.x_2 : a$

non-em

proce

e ?

endpr

where x

A predica

of an operat

DFR-7 A new

Boolean

Given o

sort o.

sorts N

spec

type

op

endt

...

proc

...

endp

ends

5.2.4 A

The activiti

tions are co

a behavior s

algebraic sp

as operator

DFR-6 A guarding condition c is formalized as a guarded expression [c].

For every state s, event e, and condition c ($s \in S$, $e \in \Sigma$, $c \in C$), distinguished sort ϕ, events Σ, and events to trigger Ξ ($\phi \in \Phi, \Sigma \subseteq \Gamma$), if the partial function φ maps (s,e) to pairs $x_1 : a_1, x_2 : a_2, ..., x_n : a_n$, a non-empty subset of $\Lambda \times \Phi$ ($\{(x_1,a_1),(x_2,a_2),...,(x_n,a_n)\} \in 2^{\Lambda \times \Phi}$), a non-empty subset of sorts Φ ($\{a_1,a_2,...,a_n\} \in 2^{\Phi}$), and $c \neq c_0$, create the following expression

process s [$\Sigma \cup \Xi$](x: ϕ): noexit :=
  e ?x_1 : a_1, ?x_2 : a_2, ..., ?x_n : a_n; ([c] → ...)
endproc,

where $x_1, x_2, ..., x_n$ are variables to hold the attributes associated with the event.

A predicate that is newly introduced in a guarding condition is formalized in terms of an operation of *Boolean* sort in the algebraic specification section.

DFR-7 A newly introduced predicate in a guarding condition is formalized as an operation of type *Boolean* in the algebraic specification section.

Given object D, for every state s, event e, condition c ($s \in S$, $e \in \Sigma$, $c \in C$), distinguished sort ϕ, and predicate p introduced in condition c, if $x_1, x_2, ..., x_n$ are the arguments for p of sorts $X_1, X_2, ..., X_n$, create the following expression

specification D [$\Sigma \cup \Xi$](x: ϕ): noexit
typedef ϕ is Boolean
  opns
    p : $X_1, X_2, ..., X_n \longrightarrow$ Bool
endtype
...
process s [$\Sigma \cup \Xi$](x: ϕ): noexit :=
...
endproc
endspec

5.2.4 Action and activity

The activities and actions are defined as operators in the algebraic specification (actions are considered instantaneous; activities may occur over a period of time). Since a behavior specification in LOTOS can reference the operators that are defined in the algebraic specification part, we model the actions and activities in the state diagrams as operator references in the LOTOS specification. The difference between activities

labeled gra

termine. It

should be m

An actio

operator (de

DFR-8 An ac

corresp

For ev

maps t

$c \neq c_0$

expres

pro

en

where

diagr

With

activities

as "the r

DFR-9 A

nor

Fo

an

ma

Con

a notat

the ita

(labeled graphically as "`do: activity`") and actions is not straightforward to determine. It depends on the experience of the designer to determine which operators should be modeled as activities and which operators should be modeled as actions.

An action can be formalized in terms of a value declaration by a reference to an operator (defined in the algebraic specification) at the corresponding gate.

DFR-8 An action a is formalized in terms of a value declaration $!a$ by an operator reference at the corresponding gate e.
For every state s, event e, and condition c ($s \in S$, $e \in \Sigma$, $c \in C$), if the partial function φ maps tuple (s, e) to sorts $a_1, a_2, ..., a_n$ (a non-empty subset of sorts Φ ($\{a_1, a_2, ..., a_n\} \in 2^{\Phi}$)), $c \neq c_0$, and the partial function δ maps tuple (s, e) to an action a ($a \in \Delta$), create the following expression

```
process s [Σ ∪ Ξ](x: φ): noexit :=
  e ?x1 : a1, ?x2 : a2, ..., ?xn : an;
    ([c] → e !a(y1, y2, ..., ym))
endproc,
```

where $\{y_1, y_2, ..., y_m\}$, a subset of $\{x_1, x_2, ..., x_n, x\}$, is specified in the corresponding state diagrams.

With the built-in *internal* event, i, predefined in LOTOS, we are able to specify the activities as value declarations preceded by an *internal* event, which can be interpreted as "the return of a result takes time to complete."

DFR-9 An activity a is formalized as a value declaration $!a$ at the corresponding gate e preceded by a nondeterministic *internal* event i.
For every s, $s \in S$, distinguished sort ϕ, events Σ, events to be triggered Ξ ($\phi \in \Phi, \Sigma \subseteq \Gamma$), and arguements $y_1, y_2, ..., y_n$ of sorts $Y_1, Y_2, ..., Y_n$ for activity a, if the partial function λ maps state s to an activity a ($a \in \Delta$), create the following expression

```
process s [Σ ∪ Ξ](x: φ, y1 : Y1, y2 : Y2, ..., yn : Yn): noexit :=
  i; e !a(y1, y2, ..., yn);
endproc
```

Consider the state diagram of the *Compressor* object shown in Figure 5.3. As a notation convention in the discussion of diagrams and specifications for diagrams, the italic font will be used to denote diagram components and the courier font will

denotes

tion fr

the *co*

compl

as acti

done e

the co

result

denotes specification components. The *compress* and *decompress* operations transition from state *idle* to *compression* and then to *decompression*, respectively. Since the *compress* and *decompress* operations, which may take some duration of time to complete, cannot be considered instantaneous, they should no longer be modeled as actions, but, instead, considered as activities in the corresponding states. The *done* event, (denoted by an unlabeled arrow leaving an activity state), representing the completion of compression or decompression, activates the actions to return the results and trigger state transitions.

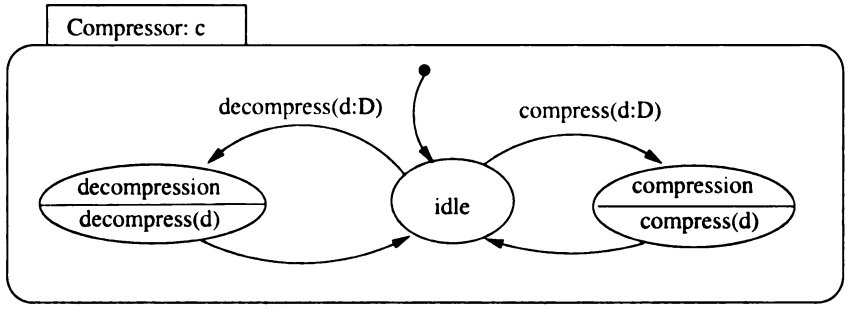

Figure 5.3: The state diagram for the *Compressor* object

Figure 5.4 contains a complete LOTOS specification for the *Compressor* class based on the formalization rules presented thus far (the exception is that the construction of specifications for transitions which is discussed next). The specification describes both the abstract data type and the behavior of a *Compressor* object. The first part of the specification describes the abstract data type of the object class in terms of ACT ONE; the second part captures the object behavior in terms of process algebras. The expression "`i;compress!compress(data)`" exactly models the *compress* activity in the state diagram. The behavior specificatoin captures:

- The *Compressor* is initially in `idle` state.
- A `decompress` request triggers the transition from `idle` state to `decompression` state.
- In the !decompression! state, the *Compressor* performs the decompression activity.
- Upon the completion of the decompression activity, the *Compressor* returns the decompresed data and transitions back to `idle` state.
- A `compress` request triggers the transition from `idle` state to `compression` state.
- In the !compression! state, the *Compressor* performs the compression activity.
- Upon the completion of the compression activity, the *Compressor* returns the compresed data and transitions back to `idle` state.

5.2.5 Transitions

The change of state caused by an event is a *transition.* Although an event, inter-object communication, or internal event, is the cause of a transition, the transition is actually triggered by the corresponding action or activity.

However, a state transition does not necessarily imply a state change. Only those transitions whose starting state is different from its ending state can cause a state change. Since only *modifier* operators can trigger a state change, the state transition that is fired by a *non-modifier* does not change state.

DFR-10 **A transition can cause a state change if and only if the action that corresponds to a transition is defined as a modifier operator.**
For every state s and event e ($s \in S, e \in \Sigma$), if the partial function ψ does not map tuple (s,e) to state s ($s \neq \psi(s,e)$), then the partial function δ maps tuple (s,e) to a modifier operator ($\delta(s,e) \in \Omega$); if the partial function δ maps tuple (s,e) to a non-modifier operator ($\delta(s,e) \in \Theta$), then the partial function ψ maps tuple (s,e) to state s ($s = \psi(s,e)$).

---

specifi
library
Nat
endlib
type (
sort
Co
opn
un
co
de
si
eqn
fo
o
o
endty
behav
Idle
where
pro
co
de
enc
pro
i
enc
pro
i
enc
endsp

```
specification Compressor [compress, decompress] (c: Compressor): noexit
library
   NaturalNumber
endlib
type Compression  is NaturalNumber
  sorts
   Compressor, Data
  opns
   undef_data  : -> Data
   compress : Data -> Data
   decompress  : Data -> Data
   sizeof : Data -> Nat
  eqns
    forall x: Data      ofsort Nat
     sizeof(undef_data) = 0;
    ofsort Bool
     sizeof(x) = 0 => x = undef_data;
     sizeof(x) ge sizeof(compress(x)) = true;
     sizeof(x) le sizeof(decompress(x)) = true;
    ofsort Data
     decompress(compress(x)) = x;
endtype
behavior
 Idle_State [compress, decompress] (c)
where
  process Idle_State[compress,decompress]: (c: Compressor)
                                          noexit:=
   compress ?data:Data; Compress_State[compress,decompress] (c, data)
   []
   decompress ?data:Data; Decompress_State[compress,decompress] (c, data)
  endproc
  process Compress_State[compress,decompress](c:Compressor,data:Data): noexit:=
    i; compress !compress(data);
                Idle_State[compress,decompress] (c)
  endproc
  process Decompress_State[compress,decompress](c:Compressor,data:Data): noexit:=
    i; decompress !decompress(data); Idle_State[compress,decompress] (c)
  endproc
endspec
```

Figure 5.4: A *Compressor* class and its behavior specified in LOTOS

The state transitions in the dynamic model of OMT are formalized in terms of LOTOS process instantiations.

DFR-11 The state transitions are formalized as process instantiations.

For every state s and event e ($s \in S, e \in \Sigma$), distinguished sort ϕ, events Σ, and events to be triggered Ξ ($\phi \in \Phi, \Sigma \subseteq \Gamma$), if the partial function φ maps (s,e) to pairs $x_1 : a_1, x_2 : a_2, ..., x_n : a_n$, a non-empty subset of $\Lambda \times \Phi$ ($\{(x_1,a_1),(x_2,a_2),...,(x_n,a_n)\} \in 2^{\Lambda \times \Phi}$), the partial function δ maps tuple (s,e) to an action a ($a \in \Delta$), the partial function ψ maps tuple (s,e) to a state s' ($s' \in S$), and a is of sort ϕ, create the following expression

```
process s [Σ ∪ Ξ](x: φ): noexit :=
  e ?x1 : a1 ?x2 : a2 ... ?xn : an;
  s'[Σ ∪ Ξ](a(y1, y2, ..., ym))
endproc
```

if a is not of sort ϕ, create the following expression

```
process s [Σ ∪ Ξ](x: φ): noexit :=
  e ?x1 : a1 ?x2 : a2 ... ?xn : an;
  e !a(y1, y2, ..., ym);s'[Σ ∪ Ξ](x)
endproc
```

In cases, where there are multiple state transitions from a single state, the multiple transitions are specified as multiple process instantiations composed by a *choice* operator ("[]").

DFR-12 Multiple state transitions from a single state are formalized as process instantiations composed by a *choice* operator (adding more transitions may add more choice operators.

For every state s, events e_1 and e_2 ($s \in S$, $e_1, e_2 \in \Sigma$), distinguished sort ϕ, events Σ, and events to be triggered Ξ ($\phi \in \Phi, \Sigma \subseteq \Gamma$), if $e_1 \neq e_2$, the partial function φ maps tuple (s,e_1) to $\{x_{11} : a_{11}, x_{12} : a_{12}, ..., x_{1n} : a_{1n}\}$, (s,e_2) to $\{x_{21} : a_{21}, x_{22} : a_{22}, ..., x_{2j} : a_{2j}\}$ (where $\{(x_{11},a_{11}),)x_{12},a_{12}),...,(x_{1n},a_{1n})\},\{(x_{21},a_{21}),(x_{22},a_{22}),...,(x_{2j},a_{2j})\} \in 2^{\Lambda \times \Phi}$), the partial function δ maps tuple (s,e_1) to a_1, (s,e_2) to a_2 ($a_1, a_2 \in \Delta$), and the partial function ψ maps tuple (s,e_1) to $s_{1'}$, (s,e_2) to $s_{2'}$, create the following expression*

```
process s [Σ ∪ Ξ](x: φ): noexit :=
  e1 ?x11 : a11 ?x12 : a12 ... ?x1n : a1n;
  e1 !a1(y11 y12 ... y1m); s1'[Σ ∪ Ξ](x)
[]
  e2 ?x21 : a21 ?x22 : a22 ... ?x2j : a2j;
  e2 !a2(y21 y22 ... y2k); s2'[Σ ∪ Ξ](x)
endproc
```

*The expression may differ according to DFR-10 depending on the sort of a_i.

5.2.6

Given

forma

DFR-1

An e

part

speci

In

and

inser

exter

"cou

obje

5.2.6 State diagram

Given the formalization rules for the components of a state diagram, we are able to formalize the diagram in terms of its components.

DFR-13 A state diagram can be formalized as either a LOTOS *specification* or a LOTOS *process definition.*
For state diagram $D = (S, \Sigma, \Lambda, \Delta, \Phi, C, \Xi, \psi, \lambda, \delta, \varphi, \zeta, s_0, c_0)$, we can have either

process D [$\Sigma \cup \Xi$] (x: ϕ) :**noexit** :=
  s_0 [$\Sigma \cup \Xi$] (x)
where
  ⟨ algebraic specification ⟩
  ⟨ process definitions ⟩
endspec
OR
specification D [$\Sigma \cup \Xi$] (x: ϕ) :**noexit**
  ⟨ algebraic specification ⟩
behaviour
  s_0 [$\Sigma \cup \Xi$] (x)
where
  ⟨ process definitions ⟩
endspec,

where the first format is used if the process will be composed within other specifications to form more complex behavior specifications; the second format is used if the specification is a top level specification.

An example. Figure 5.5 gives a dynamic model for *Storage* that corresponds to a part of the object diagram given in Figure 2.13. Its corresponding class and behavior specifications are given in Figures 5.6 and 5.7.

In Figure 5.5, the *empty* and *non_empty* states are formalized as the **empty_state** and **non_empty_state** processes, respectively in Figures 5.6 and 5.7. The events, *insert*, *retrieve*, and *delete*, are formalized as the three gates visible to the external objects. The guarding conditions are "**count(s) gt Succ(0)**" and "**count(s) eq Succ(0)**". Since *empty_state* is the default state that a *Storage* object enters when it is initially created, the behavior of a *Storage* object is specified

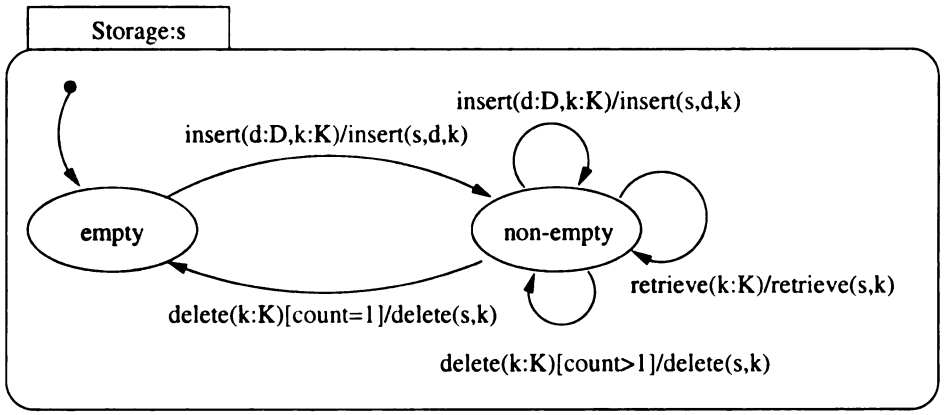

Figure 5.5: The state diagram for *Storage*

```
specification Storage[insert,retrieve,delete] (s: Storage): noexit
library
  Boolean, NaturalNumber
endlib
  sorts
      Storage, Data, Key
  opns
      new       : -> Storage
      undef_Data: -> Data
      undef_Key : -> Key
      count     : Storage -> Nat
      retrieve  : Storage, Data, Key -> Data
      delete    : Storage, Key -> Storage
      _ eq _    : Data, Data -> Bool
      _ eq _    : Key, Key -> Bool

  eqns
     forall d: Data, k, k1, k2: Key, s: Storage
    ofsort Nat
      count(new) = 0;
      count(insert(s,d,k)) = Succ(0) + count(s);
    ofsort Data
      retrieveP(new,d,k) = undef_Data;
      (k1 eq k2) => retrieve (insert(s,d,k2),k1)= d;
      not(k1 eq k2) => retrieve (insert(s,d,k2),k1) = retrieve (s, k1);
    ofsort Storage
      delete(new,k) = new;
      (k1 eq k2) => delete(insert(s,d,k2),k1)=s;
      not(k1 eq k2) => delete(insert(s,d,k2),k1)= insert(delete(s,k1),d,k2);
endtype
```

Figure 5.6: The specification (in Full LOTOS) of the *Storage* (1)

```
behavior
  empty_state[insert,retrieve,delete] (s)
where
   process empty_state[insert,retrieve,delete](s: Storage): noexit:=
     insert? d : Data ? k : Key;
         non_empty_state[insert,retrieve,delete](insert(s,d,k))
   endproc
   process non_empty_state[insert,retrieve,delete](s: Storage): noexit:=
     retrieve ?k:Key; retrieve !retrieve(s,k);
         non_empty_state[insert,retrieve,delete](s)
     []
     insert ?d:Data ?k:Key;
         non_empty_state[insert,retrieve,delete](insert(s,d,k))
     []
     delete ?k:Key; (
         [count(s) gt Succ(0)] ->
             non_empty_state[insert,retrieve,delete](delete(s,k))
       []
         [count(s) eq Succ(0)] ->
             empty_state[insert,retrieve,delete](delete(s,k))
     )
   endproc
endspec
```

Figure 5.7: The specification (in Full LOTOS) of the *Storage* (2)

as "`empty_state[insert,retrieve,delete](new)`." This specification enables a *Storage* object to enter the `empty` state when it is first initialized. The transition from `empty` to `non_empty` triggered by the `insert` event is formalized in the `empty_state` process as:

```
insert?d:D?k:K;non_empty_state[insert, retrieve,delete](insert(s,d,k)).
```

The specification "`insert ?d:D ?k:K;`" indicates that the process is to wait for an event to occur at the `insert` gate. In addition, two data values of type `D` and `K` must also be associated with the upcoming event at gate `insert`.

gat

nor

whe

nor

nor

S. w

a pr

the

stor

that

insta

obje

5.2.

So fa

objec

comp

Once an **insert** event with the two required data values occurs at the **insert** gate, the **insert** action is performed and the transition from the **empty** state to the **non_empty** state is triggered. The transition is formalized as the process instantiation

```
non_empty_state[insert,retrieve,delete](insert(s,d,k)),
```

where "**insert(s,d,k)**" executes the **insert** action, and the instantiation of the **non_empty_state** process makes the object leave the **empty** state and enter the **non_empty** state.

In the specifications of both **empty_state** and **non_empty_state** processes, sort **S**, which includes all the data values relevant to a *Storage* object, is formalized as a process parameter. The "**insert(s,d,k)**" operation of sort **S**, which formalizes the **insert** action, inserts the newly received data associated with its key into the storage space and thereby derives the updated storage. The result of this action is that the storage is no longer empty. Consequently, the **non_empty_state** process is instantiated with parameter "**insert(s,d,k))**" that represents a non-empty storage object.

5.2.7 Summary.

So far, we have addressed the formalization of the state diagram for an individual object. Objects that consist of an aggregation of objects are modeled by *parallel composition* constructs, addressed in the following discussion.

5.3

A typ

is imp

ered c

state

Th

pose

opera

•

•

•

In

the dy

transi

diagra

the ob

Th

comple

behavi

followi

object

5.3 Concurrent State Formalization

A typical system is usually composed of concurrent subsystems. In a system that is implemented by an object-oriented approach, the objects can always be considered concurrent. In OMT, this concurrency is modeled as *aggregate concurrency*; in *statecharts*, it is modeled with *AND* states.

The *parallel composition* operators of LOTOS provide sufficient semantics to compose parallel and concurrent processes. There are three types of *parallel composition* operators:

- *Full interleaving*: No communication and synchronization between processes.
- *Full synchronization*: Every action must be synchronized.
- *Normal synchronization*: Only a select set of actions are synchronized.

In the previous sections, we have shown that the formalization of the object and the dynamic models are inter-related by *attributes*, *operations*, *events*, *gates*, and *state transitions*. Similarly, the formalization of the *concurrent composition* in the state diagram is related to the *aggregation* associations that are identified and specified in the object diagrams.

Theoretically, all the objects in a system are concurrent. In order to manage large, complex systems, we advocate a hierarchical approach to modeling the concurrent behavior that follows the system structure described in the object diagrams. The following rules describe the relationship between the aggregation relationship in the object model and the synchronization in the dynamic model.

Sync
curre
synch
in te
servi
can
conc
tual
The
thro
I
star
serv
unde
still
of t
each
bora

DFR-14 Every *aggregate* object is responsible for composing the behavior of its concurrent *aggregation* objects to form one behavior specification.

DFR-15 The behavior of every *aggregation* object must be composed by and only by its *aggregate* object.

DFR-16 Only when there is an association between two *aggregation* objects, can the *aggregate* object model the corresponding synchronization between the two objects.

Synchronization mechanism and notations. The synchronization between concurrent object dynamic models are specified by name binding. If two dynamic models, synchronized in a state diagram, share a common service, which is described as a gate in terms of LOTOS, the two dynamic models are synchronized through the shared service. The services (gates) modeled in the dynamic models for individual objects can be considered as parameterized services. When an object is composed with other concurrent objects in parallel, the names of its services can be substituted by *actual names* in the composition state diagram, such as the one shown in Figure 5.9. The dynamic models that share common actual names of services are synchronized through the shared services.

In OMT, dashed lines are used to separate concurrent dynamic models in a given state diagram. If two concurrent dynamic models in a state diagram share no common service, they are composed in terms of interleaved LOTOS process algebras. However, under some circumstances, even if two dynamic models share common services, we still want to compose them as interleaving dynamic models. This happens when both of the objects use the same actual service to synchronize with objects other than each other. A new notation is introduced to represent the forced interleaving. The notation has three parallel lines drawn in vertical to the dashed line that separates

the inter

dynamic

F

The

that only

the interleaving dynamic models. A sample diagram is given in Figure 5.8, where dynamic models d_2 and d_3 are interleaved.

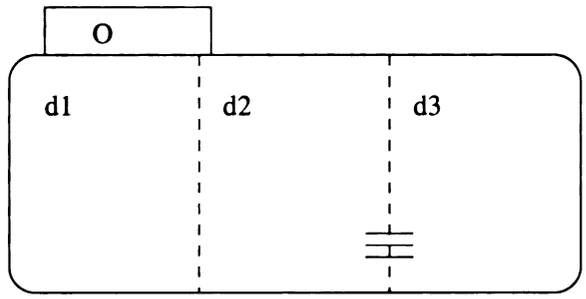

Figure 5.8: A sample state diagram with interleaving dynamic models

The following rules use LOTOS' normal synchronization construct [21] indicating that only a selected set of gates are synchronized.

DFR-17

DFR-18

DFR-17 In a state diagram that composes dynamic models of concurrent objects, the dynamic behaviors are synchronized through shared services.
The state diagram $D = (S, \Sigma, \Lambda, \Delta, \Phi, C, \Xi, \psi, \lambda, \delta, \varphi, \zeta, s_0, c_0)$ composed of
$D_1 = (S_1, \Sigma_1, \Lambda_1, \Delta_1, \Phi_1, C_1, \Xi_1, \psi_1, \lambda_1, \delta_1, \varphi_1, \zeta_1, s_{10}, c_0)$,
$D_2 = (S_2, \Sigma_2, \Lambda_2, \Delta_2, \Phi_2, C_2, \Xi_2, \psi_2, \lambda_2, \delta_2, \varphi_2, \zeta_2, s_{20}, c_0)$, ...,
and $D_n = (S_n, \Sigma_n, \Lambda_n, \Delta_n, \Phi_n, C_n, \Xi_n, \psi_n, \lambda_n, \delta_n, \varphi_n, \zeta_n, s_{n0}, c_0)$ is formalized as:

specification D[$\Sigma_1 \cup \Sigma_2 \cup .... \cup \Sigma_n \cup \Sigma$] (x:$\phi$, x_1:ϕ_1, x_2 ϕ_2, . . . , x_n:ϕ_n): **noexit**
<algebraic specification>
behaviour
s_0 [$\Sigma \cup \Xi$] (x)
|[Υ_1]|
(D_1[$\Sigma_1 \cup \Xi_1$] (x_1)
|[Υ_2]|
(D_2[$\Sigma_2 \cup \Xi_2$] (x_2)
|[Υ_3]|
(.....
|[Υ_n]|
D_n[$\Sigma_n \cup \Xi_n$] (x_n)
)))
where
<process definitions>
endspec,
where $\Upsilon_1, \Upsilon_2, ..., \Upsilon_n$ are shared services.

DFR-18 In a state diagram, the interleaving dynamic models are grouped together in terms of LOTOS interleaving process algebras.
If state diagrams $D_1 = (S_1, \Sigma_1, \Lambda_1, \Delta_1, \Phi_1, C_1, \Xi_1, \psi_1, \lambda_1, \delta_1, \varphi_1, \zeta_1, s_{10}, c_0)$,
$D_2 = (S_2, \Sigma_2, \Lambda_2, \Delta_2, \Phi_2, C_2, \Xi_2, \psi_2, \lambda_2, \delta_2, \varphi_2, \zeta_2, s_{20}, c_0)$, ...,
and $D_n = (S_n, \Sigma_n, \Lambda_n, \Delta_n, \Phi_n, C_n, \Xi_n, \psi_n, \lambda_n, \delta_n, \varphi_n, \zeta_n, s_{n0}, c_0)$ are interleaving dynamic models in a given state diagram, the following specification shall be generated:
behaviour

```
......
(
    D1[Σ1 ∪ Ξ1] (x1)
    |||
    D2[Σ2 ∪ Ξ2] (x2)
    |||
    .....
    |||
    Dn[Σn ∪ Ξn] (xn)
)
......
```

In the object diagram shown in Figure 2.13, a *Disk Manager* is an aggregation of *Storage* and a *Compressor* (notice the *one-to-one* relationship between aggregate and aggregation objects specified both in the diagram and in its corresponding formal specification). Hence, the possible synchronizations and communication are between

Dis

rent

indi

syn

ope

sor

the

Sto

per

out

con

gat

del

syn

syn

Di

con

Any

oth

spe

Disk Manager and its two components. Figure 5.9 contains the aggregated concurrent state diagram for the *Disk Manager* class. The dashed lines indicate that the individual state diagrams are active in parallel. The individual dynamic models are synchronized through shared services. The state diagram captures that: (1) the *input* operation is implemented in terms of the *compress* and *insert* operations of *Compressor* and *Storage*, respective, and (2) the *output* operation is implemented in terms of the *decompress* and *retrieve* operations of *Compressor* and *Storage*, respective.

Figure 5.10 gives the major parts of the specification with the descriptions of *Storage* and *Compressor* omitted (a complete LOTOS specification is given in Appendix D).

The expression "`Storage[ins,ret,del]|[ins,ret]|Disk_Manager[input, output,com,dec,ins,ret]`" specifies that processes *Disk_Manager* and *Storage* communicate and synchronize through actual gates *ins* and *ret*. *Storage*'s formal gates, *insert*, *retrieve*, and *delete* are substituted by the actual gates *ins*, *ret*, and *del*, respectively. These two processes, in fact, model the communication and synchronization between their corresponding objects. The communication and synchronization between *Disk Manager* and *Compressor* are similarly modeled.

Since the services *com*, *dec*, *ins*, *ret*, and *del* are actually internal services to object *Disk Manager*, they can be hidden from the external objects by using the *gate hiding* construct provided by LOTOS. Thus only gates *input* and *output* are externally visible. Any design decision or implementation change to the *Disk_Manager* will not affect other objects that are externally related to *Disk_Manager*, as long as the behavior specification of implementation meets the *Disk_Manager*'s original individual behavior

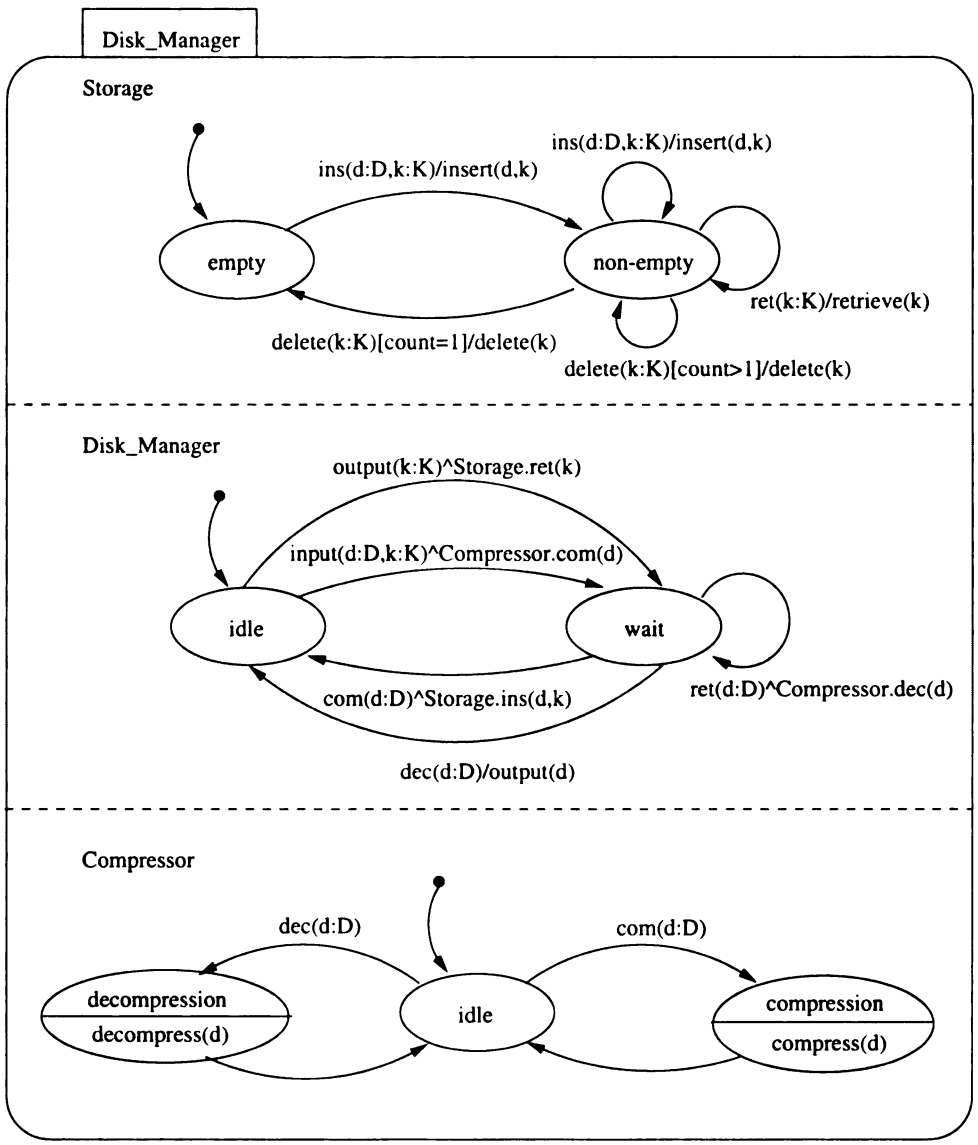

Figure 5.9: The aggregation concurrent state diagram for *Disk Manager*

```
specificati

type Disk
endtype
behavior

  (Storag
  |[ins,r
  Disk_Ma
  |[com,d
  Compres
where
  process
   idle[i
  endproc
  process
   input
   []
   output
  endproc
  process
   com ?d
   []
   ret ?d
   []
   dec ?d
  endproc
  process

  process

endspec
```

Fi

specificati

internal g

denoted b

In orde

models of

tiated. Ru

```
specification Disk_Manager[input,output,com,dec,ins,ret,delete]
                                (ds: Disk_Manager, s: Storage, c: Compressor)
type Disk_Manager_REF  is Disk_Manager, Storage, Compressor
endtype
behavior

  (Storage[ins,ret,del] (s)
  |[ins,ret]|
  Disk_Manager[input,output,com,dec,ins,ret,del] (ds))
  |[com,dec]|
  Compression[com,dec] (c)
where
  process Disk_Manager [input,output,com,dec,ins,ret,delete]: noexit:=
   idle[input,output,com,dec,ins,ret,delete] (ds)
  endproc
  process idle[input,output,com,dec,ins,ret,delete] (ds: Disk_Manager): noexit:=
   input ?d:D ?k:K; com !d !k; wait[input,output,com,dec,ins,ret,delete](ds, k)
   []
   output ?k:K; ret !k; wait[input,output,com,dec,ins,ret,delete](ds, k)
  endproc
  process wait[input,output,com,dec,ins,ret,delete](ds:Disk_Manager,k:K): noexit:=
   com ?d:D; ins !d !k; idle[input,output,com,dec,ins,ret,delete] (ds)
   []
   ret ?d:D; dec !d; wait[input,output,com,dec,ins,ret,delete](ds, k)
   []
   dec ?d:D; output !d; idle[input,output,com,dec,ins,ret,delete] (ds)
  endproc
  process Storage[insert,retrieve,delete] (s: Storage): noexit :=
                  ..... Omitted .....
  process Compressor[compress,decompress] (c: Compressor): noexit :=
                  ..... Omitted .....
endspec
```

Figure 5.10: The LOTOS specification for a simplified *Disk_Manager*

specification that only contains services input and output. Figure 5.11 shows how the internal gates *com*, *dec*, *ins*, *ret*, and *del* are hidden in the LOTOS specification, denoted by the **hide** keyword.

In order to perform analysis on the state diagrams that compose the dynamic models of concurrent objects, the parameters of distinguished sorts may be instantiated. Rules **DFR-19** and **DFR-20** formalizes the constants, newly introduced by

```
specification Disk_Manager[input,output] (ds: Disk_Manager, s: Storage, c: Compressor)
type Disk_Manager_REF is Disk_Manager, Storage, Compressor
endtype
behavior
hide com,dec,ins,ret,delete in
  (Storage[ins,ret,del] (s)
  |[ins,ret]|
  Disk_Manager[input,output,com,dec,ins,ret,del] (ds))
  |[com,dec]|
  Compression[com,dec] (c)
where
......
endspec
```

Figure 5.11: Specification for a simplified *Disk_Manager* with hidden internal gates

the instantiation of parameters of distinguished sort, in terms of LOTOS algebraic specification. The formalization rules introduce a *nullary* operation for each of the newly introduced constant and use equations to specify the equality among the constants. Specifically, **DFR-19** introduces equations asserting that a constant is equal to itself and every two constants of the same sort are not equal.

DFR-19 If c is a constant, of sort C, introduced by the instantiation of parameter x of distinguished sort O for object O in a state diagram, then create the following expression

type O is
  sorts
    .....
  opns
    $c : \longrightarrow C$
  ....
endtype

DFR-20 For all constants $c_1, c_2, ..., c_n$ of sort C introduced in object O, create the following expression

type O is Boolean
  sorts
    .....
  opns
    .....
  eqns
    ofsort C
      c_1 eq c_1 = True;
      c_2 eq c_2 = True;
      .....
      c_3 eq c_3 = True;
      c_1 eq c_2 = False;
      c_1 eq c_3 = False;
      .....
      c_1 eq c_n = False;
      .....
      c_{n-1} eq c_n = False;
endtype

For instance, the state diagram in Figure 5.12 introduces constants *storage*, *dm01*, and *zip* for the *Storage*, *Disk Manager*, *Compressor* objects, respectively. The corresponding algebraic specification generated to describe the newly introduced constants and their relationships is given in Figure 5.13.

5.4 Integration with Other OMT Models

We have, in fact, implicitly addressed the integration of the object, functional, and dynamic models through the discussion of the formalization of the dynamic model. The LOTOS specification in Figures 5.4, 5.6, and 5.7 contain two typical examples

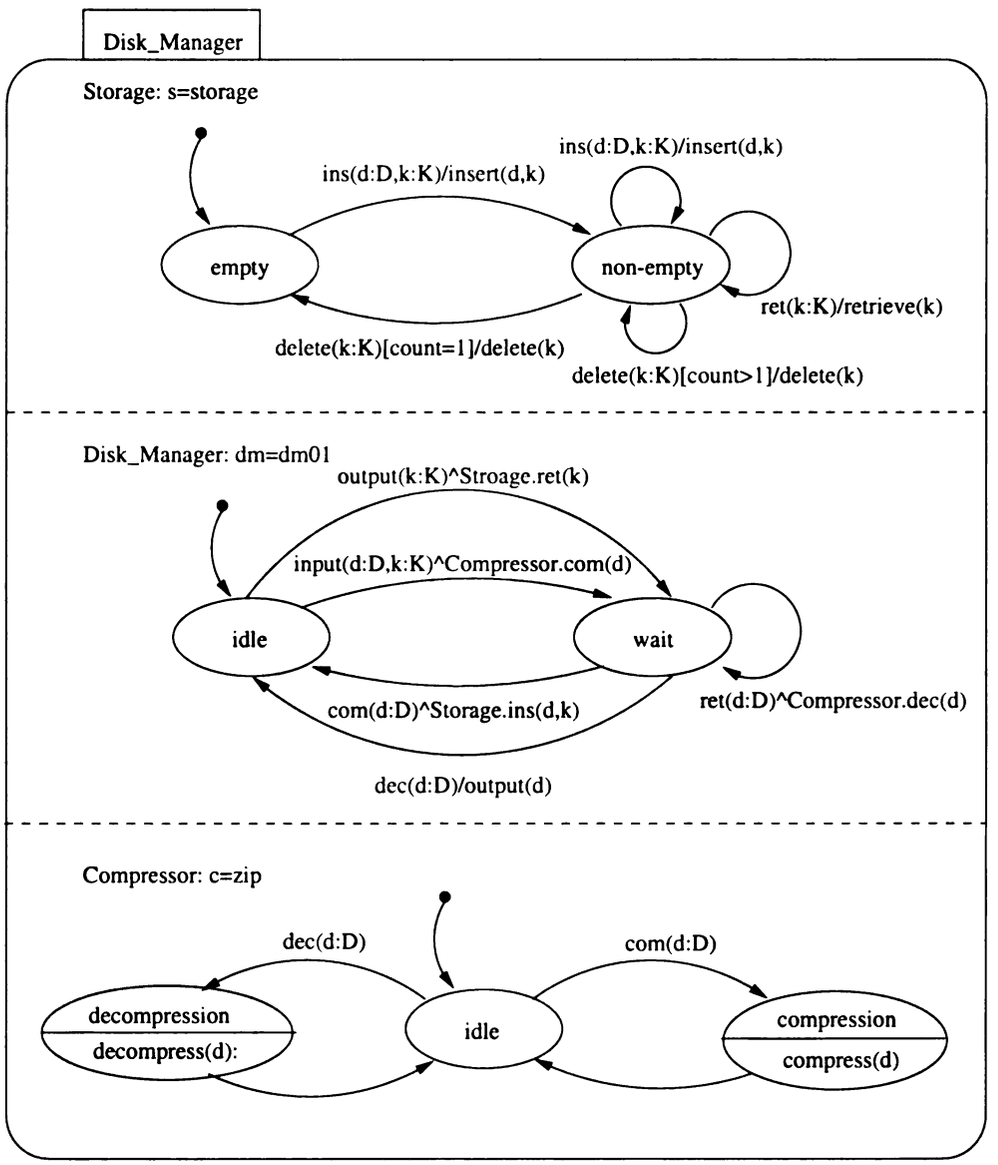

Figure 5.12: Sample state diagram with instantiated parameters of distinguished sorts

```
specification Disk_Manager[input,output,com,dec,ins,ret,delete] :noexit
type Disk_Manager_REF is Disk_Manager, Storage, Compressor
    opns
       storage : -> Storage
       dm01 : -> Disk_Manager
       zip : Compressor
    eqns
       ofsort Bool
           storage eq storage = True;
           undef_Storage eq undef_Storage = True;
           storage eq undef_Storage = False;
           undef_Storage eq storage = False;

           dm01 eq dm01 = True;
           undef_Disk_Manager eq undef_Disk_Manager = True;
           dm01 eq undef_Disk_Manager = False;
           undef_Disk_Manager eq dm01 = False;

           zip eq zip = True;
           undef_Compressor eq undef_Compressor = True;
           zip eq undef_Compressor = False;
           undef_Compressor eq zip = False;
endtype
behavior
    hide com, dec, ins, ret, del in
       (Storage[ins,ret,del] (storage)
       |[ins,ret]|
       Disk_Manager[input,output,com,dec,ins,ret,del] (dm01))
       |[com,dec]|
       Compression[com,dec] (zip)
where
   ......
endspec
```

Figure 5.13: LOTOS specification of the sample state diagram

that illustrate the integration. Although the formalization of the functional model will be addressed in Chapter 6, the services and operators whose availability we have assumed and discussed intensively in this chapter are both derived from functional models. The object and functional models together contribute to the formation of the algebraic spefication in our formalization rules.

In general, the use of LOTOS does not automatically lead to an integration of the formal specifications of the three models, though its constructs allow algebraic specifications and process algebras to share common language primitives thus providing a possibility to integrate the two types of specifications. In our formalization, the integration of the three models is achieved through explicitly imposing certain relationships upon the constructs of algebraic specifications and process algebras. The integration is three-fold. First, the dynamic model is derived in the context of object models. This approach to modeling provides a degree of integration between the two models. Second, the integration is achieved in terms of the formalization of individual state diagrams. The integration includes:

- The distinguished sort in algebraic specifications serves as state process parameters in process algebras.
- Operators defined in algebraic specifications serve as actions and activities in state diagrams.
- The data types used in process algebras are defined in algebraic specifications.
- Services (externally accessible operators) defined in algebraic specifications serve as gate lists for process algebras.
- The parameters associated with operators serve as attributes associated with events.

obj

5.

In

ON

and

pos

intu

ena

as t

two

Third, the integration is achieved by composing the dynamic models of concurrent objects hierarchically according to the system structure specified in the object model.

5.5 Summary

In this chapter, we introduced a set of formalization rules for the dynamic model of OMT. The formalization enables the precise specification of the behavior of objects and the simulation of system behavior through executable specifications. The proposed formalization also integrates the object, functional, and dynamic models in an intuitive fashion that facilitates analysis and design. Furthermore, the integration enables system developers to check both intra- and inter-model consistency, as well as to conduct system development in a stepwise fashion, where consistency between two adjacent levels of refinement can be checked.

C

F

Th

sen

mo

vid

in

inf

ob

mo

ter

DF

me

ph

con

Chapter 6

Functional Model Formalization

The data flow diagram (DFD) was first introduced in structured design for representing data and the processes that transform data [71, 72]. Since the DFD visually models a system in terms of input/output data flows and processes as well as provides a means to systematically decompose a system, it has been successfully used in the development of transaction systems, whose primary objective is to process information [6].

Based on notions such as modularity, abstraction, encapsulation, and reuse, object-oriented development methods have advantages over structured methods in modeling, maintenance, and reuse. In order to present the functional aspect of systems and objects, many object-oriented approaches have attempted to incorporate DFDs into the object-orientation paradigm [3]. However, no well-defined approach or method has been developed thus far. The difficulty is due to the incompatible philosophies of structured and object-oriented methods, particularly with respect to their contradictory system decomposition strategies. The structured approaches decom-

pose systems according to functionality while the object-based approaches conduct decomposition according to the static object structure. Therefore, how to use and integrate the DFD, as a powerful and useful tool that describes the functionalities of systems and objects, into the object-oriented methods warrants further investigation.

Given the fundamental philosophical differences, there have been two strategies for handling the situation. One strategy advocates a concurrent development of DFDs and object models, followed by an explicit association of the processes to objects [3]. Since this approach does not deal with the fundamental, philosophical difference between the structured and object-oriented methods, the association of processes with object methods at the end of the design phase is nearly impossible. The other strategy warns, and even avoids the use of DFDs, for fear that it will irrevocably bias subsequent object modeling towards a function-orientation [7, 73, 74].

Considering the fundamental contradiction between structured and object-oriented methods and the successful application of DFDs in the history of software development, neither approach discussed above is satisfactory. This chapter attempts to integrate DFDs into object-oriented methods without violating the object-orientation philosophy [75]. The proposed approach extends DFDs and introduces formal semantics to functional models in terms of algebraic specifications. The formalization of the DFD is developed in the context of the formalizations for the object model that provides the static structure of the objects and the dynamic model that describes the behavior, thereby enabling integration in terms of their underlying formal semantics.

The remainder of the chapter is organized as follows. Section 6.1 gives a brief overview of the background of DFDs. Sections 6.2 and 6.3 show how the DFD is

extended and integrated into object-oriented methods. Section 6.4 discusses the formalization of functional models in terms of algebraic specifications. Section 6.5 summarizes the integration of functional model with object and dynamic models. Section 6.6 gives the summary of this chapte.

6.1 The Data Flow Diagram (DFD)

Notations for DFDs. There are numerous variations of data flow diagrams; Figure 6.1 shows a typical conventional DFD drawn with the primitive notations, where a rectangle represents an *external entity*, a circle represents a *process* or *transformation*, a labeled arrow represents a *data flow*, and a pair of parallel lines represents a *data store*.

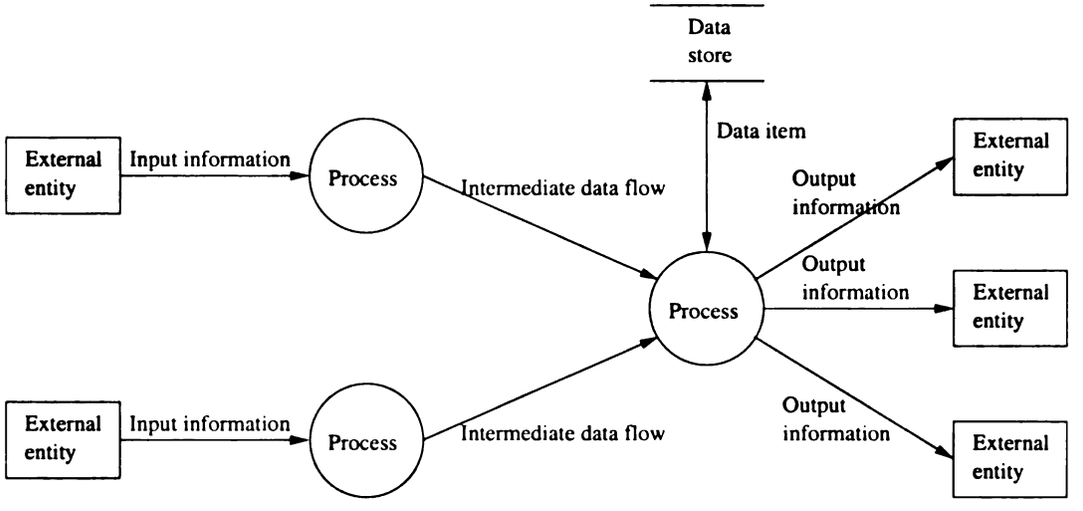

Figure 6.1: A typical data flow diagram

In addition, notations for real-time systems were also introduced by Ward [76] and Hatley [77] to capture the control information and the data flows. Since the

statechart is used in OMT to describe the dynamic aspects of a system, which already includes the control information, only the primitive notations are needed to depict the functional aspects in our object-oriented models.

DFDs in structured and object-oriented designs. Structured design is an iterative refinement process in which the DFD serves as a powerful tool. The purpose of design is to decompose a system into small pieces according to functionality, so that the fine-grained pieces can be directly implemented by programming languages. In each refinement step, processes in the DFDs are further decomposed into more detailed DFDs. A process that cannot be further decomposed will be directly implemented as a function.

In an object-oriented design, a system is usually decomposed into a collection of objects that communicate with each other. The services provided by the objects together with the inter-object communications carry out the functionalities for which the system is intended. Since the DFD models the functionality of a system or an object, it is also called a *functional model* in the object community. Although the services provided by individual objects may specify a certain aspect of functionality, without DFDs, it is difficult to capture the overall required functionality and to verify whether the requirements are satisfied.

There are two advocated strategies that deal with the functional model in the object community. One strategy is used in the object modeling technique (OMT) [3], where the functional models are developed and refined in parallel with the object and dynamic models during the analysis phase. Then, during the detailed design phase,

the pr

its ass

lying t

model

more

Re

spirit

use of

model

ing La

Howev

tional

have a

purpo

the sy

6.2

In ord

orienta

and re

Unl

object-

the processes of the functional models are assigned to individual objects according to its associated input/output data flows and data stores. Since the philosophies underlying the DFD and the object model differ dramatically, the integration of different models during the later stages of the software development life cycle is considerably more difficult and perhaps even impossible [78].

Recognizing that the concept of functional modeling directly conflicts with the spirit of object-orientation, Booch [73] and de Champeaux [74] both warn against the use of even throw-away functional models in order to avoid irrevocably biasing object modeling towards a function orientation. The latest revision of the Unified Modeling Language (UML) [70, 69] also omitted the functional model from its notations. However, this strategy goes to such an extreme that little information about the functional aspects of a system is covered in the proposed models [79]. This deficiency may have an impact and pose difficulties during the design stages, since one of the major purposes of design is to provide implementation details for the functions required for the system.

6.2 DFD Notation Modification

In order to make use of the functional model while retaining the spirit of object-orientation in object-oriented design, it is necessary to modify the DFD notations and redefine the role of a functional model.

Unlike the structured approaches, functions are distributed into objects in the object-oriented methods. Thus we model: (1) the services provided by individual

objects, and (2) how the services of individual, aggregate objects can be composed to implement the services of their corresponding aggregation object. Accordingly, we propose two types of functional models, *object functional model* (OFM) and *service refinement functional model* (SRFM), where the only notations for both models are *external objects*, *processes*, and *data flows*. Since a data store is modeled in terms of an object in the object-oriented methods, an explicit *data store* notation is no longer needed. All the interactions with a data store object are modeled as communications with the services associated with the object that represents the data store.

6.2.1 Object function model

Definition-1: Object functional model: *identifies and describes the services provided by an object and the data flows that flow in and out of services.*

Figure 6.2 contains an OFM that illustrates the services provided by an object, O_3, and how data items flow into and out of the services S_i. The rounded rectangle represents the object with its name in the upper left rectangle. The process bubbles (ovals) represent the services. The labeled arrows represent the data flows. The *ground wires* associated with services represent the accesses to the internal data of objects, which is invisible to external objects. For instance, in Figure 6.3, the OFM of an *integer stack* object provides services *push*, *pop*, and *top*. All three services need to access the internal data structure in order to accomplish the required functionality. This internal data access is represented by ground wires.

Unlike conventional DFDs, the OFM only contains services accessible to external objects; the operations/functions internal (private) to the object are not of concern

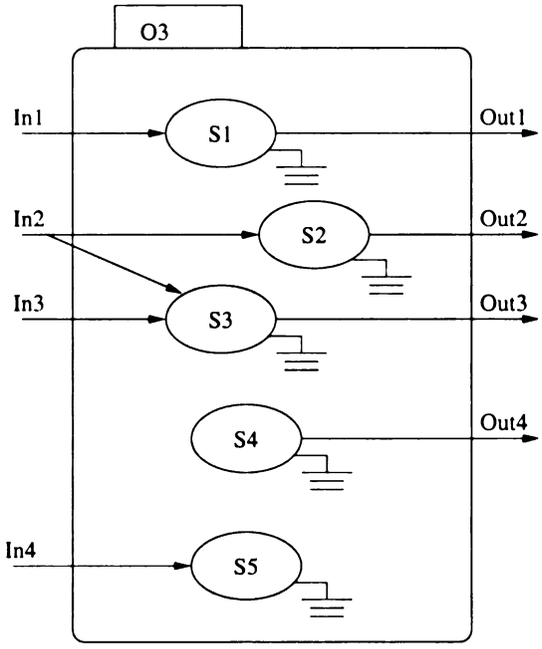

Figure 6.2: An OFM that contains services of an individual object

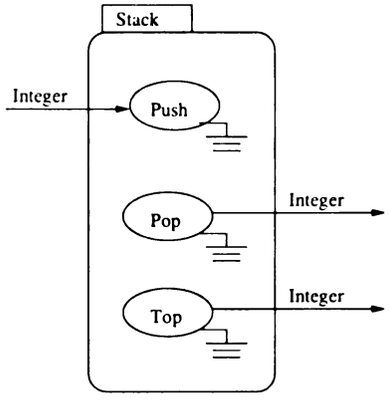

Figure 6.3: An OFM of an *integer stack* object

to the OFM, and therefore are not represented in the OFM. Given that the services will be ultimately implemented as functions, we further require that every process bubble have at most one output data flow that represents the return value from a service. This constraint guarantees that there is only one return value from any service. Indeed, the return value can be of any simple or complex data type.

In Figure 6.2, S_1 is a service that has an input and an output data flow that are In_1 and Out_1, respectively. S_3 has two input data flows and shares In_2 with S_2. S_1, S_2, and S_3 can be considered as functions with input parameters and return values. S_4 and S_5 only have output and input data flows, respectively. Example uses for two such operations are: (1) an operation that sets an attribute of an object only requests input parameters, and (2) an operation that retrieves an attribute and returns the value without input.

6.2.2 Service refinement functional model

Definition-2: Service refinement functional model (SRFM): *models a system/object service in terms of the services/operations provided by the aggregate objects.*

Unlike structured modeling approaches, functions are distributed into objects in the object-oriented methods. The SRFM shows how the service of a target system or an aggregation object is implemented in terms of the services/operations provided by the aggregate objects. Figure 6.4 shows a system level functional model, the highest level of abstraction for an OFM that is comparable to a Level 0 data flow diagram, developed during the analysis phase. The SRFM for service S_2 of the system is

illustrated in Figure 6.5, where S_2 is composed of three services, $O_3.S_1$, $O_9.S_1$, and $O_8.S_2$, provided by aggregate objects O_3, O_9, and O_8, respectively (the dashed oval encapsulates system service S_2). Similar to conventional DFDs, SRFMs focus on showing how data flows among services provided by the aggregate objects. The order in which the services are triggered is captured in the dynamic models (state diagram) but not in SRFMs.

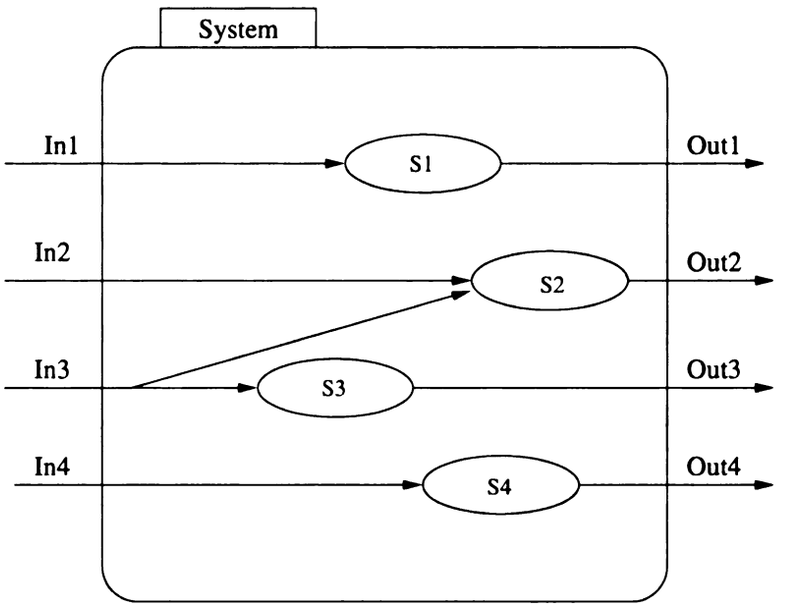

Figure 6.4: A system level OFM derived during analysis

The SRFM given in Figure 6.5 is relatively simple, because the services provided by aggregate objects are adequate to compose service S_2. When services provided by aggregate objects are inadequate to compose the services of the aggregation object, additional internal functions for the aggregation object need to be introduced. For example, in Figure 6.6, an additional internal function Fun_3, which might convert data types, is introduced to process the output from $O_8.S_2$ and generate the input for

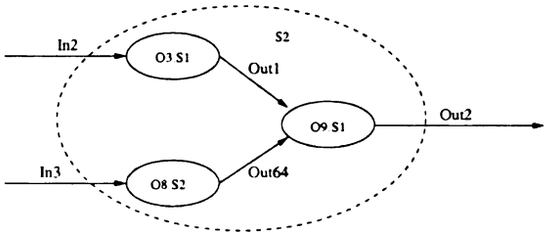

Figure 6.5: An SRFM shows how a system service is implemented in terms of object services

$O_9.S_1$. Unlike external services, an internal function represents a functional component that is imperative, together with aggregate object services, to construct a service of an aggregation object. The internal function does not have to be literally implemented as a function or a procedure. It may even be implemented as a piece of code inside the service of the aggregation object. Since each function of an aggregation object or service provided by an aggregate object has only one output data flow and circular data flow is not allowed, the services and functions of a SRFM construct an acyclic directed graph.

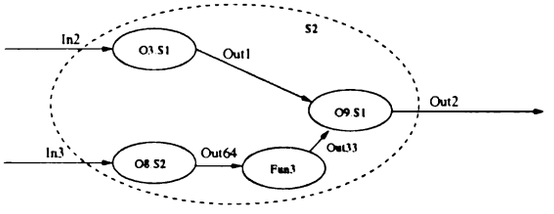

Figure 6.6: An SRFM that introduces an additional internal function

6.2.3 Data refinement

In addition to the refinement of services, an SRFM may also be used to describe data refinement. When refining services of aggregation objects in terms of services of aggregate objects, a data flow at one level of abstraction may consist of two or more data flows of a lower level of abstraction. For example, the input and output data flows, In_1 and $Oout_1$, for service S_1 in Figure 6.4 are refined into two input flows ($In_1 1$ and $In_1 2$) and two output flows ($Out_1 8$ and $Out_1 4$) shown in Figure 6.7. A solid box with one inflow and multiple outflows represents data splitting, where each split data line is a part of the original data line, and the split data lines are mutually exclusive; a solid box with multiple inflows and one outflow represent *data aggregation.* The diagram shows that: (1) the input data flow In_1 splits two input data flows, $In_1 1$ and $In_1 2$, that flow into $O_3.S_2$ and $O_9.S_3$ respectively; and (2) the output data flow consists of two data flows, $Out_1 8$ and $Out_1 4$, that flow out of the corresponding services provided by the aggregate objects. The semantics of the new notations, data aggregation and splitting, will be further discussed and formalized in Section 6.4.

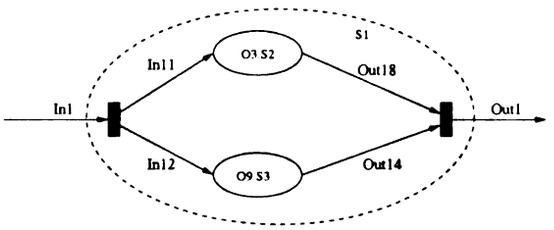

Figure 6.7: An SRFM with split and aggregate data flows

Another type of data flow refinement occurs when input/outflow of a service is supplied to multiple services without further splitting into finer-grained data flows (data duplicator), or the input/output data flow of a service is one out of a set of data flows (data selector). In order to represent this data duplicator/selector relationship, a graphical construct, empty box, is introduced. In Figure 6.8, input data In_1 might be used by both services $O_3.S_2$ and $O_9.S_3$; output data Out_1 can be either Out_18 or Out_14, but not both. The data duplicator/selector in the figure also requires In_1, In_11, and In_12 to be at least mutual convertible types (values of the types can be converted into each other); the same constraint applies to Out_1, Out_18, and Out_14. The uses of the services along with their data flows in the SRFMs must be consistent with the services and data flows depicted in the corresponding OFMs.

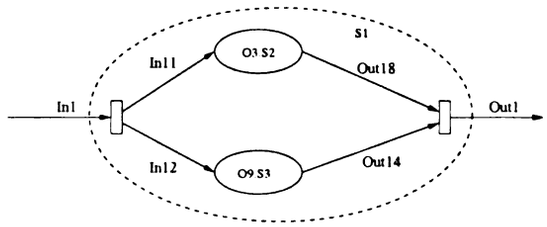

Figure 6.8: An SRFM with refined input and output data flows

6.3 Integrating the Functional Model into OOD

Instead of developing functional models in parallel with object models or excluding functional models from the object-oriented design, the OFM together with the

SRFM can be used and integrated in the object-oriented design without violating the philosophy underlying OOA and OOD.

6.3.1 System level object functional model and refinement

The system level OFM, a special OFM whose *object* is the target system, can be obtained from the results in the analysis stage. In fact, the system level functional model describes the functionalities that are required to be implemented in the target system. (The detailed rules for transformations from analysis to design is beyond the scope of this dissertation.) Figure 6.9 shows the system functional model for *Disk Manager*. As a notation convention in the discussion of diagrams and specifications for diagrams, the italic font will be used to denote diagram components and the courier font will denotes specification components. There are two services, *insert* and *retrieve*, that satisfy the requirement that the system should be able to store and retrieve data. The *insert* service takes data and its corresponding key as input, and stores them inside *Disk Manager*. The *retrieve* service outputs the data according to the key it receives. Service *insert* does not have explicit output. It either stores data in the internal data structure of *Disk Manager* or passes data to an aggregate object. However, from a perspective external to *Disk Manager*, the implementation details about how data is to be processed and stored is not of interest.

After the system is decomposed into aggregate objects and their corresponding OFMs are constructed, the services in the system level object functional model can be refined in terms of the services provided by the aggregate objects. Each SRFM

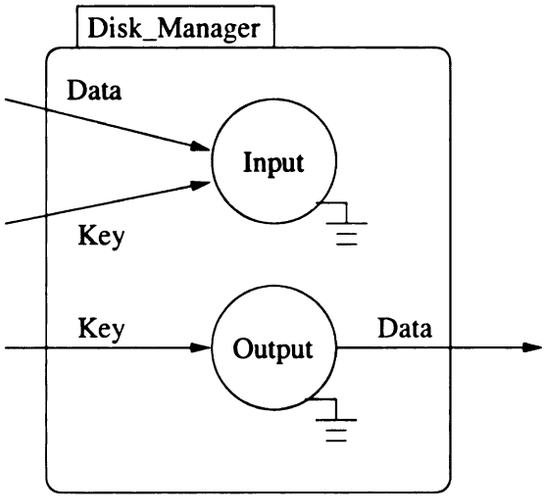

Figure 6.9: System level object functional model for *Disk Manager*

shows how a service of the system is implemented in terms of services provided by the aggregate objects.

Figure 6.10 illustrates the simplified *Disk Manager* with aggregate objects *Storage* and *Compressor*. The OFMs for *Storage* and *Compressor* are shown in Figures 6.11 and 6.12, respectively. Figure 6.13 shows the SRFM for the *Retrieve* service of the *Disk Manager*, whose system functional model is given in Figure 6.9. The SRFM (Figure 6.13) depicts how operation *Retrieve* of *Disk Manager* is implemented by composing *Retrieve* and *Decompress* services provided by aggregate objects *Storage* and *Compressor*, respectively.

In order to ensure the consistency between the different functional models, the SRFMs need to be checked against OFMs to make sure the referenced services together with their input/output data flows are consistent with their counterparts depicted in OFMs. If we check the SRFM in Figure 6.13 against the OFMs in Fig-

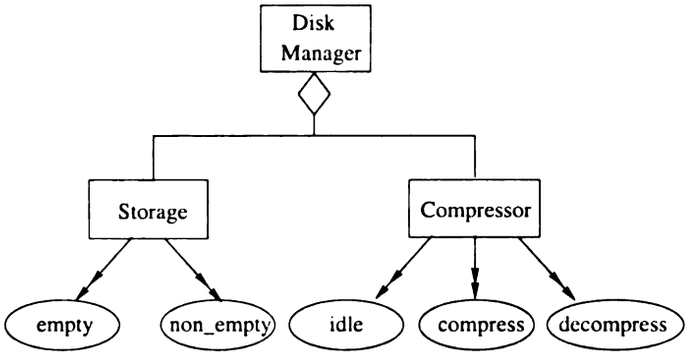

Figure 6.10: A simplified disk manager

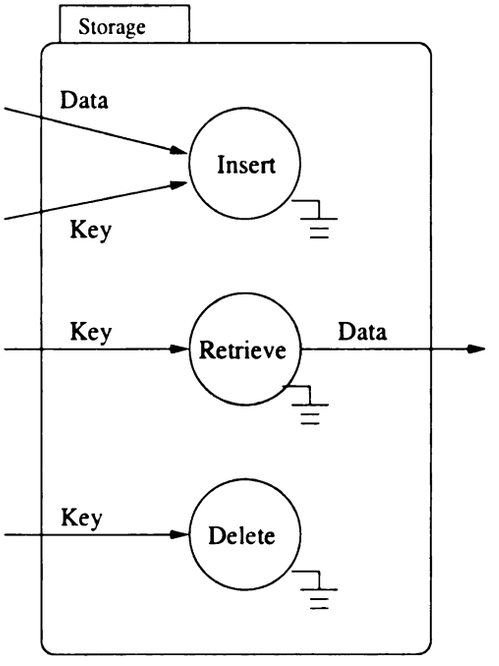

Figure 6.11: The OFM for *Storage*

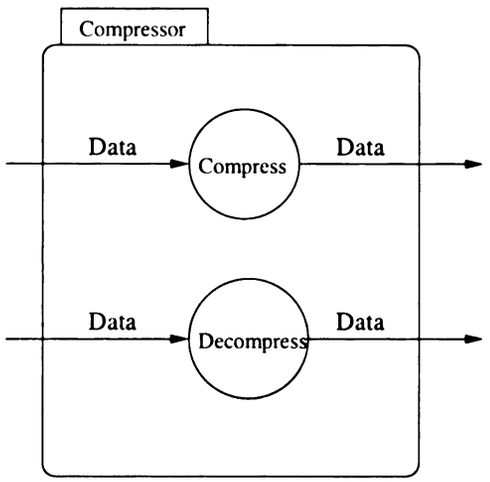

Figure 6.12: The OFM for *Compressor*

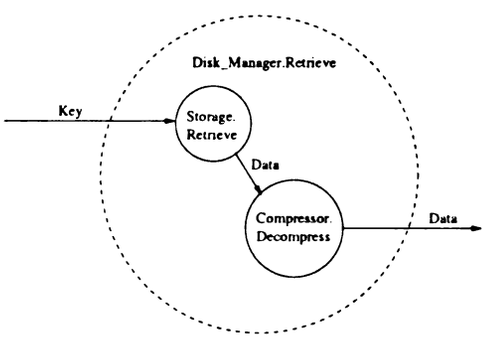

Figure 6.13: The SRFM for *Disk_Manager.Retrieve*

ures 6.11 and

associated wit

data flows are

During the

pleteness and

Service incom

gate obje

aggregate

Service redu

object se

system s

the syste

Service in

sociated wit

more service

to take. Intr

minimize the

pleteness iss

redundant se

the redundar

achieve conc

ures 6.11 and 6.12, we will find that the number of input and output data flows associated with *Storage.Retrieve* and *Compressor.Decompress* and the labels of the data flows are consistent with services *Retrieve* and *Decompress.*

During the service refinement process, two types of design defects, service incompleteness and service redundancy, can be detected.

Service incompleteness: If a system service cannot be refined in terms of the aggregate object services, then either the aggregate objects or the services provided by aggregate objects are not sufficient to support the system services.

Service redundancy: If every system service is already modeled in terms of aggregate object services and an aggregate object service $O_i.S_j$ has not been used by any system service, then aggregate object service $O_i.S_j$ is redundant in the context of the system functionality.

Service incompleteness implies either more services need to be identified and associated with aggregate objects or more objects need to be introduced to provide more services. The nature of the services that are needed determine which approach to take. Introducing services to existing objects should be given higher priority to minimize the number of objects. If this approach cannot resolve the service incompleteness issue, new objects need to be introduced. If the object that contains a redundant service is a newly designed object (in contrast to a reused object), then the redundant service should be excluded from the corresponding OFM in order to achieve conciseness and preciseness.

6.3.2 Object functional model and refinement

The object functional modeling and service refinement functional modeling is an iterative process. Similar to the system level object functional modeling and its refinement, the objects obtained in the previous iteration of modeling are further decomposed into aggregate objects, and their corresponding services are further refined in terms of the services provided by the aggregate objects.

6.4 Functional Model Formalization

The OFMs and the SRFMs are tightly coupled with the object and dynamic models. In order to integrate the three models formally, the functional models are formalized in terms of algebraic specifications and process algebras. In this section, we introduce the formalization rules for the OFM and the SRFM.

6.4.1 Formalization of the object functional model

The OFM specifies the services that an object provides to the external world. In our proposed formalization rules, the input/output data items are formalized as *sorts* of an algebraic specification. Sorts are similar to the programming language concept of type, but it is important not to confuse these concepts. Sorts represent disjoint nonempty sets of values and are used to indicate the domains and ranges of operators. A sort can be implemented as different types in a specific language. Since the implementation details are gradually added during the design process, we choose sorts

instead of ty

rule formaliz

FFR-1 Each da
For all t
declarati

In order t

resenting an

FFR-2 For ever
undef_s

The servi

external worl

Though objec

projects [31].

ture of opera

from tuples c

a sort for its

relationship b

the properties

language shar

tinction betwe

operations tha

defined in alge

instead of types of a specific programming language to represent data. The following rule formalizes the data flows in an OFM.

FFR-1 Each data item in an OFM is formalized as a sort in the corresponding algebraic specification. For all the data items, $D_i (1 \leq i \leq n)$, in an OFM, create the following expression in the sort declaration part of an ACT ONE specification of Full LOTOS

sorts

D_1, D_2, ..., D_n

In order to handle potential exceptions, we also introduce a nullary operator representing an undefined data value for each sort.

FFR-2 For every sort, S, used in the object specification, introduce operator **undef_s:** $\rightarrow S$.

The services provided by an object are the only means for interaction with the external world. The gates in Full LOTOS specifications serve an analogous purpose. Though object services are formalized as operators of algebraic specifications in some projects [31], this approach has not been adopted in this research because of the nature of operators of algebraic specifications. That is, an operator represents a map from tuples of values to values; its signature is a tuple of sorts for its domain and a sort for its result. The objective of an operator is to describe the mathematical relationship between sorts of values. Thus the operators are often used to capture the properties of systems. Though procedures/functions of a specific programming language share some similarity with operators of algebras, there is a fundamental distinction between the two. A procedure/function consists of a sequence of data/control operations that are intended to perform certain actions on data, whereas operations defined in algebraic specifications describe mathematical relations between sorts. The

former focuses on operation, the latter focuses on property. Therefore services are not formalized as operators of algebraic specifications in our proposed formalization rules. Instead, the services are formalized in terms of gates in a Full LOTOS specification to capture the interface of an object with its external world. The following rule formalizes the services in an OFM.

FFR-3 Each service in an OFM is formalized as a gate that is associated with input and output in the corresponding Full LOTOS specification.
For all the services, $S_i(1 \leq i \leq n)$, with inputs, $I_{i_j}(1 \leq i_j \leq |I_i|$ (I_i represents the permutation of the input data), and outputs, O_i, in the OFM of object $OBJECT$, create the following expression

specification $OBJECT$ [S_1, S_2, ..., S_n] **: noexit :=**
(* S_1 : $i_{1_1} : I_{1_1}$, $i_{1_2} : I_{1_2}$, ..., $i_{1_{|I_1|}} : I_{1_{|I_1|}}$ -> O_1 *)
(* S_2 : $i_{2_1} : I_{2_1}$, $i_{2_2} : I_{2_2}$, ..., $i_{2_{|I_2|}} : I_{2_{|I_2|}}$ -> O_2 *)
...
(* S_n : $i_{1_n} : I_{1_n}$, $i_{1_n} : I_{1_n}$, ..., $i_{n_{|I_n|}} : I_{n_{|I_n|}}$ -> O_n *)
.....
typedef $OBJECT$ **is**
.....
opns
S_1 : I_{1_1}, I_{1_2}, ..., $I_{1_{|I_1|}}$ -> O_1
S_2 : I_{2_1}, I_{2_2}, ..., $I_{2_{|I_2|}}$ -> O_2
.....
S_n : I_{1_n}, I_{1_n}, ..., $I_{n_{|I_n|}}$ -> O_n
.....
endtype endspec

Since LOTOS does not currently support typed gates [80], the input/output arguments for gates/services are given in annotations delimited by the (* and *) pairs. For those services without input arguments, the left side of the arrow "->" will be left blank; for those services without output arguments, a special sort "`Void`", representing an empty set, is used as the output argument. The i's represent parameter variables of the corresponding sorts. Currently the description of services in terms of typed gates are not supported by LOTOS. Therefore, in order to facilitate the analy-

sis of the automatically generated formal specification, the services are also described as operations in the *opns* section of algebraic specification.

According to formalization rules **FFR-1**, **FFR-2**, and **FFR-3**, the OFMs for system *Disk Manager* and object *Storage* lead to the generation of the algebraic specifications in Figures 6.14 and 6.15, respectively. Sort `Void` is a special sort that indicates no value will be returned. The `Storage` algebraic specification of includes `Disk\_Manager`, because the aggregate object usually uses the data sorts defined by the corresponding aggregation object.

```
specification Disk_Manager [Insert, Retrieve]  : noexit :=
   (* Insert     : d: Data, d: Key -> Void *)
   (* Retrieve   : k: Key -> Data *)
type Disk_Manager is
    sorts
      Data, Key, Void
    opns
      undef_Data: -> Data
      undef_Key : -> Key
      Insert    : Data, Key -> Void
      Retrieve  : Key -> Data
endtype
endspec
```

Figure 6.14: The ACT ONE specification for *Disk Manager*

6.4.2 Derivation of pre- and postconditions for services

The rules given thus far have focused on how to formalize the graphical notations in an OFM in terms of LOTOS constructs. Next, we discuss an extension to the formalized OFM to include pre- and postconditions in order to further describe services.

```
specification Storage [Insert, Retrieve]  : noexit :=
   (* Insert    : d: Data, k: Key -> Void *)
   (* Retrieve  : k: Key -> Data *)
type Storage  is Disk_Manager
    sorts
      Storage
    opns
      Insert    : Data, Key -> Void
      Retrieve  : Key -> Data
endtype
endspec
```

Figure 6.15: The ACT ONE specification for *Storage*

Algebraic specifications for objects

Before pre- and postconditions for services can be addressed, we must investigate the abstract algebraic specifications generated from the FRMs that describe the properties of objects. The algebraic specifications form the foundation upon which pre- and postconditions of services can be stated and about which the properties of the objects can be reasoned.

Although such algebraic specifications cannot be generated automatically, we are not without some general ideas of how to construct these specifications. The sorts and services depicted and formalized in the previous subsections together with the results of requirements analysis may all serve as clues to derive algebraic specifications. Some high-level guidelines, denoted by **GL-X**, will be given in the following discussion about algebraic specification derivation.

The derived algebraic specifications shall include a set of sorts that represent the attributes for objects, a set of operators, and a set of axioms that describes

the properties of objects in terms of attributes and their operators. The operators together with axioms should be sufficient to support the construction of pre- and postconditions.

The purpose of the algebraic specifications is to describe the properties of the modeled objects. Pre- and postconditions of services are themselves expressions of the properties of an object at a certain time. Since attributes may change over time, operators that manipulate the attributes need to be introduced to reflect and describe the potential change. Based upon the operators, pre- and postconditions can be constructed to assert the property of an object before and after an external service is invoked.

GL-1 Sufficient operators for the sorts declared in the algebraic specification shall be provided to construct pre- and postconditions.

The operators of data sorts only represent a set of manipulations that may be applied to corresponding sorts. The exact semantics of the operators are described through axioms. Therefore, without a set of axioms, no reasoning can be performed against pre- and postconditions.

GL-2 If reasoning about pre- and postconditions is expected, related axioms for algebraic specifications need to be derived.

Formalization of attributes.

An algebraic specification is a high-level abstraction intended to describe a class of objects instead of a particular object. For instance, the algebraic specification given

in Figure 4.1 only describes the abstract data types of a class of objects, it provides no means to represent a particular object with concrete values for it attributes. In order to present individual objects, a mechanism, other than algebraic specifications, that can represent objects in terms of concrete values for their attributes is desired.

As mentioned previously, in Full LOTOS, in which algebraic specifications is a component, a specification can have a set of parameters of sorts that are defined in algebraic specifications. This capability provides a convenient mechanism to describe individual objects. By making the distinguished sort of an object class as a parameter of its Full LOTOS specification, we are able to represent particular objects. According to the formalization rules for dynamic models, processes of a Full LOTOS specification represent states of an object, therefore the distinguished sort is also formalized as a parameter for the corresponding processes of the specification.[1]

FFR-4 The set of attributes of an object is formalized as a parameter of the distinguished sort for the corresponding Full LOTOS specification.
For object class O, and its distinguished sort ϕ and a set of services Σ, create the following

	specification O [Σ] ($o : \phi$) **: noexit :=**
	...
expression	**endspec**

Based upon the guidelines to derive algebraic specifications and rule **FFR-4**, the Full LOTOS specification for *Disk Manager* is given in Figure 6.16, where the italic text denotes the newly added specification components to what was previously derived in Figure 6.14. Sort `Disk` is the distinguished sort for *Disk Manager*. Since the design is still at a high level, the distinguished sort does not contain other sorts but

[1] This is related to the formalization of the dynamic model, in which processes represent object states.

is an abstract sort representing the potential attributes of *Disk Manager*. Operators **insert** and **retrieve** are derived to describe potential manipulations on sort **Disk**. The services provided by *Disk Manager* imply that data items may be inserted into and retrieved from *Disk Manager* thus giving guidance for the derivation of operators for sort **Disk**. Though the operators and the services share similar names, it is not a necessity.

```
specification Disk_Manager [Insert, Retrieve] (disk: Disk)  : noexit :=
   (* Insert    : d: Data, k: Key -> Void *)
   (* Retrieve  : k: Key -> Data *)
type Disk_Manager  is
    sorts
      Data, Key, Void
      Disk (* Distinguished sort *)
    opns
      undef_Data: -> Data
      undef_Key : -> Key
      empty : ->  Disk
      insert    : Disk, Data, Key -> Disk
      retrieve  : Disk, Key -> Data
    eqns
      forall disk: Disk, data: Data, k1, k2: Key
      ofsort Data
          retrieve (empty, key) = undef_Data;
           retrieve (insert(disk, data, k1), k2) = if k1=k2 then data
               else retrieve(disk, k2)
endtype
endspec
```

Figure 6.16: The Full LOTOS specification for *Disk Manager* with algebraic specifications and distinguished sort

---

Pre- and postconditions for services.

The services along with their input/output data flows only describe the interfaces of an object to the external world. The object properties exhibited by services as well

as constraints for services cannot be represented by gates and their corresponding signatures alone. In order to describe the constraints and requirements for object services, we make use of concepts used in the Larch Interface Languages (LILs) [19]. The specific approach is to use pre- and postconditions to depict requirements and constraints for a service. Thus the specification of a service consists of a service declaration followed by a body of the form:

```
<service declaration>
    requires <requirement expression>
    modifies <modification expression>
    ensures <ensuring expression>
```

For every service, we presume two states for the objects: the state when the service is invoked, the *pre-state*, and the state when it terminates, the *post-state*. More specifically, the pre- and post-states represent two sets of values of the attributes of an object. In order to refer to the values contained by the attributes of a distinguished sort in the pre- and post-states of a service, the postfix operators ^ and ′, respectively, are introduced. When applied to a variable/parameter of a simple sort, for instance variable x of sort s, $x^{\wedge}$ and x' yield the stored value in pre- and post-states, respectively. When applied to a variable/parameter of a complex sort, such as a distinguished sort containing sorts for attributes, ^ and ′ yield a tuple containing the values stored in the variable or parameter.

A *requires* clause (precondition) is used to state restrictions on the values in the pre-state of both attributes of the corresponding object and parameters for the service.

An omitted *requires* clause is equivalent to the weakest possible requirement, "*requires true*".

A *modifies* clause states which attribute(s) of the object can be changed during the invocation of the service. Unless an attribute of an object is listed in the *modifies* clause, the service must not change the value of the attribute, and the attribute must have the same value in the pre- and post-states. If no *modifies* clause is given, then nothing may be changed.

An *ensures* clause (postcondition) places requirements and constraints on the functionality of a service. It relates the pre-state and post-state of a service. The reserved keyword **result** represents the value/output (if any) returned by the service.

Figure 6.17 shows the specification for *Disk Manager* with pre- and postconditions for services `Insert` and `Retrieve`, delimited by `(*, *)` pairs. For service `Insert`: the *requires* clause of `Insert` requires that the input parameters of sort `Data` and `Key` contains valid values; the *modifies* clause states that the values of the attributes of sort `Disk` will be changed when `Insert` is invoked; the *ensures* clause guarantees that after the attribute `disk` is changed, its value will reflect the fact that a pair of data and key is inserted. The *ensures* clause for service `Retrieve` ensures that the returned value will be exactly what `retrieve(disk,k)` describes.

Integrating pre- and postconditions into dynamic models.

Our specification approach is attempting to integrate the Larch and LOTOS specification frameworks in order to provide a three-pronged approach to specification, where the algebraic specifications provide the integration mechanism since both Larch

```
specification Disk_Manager [Insert, Retrieve] (disk: Disk) : noexit :=
   (* Insert    : d: Data, k: Key -> Void *)
   (* requires d eq undef_Data = false and k eq undef_Key = false *)
   (* modifies disk *)
   (* ensures disk' eq insert(disk^, d, k) = true *)
   (* Retrieve  : k: Key -> Data *)
   (* requires disk eq empty = false and k eq undef_Key = false *)
   (* ensures result = retrieve (disk, k) *)
type Disk_Manager  is
    sorts
      Data, Key, Void, Disk
    opns
      undef_Data: -> Data
      undef_Key : -> Key
      empty     : -> Disk
      insert    : Disk, Data, Key -> Disk
      retrieve  : Disk, Key -> Data
    eqns
      forall disk: Disk, data: Data, k1, k2: Key
      ofsort Data
         retrieve (empty, key) = undef_data;
         retrieve (insert(disk, data, k1), k2) = if k1=k2 then data
            else retrieve(disk, k2)
endtype
endspec
```

Figure 6.17: The Full LOTOS specification for *Disk Manager* with pre- and postconditions for services

and LOTOS have algebraic components. In both cases, the algebraic specifications provide the foundation to describe and reason about properties of objects. The services/functions specifications describe the services accessible to the external world as well as the pre- and postconditions that shall be satisfied. The process algebras depict the behavior of the objects and illustrate how the services/functions are invoked and interact during the life cycle of their objects. However, the services/functions in our approach are more abstract than those typically specified in the Larch Interface Language, whose interfaces are usually specified in the syntax of a specific programming

language. Since LOTOS does not currently support specifications for pre- and post-conditions of services, they are currently introduced as annotations for Full LOTOS specifications. A new language that supports our needs, where services/functions, together with their pre- and postconditions, of an object need to be explicitly developed; and corresponding tools are also desired to support the construction, refinement, and analysis of these heterogeneous specifications.

Though service declarations and pre- and postconditions specification are not explicitly supported by LOTOS currently, we can still formalize them in terms of constructs provided by LOTOS. The functional models and dynamic models share some common features, one of which is that the actions invoked by events in the dynamic models usually are services/functions defined in functional models. Figure 6.18 gives a typical state diagram with an initial state and three state transitions. A label on an arc in the format of $e(d_1 : D_1, d_2 : D_2, ..., d_n : D_n)[c]/a(d_{x_1}, d_{x_2}, ..., d_{x_m})\hat{}O.e'(d_{y_1}, d_{y_2}, ..., d_{y_l})$ denotes an event e, data items $(d_1, d_2, ..., d_n)$ associated with the event e, a guarding condition c, the corresponding action a (with parameters $d_{x_1}, d_{x_2}, ..., d_{x_m}$), and the event e' of object O to be triggered. An event is a request for a services that occurs at a gate representing the corresponding service (see **FFR-3**). Usually, if the service is not further refined into internal functions and services provided by aggregate objects, then the event will directly trigger the service to which it refers to. Either a service or an internal function is invoked, the associated pre- and postconditions can be specified in the corresponding process algebras that describe the dynamic behavior of the object.

The related formalization rules along with the rules for the dynamic model together contribute to generate formal specifications in terms of process algebras.

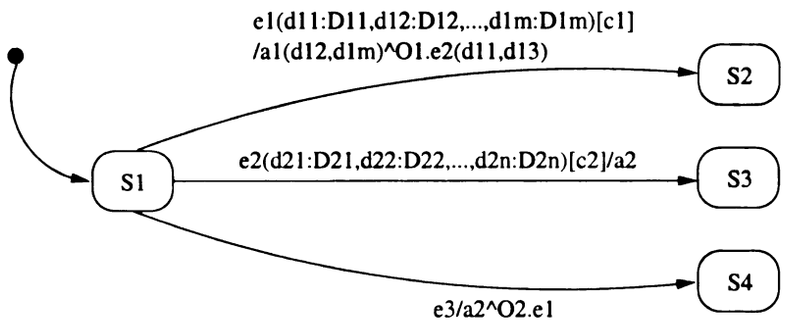

Figure 6.18: A typical state diagram

In Chapter 5, we developed a formalization for dynamic models. We will not go into the details of that formalization, but, instead, concentrate on integrating pre- and postconditions into the process algebras generated according to the dynamic model formalization rules given in Chapter 5.

Usually, both pre- and postconditions are used to describe the properties and constraints of a service/function in terms of the pre- and post-states of the attributes of their corresponding object in addition to the input and output data. Since the value of an attribute may change after a service/function is triggered, a mechanism is needed to save the pre-state values of attributes. In the course of the dynamic model formalization, we introduced a local variable to save the values of attributes when a process, representing a state in the dynamic model, is entered. The values of the attributes may be changed by a service/function invoked by an event, thus making the object transition into another state. Therefore, the values stored in the local vari-

able represent the attributes of the object in the pre-state of the services/functions. The following rule formalizes the introduction of a variable to store the value of the attributes prior to the execution of a service.

FFR-5 The pre-state of the attributes of an object is stored in a local variable when a process, representing a state in the dynamic model, is entered.
For object class O, and its distinguished sort ϕ and a set of services Σ, if p is a process, representing a state in the dynamic model, defined in the process algebras, then create the following expression that introduces local variable PRE to store values of attributes at the entrance of the process

```
specification O [Σ] (o : φ) : noexit :=
...
   process p [Σ] (o : φ) : noexit :=
      ( let PRE:= o in
      ...
      )
   endproc
...
endspec
```

A non-trivial (not a *true* Boolean expression) precondition, expressed in a *requires* clause, refers to input parameters of a service/function and attributes in the pre-state of the corresponding object in order to state the conditions that must be satisfied before the service/function can be invoked. A guarding condition of a transition in the dynamic models is used to describe circumstances under which a state transition can take place. We formalize both the guarding conditions of the dynamic models and the preconditions of the functional models in terms of guarded expressions supported by LOTOS. Only when both the guarding conditions and the preconditions hold can the corresponding service/function be invoked.

FFR-6 The guarding condition and the precondition that are associated with the action of a state transition are conjuncted and formalized in terms of a guarded expression in LOTOS process algebras.
Suppose event e (associated with arguments $x_1 : a_1, x_2 : a_2, ..., x_n : a_n$), condition c, action a (a refers to a service/function and is associated with parameters y_1, y_2, ..., y_m, if event e directly refers to service a, then the y's and x's are identical too), together constitute a state transition from state s of an object with distinguished sort ϕ and services Σ, if p is the precondition for service/function a, create the following expression

```
process s [Σ](o: φ): noexit :=
    e ?x1 : a1, ?x2 : a2, ..., ?xn : an;
      ([c and p] → <a(y1, y2, ... ym)>)
endproc
```

The postcondition, expressed in an *ensures* clause, of a service/function states the conditions that must be satisfied at the termination of the service/function. If it is formalized in the process algebras fashion similar to the formalization rule for preconditions in **FFR-6**, then a guarding expression is introduced. The guarding condition serves to check whether the postcondition is satisfied before leaving the state of the corresponding object. Specifically, for **FFR-6**, the guarding expression will be given after <a(y_1, y_2, ..., y_m)> in order to ensure that the postcondition is satisfied.

FFR-7 The postcondition that is associated with the action of a state transition is formalized in terms of a guarded expression in LOTOS process algebras.
Suppose event e (associated with arguments $x_1 : a_1, x_2 : a_2, ..., x_n : a_n$), condition c, action a action a (associated with parameters y_1, y_2, ..., y_m, if event e directly refers to service a, then the y's and x's are identical too), together constitute a state transition from state s of an object with distinguished sort ϕ and services Σ, if p and q are the pre- and postconditions for service/function a, create the following expression

```
process s [Σ](o: φ): noexit :=
    e ?x1 : a1, ?x2 : a2, ..., ?xn : an;
      ([c and p] → <a(y1, y2, ... ym)>);
      ([q]→...)
endproc
```

There is no doubt that **FFR-7** is a legitimate formalization rule for postconditions. However, unless we presume that the invoked service/function performs certain actions that either modify the attributes of an object or returns a value, then it is impossible to check whether the postcondition is satisfied. It is only during the later stages of software development, when individual services/functions already have a specific implementation, that we can make such a presumption. During the early stages of design, when services/functions are not implemented, the postcondition cannot be analyzed and checked, but, instead, can only serve as a specification. In order to facilitate the validation, verification, and analysis of design, we further investigated the use of postconditions and developed two alternative formalization rules. Given these two rules, the derived formal specification of an object is amenable to symbolic execution for simulation purposes.

A postcondition contains assertions about constraints/requirements in three basic format categories. The first category includes the assertions of the form: "**result := <expression>**." This form of postcondition is usually associated with services/functions that return values. The *expression* on the right hand side of keyword *result* specifies the value to be returned in terms of the operators of the algebraic specification for the corresponding object. Since the exact return value of a service/function is not available during the early stages of design, in order to facilitate design analysis and simulation, the abstract, declarative return value given in the assertion, **result = <expression>**, is used for symbolic execution.

FFR-8 The postcondition that is associated with the action of a state transition, in the form of *result* = ⟨*expression*⟩, is formalized in terms of a value declaration in LOTOS process algebras.
Suppose event e (associated with arguments $x_1 : a_1, x_2 : a_2, ..., x_n : a_n$), condition c, action a (associated with parameters y_1, y_2, ..., y_m, if event e directly refers to service a, then the y's and x's are identical too), together constitute a state transition from state s of an object with distinguished sort ϕ and services Σ, if q is an expression, in the form of $result = op\ (z_1, z_2, ..., z_l)$ (op is an operator defined in the corresponding algebraic specifications, z_i $(1 \leq i \leq l)$ is a subset of y_i $(1 \leq i \leq m)$ and attributes) within the postcondition for service/function a, then create the following expression

```
process s [Σ](o: φ): noexit :=
    e ?x1 : a1, ?x2 : a2, ..., ?xn : an;
      ([c and p] → e! op(z1, z2, ... zl))
endproc
```

The *modifies* clause lists the object attributes that may be changed during the execution of a service/function. As a consequence, the *ensures* clause that follows the *modifies* clause always describes the constraints/requirements on the changes of the specified attributes. Among the descriptions of attribute changes, several assertions have the form: "`o^ := ...`" or "`o^.a := ...`", where `o` represents the set of attributes, and `a` represents one attribute. This format defines the second category of assertions of postconditions. Similar to the first category, the second category of assertions specifies the value, in terms of operators of the corresponding algebraic specifications, to be assigned to the corresponding attributes. Since the service/function at such an early phase of design does not change the attributes specified in the *modifies* clause, we use the abstract value given in the second category of assertions to facilitate the validation and verification of design. For instance, the specification in Figure 6.17 gives the pre- and postconditions for services **Insert** and **Retrieve**. During the design phase, no implementation of the service is available for **Retrieve**. However, its postcondition **result = retrieve (disk, k)** precisely

specifies the returned value in terms of algebras. Therefore, `retrieve (disk, k)` can be used symbolically to simulate the system behavior or to validate the design during design phase.

FFR-9 The postcondition that is associated with the action of a state transition can be formalized in terms of a local variable declaration in LOTOS process algebras.
Suppose event e (associated with arguments $x_1 : a_1, x_2 : a_2, ..., x_n : a_n$), condition c, action a(associated with parameters y_1, y_2, ..., y_m, if event e directly refers to service a, then the y's and x's are identical too), together constitute a state transition from state s of an object with distinguished sort ϕ and services Σ, if Q is a set of expression, in the form of $o^{\wedge}.a_i = op_i\ (z_{i_1}, z_{i_2}, ..., z_{i_l})(1 \leq i \leq k)$ (op_i is an operator defined in the corresponding algebraic specifications, z_{i_j} $(1 \leq j \leq l)$ is a subset of y_i $(1 \leq i \leq m)$ and attributes) within the postcondition for service/function a, create the following expression

```
process s [Σ](o: φ): noexit :=
    e ?x1 : a1, ?x2 : a2, ..., ?xn : an;
      ([c and p] →
         (let POST.a1 := op1(z1_1, z1_2, ... z1_l),
            POST.a2 := op2(z2_1, z2_2, ... z2_l),
            ...,
            POST.ak := opk(zk_1, zk_2, ... zk_l)  in
            ...
         )
      ...
      )
endproc
```

An assertion in a postcondition that does not belong to either of the first two categories falls into the third category that describes the constraints/requirements to a service/functions indirectly. By *indirectly*, we mean that the specifications of a postcondition only assert the conditions that shall be satisfied after the execution of the service rather than giving a symbolic value that can be rewritten by the algebraic constructors. Thus we are not able to use the abstract values to represent the return value of a service or the attributes after modification. If a postcondition contains such an assertion of the third category, the only applicable formalization rule is **FFR-7**. Since the services/functions, at such a high level of design, do not cause any actual change to object attributes nor return a specific value, no mechanism exists to verify if

a postcondition of the third category can be satisfied. Therefore, symbolic execution and analysis are not possible. However, in most cases, the postconditions can be covered by the assertions of the first two categories, thus enabling us to perform formal analysis.

6.4.3 Formalization of the service refinement functional model

This subsection discusses the formalization of the service refinement functional model (SRFM).

Formalization of refined objects.

The purpose of the SRFM is to refine the services of aggregation objects in terms of services provided by aggregate objects. Therefore, the formalization of SRFMs leads to the refinement of the existing algebraic specifications that correspond to aggregation objects. Since the refinement is based upon existing models and their corresponding algebraic specifications, the refined algebraic specifications should include their specifications before the refinement for reference use.

FFR-10 The refinement of object O, results in a refined algebraic specification, S_REF, that includes additional operators and axioms.
For every object O that is further refined into a set of aggregate objects O_1, O_2, ..., O_n, create the following expression

```
type O_REF is
    ...
endtype
```

FFR-11 In a refined algebraic specification for an object (or system), the algebraic specifications for the aggregate objects are included in order to make use of the services defined in the aggregate objects.
For every object O that is further refined into a set of aggregate objects O_1, O_2, ..., O_n, create the following expression

```
type O_REF is O1, O2, ..., On
    ...
endtype
```

For instance, given the SRFMs for the services of *Disk Manager*, the algebraic specification **Disk_Manager** will be refined to form a new specification, **Disk_Manager_REF**.

```
type Disk_Manager_REF  is Storage, Compressor, Disk_Manager
   ......
endtype
```

Figure 6.19: A part of the refined ACT ONE specification for *Disk Manager*

Formalization of internal functions and data refinement.

Similar to the formalization of services in OFMs, new internal functions introduced in the SRFMs are declared along with the services as annotations. Since the internal functions are not accessible to external objects, they are not formalized in the gate list of the corresponding specification.

FFR-12 Each internal function in an SRFM is declared along with services as annotations and operators of algebraic specification.
For all the internal functions, $F_i(1 \leq i \leq n)$, with inputs, $I_{i_j}(1 \leq i_j \leq |I_i|$ (I_i represents the permutation of the input data), and outputs, O_i, in the OFM of object $OBJECT$, create the following expression

specification $OBJECT$ [< *service list*>]] **: noexit :=**
(* F_1 : $i_{1_1} : I_{1_1}$, $i_{1_2} : I_{1_2}$, ..., $i_{1_{|I_1|}} : I_{1_{|I_1|}}$ -> O_1 *)
(* F_2 : $i_{2_1} : I_{2_1}$, $i_{2_2} : I_{2_2}$, ..., $i_{2_{|I_2|}} : I_{2_{|I_2|}}$ -> O_2 *)
...
(* F_n : $i_{1_n} : I_{1_n}$, $i_{1_n} : I_{1_n}$, ..., $i_{n_{|I_n|}} : I_{n_{|I_n|}}$ -> O_n *)
.....
typedef $OBJECT$ **is**
.....
opns
F_1 : I_{1_1}, I_{1_2}, ..., $I_{1_{|I_1|}}$ -> O_1
F_2 : I_{2_1}, I_{2_2}, ..., $I_{2_{|I_2|}}$ -> O_2
.....
F_n : I_{1_n}, I_{1_n}, ..., $I_{n_{|I_n|}}$ -> O_n
.....
endtype endspec

According to this rule, function Fun_3 introduced in Figure 6.6 is formalized as $Fun_3 : out_{64} : Out_{64} \rightarrow Out_{33}$ in the corresponding refined algebraic specification. In addition to the function declaration, pre- and postconditions shall also be specified for the internal functions.

The splitting/aggregation data flow refinements described in SRFMs (denoted by solid box) map an abstract sort into a set of sorts, which is similar to a *record* or *struct* type definition in commonly used programming languages. The formalization is achieved by redefining the abstract sort in terms of a set of LOTOS sorts, operations, and equations. However, there is a significant difference between the splitting and aggregation data flow refinements. The composition of aggregate data flows form a data item that will be supplied to a process to perform a data transaction. Therefore, all the inflows, and only those inflows, shall constitute the data components for the

refined data sort. This feature of data aggregation introduces a constraint to the functional models: if there are multiple data aggregations, given in functional models, for the same outflow data item, then their inflows must be identical to each other. The constraint requires that there only be one refinement for every data item. Unlike data aggregation, when a data is split into a set of outflows, it does not necessarily imply that the outflow data items cover all data components of the split data type. It only specifies that the split data item shall, at least, contain the data items to which the outflow data items correspond. Thus a specific data splitting cannot itself be formalized as a record data type. Instead the union of data splitting for the same split data item shall be formalized in terms of record data items.

FFR-13 If a data flow D is aggregated into a set of data flows, D_1, D_2, ..., D_n, then D is redefined/refined as a set of operations that constitute D from D_1, D_2, ..., D_n and single out D_1, D_2, ..., D_n from D.
For every data flow D that is aggregated into a set of data flows, D_1, D_2, ..., D_n, create the following expression

```
sorts
   D_1, D_2, ..., D_n
opns
   D : D_1, D_2, ..., D_n ⟶ D
   get_d_1 : D ⟶ D_1
   get_d_2 : D ⟶ D_2
   ......
   get_d_n : D ⟶ D_n
eqns
   forall d_1 : D_1, d_2 : D_2, ..., d_n : D_n
   ofsort D_1
      get_d_1(D(d_1, d_2, ..., d_n)) = d_1
   ofsort D_2
      get_d_2(D(d_1, d_2, ..., d_n)) = d_2
      ......
   ofsort D_n
      get_d_n(D(d_1, d_2, ..., d_n)) = d_n
```

FFR-14 All the data splittings for a data flow D together with D is redefined/refined as a record data type.
For data flow D and all the sets of data flows, S_1, S_2, ..., S_n, into which D is split, if data items D_1, D_2, ..., D_m cover all the data items in S_i $(1 \leq i \leq n)$ and do not contain redundant data items, then create the following expression

```
sorts
   D_1, D_2, ..., D_n
opns
   D : D_1, D_2, ..., D_n ⟶ D
   get_d_1 : D ⟶ D_1
   get_d_2 : D ⟶ D_2
   ......
   get_d_n : D ⟶ D_n
eqns
   forall d_1 : D_1, d_2 : D_2, ..., d_n : D_n
   ofsort D_1
      get_d_1(D(d_1, d_2, ..., d_n)) = d_1
   ofsort D_2
      get_d_2(D(d_1, d_2, ..., d_n)) = d_2
      ......
   ofsort D_n
      get_d_n(D(d_1, d_2, ..., d_n)) = d_n
```

Given formalization rules **FFR-13** and **FFR-14**, the SRFM in Figure 6.7 will result in a redefinition of `Out1` and `In1` (assuming that there is only one data splitting for `In1`). The diagram is formalized in Figure 6.20.

```
sorts
  Out1, Out14, Out18, In1, In11, In12
opns
  Out1 : Out14, Out18 -> Out1
  get_out14 : Out1 -> Out14
  get_out18 : Out1 -> Out18
  In1  : In11, In12 -> In1
  get_in11  : In1 -> In11
  get_in12  : In1 -> In12
eqns
  forall out1: Out1, out14: Out14, out18: Out18, in1: In1, in11: In11, in12: In12
  ofsort Out14
     get_out14(Out1(out14,out18)) = out14;
  ofsort Out18
     get_out18(Out1(out14,out18)) = out18;
  ofsort In11
     get_In11(In1(in11, in12)) = in11;
  ofsort In12
     get_In12(In1(in11, in12)) = in12;
```

Figure 6.20: The formalization of the refined data items shown in Figure 6.7

The axiom presented in Figure 6.20 states that: (1) $S1$ is fully implemented in terms of $O_3.S_2$ and $O_9.S_3$; (2) the input of S_1 is split into input for $O_3.S_2$ and $O_9.S_3$; (3) the output of S_1 comprises the outputs from $O_3.S_2$ and $O_9.S_3$.

The duplicator/selector data flow refinement requires or implies that the involved data types are mutually convertible. So that for any pair of mutually convertible data types, D_1 and D_2, two operators, $D_1\_2\_D_2$ and $D_2\_2\_D_1$, and corresponding axioms are introduced to describe the mutual convertibility. The difference between

a duplicator and a selector lies in that the former can have multiple effective outflows while the latter can have only one effective inflow at a given time. The mutual exclusiveness between the inflows of a data selector has direct impact on its semantics and the formalization. This issue will be further addressed in the discussion of the formalization of service composition.

FFR-15 If a data flow D is duplicated to or selected from a set of data flows, D_1, D_2, ..., D_n, operators $D\_2\_D_1$, $D_1\_2\_D$, $D\_2\_D_2$, $D_2\_2\_D$, ..., $D\_2\_D_n$, and $D_n\_2\_D$, in the formats of

opns
$D\_2\_D_1$: $D \rightarrow D_1$
$D\_2\_D_2$: $D \rightarrow D_2$
...
...
$D\_2\_D_n$: $D \rightarrow D_n$

are introduced in the refined algebraic specification. In addition, the following axioms are used to specify the mutual convertibility:

eqns
forall x: D
ofsort D
$D_1\_2\_D$ ($D\_2\_D_1$ (x)) = x;
$D_2\_2\_D$ ($D\_2\_D_2$ (x)) = x;
....
$D_n\_2\_D$ ($D\_2\_D_n$ (x)) = x;

Given formalization rule **FFR-15**, the SRFM in Figure 6.8 will introduce operators `In1_2_In11`, `In11_2_In1`, `In1_2_In12`, `In12_2_In1`, `Out1_2_Out14`, `Out14_2_Out1`, `Out1_2_Out18`, and `Out18_2_Out1`. The diagram is formalized in Figure 6.21.

Formalization of service composition.

The SRFM only presents which aggregate object services and internal functions are composed to implement a higher-level object service and illustrates how information flows among services and functions. The control information and the order in which

```
opns
  In1_2_In11:    In1 -> In11
  In11_2_In1:    In11 -> In1
  In1_2_In12:    In1 -> In12
  In12_2_In1:    In12 -> In1
  Out1_2_Out14:  Out1 -> Out14
  Out14_2_Out1:  Out14 -> Out1
  Out1_2_Out18:  Out1 -> Out18
  Out18_2_Out1:  Out18 -> Out1
eqns
   forall x: In1, y: Out1
   ofsort In1
     In11_2_In1(In1_2_In11(x)) = x;
     In12_2_In1(In1_2_In12(x)) = x;
   ofsort Out1
     Out14_2_Out1(Out1_2_Out14(y)) = y;
     Out18_2_Out1(Out1_2_Out18(y)) = y;
```

Figure 6.21: The formalization of the data duplicator and selector shown in Figure 6.8

the services and functions are activated are beyond the scope of functional models; they are described in dynamic models. However, the functional model does capture the data dependency relationships between the services provided by aggregate objects. Since each service has at most one output data flow and circular data flow is not allowed, each SRFM forms an acyclic directed graph where services are depicted as vertices and data flows as directed edges. Therefore, algebraic axioms can be introduced to describe the constraints on the hierarchical acyclic directed structure in which the aggregate object services are related with each other. Because the services are specified in terms of pre- and postconditions, we pursue the use of the pre- and postconditions as a means to impose design constraints.

The pre- and postconditions of two services, suppose s_1 and s_2, that share a common data flow are related. If the output data flow of s_1 serves as an input data

flow of s_2, a basic constraint is that the postcondition of s_1 does not imply *false* of the precondition of s_2, in other words, they are not contradictory. Thus we impose a verification obligation upon SRFMs to check that the pre- and postconditions of two adjacent services do not form a contradiction.

FFR-16 The post- and preconditions of two adjacent services in a SRFM shall not form a contradiction. For any two services s_1 and s_2 in a SRFM, if (1) the output of s_1 serves as an input of s_2, (2) p_{s_1} and q_{s_1} are the pre- and post conditions for s_1, and (3) p_{s_2} is the precondition for s_2, create and prove the following expression

$(p_{s_1} \Rightarrow ((q_{s_1} \text{ and } p_{s_2}) \text{ eq false})) = \text{false}$

Figure 6.22 and 6.23 show the specifications for *Storage* and *Compressor* with pre- and postconditions for the corresponding services. Given the SRFM in Figure 6.13, according to rule **FFR-16**, the postcondition of service `Storage.Retrieve` and the precondition of service `Compressor.Decompress` can be analyzed to check the consistency of function refinement. Since the result of `Storage.Retrieve` will be fed into `Compressor.Decompress`, given the pre- and postconditions of the two services, the constraint is

```
(s neq empty =>(result=retrieve(s,k) and result eq undef_d) eq false)=false.
```

There are basically two approaches to perform the consistency checking. One is to conduct a manual proof; the other is to use proof tools. Unfortunately, there is currently no sophisticated proof tool for the algebraic specifications of LOTOS. Since ACT ONE is a subset of the LSL of Larch, we translated the LOTOS algebraic specifications into LSL specifications and used Larch Prover (LP) [19] to check the consistency of pre-and postconditions. The corresponding LSL specifications are given in Appendix F. Figure 6.24 gives the LSL specification for the refined *Disk Manager*.

```
specification Storage [Insert, Retrieve] (s: S) : noexit :=
   (* Insert    : d: Data, k: Key -> Void *)
   (* requires d eq undef_Data = false and k eq undef_Key = false *)
   (* modifies s *)
   (* ensures s' eq insert(s^, d, k) = true *)
   (* Retrieve  : k: Key -> d: Data *)
   (* requires s eq empty = false *)
   (* ensures result = retrieve (s, k) *)
type Storage is Disk_Manager
    sorts
      S, Data, Key
    opns
      empty     : -> S
      insert    : S, Data, Key -> S
      retrieve  : S, Key -> Data
    eqns
      forall s: S, data: Data, k1, k2: Key
      ofsort Data
         retrieve (empty, key) = undef_data;
         retrieve (insert(s, data, k1), k2) = if k1=k2 then data
            else retrieve(s, k2)
endtype
endspec
```

Figure 6.22: The Full LOTOS specification for *Storage* with pre- and postconditions for services

The proof obligation is simplified to `~(s=empty)=>~(retrieve(s,n)=undef_d)` and given in the *implies* clause (the logic simplification can also be assisted by tools, such as PVS [81]). Figure 6.25 shows the result of the proof by using LP.

By using the *induction* proof method provided by LP, we have reached a point that shows `Current subgoal: false` (in Figure 6.25) that means there are cases that yield false for the constraint. The proof detects an inconsistency between the postcondition of *Storage.Retrieve* and the precondition of *Compressor.Decompress*, thus requiring us to perform a further inspection into the problem. This problemm is further discussed in Chapter 8 in the context of the overall development process.

```
specification Compressor [Compress, Decompress] (c: C) : noexit :=
   (* Compress     : d: Data -> Data *)
   (* requires d eq undef_Data *)
   (* ensures result = compress(d) and size(result) le size(d) *)
   (* Decompress   : d: Data -> Data *)
   (* requires d eq undef_Data *)
   (* ensures result = decompress(d) and size(result) ge size(d) *)
type Storage  is Disk_Manager, NaturalNumber
    sorts
      C (* Distinguished sort *)
    opns
      compress : Data -> Data
      decompress : Data -> Data
      size : Data -> Nat
    eqns
      forall data: Data
      ofsort Bool
         size(data) ge size(compress(data)) = true;
         size(data) le size(decompress(data)) = true
endtype
endspec
```

Figure 6.23: The Full LOTOS specification for *Compressor* with pre- and postconditions for services

```
Disk_ref (D, K): trait
    includes Storage(D,K), Compressor(D)
    implies
      ∀ s: S, n: N
      ~(s=empty) => ~(retrieve(s,n) = undef_d);
```

Figure 6.24: The LSL specification for the refined *Disk Manager*

```
LP2: display proof

Conjecture Disk_refTestTheorem.1: ~(s = empty) => ~(retrieve(s, n) = undef_d)
  Attempting a proof by depth 2 structural induction on 's'
Level 2 subgoal 2 (basis step) for proof by induction on s:
  ~(insert(empty, n1, n2) = empty)
    => ~(retrieve(insert(empty, n1, n2), n) = undef_d)
  Current subgoal: ~((if n = n2 then n1 else undef_d) = undef_d)
  Attempting a proof by depth 2 structural induction on 'n'
Level 3 subgoal 1 (basis step) for proof by induction on n:
  ~((if 0 = n2 then n1 else undef_d) = undef_d)
  Attempting a proof by depth 2 structural induction on 'n1'
Level 4 subgoal 1 (basis step) for proof by induction on n1:
  ~((if 0 = n2 then 0 else undef_d) = undef_d)
  Attempting a proof by depth 2 structural induction on 'n2'
Level 5 subgoal 2 (basis step) for proof by induction on n2:
  ~((if 0 = succ(0) then 0 else undef_d) = undef_d)
  Current subgoal: false
  Attempting a proof by normalization

LP3:
```

Figure 6.25: The proof of the example constraint using LP

6.5 Integration with Object and Dynamic Models

The functional model is integrated with the object and dynamic models in terms of the underlying formalization rules. The integration is three-fold. First, both OFMs and SRFMs are derived in the context of object models. This approach to modeling provides a degree of integration between the two models.

Second, the integration is achieved by sharing common language constructs among specifications derived from the object, functional, and dynamic models. The integration includes:

- Operators defined in algebraic specifications are used to define pre- and post-conditions for services.
- Services defined by functional models serve as actions and activities in state diagrams.

- Algel
repla
enab
- The
and
- Servi
gate
- The
const
- The
tiate

Third,

ing to the

6.6 S

In this ch

integrated

formally a

for derivi

pre- and p

for service

high-level

and valida

for pre- an

conflicts th

- Algebraic specifications used to describe the postconditions of a service can replace the service reference in the process algebraic specification in order to enable symbolic simulations.
- The data types used in process algebras are defined in the functional models and are formalized in terms of algebraic specifications.
- Services (externally accessible operators) defined in functional models serve as gate lists for process algebras.
- The pre- and postconditions specified during functional modeling impose extra constraints to the corresponding actions in the dynamic models.
- The postconditions specified during functional modeling can be used to instantiate the corresponding actions in the dynamic models.

Third, the integration is achieved by composing the SRFMs hierarchically according to the system structure specified in the object model.

6.6 Summary

In this chapter, we showed how the functional model is modified, formalized, and integrated into object-oriented design. Formalization rules are also introduced to formally and rigorously specify both the OFMs and SRFMs. In addition, guidelines for deriving algebraic specifications are given. Based upon algebraic specifications, pre- and postconditions are introduced to describe the requirements and constraints for services. The pre- and postconditions of services enable symbolic execution of the high-level design model thus providing a means to perform simulation, verification, and validation during the early phases of software development. Proof obligations for pre- and postconditions of adjacent services in SRFMs serve as a means to detect conflicts that may be introduced into design during refinement.

Cha

Mod

We have,
dynamic
explicitly
dynamic
ization.

7.1

In genera
formal sp
structs al
primitive
our forma
imposing

Chapter 7

Model Integration and Analysis

We have, in fact, implicitly addressed the integration of the object, functional, and dynamic models through the discussion of their formalization rules. This chapter explicitly summarizes the integration of the three models and discusses the static and dynamic analysis of inter-model and intra-model integration enabled by the formalization.

7.1 Integration of the Three Models of OMT

In general, the use of LOTOS does not automatically lead to an integration of the formal specifications of the object, functional, and dynamic models, though its constructs allow algebraic specifications and process algebras to share common language primitives thus providing a possibility to integrate the two types of specifications. In our formalization, the integration of the three models is achieved through explicitly imposing certain relationship upon the three models as well as the constructs of al-

gebraic specifications, predicate specifications, and process algebras. The integration is three-fold.

First, the functional and dynamic models are derived in the context of object models. By associating a functional model and a dynamic model to every individual object, we resolved the conflicts between the models with respect to their respective development philosophies. The functional model and dynamic model describe the services and dynamic behavior of their corresponding object, respectively. Thus the modeling technique provides a degree of integration between the three models. A formal specification for a specific object shall be derived from three models: object model, object functional model, and dynamic model. The three complementary models of an object describe different aspects of the object and forms a single formal specification.

Figure 7.1 shows an outline of a specification derived from object, functional, and dynamic models. The words in bold fonts are keywords; the words in "< >" pairs are the components of formal specifications.

The components of a formal specification includes:

- **Class name**: *identifies a class of objects that share the same interfaces, attribute types, and behavior.*
- **Services**: *services provided/needed by a class of objects.*
- **Distinguished sort**: *a sort that consist of sorts of all the attributes to a class of objects.*
- **Services specification**: *specifies the signatures and pre- and postconditions of the services (the pre- and postconditions shall be supplied by specifiers).*
- **Algebraic specification**: *specifies the properties that an object may have in terms of algebraic specifications (this part of specifications shall be supplied by specifiers).*

```
specification <class name> [<services>](<distinguished sort>) noexit:=
   (* ..........................*)
   (* <services specifications> *)
   (* ..........................*)
   typedef
      ......
      <algebraic specification>
      ......
    endtype
   <starting state> [<services>] (<distinguished sort>)
   process <state name> [<services>](<distinguished sort>) noexit:=
      ......
      <behavior specification>
      ......
    endproc
endspec
```

Figure 7.1: The format of formal specifications

- **Behavior specification**: *specifies the patterns that an object interacts with other objects.*
- **State name**: *identifies different states in which an object may be during its life cycle.*
- **Starting state**: *the state that an object enters after being instantiated.*

A specification depicts four aspects of a class: identity, interface, functionality, and behavior. **Class name** and the **distinguished sort** together contribute to the identity of classes and objects. **Services** describe the interface of a class. **Services specifications** delineate the functionalities that the services provide. **Behavior specifications, state names**, and the **starting state** capture the behavior of a class. With the exception of a part of the algebraic specifications that specify the properties of objects and the pre- and postconditions that describe services provided by object classes, all other components of the formal specifications can be automatically generated from diagrammatic models according to our formalization rules.

Second, the integration is achieved by sharing common language constructs among specifications derived from the object, functional, and dynamic models. The integration includes:

- The distinguished sort in algebraic specifications serves as state process parameters in process algebras.
- Operators defined in algebraic specifications are used to define pre- and postconditions for services.
- Services defined by functional models serve as actions and activities in state diagrams.
- Operators defined in algebraic specifications can be used to substitute services referenced in process algebras.
- The data types used in process algebras are defined in algebraic specifications.
- Services (externally accessible operators) defined in functional models serve as gate lists for process algebras.
- The parameters associated with operators serve as attributes associated with events.
- The pre- and postconditions specified during functional modeling impose extra constraints to the corresponding actions in the dynamic models.
- The postconditions specified during functional modeling can be used to instantiate the corresponding actions in the dynamic models.

Third, the integration is achieved by composing (1) the dynamic models of concurrent objects hierarchically according to the system structure specified in the object model, and (2) the SRFMs of an object in terms of the services provided by the aggregate objects.

7.2 The Correctness, Consistency, and Completeness of the Formalization Rules

Since both the formalization and integration of the models greatly rely on the formalization rules, it is worthwhile to discuss the correctness, consistency, and completeness of our proposed formalization rules. The informal graphical notations do not have precise semantics. The purpose of formalization rules are two-fold. First, the rules introduce precise meaning to the graphical notations. Second, the rules provide a graphical front-end for the constructs of formal specifications.

Correctness. Although the informal graphical notations do not have precise semantics, they still have underlying general semantics. The formalization rules attempt to give precise meanings to the graphical notations by introducing formal specifications to describe the graphical models. A formalization rule can be considered *correct* if it imposes the *right* semantics to the graphical notations. Because, there is no specific, formal definition for the semantics of graphical notations, we have no means to formally prove the correctness of the formalization rules. Given this situation, the correctness of the formalization rules is pursued in the context of general reasoning, which is justified in the chapters of formalization, the model construction and use presented by Rumbaugh [3], and empirical experience through the case studies.

Consistency. If the formalization rules do not conflict with each other, they can be considered consistent. From the perspective of graphical notations, because each graphical notation is formalized in terms of one language construct of formal spec-

ifications, the formalization rules are consistent from the perspective of graphical notations. From the perspective of formal languages, the consistency of formalization rules can be discussed under two circumstances. First, for the formalization rules whose corresponding constructs of formal specification language are not related with each other in the formal specifications, consistency is automatically inherited from the perspective of formal specifications. Second, for the formalization rules whose corresponding constructs of formal specification language are related with each other, because the relationship is maintained through language semantics developed by Larch and LOTOS, the consistency of the formalization rules can be reduced to the consistency between different types of specifications in Larch and LOTOS. Neither Larch nor LOTOS has addressed this consistency issue. However, the large number of applications of Larch and LOTOS have not reported any inconsistency between different types of formal specifications thus far.

Completeness. Since the rules serve as a bridge between informal graphical notations and language constructs of formal languages, the completeness of the rules can be discussed from two perspectives. From the perspective of formal languages, the rules can be considered complete if every part of a formal specification has corresponding graphical notations from which formal specifications can be automatically generated. From the perspective of graphical notations, the rules can be considered complete if every graphical notation can be described in terms of formal specifications. Because the axioms of algebraic specifications cannot be easily expressed by graphical

notations, our formalization rules only preserve the property of completeness from the perspective of graphical notations.

7.3 Specification Analysis Techniques

The integration and formalization of the three models that describe a system from complementary aspects enable designers to perform analysis tasks by using the derived formal specifications.

LOTOS is a formal abstract description language that can be used to precisely describe a system, abstracting away realization details. This is particularly useful in the design and analysis of the behaviors of systems that contain communicating objects, where the interactions often have complex interdependencies. LOTOS is based upon several mathematical models that make it possible to check the validity of the specification, ensuring the correctness of the design. Because our formalization and integration rules can serve to derive a single, integrated formal specification for an object from its corresponding object, functional, and dynamic models, both static and dynamic analysis tasks can be performed for specification analysis by using available LOTOS tools. Currently, a rich set of LOTOS tools [82] are available for analyzing LOTOS specifications. Among the available tools, TOPO [64, 83], LOLA [84, 85, 86, 87], SMILE [88, 89], and XELUDO [90] for Full LOTOS are promising tools.

- TOPO is a toolset that supports product realization from LOTOS specifications. TOPO includes tools to perform static semantics analysis, specification investigation (cross references, data type dependencies, etc.), and it is able to generate C or Ada code that may be compiled into a prototype.

- LOLA is a transformational and state exploration tool with applications to testing, simulation, and debugging. The transformation functionality allows the generation of the equivalent Extended Finite State Machines (EFSM) [91][1] from given LOTOS specifications. The execution functionality enables designers to simulate LOTOS specifications in a stepwise fashion. The testing functionality calculates the dynamic response of a system specification to a test according to the testing equivalence properly.
- SMILE is a symbolic evaluation tool for LOTOS. It contains functions for the analysis of the abstract data type part of the specification, execution and debugging of the LOTOS specification or parts thereof, transformation of (parts of) the specification into strong equivalent EFSMs.
- XELUDO is a toolset that support specification transformation and execution. The execution functionality allows step-by-step execution of a specification. The symbolic expansion transformation functionality generates equivalent symbolic trees of a specification.

TOPO and LOLA, which cover most of the critical functionalities for LOTOS specification analysis, are two intensively used tools in this research. Our discussion of specification analysis will focus on the functionalities provided by TOPO and LOLA.

7.3.1 Static analysis

A static semantic checker for full LOTOS can check the consistency between the algebraic and behavior specifications, such as operator references, parameter types, and so on. Therefore, given a formal specification that describe a set of corresponding models, inconsistencies among models that are introduced during the the construction of a diagram will be detected by such a semantics checker. LSA (LOTOS semantics analyzer) included in TOPO is an example of such a semantics checker. Figure 5.6 and 5.7 of Chapter 5 show an instance of static analysis performed by LSA.

[1] EFSMs are finite state machines that have output functions associated with transitions or states.

The underlying formal semantics of LOTOS is a Labeled Transition System (LTS) [92, 93]. Based upon the formal semantics described by the LTS, equivalence between behaviors of LOTOS specifications is also defined [92]. The expansion transformations provided by LOLA produces a compressed version of Extended Finite State Machines (EFSM) from a given LOTOS behavior specification. The behavior of the generated EFSM is equivalent to the original LOTOS specification. The effect of an expansion is the removal of the most complex LOTOS operations (e.g., parallel operators) from the specification, producing an equivalent specification in terms of *action prefix*, *behavior choice*, *guards*, and *choice*, etc. This transformation can be used for state exploration, deadlock detection, deriving efficient implementations, etc. Because the dynamic models of the individual objects can be synchronized by parallel operators, we can use the expansion transformation to check if the behavior of the synchronized objects can be expressed in terms of an equivalent EFSM without parallel operators. This transformation can be considered as a static analysis of state exploration. A deadlock detected by the transformation suggests that the behavior of the communicating objects has a synchronization conflict.

7.3.2 Dynamic analysis

Although a static expansion transformation can exhaustively explore all the possible execution paths for a given LOTOS specification, the variables in the given LOTOS specification are treated symbolically during the static analysis. Therefore, potential problems and conflicts caused by variable instantiation cannot be detected by static

analysis. By running a test (a formal specification of desired behavior) in parallel with the target specification, we are able to instantiate the variables in the given specification thus dynamically analyzing its behavior. This type of dynamic analysis is based on the definition of *Testing Equivalence* [94] of LOTOS specifications. If the behaviors exhibited by two LOTOS specifications cannot be distinguished from external observation, then the specifications are considered to be *testing equivalent.* Therefore, if none of the execution paths of a test running in parallel with the specification under test results in a deadlock, then the test and the specification under test can be considered testing equivalent to one another. That is, we can assert that the specification under test satisfies (or passes) the test.

LOLA provides functionality that enables designers to analyze the behavior of a given specification dynamically in terms of *Testing Equivalence* [84]. Tests are passed by specifying a test process and obliging it to synchronize with the behavior under test. The successful termination of a test in a given execution consists in reaching a state where the termination event (usually specified as *success*) is offered. A test does not terminate in a given execution if it reaches a deadlock situation. The results of the test are classified into three classes: *Reject*, *Must Pass*, and *May Pass.* If all the execution paths terminate successfully, then the test is a *Must Pass.* If there is least one execution that leads to the successful termination, then the test is considered *May Pass.* Otherwise, the test is rejected, the specification under test will not satisfy/pass the test under any means.

We can use this functionality provided by LOLA to check the behavior specifications derived from design models. The ability to specify test cases enables us to

perform dynamic analysis on the design models. Given this analysis tool, test cases can be specified to check the property of a given specification. The variables in the specification under test can also be instantiated with concrete values to detect communication conflicts between interacting objects. Furthermore, if the specifications derived from models of different levels of abstraction both pass the same set of testing processes, they can be considered testing equivalent to each other in the context of the given set of testing processes. That is, their design models are considered consistent in terms of testing equivalence.

The executable nature of LOTOS behavior specifications enables designers to interactively execute any LOTOS specification in a stepwise fashion, assigning values to the variables defined in the specification and examining the response of the specification. We can use this functionality to simulate the behavior of design models in order to achieve a better understanding of the design as well as a means to communicate with customers and team members of software development. In addition, we can use the interactive execution to detect defective design models. If either a static or a dynamic analysis of the behavior of a given specification leads to a deadlock, we can recover the paths to deadlocks by investigating the log files, then run the specification interactively in accordance with the deadlock paths. Such a stepwise execution may help to identify the cause of deadlock.

7.4 Analysis of Model Integration

The specification analysis techniques discussed in the previous section can be used to perform analysis of model integration.

7.4.1 Inter-model analysis

Based on our formalization rules, beginning with the initial stages of modeling, the models are no longer considered disjoint. A static semantic checker for full LOTOS, LSA (LOTOS semantics analyzer), included in TOPO [64, 83], checks the consistency between the algebraic and behavior specifications, such as operator references, parameter type, and so on. For example, an error in the behavior specification in Figures 5.6 and 5.7 was detected by LSA, where the transcript of the semantic checking with LSA is given in Figure 7.2. Further investigation reveals that the error was caused by an incorrect operator reference in the behavior specification (derived from the state diagram), *retrieve(s)* that should have been *retrieve(s, k)*. Therefore given a formal definition for the model integration, inconsistencies between models that may be introduced in the construction of a diagram can be detected when the corresponding formal specifications are analyzed.

```
lsa -l /home/wangyi/Research/Tools/TOPO/stdlib/mod-is
-p com2 com2.lfe
com2.lot:116: lsa: undefined operation: retrieve
***  lsa: errors detected
```

Figure 7.2: The error in the behavior specification of the *Storage* detected by LSA

7.4.2 Intra-model analysis

Model integration is also achieved by composing concurrent objects hierarchically according to the system structure that is specified in the object model.

The benefits brought by this integration include the ability to refine both the structure and the behavior of a system in a parallel, hierarchical, and systematic fashion. In addition, a behavior specification, obtained by composing concurrent aggregation objects, can be checked against the behavior specification of their corresponding individual aggregate objects to guarantee the consistency between the behavior specifications during the refinement and development processes. For example, prior to the design stage, the *Disk Manager* may have a simple behavior specification with *input* and *output* as the only gates. In order to validate the behavior specification, a set of *test* processes, can be written in LOTOS to simulate the environmental events. Then the same set of *test* processes can also be used to test the refined and *concurrent* behavior specification, where the refined specification must also satisfy all the *test* processes before it can be accepted.

The testing equivalence between two process algebras means that one cannot detect the difference of the behavior from outside the processes in terms of testing. Currently, we term intra-model consistency as the test equivalence in terms of the LOTOS specification language. This type of intra-model consistency checking is already supported by several tools designed for Full LOTOS, such as LOLA [84].

The process "*accept_test*" in Figure 7.3 was used to test the behavior of the *Disk Manager* before it was decomposed into *Storage* and *Compressor*. This testing pro-

cess can also be used to test the behavior specified in Figure 5.10 to check if it is also satisfied by the modified specification. In process *accept_test*, three data items associated with keys 0, 1, and 2, respectively are given, then two items of data with keys 0 and 2 are requested.

```
process accept_test [input, output, success] : noexit :=
   let x:D=Hex(b),y1:K=succ(0),y0:K=0,y2:K=succ(succ(0)) in
      input !x !y0;
      input !x !y1;
      input !x !y2;
      output !y0; output ?dx: D;
      output !y2; output ?dx: D;
      success;
      stop
endproc
```

Figure 7.3: The testing process used to test the behavior of *Disk Manager*

Figure 7.4 captures the transcript of *oneexpand*, a testing command in LOLA, that composes the behavior of *disk manager* in parallel with the testing process *accept_test* through gates *input and output*, and executes a random trace of the composed process. The *success* event that appears in the tenth step shows that the test successfully terminates.

Since the behavior specification of *Disk Manager* only has a small number of states, we can exhaustively explore all the possible executions. Figure 7.5 shows such a test. (For those cases with a large number of states, LOTOS has facilities for performing partial analysis.) There are three types of termination for the testing analysis: *must*, *may*, and *reject*. If the test terminates for every execution of the composed process, then the result is a *must* pass. If the test does not terminate for any execution, then the test is a *reject*. If the test terminates for at least one execution, then it is a

```
lola> oneexpand 10 success accept_test -v
oneexpand 10 accept_test 0 1 -v

  Composing behaviour and test :

     accept_test [input,output,success]
   |[input,output]|
     (hide com,dec,ins,ret,delete in
        storage [ins,ret,delete]
      |[ins,ret,delete]|
          disk_manager_1 [input,output,com,dec,ins,ret,delete]
        |[com,dec]|
          compression [com,dec]
     )

    1  input  ! hex(b) of d ! 0 of k;
    2  input  ! hex(b) of d ! succ(0) of k;
    3  input  ! hex(b) of d ! succ(succ(0)) of k;
    4  output  ! 0 of k;
    5  i;
    6  output  ! decompress(data(entry(compress(hex(b)),0)));
    7  output  ! succ(succ(0)) of k;
    8  i;
    9  output  ! decompress(data(entry(compress(hex(b)),
                 succ(succ(0)))));
   10  success;

    Process Test = accept_test
    Test result  = SUCCESSFUL EXECUTION.

       Transitions generated = 10
```

Figure 7.4: Using *accept_test* testing process to test the specified *Disk Manager*

may pass result. The result of the test in Figure 7.5 is a *may* pass. In this test, five executions were performed, where two of the executions were successful, and three were aborted due to deadlock.

In Figure 7.6, we conducted a step-by-step interactive simulation of the original behavior specification composed with the test process to determine the source of the deadlock problems. In the first seventeen steps of the simulation, there were no choices for the developer to make. At the eighteenth step, three choices were presented to

```
lola> testexpand -1 success accept_test -v -y
testexpand -1 success accept_test -v 0 -y

 Rewriting expressions in the specification.
 Rewriting done.
 Analysing unguarded conditions.
 Analysis done.

  Composing behaviour and test :

     accept_test [input,output,success]
   |[input,output]|
     (hide com,dec,ins,ret,delete in
        storage [ins,ret,delete]
      |[ins,ret,delete]|
          disk_manager_1 [input,output,com,dec,ins,ret,delete]
        |[com,dec]|
          compression [com,dec]
     )

                Exploration Tree             |Transits| States
   0 - 10 / 7 - 9 / 7 - 10 / 4 - 6 / 4 - 8/ |      18|     17

   Analysed states       = 17
   Generated transitions = 18
   Duplicated states     = 0
   Deadlocks             = 3

   Process Test = accept_test
   Test result  = MAY PASS.

   5 executions analysed:

                  successes = 2
                      stops = 3
                      exits = 0
              cuts by depth = 0
```

Figure 7.5: An exhaustive test using process *accept_test*

the developer. This nondeterminism is introduced due to gate hiding in the behavior specification of *Disk Manager*. Hiding the gates transformed the explicit event into an internal event *i*. The three internal events appearing in a *choice* construct resulted in nondeterminism. The first choice may lead to a success event; the last two choices will result in deadlock. This interactive simulation analysis answers why deadlocks were detected in the previous testing. However, since the detected deadlocks were caused by hidden gates, which were only hidden for the purposes of testing, it is not a real problem and will not cause deadlocks when the gates are not hidden.

```
lola> step success accept_test

 Rewriting expressions in the specification.
 Rewriting done.
 Analysing unguarded conditions.
 Analysis done.

  Composing behaviour and test :
     accept_test [input,output,success]
   |[input,output]|
     (hide com,dec,ins,ret,delete in
        storage [ins,ret,delete]
      |[ins,ret,delete]|
          disk_manager_1 [input,output,com,dec,ins,ret,delete]
        |[com,dec]|
          compression [com,dec]
     )
1.  [  1]  input  ! hex(b) of d ! 0 of k;
<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> 1
  ==> input  ! hex(b) of d ! 0 of k;
    .......... (omitted)
<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> Trace
1.  [  1] - input  ! hex(b) of d ! 0 of k;
2.  [  1] - i; (* com ! hex(b) of d *)
3.  [  1] - i;
4.  [  1] - i; (* com ! compress(hex(b)) *)
5.  [  1] - i; (* ins ! compress(hex(b)) ! 0 of k *)
6.  [  1] - input  ! hex(b) of d ! succ(0) of k;
7.  [  1] - i; (* com ! hex(b) of d *)
8.  [  1] - i;
9.  [  1] - i; (* com ! compress(hex(b)) *)
10. [  1] - i; (* ins ! compress(hex(b)) ! succ(0) of k *)
11. [  1] - input  ! hex(b) of d ! succ(succ(0)) of k;
12. [  1] - i; (* com ! hex(b) of d *)
13. [  1] - i;
14. [  1] - i; (* com ! compress(hex(b)) *)
15. [  1] - i; (* ins ! compress(hex(b)) ! succ(succ(0)) of k *)
16. [  1] - output  ! 0 of k;
17. [  1] - i; (* ret ! 0 of k *)
<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> Menu
18. [  1]  i; (* ret ! data(entry(compress(hex(b)),0)) *)
    [  2]  choice d_189:d []
           i; (* com ! d_189 *)
    [  3]  choice d_191:d []
           i; (* dec ! d_191 *)

<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?>
```

Figure 7.6: An interactive simulation of the process composed for testing

Chapter 8

Design Process

If the objective of software requirements analysis is to achieve a comprehensive understanding of software requirements for the target system, then the purpose of design is to *apply various techniques and principles for the purpose of defining a device, a process, or a system in sufficient detail to permit its physical realization* [95]. Given requirements specifications derived from the requirements analysis stage, software design is the first step in the development process that involves a significant amount of engineering effort. The primary objective of software design is to build models/specifications with sufficient detail such that there is enough information to develop the corresponding source code. A design process typically incorporates intuition and judgment based on the experience of engineers in building similar systems, a set of principles and/or heuristics that guide the evolution of the system, a set of criteria that enable quality to be judged, and an iterative process that ultimately leads to a final design representation [6].

Several design methods [3, 7, 41, 51, 72, 96] have been proposed during the last three decades. However, unlike other engineering disciplines, such as mechanical or electronic engineering, where strict mathematics is heavily involved in the design, little or no mathematics is explicitly employed by most of the proposed software design methods. This can be partly due to the nature of concrete mathematics, the foundation of computing, which is more abstract and perceived to be more difficult to master and manipulate than general mathematics. We will not go into an in-depth discussion about concrete mathematics at this point. Nonetheless, mathematics is the only means that we have, thus far, to achieve precise descriptions of software systems. The application of formal methods to software engineering attempts to introduce mathematical rigor as well as systematic methods into software development. The well-defined notations used by the formal methods are amenable to automated processing for numerous analysis tasks [2], including verification of correctness of resulting systems. Based on investigations by Dart *et al.* [97] and Brinkkemper [98], Fraser *et al.* [5] classified design methods into three categories according to the degree of formal methods that are involved.

- *The informal category includes the techniques that do not have complete sets of rules to constrain the models that can be created. Natural language and unstructured pictures are typical examples.*
- *The semiformal category includes the techniques that have defined syntax. Typical instances are diagrammatic techniques with precise rules that specify conditions under which constructs are allowed and textual and graphical descriptions with limited checking facilities. Example methods include entity-relationship diagrams and variations on data/control flow diagrams [97, 98].*
- *The formal category includes the techniques that have rigorously defined syntax and semantics. There is an underlying theoretical model against which a*

description expressed in a mathematical notation can be verified. Specification languages based on predicate logic are typical instances. Example methods include Petri nets, state machines, Z (Zed), and VDM [97, 98, 99].

Although formal specifications are gaining increasing attention as a means to rigorously document requirements and design information given that the well-defined notations are amenable to automated processing for numerous analysis tasks [2], most of the popular design methods fall into the *semiformal* category. This phenomenon can be partly attributed to the fact that (1) formal methods research has largely focused on the development of formal notation and inference rules, and (2) the notation and the conceptual grammar of formal specification languages require familiarity with discrete mathematics and symbolic logic that most practicing software engineers, designers, and implementors do not have [5].

Most design techniques in the semiformal category use various diagrammatic techniques with formal syntax for graphical notations to specify design models. However, the lack of formal semantics of graphical notations deprives designers of analysis techniques that can help designers to detect and eliminate design error during the early stages of software development process. In order to take advantages of both the intuitive diagrammatic and rigorous formal techniques, the integration of the two approaches is clearly motivated.

This chapter proposes a design method, based upon our formalization and integration rules in the previous chapters, that provides designers with both intuitive diagrammatic and formal methods techniques. The remainder of the chapter is organized as follows. Section 8.1 gives a brief discussion of strategies for incorporating

formal methods into software development and an overview of our proposed design approach. Section 8.2 briefly describes the graphical models, the formal specifications used to describe these models, and the proposed design process. Section 8.3 gives a detailed presentation about the models and the design process through the model and design of an example. Section 8.4 gives the summary of this chapter.

8.1 Overview of Strategies Proposed Approach

In this section, a brief discussion of the strategies that incorporate formal specifications into design is given to serve as background and provide context for our research. Then a short description of the proposed approach is presented.

8.1.1 The integration strategies

Fraser *et al.* [5] developed a taxonomy for describing and assessing strategies that incorporate formal specification techniques in software development. Our design process adheres to the taxonomy. The framework consists of two dimensions: *formalization support* and *formalization process.* From the *formalization support* dimension, the techniques are divided into *computer assisted* and *unassisted* categories according to whether the formalization process is supported by computer or not. In the formalization process dimension (shown in Figure 8.1), there are two approaches: direct and transitional. The transitional approach can be further divided into sequential and parallel successive refinement approaches. The definitions are as follows:

Direct: directly moving from informal requirements to formal specifications without the use of (semiformal) intermediate specifications.

Transitional: moving from informal requirements to formal specifications through the use of (semiformal) intermediate representations. There are two further refinements of transitional approaches.

Sequential: formal specifications are derived from a final set of semiformal models.

Parallel successive refinement: the semiformal specifications are produced through successive refinements, where the formal specifications are derived from the semiformal specifications in parallel with the refinement process.

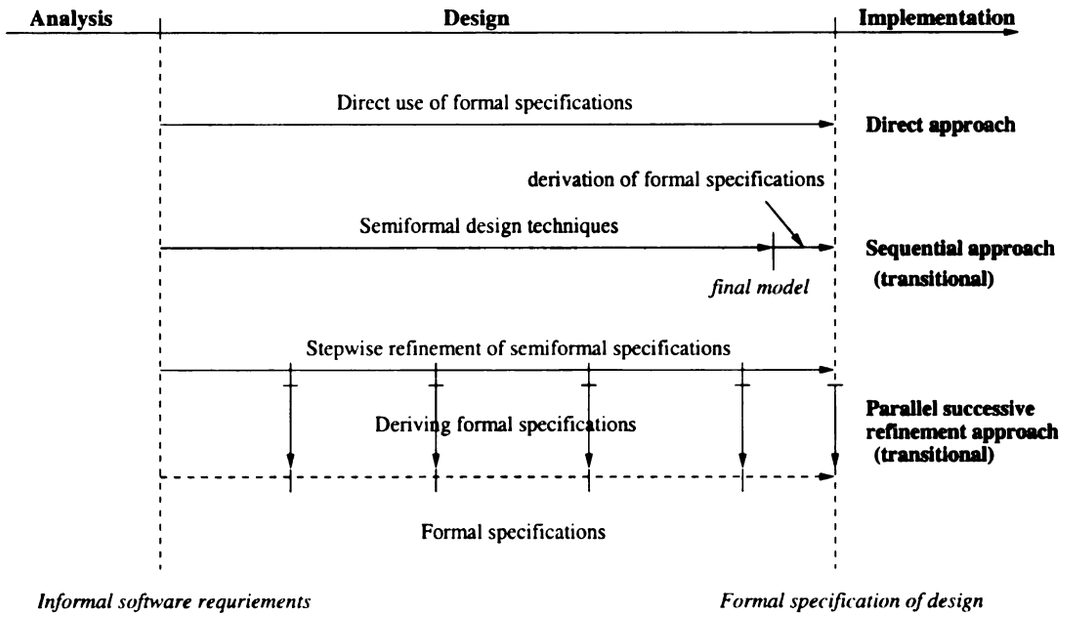

Figure 8.1: The *formalization process* dimension of the suggested framework

Among the strategies that produce formal specifications, the computer-assisted, parallel approach appears to be the most promising. During the design process of this type of approach, the semiformal methods guide the stepwise refinement. At each design step, the computer-assisted supporting tools facilitate the generation of

formal specifications from newly refined semiformal models. The derived, refined formal specifications can be verified against the formal specifications of the higher-level abstraction in order to eliminate design inconsistencies. Detected problems may also be traced back to the semiformal models. Thus the techniques of this category have the potential of enabling semiformal and formal specifications to be developed in a synergistic fashion during the design refinement process. However, to date, there do not appear to be examples of a computer-assisted transitional-parallel formalization strategy [5].

8.1.2 The proposed approach

The objective of our investigation is to propose a design process that employs the computer-assisted transitional-parallel formalization strategy. The Object Modeling Technique (OMT) [3] is the semiformal design technique upon which our proposed method is based.

We formalized the three OMT models and integrated their corresponding formal specifications in terms of the underlying semantics in Section 2.4 and Chapters 4, 5, 6, and 7. Based upon the proposed formalization rules, programs can be written to generate formal specifications directly from the semiformal models of OMT. The object model is formalized in terms of algebraic specifications [27] that provide a framework of the formal specifications; the formalization of the functional model [75] introduces formal interface functions and pre- and postconditions for services provided by the system; the dynamic model is formalized in terms of process algebras [62, 63]

that capture the dynamic behavior of the objects. The formal specifications generated from different models are complementary, and they, together, form a single formal specification of an object. Based on the formalization and integration of the three models and the design process of OMT, we have proposed a stepwise design process that conducts design activities in an iterative fashion.

8.2 The Proposed Rigorous Design Process

In this section, a formal, rigorous development paradigm for design is proposed. The design paradigm explicitly addresses the consistency between the formal specifications of two adjacent levels of abstraction thus enabling stepwise refinement and consistency checking.

8.2.1 Format of formal specifications

Figure 7.1 in Chapter 7 shows an outline of a specification that describes an object class, including its identity, interface, functionality, and behavior. The formal specifications that describe the OMT models serve as the foundation of our proposed design process that employs a parallel transitional approach that incorporates formal specifications into design.

8.2.2 Design models

Figure 8.2 shows the graphical models and the order in which they should be developed during the development process. In this figure, a rectangle represents a model

to be developed, an arrow describes a dependency relationship, a dashed arrow indicates iteration, the numbers in the upper left hand corner indicate the order that the model should be developed in a given iteration. The proposed development paradigm includes iterations of model refinement where each subsequent iteration adds more design detail. In Figure 8.2, the first three models are system level object, functional, and dynamic models that describe the structural, functional, and dynamic behavioral aspects of a system, respectively. The subsequent three models capture information that is similar to the first three models, except they depict the structural, functional, and behavioral features of objects at a level more detailed than that at the system level. The *service refinement functional model* refines a service of an object in terms of the services provided by its aggregate objects at a lower level of abstraction. The refined dynamic model refines the dynamic behavior of an object in terms of its interaction with objects at a lower level of abstraction. The composed parallel dynamic model is a composition of a set of dynamic models of objects that are synchronizing with each other. Once the composed parallel dynamic model at a certain level of abstraction is obtained and analyzed, the development enters the next iteration of model refinement and derivation. Table 8.1 describes the models and the formal specifications that are derived from the corresponding models. The double lines are used to separate the (1) models that describe the system as a whole, (2) individual aggregate objects, and (3) the composition of the aggregate objects.

Models	Use	Formal Specifications
(*System-Level*) **Object Model**	depicts the static structure of a system	derives class name and class attributes
(*System-Level*) **Functional Model**	contains services provided by a system	derives services and service specifications
(*System-Level*) **Dynamic Model**	specifies all the possible interaction patterns of a system	derives state names, behavior specifications, and starting states
Refined Object Model	decomposes an object into aggregate objects and describes the inter-relationship between the aggregate objects	derives class name and class attributes
Object Functional Model (OFM)	contains services provided by an object	derives services and service specifications
Object Dynamic Model	specifies all the possible interaction patterns of an object	derives state names, behavior specifications, and starting states
Service Refinement Functional Model (SRFM)	models a system/object service in terms of the services provided by aggregate objects	refines algebraic specifications
Refined Dynamic Model	refines the dynamic behavior model of a system/object and expresses its behavior in terms of the interaction with aggregate objects	refines states and behavior specifications
Composed Parallel Dynamic Model	composes the dynamic models of an object and its corresponding aggregate objects in parallel	composes behavior specifications

Table 8.1: The models that will be developed during a given iteration of design

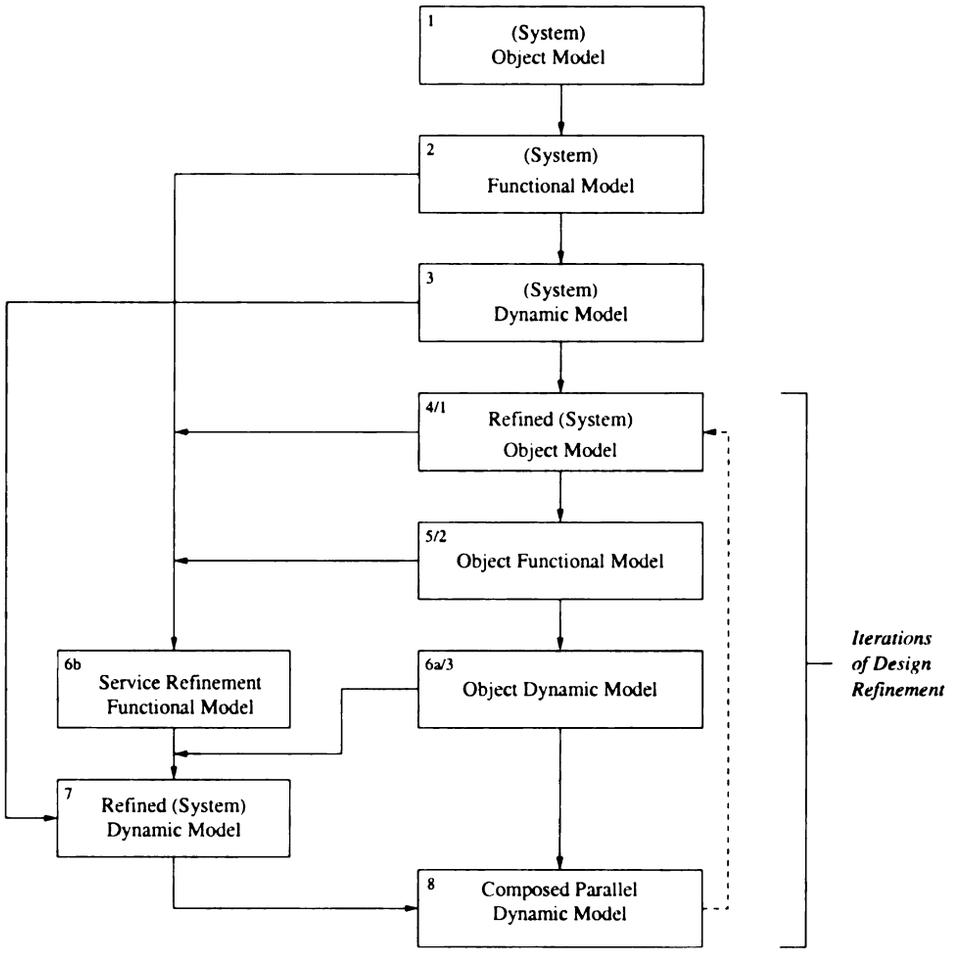

Figure 8.2: Models in the order of development

8.2.3 Design process

Booch [49] asserted: "*We have observed two traits that are common to virtually all of the successful object-oriented systems we have encountered, and noticeably absent from the ones that we count as failures: (1) the existence of a strong architectural vision; (2) the application of a well-managed iterative and incremental development life cycle.*" Our investigation attempts to address the second trait for successful development of systems.

The proposed design paradigm focuses on the process in order to facilitate a stepwise refinement of designs. Since formal specifications can be generated for each refinement step, analysis and verification technique can be used to check the consistency and correctness of design. Instead of providing specific decomposition heuristic methods, the proposed design process serves as a framework within which decomposition heuristics can be conducted and design consistency can be checked. Sufficient freedom is left to designers to choose appropriate decomposition approaches during design; decomposition heuristics is outside the scope of this research.

The design process contains iterations of model development that are introduced in the preceding discussion. For each step of model development during a given iteration, corresponding formal specifications are derived or refined. As a consequence, a progressive, incremental development of formal specifications can be performed concurrently with the development of graphical models. At the end of each iteration, analysis of formal specifications can be performed to check the consistency of diagrams and their refinements.

System design and architectural styles. The first 8 steps of model and formal specification development are considered at the system design level. Since the object being decomposed in step 4 is a system, architectural styles [100, 101, 102] should be taken into consideration to facilitate the decomposition. An *architectural style* is a recurring pattern of system organization that provides an abstract architecture for some family of applications [103] (e.g., client-server architectural style). Based upon the analysis results, choosing a specific architectural style for design is a step that

transforms the results of requirements analysis to design. Since the decomposition of a system is typically dependent upon the creativity, wisdom, and experience of a designer, we leave freedom for designers to determine which architectural style to use. The proposed design framework provides the necessary means for designers to describe various architectural styles by using graphical models that can then be analyzed according to their formal specifications.

Detailed design and design patterns. After the first iteration (8 steps), individual objects are subject to decomposition and refinement in order to achieve higher modularity and cohesion for implementation purposes. An object is subject to decomposition, unless it can be directly implemented by programming languages. Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context [104]. The designers can choose various design patterns as means to object decomposition for detailed designs. We employ the same philosophy for detailed design as we did for system design: freedom is left to the designers to determine which design pattern or decomposition approach to use; the design process only provides the necessary means for designers to specify their design models and the formalization of the OMT models enable the automated analysis of the user's design.

8.2.4 Consistency checking of refinements

The advantage of a computer-assisted transitional-parallel approach is that the formal specifications can be used to perform rigorous analyses during the design process in

order to detect and eliminate design errors. The available formal analysis has already been discussed in Chapter 7. Here we particularly emphasize two types of analysis to conduct during the design process.

Consistency checking for function refinement. SRFMs describe the refinement of the services. An SRFM captures the data dependency relationships between the services provided by aggregate objects. Therefore, algebraic assertions can be introduced to describe the constraints on the hierarchical acyclic directed structure in which the aggregate object services are related with each other. Because the services are specified in terms of pre- and postconditions, we make use of them as a means to specify design constraints and proof obligations. Given two services s_1 and s_2, if the output data flow of s_1 serves as an input data flow of s_2, a basic constraint is that the postcondition of s_1 should not imply the *falseness* of the precondition for s_2, in other words, they are not contradictory. Thus we impose a verification obligation upon SRFMs to check that the pre- and postconditions of two adjacent services are not contradictory.

Specification analysis for behavior refinement. Since the proposed design process advocates an approach that generates formal specifications in parallel with the successive refinement of semiformal design models, a behavior specification, obtained by composing concurrent aggregation objects, can be checked against the behavior specification of their corresponding individual aggregate object to detect inconsistency. In order to validate the behavior specification, a set of *test* processes, can

be written in LOTOS to simulate the environmental events. Then the same set of *test* processes can also be used to test the refined behavior specification, where the refined specification must also satisfy all the *test* processes before it can be accepted. In addition to the consistency checking, behavior simulation can also be conducted in order to interactively verify and validate the design.

8.3 The Design Process Applied to An Example

This section gives detailed descriptions about the design models and their corresponding formal specifications in the order that they are developed during the proposed design process. In order to facilitate the understanding of the models and the design process, a simple sample system is developed. The sample system is a *Disk Manager* that stores data. Each step of the process is described in the remainder of this section, where the step number is enclosed in parenthesis. Although some diagrams and the corresponding formal specifications have appeared in the previous chapters, in order to facilitate the reading, we present them again in this chapter.

System Level Object Model (1). The first step in the process is to create the system level object model, the highest level (most abstract) of the object model. It depicts the system as a single entity to be developed, and it establishes the starting point for system development. Given the requirements specifications of analysis, this is the first step of design. Usually, requirements analysis may generate an object model that includes multiple objects. Some of the objects represent environment

entities; some describe a part of the target system to be developed. Therefore, by giving the system level object model, software engineers clearly identify the software system to be developed. A system level object model generates a framework for specification within which specifications for functional and dynamic features of the system will be filled.

Figure 8.3 shows the system level object diagram for *Disk Manager*. Since *Disk Manager* is the only *object* to be built at the system level, there is only one object. Conceptually, every part of the system will be organized under object *Disk Manager*, thus forming a set of hierarchically aggregated objects. The formal specification (in Figure 8.4) for the object diagram only contains a specification declaration, where detailed information will be added based on the development of the model.

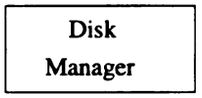

Figure 8.3: The system object model for *Disk Manager*

```
specification Disk_Manager [ gates] :  noexit
type Disk_Manager  is

endtype
endspec
```

Figure 8.4: The ACT ONE specification for system level object model of *Disk Manager*

System Level Object Functional Model (2). A system level object functional model (OFM) describes the services, in terms of data flow diagrams, provided to the external world by the system. This model is the highest level (most abstract) object. Given a system level OFM, the formalization rules in Chapter 6 can be used to automatically generate specifications that describe the services in terms of a LOTOS gate list. Meanwhile, based upon the requirements analysis results and the services identified in the system level OFM, algebraic specifications can be derived by designers to describe the properties of the object. A set of high-level guidelines that describes how to derive algebraic specifications is given in Chapter 6. In order to capture the functionalities of the services, pre- and postconditions for the services should also be specified by designers in terms of the operations of algebraic specifications.

The system level OFM for *Disk Manager* is given in Figure 8.5, where services, including their interface signatures, that *Disk Manager* provides to the external world are depicted in terms of processes and input/output data flows. This diagram introduces three data types, *Data*, *Key*, and *Void*, and two operations, *Input* and *Output*, to the ACT ONE algebraic specifications in Figure 8.4. The *ground wires* associated with services *Input* and *Output* indicate that the services need to access internal data structures in order to carry out the required functionality. The specification (Figure 8.4) augmented with algebraic specifications that describe the properties of the *Disk Manager* is given in Figure 8.6. The algebraic specification that describes the properties of the *Disk Manager* is derived by the designer based upon the services of the object. Since LOTOS currently does not support typed gates and pre- and

postconditions, the input/output arguments for services and their corresponding pre- and postconditions are given in annotations delimited by the (* and *) pairs.

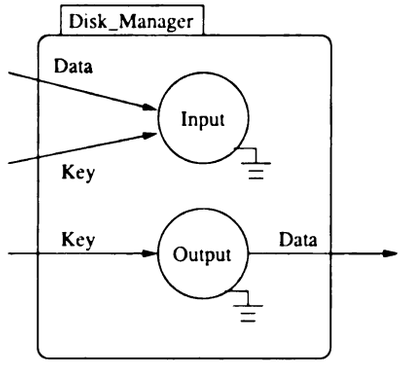

Figure 8.5: The system functional model for *Disk Manager*

System Level Dynamic Model (3). Similar to the system level functional model, the system level dynamic model is the object dynamic model for the system to be developed. Based upon the system functional model, the system level dynamic model captures the dynamic behavior (interaction patterns) of a system in terms of the services that the system provides. According to the formalization rules given in Chapter 5, a formal specification in terms of process algebras can be automatically generated from the dynamic model, where the newly introduced specification is largely under the '**behavior**' section of the specification.

The system level dynamic model in Figure 8.7 contains only one state, *Idle*, associated with two transitions. If an event for either *Input* service or *Output* service occurs, the corresponding operation will be triggered. In Figure 8.8, the formal specification

```
specification Disk_Manager [Input, Output] (disk: Disk)  :noexit :=
    (*| Input     : d: Data, k: Key -> Void |*)
    (*|requires d eq undef_Data = false and k eq undef_Key = false |*)
    (*|modifies disk |*)
    (*|ensures disk' eq input(disk^, d, k) = true |*)
    (*| Output   : k: Key -> Data |*)
    (*|requires disk eq empty = false and k eq undef_Key = false |*)
    (*|ensures result = output (disk, k) |*)
type Disk_Manager  is
     sorts
       Data, Key, Void, Disk
     opns
       undef_Data : -> Data
       undef_Key  : -> Key
       empty    : -> Disk
       input    : Disk, Data, Key -> Disk
       output   : Disk, Key -> Data
     eqns
       forall disk: Disk, data: Data, k1, k2: Key
       ofsort Data
          output (empty, key) = undef_data;
          (k1 = k2) => output (input(disk, data, k1), k2) = data
          not(k1 = k2) => output (input(disk, data, k1), k2) = output(disk, k2)
endtype
endspec
```

Figure 8.6: The specification for *Disk Manager* that specifies its services and properties

is augmented with the process algebras (delimited by "*behavior*" and "*endproc*") that capture the dynamic behavior of *Disk Manager* depicted in the system dynamic model.

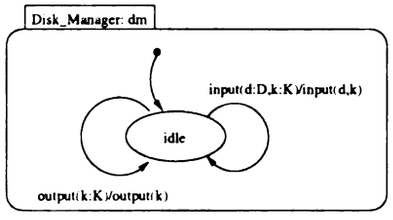

Figure 8.7: The system dynamic model for *Disk Manager*

```
specification Disk_Manager [Input, Output] (disk: Disk) : noexit :=
   (*| Input     : d: Data, k: Key -> Void |*)
   (*|requires d eq undef_Data = false and k eq undef_Key = false |*)
   (*|modifies disk |*)
   (*|ensures disk' eq input(disk^, d, k) = true |*)
   (*| Output   : k: Key -> d: Data |*)
   (*|requires disk eq empty = false and k eq undef_Key = false |*)
   (*|ensures result = output (disk, k) |*)
type Disk_Manager is
    sorts
      Data, Key, Void, Disk
    opns
      undef_Data : -> Data
      undef_Key  : -> Key
      empty      : -> Disk
      input      : Disk, Data, Key -> Disk
      output     : Disk, Key -> Data
      endtype
    opns
      output     : Disk, Key -> Data
    eqns
      forall disk: Disk, data: Data, k1, k2: Key
      ofsort Data
         output (empty, key) = undef_data;
         (k1 = k2) => output (input(disk, data, k1), k2) = data
         not(k1 = k2) => output (input(disk, data, k1), k2) = output(disk, k2)
endtype
behavior
      Idle [Input, Output] (empty)
where
    process Idle [Input, Output] (disk: Disk)  : noexit :=
      (let PRE = disk in
      Input ? d:Data ? k:Key;
         [not(d eq undef_Data] and not(k eq undef_Key)] ->
            (let POST = input(PRE,d,k) in Idle [Input, Output, Void] (POST))
       []
      Output ? k:Key;
         [not(PRE eq empty) and not(k, undef_Key)] ->
            Output ! output(PRE, k); Idle [Input, Output] (PRE)
      )
    endproc
endspec
```

Figure 8.8: The complete Full LOTOS specification for system level object model of *Disk Manager*

Because a behavior specification in Full LOTOS is executable, given the specification for *Disk Manager*, simulation and symbolic testing can be conducted to perform design verification and validation. In order to conduct simulation and testing, we instantiated sorts *Data* and *Key* in terms of predefined sorts *HexString* and *NaturalNumber* that are well defined with complete algebraic theories. The instantiated specification is given in Appendix E. The process "*accept_test*" in Figure 8.9 was used to test the behavior of the *Disk Manager* before it was decomposed into *Storage* and *Compressor*. In process *accept_test*, four data items associated with keys 4, 1, 3, and 2, respectively are given, then two data items of data with keys 1 and 3 are requested. The test process checks whether the *Disk Manager* returns correct values in accordance with the corresponding keys.

```
process accept_test [input, output, success] :  noexit :=
     input !Hex(0) !succ(succ(succ(succ(0))));
     input !Hex(3)+9 !succ(0);
     input !Hex(5) !succ(succ(succ(0)));
     input !Hex(8) !succ(succ(0));
     output !succ(0); output !3+Hex(9);
     output !succ(succ(succ(0))); output ! hex(5);
     success;
     stop
endproc
```

Figure 8.9: The testing process used to analyze the behavior of *Disk Manager*

Figure 8.10 captures the transcript of *oneexpand*, a testing command in LOLA, that composes the behavior of *Disk Manager* in parallel with the testing process `accept_test` through gates `Input` and `Output`, and executes a random trace of the

composed process. The **success** event that appears in the 9th step shows that the test successfully terminates. This implies that the returned values by *Disk Manager* are correct.

```
lola> oneexpand 10 success accept_test -v
oneexpand 10 accept_test 0 1 -v

 Rewriting expressions in the specification.
 Rewriting done.
 Analysing unguarded conditions.
 Analysis done.

  Composing behaviour and test :

     accept_test [input,output,success]
   |[input,output]|
     idle [input,output] (empty)

    1  input  ! hex(0) ! succ(succ(succ(succ(0))));
    2  input  ! 3 + hex(9) ! succ(0);
    3  input  ! hex(5) ! succ(succ(succ(0)));
    4  input  ! hex(8) ! succ(succ(0));
    5  output  ! succ(0);
    6  output  ! 3 + hex(9);
    7  output  ! succ(succ(succ(0)));
    8  output  ! hex(5);
    9  success;

    Process Test = accept_test
    Test result  = SUCCESSFUL EXECUTION.

       Transitions generated = 9
```

Figure 8.10: Using *accept_test* testing process to analyze the specified *Disk Manager*

Similarly, other testing cases can also be written to check the behavior of the *Disk Manager*. In addition, a stepwise simulation can also be conducted to further analyze the behavior of *Disk Manager*. Since the behavior specifications, case testing, and stepwise simulation are event-based, both case testing and simulation enables the

designers to detect design flaws that two or more related behavior sequences are inconsistent.

Refined (System Level) Object Model (4). Decomposition is one of the most important activities during design. Although decomposition heuristics is outside the scope of this research, it is necessary to give an overview because of its importance during design.

Decomposition heuristics is an important issue of design and is interwoven with design refinement during the design process. At each step of design refinement, decomposition may be performed in order to decompose a part of a system into smaller modules for implementation. Software architectures and design patterns are two major decomposition heuristics that have gained extensive interest in recent years. They are also applicable during our proposed design process.

Software architecture: *The component structure of a program or system, their interrelationships, and principles and guidelines governing their design and evolution over time* [100].

Architectural style: *A recurring pattern of system organization that provides an abstract architecture for some family of applications* [103].

Design pattern: *Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context* [104].

Software architecture has long been recognized as an important issue for software development [6] and has recently received significant attention [101, 102]. The emergence of research in software architecture is due to two primary reasons [100]. One reason is that over the years designers have begun to develop a shared repertoire of methods, techniques, patterns, and idioms for structuring complex software systems.

The other reason is that the practice of application development for different domains now, after many years of hard work, has resulted in reusable frameworks for product families. The concern of software architecture mainly focuses on the organization and the connection of the components of a large system. The architecture style refers to a class of architectures that share common characteristics. We, the Software Engineering Research Group at Michigan State University, have further characterized the architectural style in terms of architectural topology, characteristics/constraints of components, and characteristics of connections between components [105]. The most popular architectural styles include client-server, pipeline, layered, and so on.

Design patterns [104] provide reusable design blocks that help to alleviate software complexity during system design (for simple systems) and detail design stages. They supply the designers with design alternatives that have proven to be successful. Design patterns can be used for architecture design if a system is simple.

Though a design process and design decomposition have entirely different scopes of concern, there is a close relationship between the two. A design process focuses on providing a process to facilitate the decomposition and refinement of design information. Architectures and patterns focus on the advantages and disadvantages of different architectural styles and design patterns and their ability to solve specific types of problems. Thus choosing a certain architectural style or design pattern is an important step during design.

During the object-oriented design process, the objects are decomposed into sets of finer-grained objects in order to achieve modularity that reduces complexity. This decomposition activity is facilitated by object model refinement in our proposed de-

sign process. A refined object model depicts the decomposition of objects of a higher level of abstraction. Since a refined object model decomposes an object into aggregate objects, it looks like a tree structure with a root and a set of leaves. In this step, architectural styles as well as design patterns should also be taken into consideration in order to conduct better decomposition. The former is applicable for a system and the latter for the decomposition of an individual object. During the first iteration of refinement, where the refinement of a system-level object model decomposes a system, architectural styles should be considered. For object decomposition, existing design patterns may help the designers to conduct the refinement process. From the refined object model, frameworks of formal specifications for newly introduced objects can be automatically generated by using our proposed formalization rules.

We have thus far achieved a relatively complete formal specification in Full LOTOS for the *Disk Manager*. The term "complete" used here means that the specification includes the static, functional, and behavioral description of an object instead of implying that the design is completed. In order to conduct the design to a more detailed level, further refinement is needed.

Figure 8.11 gives a refined object model for *Disk Manager*. The refined object model shows that *Disk Manager* is composed of aggregate objects *Storage* and *Compressor*. The object diagram in Figure 8.11 generates two specification frameworks for objects *Storage* and *Compressor* in Figure 8.12. Once the aggregate objects, *Storage* and *Compressor*, are further specified in terms of functional and dynamic models, more design details of *Disk Manager* can be added by using formal specifications generated by aggregate objects.

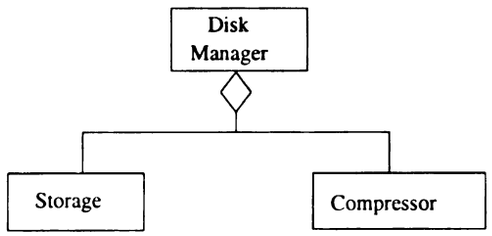

Figure 8.11: The refined object model for *Disk Manager*

```
specification Storage [ gates] :  noexit
type Storage  is Disk_Manager

endtype
endspec

specification Compressor [ gates] :  noexit
type Storage  is Disk_Manager

endtype
endspec
```

Figure 8.12: The ACT ONE specifications for object models of *Storage* and *Compressor*

Object Functional (5) and Dynamic Models (6a). The OFM describes the services provided by an individual object to its external world. When an aggregate object is introduced into a system for the first time during the refinement of object models, its corresponding OFM should also be developed to specify what services the object provides. Similar to the system-level OFM, based upon OFMs for individual objects, formal specifications that describe the signatures of the services can be automatically generated according to formalization rules, and pre- and postconditions that describe the behavior of services derived by designers.

Based upon the OFM for an object, an object dynamic model can be developed to capture the dynamic behavior (interaction patterns) of the object in terms of the services provided by the object. The formal specifications that are automatically derived from dynamic models describe the behavior of the objects in terms of process algebras.

Similar to the process used to generate formal specification for the *Disk Manager*, Figures 8.13, 8.14, 8.17, and 8.18 contain the OFMs and dynamic models for objects *Storage* and *Compressor*, respectively. Their corresponding specifications in Full LOTOS are given in Figures 8.15, 8.16, and 8.19. The used rules include **OM1**, **OM2**, **OM10**, **DFR-1**, **DFR-2**, **DFR-3**, **DFR-4**, **DFR-6**, **DFR-7**, **DFR-8**, **DFR-11**, **DFR-12**, **DFR-13**, **FFR-1**, **FFR-2**, **FFR-3**, and **FFR-4**.

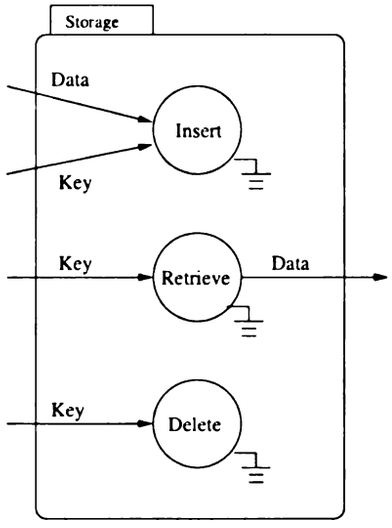

Figure 8.13: The object functional model for *Storage*

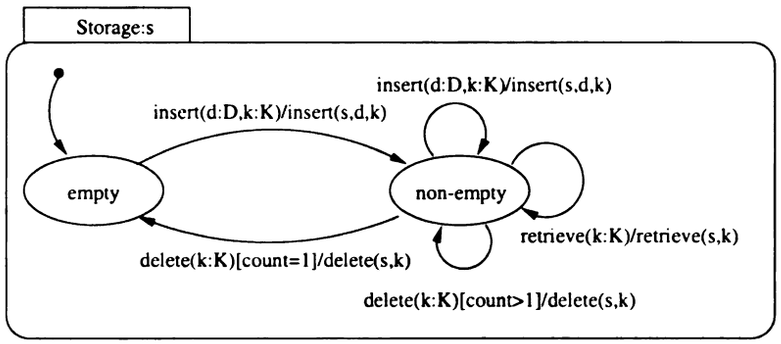

Figure 8.14: The object dynamic model for *Storage*

Service Refinement Functional Model (6b). The description of a service refinement functional model involves a few subtleties because its derivation involves several models at different levels of abstraction. Figure 8.20 gives an example of a refined object model.

In Figure 8.20, object O is a more abstract object model that is decomposed (refined) into objects O_1, O_2, ..., O_n in the refined object model. Objects O, O_1, O_2, ..., O_n are all associated with corresponding OFMs identifying the services they provide. If S is a service provided by O, then its corresponding SRFM depicts how S is composed in terms of the services provided by O_1, O_2, ..., O_n. However, the SRFM only illustrates the data transformation among the provided services, the control information and the order in which the provided services are activated are not in the scope of SRFM. However, the SRFMs do capture the data dependency relationships between the services provided by aggregate objects. Therefore, algebraic axioms can be specified to describe the constraints on the SRFM in which the aggregate object

```
speci
(
(
(
(
(
(
(
(
(
(
(
type
s
endt
ends
```

F

```
specification Storage [Insert, Retrieve, Delete] (s: S)  : noexit :=
   (*| Insert     : d: Data, k: Key -> Void |*)
   (*|requires d eq undef_Data = false and k eq undef_Key = false |*)
   (*|modifies s |*)
   (*|ensures s' eq insert(s^, d, k) = true |*)
   (*| Retrieve  : k: Key -> d: Data |*)
   (*|requires s eq empty = false |*)
   (*|ensures result = retrieve (s, k) |*)
   (*| Delete : k: Key -> Void |*)
   (*|requires k eq undef_Key = false and s eq empty = false|*)
   (*|modifies s |*)
   (*|ensures s' eq delete(s^, k) = true |*)
type Storage  is Disk_Manager, NaturalNumber
    sorts
      type S is (*| Distinguished sort |*)
         empty | insert (S, Data, Key)
      endtype
    opns
      retrieve  : S, Key -> Data
      delete    : S, Key -> S
      count     : S -> Nat
    eqns
      forall s: S, data: Data, k1, k2: Key
      ofsort Data
         retrieve (empty, key) = undef_data;
         retrieve (insert(s, data, k1), k2) = if k1=k2 then data
            else retrieve(s, k2);
         delete (empty, key) = empty;
         delete (insert(s, data, k1), k2) = if k1=k2 then s
            else delete(s, k2);
         count (empty) = 0;
         count (insert(s, data, key)) = Succ(count(s));
endtype
endspec
```

Figure 8.15: The specification for object *Storage* in Full LOTOS

```
behavior
  Empty_State [Insert, Retrieve, Delete] (empty)
where
   process Empty_State [Insert, Retrieve, Delete] (s: S) :  noexit :=
      (let PRE = s in
         Insert ?d:D ?k:K;
            (let POST = insert(PRE, d, k)
               in None_Empty_State [Insert, Retrieve, Delete] (POST))
   endproc
   process None_Empty_State [Insert, Retrieve, Delete] (s : S) :  noexit :=
      (let PRE = s in
      Retrieve ?k:K;
      [not(s eq empty)] ->
         Retrieve ! retrieve (PRE, k);
         None_Empty_State [Insert, Retrieve, Delete] (PRE)
       []
      Insert ?d:D ?k:K;
      [not(d eq undef_Data) and not(k eq undef_Key)] ->
         (let POST = insert(PRE, d, k) in
            None_Empty_State [Insert, Retrieve, Delete] (POST)
         )
       []
      Delete ?k:K;
      [not(k eq undef_Key) and not(s eq empty)] ->
         ([count(s) gt Succ(0)] ->
         (let POST = delete (PRE, k)
            in None_Empty_State [insert, retrieve, delete] (POST))
          []
         [(count(s) eq Succ(0)) and (delete(s,k) eq empty)] ->
         (let POST = delete (PRE, k)
            in Empty_State [insert, retrieve, delete] (POST))
         )
      )
   endproc
endspec
```

Figure 8.16: The specification for object *Storage* in Full LOTOS

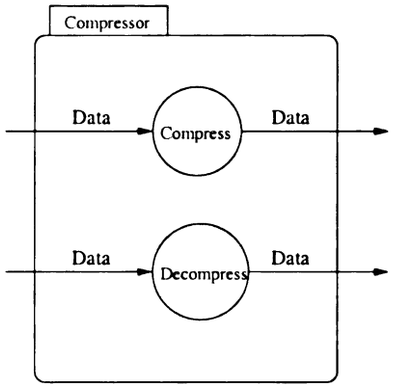

Figure 8.17: The object functional model for *Compressor*

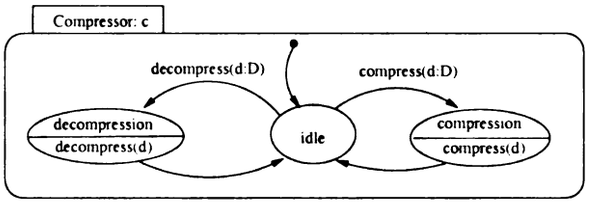

Figure 8.18: The object dynamic model for *Compressor*

services are related with each other. Because the services are specified in terms of pre- and postconditions, we can use them to describe design constraints.

In Figures 8.21 and 8.22, the SRFMs for services *Input* and *Output* provided by *Disk Manager* are illustrated. The constraints introduced by the refinement of the service *Output* is described in terms of ACT ONE axioms in Figure 8.23. Given the SRFM in Figure 8.22, according to the formalization rules for SRFMs in Chapter 6, the postcondition of service *Storage.Retrieve* and the precondition of service *Compressor.Decompress*, which are given in Figures 8.15, 8.16, and 8.19 respectively, should be consistent. The conjunction of the two can be analyzed to check the con-

```
specification Compressor [Compress, Decompress] (c: C) : noexit :=
   (*| Compress     : d: Data -> Data |*)
   (*|requires d eq undef_Data = false |*)
   (*|ensures result = compress(d) and size(result) le size(d) |*)
   (*| Decompress  : d: Data -> Data |*)
   (*|requires d eq undef_Data = false |*)
   (*|ensures result = decompress(d) and size(result) ge size(d) |*)
type Storage is Disk_Manager, NaturalNumber
    sorts
      C (*| Distinguished sort |*)
    opns
      compress : Data -> Data
      decompress : Data -> Data
      size : Data -> Nat
    eqns
      forall data: Data
      ofsort Bool
         size(data) ge size(compress(data)) = true;
         size(data) le size(decompress(data)) = true;
         decompress(compress(data)) = data;
endtype
behavior
   Idle_State [Compress, Decompress]
where
   process Idle_State [Compress, Decompress] : noexit:=
      Compress ?d:D; Compress_State [Compress, Decompress] (d)
      []
      Decompress ?d:D; Decompress_State [Compress, Decompress] (d)
   endproc
   process Compress_State [Compress, Decompress] (d:D): noexit:=
      i; Compress ! compress(d); Idle_State [Compress, Decompress]
   endproc
   process Decompress_State [Compress, Decompress] (d:D): noexit :=
      i; Decompress ! decompress(d); Idle_State [Compress, Decompress]
   endproc
endspec
```

Figure 8.19: The specification for *Compressor* in Full LOTOS

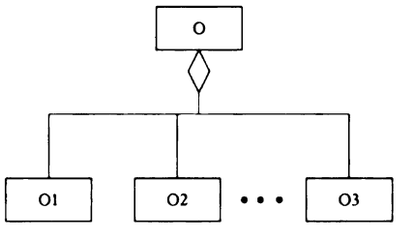

Figure 8.20: An example refined object model

sistency of function refinement. Since the result of *Storage.Retrieve* will be supplied to *Compressor.Decompress*, given the pre- and postconditions of the two services, the constraint is

```
((s neq empty =>(result=retrieve(s,k) and result neq undef_Data) eq false)=false).
```

The equation specifies that, given a non-empty *Storage* object, the retrieved data for key k should be a valid data item that can be fed into the *Decompress* service provided by the *Compressor* object.

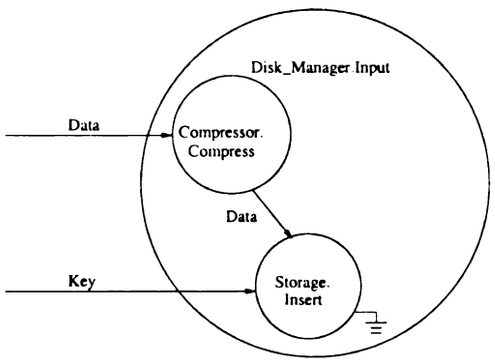

Figure 8.21: The service refinement functional model for function *Input* of *Disk Manager*

There are basically two approaches to performing the consistency checking. As mentioned previously, due to the lack of proof tool support for ACT ONE spec-

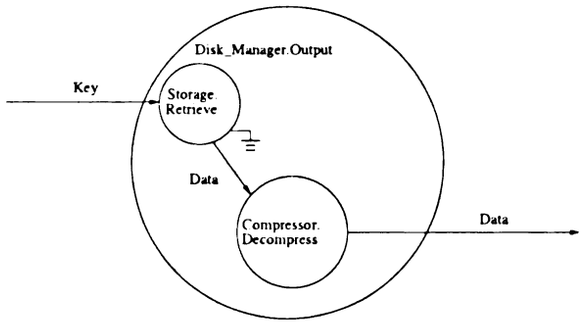

Figure 8.22: The service refinement functional model for function *Output* of *Disk Manager*

```
type Disk_Manager_REF is Storage, Compressor, Disk_Manager
    eqns
      forall s: S, d: Data, k: Key
      ofsort Bool
         not(s=empty) => not(retrieve(s,k) = undef_Data) = true
endtype
```

Figure 8.23: The refined ACT ONE specification for *Disk Manager*

ifications, we translated the specifications into LSL in order to use LP to check for consistency between pre- and postconditions. The corresponding LSL specifications are given in Appendix F. Figure 8.24 gives the LSL specification translated from the ACT ONE specification for the refined *Disk Manager* given in Figure 8.23. The translation is currently performed manually, though a parer can be written to perform the translation automatically. The proof obligation is simplified to `~(s=empty)=>~(retrieve(s,n)=undef_d)` and is given in the *implies* clause (the simplification can also be assisted by tools, such as PVS). Figure 8.25 shows the result of the proof using LP.

```
Disk_ref (D, K): trait
    includes Storage(D,K), Compressor(D)
    implies
      ∀ s: S, n: N
      ~(s=empty) => ~(retrieve(s,n) = undef_d);
```

Figure 8.24: The LSL specification for the refined *Disk Manager*

```
LP2: display proof

Conjecture Disk_refTestTheorem.1: ~(s = empty) => ~(retrieve(s, n) = undef_d)
  Attempting a proof by depth 2 structural induction on 's'
Level 2 subgoal 2 (basis step) for proof by induction on s:
  ~(insert(empty, n1, n2) = empty)
    => ~(retrieve(insert(empty, n1, n2), n) = undef_d)
  Current subgoal: ~((if n = n2 then n1 else undef_d) = undef_d)
  Attempting a proof by depth 2 structural induction on 'n'
Level 3 subgoal 1 (basis step) for proof by induction on n:
  ~((if 0 = n2 then n1 else undef_d) = undef_d)
  Attempting a proof by depth 2 structural induction on 'n1'
Level 4 subgoal 1 (basis step) for proof by induction on n1:
  ~((if 0 = n2 then 0 else undef_d) = undef_d)
  Attempting a proof by depth 2 structural induction on 'n2'
Level 5 subgoal 2 (basis step) for proof by induction on n2:
  ~((if 0 = succ(0) then 0 else undef_d) = undef_d)
  Current subgoal: false
  Attempting a proof by normalization

LP3:
```

Figure 8.25: The proof of the example constraint using LP

By using the *induction* proof method provided by LP, we reach a point that shows `Current subgoal: false` (in Figure 8.25), which means there are cases that leads to false under that constraint. The proof detects an inconsistency between the post-condition of *Storage.Retrieve* and the precondition of *Compressor.Decompress*, thus requiring us to perform a further inspection into the problem. Further investigation reveals that the *Storage.Retrieve* operation is specified to return *undef_d* if it is given a key that has never been inserted into the storage. In this case the precondition of *Compressor.Decompress*, which specifies that the input is not an invalid value, cannot be satisfied. This situation must be taken into consideration when the dynamic model of *Disk Manager* is refined.

Refined (System) Dynamic Model (7). Once the services of object O are refined in terms of services provided by aggregate objects O_1, O_2, ..., O_n, and the dynamic models of O_1, O_2, ..., O_n are derived, the dynamic model of O is refined accordingly. The refined dynamic model substitutes the services of O with services provided by O_1, O_2, ..., O_n and interacts with the dynamic models of O_1, O_2, ..., O_n. From the refined dynamic model, a specification in terms of process algebras can be automatically generated. The specification describes the refined behavior of the corresponding object. For instance, Figure 8.26 shows an example dynamic model for object O. The dynamic model contains only one state. Whenever an event S_1 occurs, the two arguments associated with the event are taken, the corresponding service, S_1 is triggered to provide the requested service.

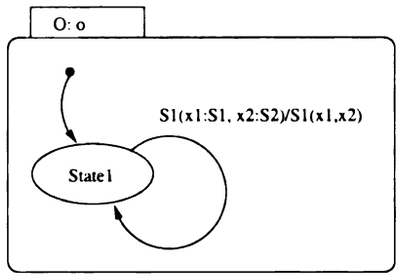

Figure 8.26: An example dynamic model for object O

Figure 8.27 shows an example SRFM that refines service S_1 of object O in terms of the services provided by objects O_1 and O_2 that constitute O. In this diagram, the data flows and the related services are depicted. Service $O_2.S_3$ takes the output of service $O_1.S_2$ as input, thus having a data dependency.

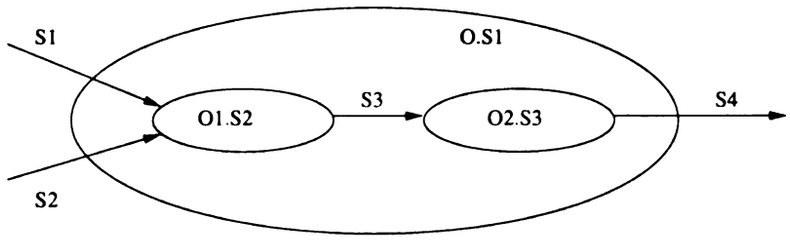

Figure 8.27: An example SRFM for service $S1$ of object O

The dynamic model in Figure 8.28 further refines the dynamic model of object O given in Figure 8.26 according to the SRFM in Figure 8.27. The diagram depicts the behavior sequence of object O for a given event that requests for service S_1. Once O receives a request for service S_1: (1) S_1 of O takes two input arguments; (2) feeds them into $O_1.S_2$; (3) takes the output from $O_1.S2$ and feeds it into $O_2.S_3$; (4) then takes the output from $O_2.S_3$ and returns it to the caller of service S_1.

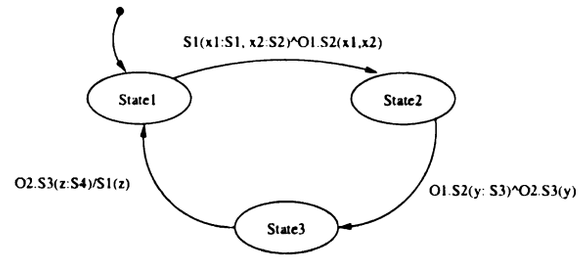

Figure 8.28: An example refined dynamic model for object O

Based upon the SRFMs, the dynamic model of *Disk Manager* is further refined and given in Figure 8.29. Three intermediate states *IS1*, *IS2*, and *IS3* are added in order to describe the refined dynamic behavior. The refined dynamic model reflects the fact that the services *Input* and *Output* of *Disk Manager* are implemented in terms of the services provided by objects *Storage* and *Compressor*. However, the refined dynamic model was derived before we conducted the consistency for the SRFMs. As we have shown in the above discussion, the returned value from *Storage.Retrieve*, under some circumstances, may be *undef_Data*, which is contradictory to the precondition for *Compressor.Decompress*. This needs to be addressed in the refined dynamic model. Therefore a guarding condition is added to the transition that triggers the *Compressor.Decompress* event in order to guarantee that the value passed to *Compressor.Decompress* is valid. In addition, once an invalid value is returned from *Storage.Retrieve*, the state of *Disk Manager* should return to *Idle* instead of conducting further transactions. The revised dynamic model is given in Figure 8.30. The corresponding refined specification in terms of process algebras is given in Figure 8.31. The formalization rules used to generate the formal specification include

DFR-1, **DFR-2**, **DFR-3**, **DFR-4**, **DFR-5**, **DFR-6**, **DFR-7**, **DFR-8**, **DFR-11**, **DFR-12**, and **DFR-13**.

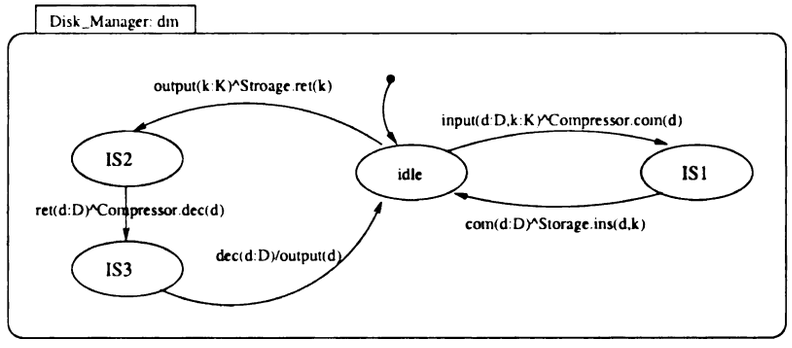

Figure 8.29: The refined dynamic model for *Disk Manager*

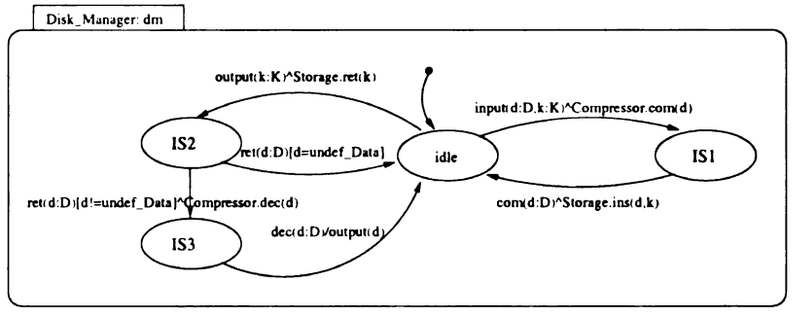

Figure 8.30: The revised refined dynamic model for *Disk Manager*

Composed Parallel Dynamic Model (8). Given the refined dynamic model of object O and dynamic models of aggregate objects O_1, O_2, ..., O_n, a parallel dynamic model can be composed to model communication and interaction of the aggregate objects. In the parallel dynamic model, the refined dynamic model of O and the dynamic models of O_1, O_2, ..., O_n communicate with each other. With respect to information encapsulation, one key characteristic of any object-oriented methodology

```
type Disk_Manager_REF is Storage, Compressor, Disk_Manager
behaviour
   Idle [Input,Output,Compress,Decompress,Insert,Retrieve]
where
    process Idle[Input,Output,Compress,Decompress,Insert,Retrieve]: noexit :=
      Input ?d:D ?k:K; Compress !d !k;
         IS1[Input,Output,Compress,Decompress,Insert,Retrieve](k)
       []
      Output ? k:K; Retrieve ! k;
         IS2[Input,Output,Compress,Decompress,Insert,Retrieve]
    endproc
    process IS1[Input,Output,Compress,Decompress,Insert,Retrieve]: noexit :=
      Compress ?d:D; Insert !d !k;
         Idle[Input,Output,Compress,Decompress,Insert,Retrieve]
    endproc
    process IS2[Input,Output,Compress,Decompress,Insert,Retrieve]: noexit :=
     Retrieve ?d:D;
        ([d eq undef_Data] ->
           Idle[Input,Output,Compress,Decompress,Insert,Retrieve]
        []
        [not(d eq undef_Data)] -> Decompress !d;
           IS3[Input,Output,Compress,Decompress,Insert,Retrieve]
        )
    endproc
    process IS3[Input,Output,Compress,Decompress,Insert,Retrieve]: noexit :=
      Decompress ?d:D; Output !d;
         Idle[Input,Output,Compress,Decompress,Insert,Retrieve]
    endproc
endspec
```

Figure 8.31: The refined ACT ONE specification for *Disk Manager*

is that a change in an object is not visible to external objects. The only means that a change can be reflected to other objects is through inter-object communication. Accordingly, the inter-object communication is the only communication mechanism in the formalized model; the visible events of an object are realized by the inter-object communication external to the object. The inter-object communication is described in terms of LOTOS gate synchronizations (see Chapter 5). Formal specifications can also be automatically generated from the composed parallel dynamic model (see

Chapter 5). The formal specifications precisely capture the dynamic interaction of the corresponding models. In addition to behavior simulation, the formal specifications can used to perform consistency checking against output O's original dynamic model before refinement.

Since the refined dynamic model of *Disk Manager* triggers the services provided by *Storage* and *Compressor*, a dynamic model that comprises the three behavioral dynamic models in parallel is constructed in Figure 8.32. The corresponding formal specification include both specifications for *Storage* and *Compressor*, in addition to that for *Disk Manager*, in Full LOTOS. The specifications for *Storage* and *Compressor* are treated as a single complex process instead of as separate specifications because of the specification structure of the LOTOS language. The entire specification is given in Appendix G. The specification given in Figure 8.33 describes how the three dynamic models are composed in parallel in terms of LOTOS.

Given the refined LOTOS specification for *Disk Manager* in Figure 8.33, we can use the same set of testing processes for *Disk Manager* that were developed before the refinement and decomposition to check the consistency of the refined models. Figures 8.34 and 8.35 captures the transcript of *oneexpand* that composes the behavior of the refined *Disk Manager* in parallel with the testing process **accept_test** (given in Figure 8.9) through gates **Input** and **Output**, and executes a random trace of the composed process. The **success** event that appears in the 35th step in Figure 8.34 shows that the test successfully terminates. This result implies that the values returned by *Disk Manager* are correct. If the internal events, `i`, are removed from the

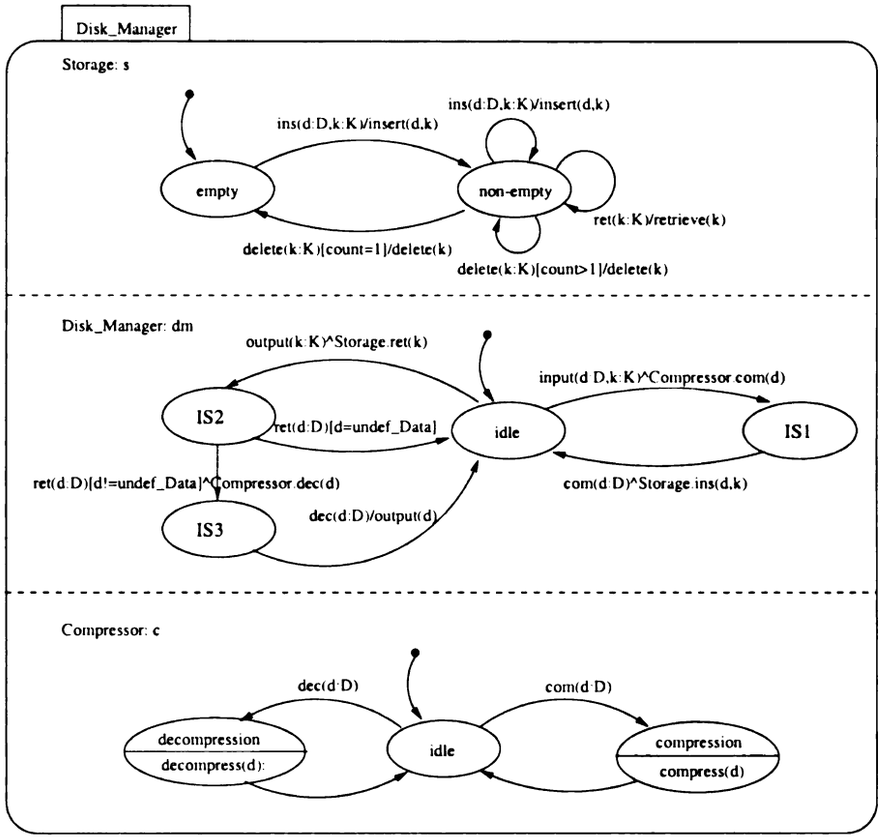

Figure 8.32: The refined dynamic model for *Disk Manager* that is composed by parallel dynamic models

```
type Disk_Manager_REF is Storage, Compressor, Disk_Manager
behaviour
   hide Compress, Decompress, Insert, Retrieve, Delete in
      Storage [Insert, Retrieve, Delete] |[Insert, Retrieve]|
      Idle [Input, Output, Compress, Decompress, Insert, Retrieve]
      |[Compress, Decompress]| Compressor [Compress, Decompress]
endspec
```

Figure 8.33: The composed specification for *Disk Manager*

result, we can find that the execution result is identical to that given in Figure 8.9 (the testing process for *Disk Manager* before refinement).

```
lola> oneexpand 50 success accept_test -v -i
oneexpand 50 accept_test 0 1 -v -i

  Composing behaviour and test :

     accept_test [input,output,success]
   |[input,output]|
     (hide ins,ret,del,com,dec in
        storage [ins,ret,del]
      |[ins,ret,del]|
          idle [input,output,com,dec,ins,ret,del]
        |[com,dec]|
          compression [com,dec]
     )
```

Figure 8.34: Using *accept_test* testing process to analyze the behavior of the refined *Disk Manager* (1)

8.4 Summary

In this chapter, based upon the formalization and integration of the models of OMT (presented in Section 2.4 and Chapters 4, 5, and 6), we proposed a design process that facilitates the development of formal design specifications in parallel with the development of OMT's semi-formal, graphical models. Because of the rigorous mathematical foundation of formal specifications, both customers and designers can have a more precise means to describe the design thus avoiding ambiguities. In addition, symbolic simulation of the design can help designers better understand the design as well as facilitate the communication among designers and even with the customers.

```
 1  input  ! hex(0) ! succ(succ(succ(succ(0))));
 2  i; (* com ! hex(0) *)
 3  i;
 4  i; (* com ! compress(hex(0)) *)
 5  i; (* ins ! compress(hex(0)) ! succ(succ(succ(succ(0)))) *)
 6  input  ! 3 + hex(9) ! succ(0);
 7  i; (* com ! 3 + hex(9) *)
 8  i;
 9  i; (* com ! compress(3 + hex(9)) *)
10  i; (* ins ! compress(3 + hex(9)) ! succ(0) *)
11  input  ! hex(5) ! succ(succ(succ(0)));
12  i; (* com ! hex(5) *)
13  i;
14  i; (* com ! compress(hex(5)) *)
15  i; (* ins ! compress(hex(5)) ! succ(succ(succ(0))) *)
16  input  ! hex(8) ! succ(succ(0));
17  i; (* com ! hex(8) *)
18  i;
19  i; (* com ! compress(hex(8)) *)
20  i; (* ins ! compress(hex(8)) ! succ(succ(0)) *)
21  output  ! succ(0);
22  i; (* ret ! succ(0) *)
23  i; (* ret ! compress(3 + hex(9)) *)
24  i; (* dec ! compress(3 + hex(9)) *)
25  i;
26  i; (* dec ! 3 + hex(9) *)
27  output  ! 3 + hex(9);
28  output  ! succ(succ(succ(0)));
29  i; (* ret ! succ(succ(succ(0))) *)
30  i; (* ret ! compress(hex(5)) *)
31  i; (* dec ! compress(hex(5)) *)
32  i;
33  i; (* dec ! hex(5) *)
34  output  ! hex(5);
35  success;

 Process Test = accept_test
 Test result  = SUCCESSFUL EXECUTION.

    Transitions generated = 35
```

Figure 8.35: Using *accept_test* testing process to analyze the behavior of the refined *Disk Manager* (2)

Finally, consistency can be checked during model refinement of the design process to detect and eliminate design flaws during earlier stages of software development. A simple example that employed the proposed process to conduct design is also given.

Chapter 9

Case Study

As a means to obtain empirical credibility [106] for our research, we applied the formalization and integration rules and design process to an industrial project. This chapter shows a case study that applied our proposed integration and formalization rules and design process to an industrial project, the *Environmental Information System* (ENFORMS) [107, 108], developed by the Software Engineering Research Group (SERG) at Michigan State University over a three year period involving 15 software developers. The remainder of this chapter is organized as follows. Section 9.1 introduces background information about the project. Section 9.2 discusses the focus of the case study. An overview of the requirements analysis for ENFORMS project is given in Section 9.3. Section 9.4 describes the system level modeling of ENFORMS. Based upon the system modeling, Section 9.5 discusses the system design of ENFORMS. The design analysis and refinement of ENFORMS is given in Section 9.6. A summary and conclusions drawn from the case study are given in Section 9.7.

9.1 The Project

During this decade, NASA will launch many new platforms into earth orbit, including the satellites that will make up the Earth Observing System (EOS). The remotely sensed data obtained from EOS can be used to promote global and national security, extend international cooperation, and improve our ability to understand and manage global environmental, economic, and social problems. In the past, NASA and other agencies have focused on the acquisition of data rather than the integration or the dissemination of data. Many organizations addressing grand challenge problems, such as those defined by earth sciences, require the integration of both physical and human resource databases in an interactive manner. Such a capability allows reasonably informed policy analysts and related staff to query an "Environmental Science Workstation" so as to better understand how human uses impact our natural resource base.

Scientific research addressing global change continues to generate large quantities of information for analysis and understanding. However, the volume, distributed nature, and diversity of this information prohibits convenient access by many potential users, including policy analysts, federal agency staff, and local township planners.

ENFORMS [107, 108] has been developed to facilitate the access, integration, and analysis of data relevant to a regional study that has local and global impacts. Specifically, information from studies of the Saginaw Bay Watershed region has been used to populate the ENFORMS archive. ENFORMS is the result of a multidisciplinary effort, largely sponsored by NASA, EPA, USDA, and the Consortium of

International Earth Science and Information Network (CIESIN). The project team consists of researchers and practitioners from fields including computer science, entomology, ecology, land and resource management, sociology, and geography. The project team is divided into user needs analysts, application scientists, and computer scientists.

There were several motivating factors that led to the selection of the Saginaw Bay Watershed region as the focus of the regional study [107, 108]. The Saginaw Bay Watershed region is located in central Michigan around the mouth of the Saginaw Bay of Lake Huron. It is over 8,700 square miles in size. Large areas are devoted toward industry and agriculture. Several state and national forests are located in the area. There is also one major river in the area (Saginaw River) with many tributaries that flow into the Saginaw Bay. Thus, the entire watershed affects the quality of the Great Lakes. The region is an excellent area to study how humans affect the environment because the major types of land uses exist in the area. In addition, the Saginaw Bay Watershed region is an important resource for the state of Michigan as well as for the United States. It is the center of some of the most productive agricultural soils in the country. It contains several large industrial centers and thus supports many facets of the states' economy.

The Saginaw Bay Watershed region has been identified by several United States government agencies (e.g. EPA and USDA) as an area of environmental concern due to the massive amount of soil erosion that occurs in the area. The Great Lakes Commission has reported that nearly 606 million tons of soil are lost annually from

the Great Lakes area, at the cost of over 3 billion U.S. dollars worth of lost nutrients and productivity [109].

The objective of ENFORMS is to provide users with convenient access to large amounts of multimedia information that may be distributed across many sites. Figure 9.1 shows a high-level view of the organization, where the rectangles represent participating sites.

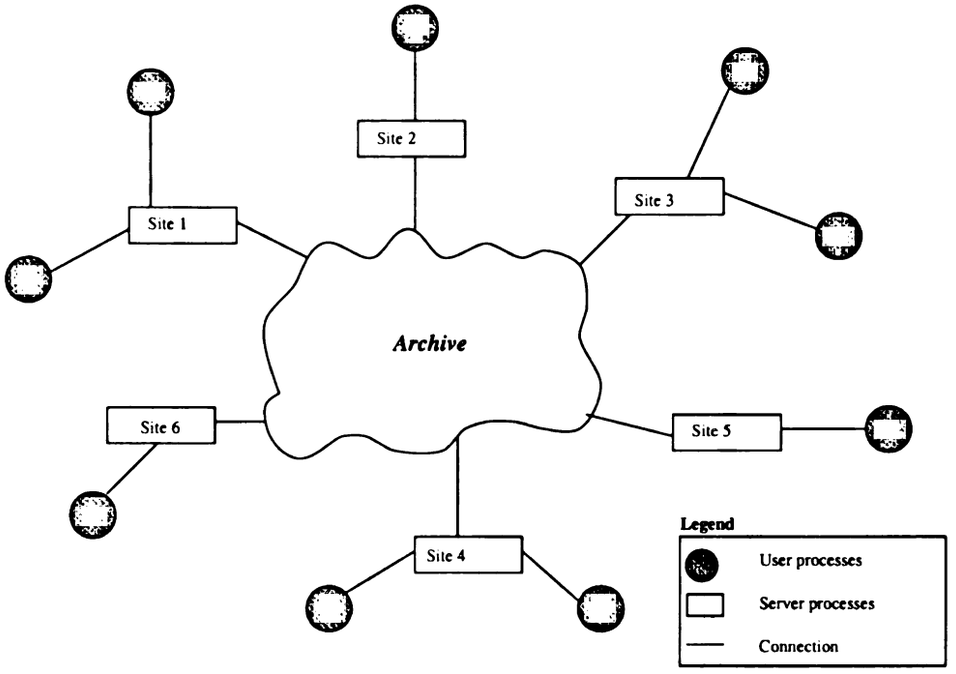

Figure 9.1: A high-level view of the architecture

The main requirements of the system are threefold. First, the archive itself is potentially large, and thus some sort of selective browsing mechanism is necessary to help the user navigate through the archive. In order to keep the system flexible for new users and new features, it it was necessary that the system design be able to support a variety of browsing mechanisms and not be tightly coupled with either

the user interface or the user interface technology [110]. Second, information items may be stored in a variety of forms [111]: simple databases and data files, images, documents, GIS maps, animations, parameterized models, and so on; accordingly, the system must be able to determine which software tools are needed to examine each item. Finally, when accessing the archive, the distributed nature of the stored items should be transparent to the user with regards to the browser [111]; this requirement also suggests that the system should be able to operate both as a stand-alone system and as a component of a distributed system.

9.2 Focus of the Case Study

ENFORMS is a relatively large system that cost about 10 person years. In this case study, we focus on the requirement that *the distributed nature of the stored items should be transparent to the user with regards to the browser when accessing the archive.* The case study shows that:

- The formalization and integration rules for OMT models can help the designers to derive more precise and integrated descriptions of the design.
- The proposed design process facilitates the refinement of design information.
- Based on the formal specifications derived from OMT models, analysis can be performed to check design consistency and to facilitate the understanding about the design.

9.3 Overview of the Requirements Analysis

This section focuses on the distributed query feature of the ENFORMS system.

A high-level description of requirements analysis based on [112] is given as follows to provide an overview.

From the most abstract point of view, the ENFORMS system allows a user to gain access to an archive of data, where the identification of the data of interest is achieved by browsing related indices (i.e., a classification scheme) about the data.

The ENFORMS uses a *browse-analysis* approach for decision support. Browsing is the method used to facilitate the construction of queries about topics of interest, retrieval of data related to those interests. Analysis services utilize data that is retrieved from the archive after browsing.

In the browse-analysis framework, each browse-analysis *task* begins in the **browse** state, where a user browses based on the type of *search model* being employed. Search models can vary from *hierarchical* and *spatial* to *thematic.* Once the search criteria have been defined, data is retrieved, placing the browse-analysis task in the **retrieve** state. If a search is successful (i.e., desired data has been found), the browse-analysis task transitions into the **analysis** state, where a user can analyze the data using tools that can manipulate the data.

From the user perspective, there will be two modes of operation, browsing and analysis. Browsing is a process that results in the selection of datasets from the data archive. Selected datasets become part of a project that may be used by the analysis services. Analysis services provide tools for exploring and manipulating the content of selected data sets. Analysis services allow for data to be subsetted, aggregated both spatially and temporally, and the results displayed in a variety of manners. The results of data analyses are data products, and the sequence of operations that define

a given product (the product specification) can be used to represent the data product rather than to actually require the product to be explicitly stored.

While original sources of data may be single repositories (i.e., the EPA mainframe), users have been moving towards network topologies in terms of both computer use and data distribution. The ENFORMS system has been designed and implemented with the assumption that users, as well as the data that they wish to access, is distributed in nature. Figure 9.1 depicts this idea of an archive that is accessible via many different network sites and by many different users. In order for ENFORMS to be usable by a wide group of users, it must be able to operate within the distributed computing paradigm. Requirements **R01004** and **R04001** are from the original requirement analysis document [112].

R01004 The system must provide distributed data access services.

R04001 The system will provide services that allow for the transparent distribution of data across a network of data archives.

9.4 System Level Modeling

Based on the requirement analysis, the system level object model, object functional model (OFM), and dynamic model shown in Figure 9.2 give a description of the system to be developed at a very high level of abstraction. The object model identifies the ENFORMS system that includes both data archives and corresponding data indices as the target system for development. The OFM depicts three services, *Browse*, *Retrieve*, and *Analysis*, that the system is supposed to provide. Based on data indices, service *Browse* takes user input and formulates retrieve request. Ser-

vice *Retrieve* retrieves records from data archives for given retrieve requests. Service *Analysis* performs analysis on the retrieved records according to user input. The dynamic model captures the behavior of the system where a *Retrieve* service request must be preceded by *Browse* requests, and *Analysis* service request must be preceded by *Retrieve* requests.

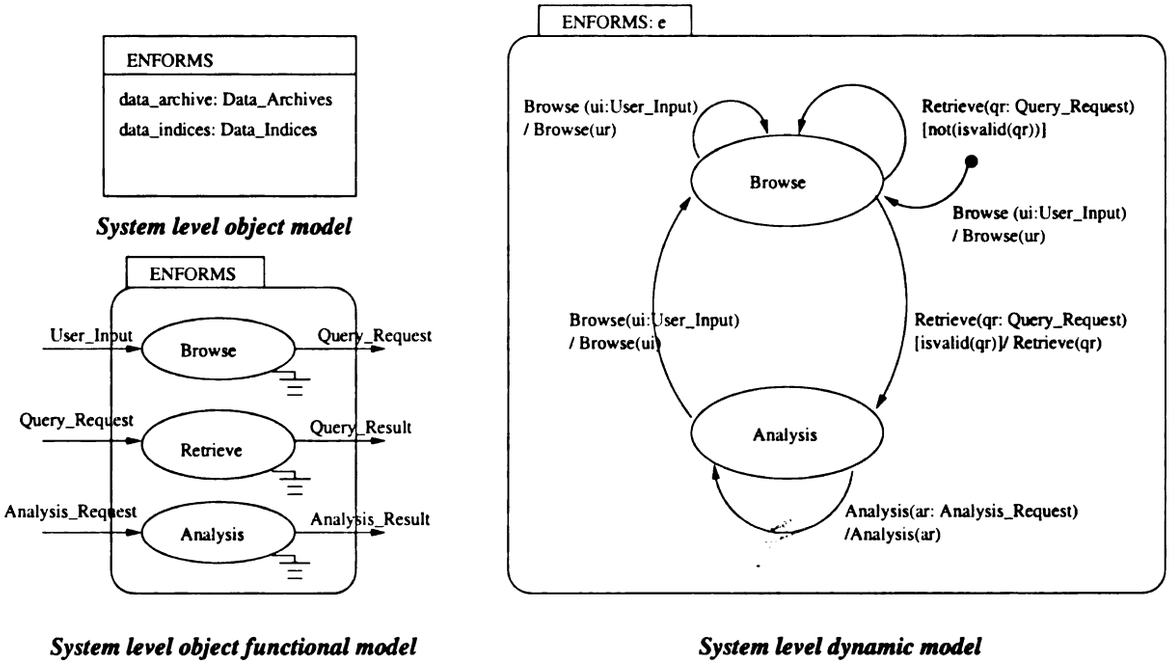

Figure 9.2: The design models of ENFORMS at the system level

Figures 9.3 9.4 show the formal specification of the system. The formalization rules used to generate the formal specification include **OM1**, **OM2**, **OM10**, **DFR-1**, **DFR-2**, **DFR-3**, **DFR-4**, **DFR-6**, **DFR-7**, **DFR-8**, **DFR-11**, **DFR-12**, **DFR-13**, **FFR-1**, **FFR-2**, **FFR-3**, and **FFR-4**. The specification can be automatically generated by applying the formalization rules given in the previous chapters. The specifications generated automatically from the diagrams give a formal description

of the services that the system provides and the behavioral dynamic aspects of the system.

```
specification ENFORMS [Browse, Retrieve, Analysis] (e: ENFORMS) : noexit
(* Browse:    ui: User_Input -> Retrieve_Request *)
(* Retrieve:    rr: Retrieve_Request -> Retrieve_Result *)
(* Analysis:    ar: Analysis_Request -> Analysis_Result *)
library
    Boolean
endlib
type ENFORMS is Boolean
    sorts
        Data_Archives, Data_Indices, User_Input, Retrieve_Request,
        Retrieve_Result, Analysis_Request, Analysis_Result, ENFORMS
    opns
        undef_Data_Archives : -> Data_Archives
        _ eq _ : Data_Archives, Data_Archives -> Bool
        undef_Data_Indices : -> Data_Indices
        _ eq _ : Data_Indices, Data_Indices -> Bool
        undef_User_Input : -> User_Input
        _ eq _ : User_Input, User_Input -> Bool
        undef_Retrieve_Request : -> Retrieve_Request
        _ eq _ : Retrieve_Request, Retrieve_Request -> Bool
        undef_Retrieve_Result : -> Retrieve_Result
        _ eq _ : Retrieve_Result, Retrieve_Result -> Bool
        undef_Analysis_Request : -> Analysis_Request
        _ eq _ : Analysis_Request, Analysis_Request -> Bool
        undef_Analysis_Result : -> Analysis_Result
        _ eq _ : Analysis_Result, Analysis_Result -> Bool
        undef_ENFORMS : -> ENFORMS
        _ eq _ : ENFORMS, ENFORMS -> Bool
        Browse:    User_Input -> Retrieve_Request
        Retrieve:  Retrieve_Request -> Retrieve_Result
        Analysis:  Analysis_Request -> Analysis_Result
        isvalid :  Retrieve_Request -> Bool
        ENFORMS      : Data_Archives, Data_Indices -> ENFORMS
        getArchives  : ENFORMS -> Data_Archives
        getIndices   : ENFORMS -> Data_Indices
endtype
```

Figure 9.3: The formal specification automatically generated from system models (1)

The automatically generated specification should be syntactically correct. Inter-model checking can be performed to check the consistency between the three models

```
behavior
    Browse?ui:User_Input; Browse!Browse(ui); Browse[Browse,Retrieve,Analysis](e)
where
process Browse[Browse,Retrieve,Analysis](e:ENFORMS):noexit:=
    Browse?ui:User_Input; Browse!Browse(ui);
        Browse[Browse,Retrieve,Analysis](e)
    []
    Retrieve?rr: Retrieve_Request;(
        [isvalid(rr)] -> Retrieve!Retrieve(rr);
            Analysis[Browse,Retrieve,Analysis](e)
        []
        [not(isvalid(rr))] -> Browse[Browse,Retrieve,Analysis](e)
    )
endproc
process Analysis[Browse,Retrieve,Analysis](e:ENFORMS):noexit:=
    Browse?ui:User_Input; Browse!Browse(ui);
        Browse[Browse,Retrieve,Analysis](e)
    []
    Analysis?ar:Analysis_Request; Analysis!Analysis(ar);
        Analysis[Browse,Retrieve,Analysis](e)
endproc
endspec
```

Figure 9.4: The formal specification automatically generated from system models (2)

of the ENFORMS system (see Figure 9.2). We use the *LOTOS semantics analyzer* (LSA) [64, 83] to ensure that the sorts and operations used in the dynamic models are previously modeled in the object and functional models and have corresponding definitions in the automatically generated formal specification. Figure 9.5 gives the transcript that runs LSA over the formal specification shown in Figure 9.3. The successful termination of the analysis indicates that no inter-model inconsistency is detected by LSA.

Although the automatically generated formal specification does not include a sophisticated algebraic specification that describe the properties of ENFORMS, the executable feature of the process algebras that describes the dynamic aspect of the

```
<131 aynow: > topo enforms_system01 -v -s
TOPO_3R6 (Mon Jan 23 15:19:15 MET 1995) /usr/local/LOTOS/Tools/TOPO
semantics analysis ...
lfe  enforms_system01.lot > Ta01614
mv Ta01614 enforms_system01.lfe
lsa -l design   -p enforms_system enforms_system.lfe
<132 aynow: >
```

Figure 9.5: The transcript of running LSA over ENFORMS formal specification to check inter-model consistency

system enables designers and end customers to symbolically execute the formal specification. The simulation provides an intuitive means to understand the system's functionality and behavior and a concrete approach that facilitates the communication among designers and customers. Figure 9.6 shows a transcript of the execution of the formal specification by using LOLA. Because no constants for the sorts are introduced in the automatically generated formal specification, the interactive simulation is quite high-level. No concrete values is given during the simulation process. The interactive execution confirms the requirement that a *Browse* request must precede a *Retrieve* request, a *Retrieve* request must precede an *analysis* request.

The formal specification automatically generated from diagrammatic models provides a framework into which algebraic specification can be added by designers. In order to further specify the properties of ENFORMS and to perform better simulation and analysis, more effort is needed to supply a well-defined algebraic specification for ENFORMS. Figure 9.7 shows the refined formal specification for ENFORMS with a better-defined algebraic specification manually derived by using the guidelines

```
 [  1]  analysis  ! analysis(ar_18);

<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> 1

  ==> analysis  ! analysis(ar_18);

-------------------------------------------------------------------------


 [  1]  browse  ? ui_19:user_input;
 [  2]  analysis  ? ar_20:analysis_request;

<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> t

 [  1] - browse  ? ui_4:user_input;
 [  1] - browse  ! browse(ui_4);
 [  2] - retrieve  ? rr_14:retrieve_request;
 [  1] - [isvalid(rr_14) = true]->
         retrieve  ! retrieve(rr_14);
 [  2] - analysis  ? ar_18:analysis_request;
 [  1] - analysis  ! analysis(ar_18);

<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?>
```

Figure 9.6: The transcript of running LOLA over ENFORMS formal specification for simulation

given in Chapter 6. The refined specification captures system properties, and contains pre- and postconditions that depict the functionalities of *Browse* and *Analysis* services.

In the refined formal specification, two operations (*Browse* and *Retrieve*), four constants (*dataArchives*, *dataIndices*, *valid_Retrieve_Request*, *valid_Retrieve_Result*), and a section that describes the properties of the operations in terms of equations are introduced into the algebraic specification part of the formal specification. The signatures of the newly introduced operations *Browse* and *Retrieve* are different from that of the original services *Browse* and *Retrieve* in Figures 9.3 and 9.4: operation

Retrieve has an additional input argument of sort *Data_Archives*; operation *Browse* has additional input argument of sort *Data_Indices*. The additional input arguments reflect the fact that internal data structures are involved for services *Browse* and *Retrieve*. The involvement of internal data structures is already expressed in terms of the *ground wire* in the corresponding OFM for services *Browse* and *Retrieve* in Figure 9.2. The two constants, *dataArchives* and *dataIndices*, of high-level abstraction, represent the available data archives and data indices. Constants *valid_User_Input*, *valid_Retrieve_Request* and *valid_Retrieve_Result* refer to the valid request arguments for services *Browse* and *Retrieve*, and the result that *Retrieve* issues. Given the constants, the equation section precisely defines the semantics of operations *isValid* and *Retrieve*. Since the distributed query feature is the focus of this case study, services *Browse* and *Analysis* are not further refined.

The newly introduced operations are used to describe the functionalities of the services in terms of pre- and postconditions. The pre- and postconditions for service *Retrieve* state that for a given valid retrieve request, ENFORMS checks the data archives and returns a valid result. Based upon the formalization rules for the pre- and postconditions, the pre- and postconditions are also integrated into the process algebra section of the specification, thus enabling more concrete simulation and testing.

Figure 9.8 contains a test process for the refined specification of ENFORMS. The test process contains two consecutive browse-retrieve tests each of which specifies that after a *Browse* service request, given a valid retrieve request, the ENFORMS system

```
specification ENFORMS [Browse, Retrieve, Analysis] (e: ENFORMS) : noexit

(* Browse:    ui: User_Input -> Retrieve_Request *)
(*    ensures    result = Browse (ui, getIndices(e)) *)
(* Retrieve:    rr: Retrieve_Request -> Retrieve_Result *)
(*    requires isValid(rr)    *)
(*    ensures result = Retrieve (rr, getArchives(e)) *)
(* Analysis:    ar: Analysis_Request -> Analysis_Result *)

library
    Boolean
endlib

type ENFORMS is Boolean
    sorts
        Data_Archives, Data_Indices, User_Input, Retrieve_Request,
        Retrieve_Result, Analysis_Request, Analysis_Result, ENFORMS
    opns
        ......
        dataArchives  : -> Data_Archives
        dataIndices   : -> Data_Indices
        valid_User_Input        : -> User_Input
        valid_Retrieve_Request  : -> Retrieve_Request
        valid_Retrieve_Result   : -> Retrieve_Result
        Retrieve  : Retrieve_Request, Data_Archives -> Retrieve_Result
        Browse    : User_Input, Data_Indices -> Retrieve_Request
    eqns
        forall rr, rrx, rry: Retrieve_Request, da: Data_Archives
        ofsort Bool
            isValid (valid_Retrieve_Request) = True;
            isValid (undef_Retrieve_Request) = False;
            isValid (valid_Retrieve_Result) = True;
            valid_Retrieve_Request eq valid_Retrieve_Request = True;
            undef_Retrieve_Request eq undef_Retrieve_Request = True;
            undef_Retrieve_Request eq valid_Retrieve_Request = False;
            rrx eq rry = rry eq rrx;
        ofsort Retrieve_Result
            isValid (rr) => Retrieve (rr, da) = valid_Retrieve_Result;
            not(isValid(rr)) => Retrieve (rr, da) = undef_Retrieve_Result;
endtype

behavior
......
endspec
```

Figure 9.7: The formal specification with algebraic specifications and pre/post-conditions

should return a valid result. If both tests are satisfied by the given system, a *success* event is reached.

```
process test [Browse, Retrieve, Analysis, success] : noexit :=
    Browse ! valid_User_Input;
    Browse ? rr: Retrieve_Request;
    Retrieve  ! valid_retrieve_request;
    Retrieve  ? rr: Retrieve_Result;
    ([isValid(rr)] -> Browse ! valid_User_Input;
                      Browse ? rr: Retrieve_Request;
                      Retrieve  ! valid_retrieve_request;
                      Retrieve  ? rr: Retrieve_Result;
                      ([isValid(rr)] -> success; stop))
endproc
```

Figure 9.8: A test process for the refined formal specification of ENFORMS

OneExpand is a LOLA function that makes a random symbolic execution of the current behavior. We use it to test the refined ENFORMS specification. The test process and the behavior specification are composed in parallel, synchronizing all the gates, except for the *success* event. LOLA analyzes whether the executions reach the *success* event or not. A testing transcript that runs *OneExpand* with process *test* is shown in Figure 9.9. The test reaches the *success* event at the ninth step of symbolic execution.

Unlike *OneExpand*, which picks up a random symbolic execution, *TestExpand* can perform an exhaustive test that explores all the symbolic execution paths of a specified system. Figure 9.10 shows the testing transcript that applies the LOLA *TestExpand* operation to the refined ENFORMS formal specification. Although unlimited depth of exploration is permitted for the exhaustive test, there is actually only one execution

```
lola> oneexpand -1 success test 23 -v
oneexpand -1 test 23 1 -v

 Rewriting expressions in the specification.
 Rewriting done.
 Analysing unguarded conditions.
 Analysis done.

  Composing behaviour and test :

     test [browse,retrieve,analysis,success]
   |[browse,retrieve,analysis]|
     browse  ? ui_8:user_input;
     browse  ! browse(ui_8,getindices(e_7));
     browse [browse,retrieve,analysis] (e_7)

    1  browse  ! valid_user_input;
    2  browse  ! browse(valid_user_input,getindices(e_7));
    3  retrieve  ! valid_retrieve_request;
    4  retrieve  ! valid_retrieve_result;
    5  browse  ! valid_user_input;
    6  browse  ! browse(valid_user_input,getindices(e_7));
    7  retrieve  ! valid_retrieve_request;
    8  retrieve  ! valid_retrieve_result;
    9  success;

    Process Test = test
    Test result  = SUCCESSFUL EXECUTION.

       Transitions generated = 9
```

Figure 9.9: The transcript of testing that runs *OneExpand* of LOLA over the refined ENFORMS formal specification

path for the given test process. During the exhaustive exploration of testing, 9 states are analyzed, 9 transitions are generated, but no deadlock is detected. The execution reached the *success* event at the ninth step. Because no deadlock is detected during the exhaustive testing, *MUST PASS* is given as the *test result* indicating that the *success* event is always reached. The symbolic test, thus far, has shown very promising testing results. Based upon the testing result, we can confidently claim that for valid retrieve requests, the designed system is very likely to return a valid result. Since the objective of testing is to detect errors in design instead of proving correctness, the fact that no design flaw is reported after a large amount of exhaustive testing does not guarantee that the design is flawless.

9.5 System Design

The system design of the ENFORMS system is based on the design presented in [112].

Overview

ENFORMS is designed as a distributed system, where the functionality of the system is realized through a collection of communicating software components. The distributed operation of ENFORMS is supported via a *client-server* architectural style. In this style, *servers* accept requests over a network, perform the relevant services, and return the results. The *Client* is a software component that requests the services. In the case of the ENFORMS client, the purpose of the requested services

```
lola> test success test -y -i
testexpand -1 success test -s -v 0 -y

 Rewriting expressions in the specification.
 Rewriting done.
 Analysing unguarded conditions.
 Analysis done.

  Composing behaviour and test :

     test [browse,retrieve,analysis,success]
   |[browse,retrieve,analysis]|
     browse  ? ui_8:user_input;
     browse  ! browse(ui_8,getindices(e_7));
     browse [browse,retrieve,analysis] (e_7)

    Analysed states       = 9
    Generated transitions = 9
    Duplicated states     = 0
    Deadlocks             = 0

    Process Test = test
    Test result  = MUST PASS.

                  successes = 1
                      stops = 0
                      exits = 0
              cuts by depth = 0
```

Figure 9.10: The transcript of testing that runs *TestExpand* of LOLA over the refined ENFORMS formal specification

is to support the functionality realized at the GUI level. The requests that may be issued by a client are categorized into two types. First, *Query* requests may be issued for selecting data items from data archives. Second, *Server Table* requests may be issued for determining the set of available servers. For each of these cases, the communication between a client and a server is essentially hidden from the user. That is, the underlying details of establishing connections and transmitting messages are handled without user intervention.

Two main types of servers are supported by ENFORMS. The first is an *Archive Server* that manages the searching and the manipulation of items for an archive site. The services provided by an archive server directly support the *Query* request described above. The other type of server supported by ENFORMS is referred to as the *Name Server.* The name server is responsible for maintaining a table of the active archive servers, referred to as the *Server Table.* The address and port number of the name server is established prior to the activation of ENFORMS, such that when an archive server is activated, the location of the name server is already known. Using this information, the archive server sends a *registration* message to the name server, providing its address for communicating with clients. The name server then stores that information as an entry in the server table. Likewise, when an archive server is deleted, the corresponding entry in the server table is removed. The name server is also responsible for handling server table requests from clients.

Multiple archive servers may be active at one time, where each server provides services for a particular archive site. However, a single instance of a name server answers the server table requests for the entire collection of clients in ENFORMS. An additional characteristic of the distributed operation of ENFORMS is that a client may send requests to any active archive server, and, likewise, an archive server can provide services to any of the active clients. The relationship between the name server, the archive servers, and the clients is illustrated in the architectural overview shown in Figure 9.11. Figure 9.11 describes that the ENFORMS system is composed of (diamond) multiple (0 or more) *clients* and *Archive Servers*, and a single *Name Server.*

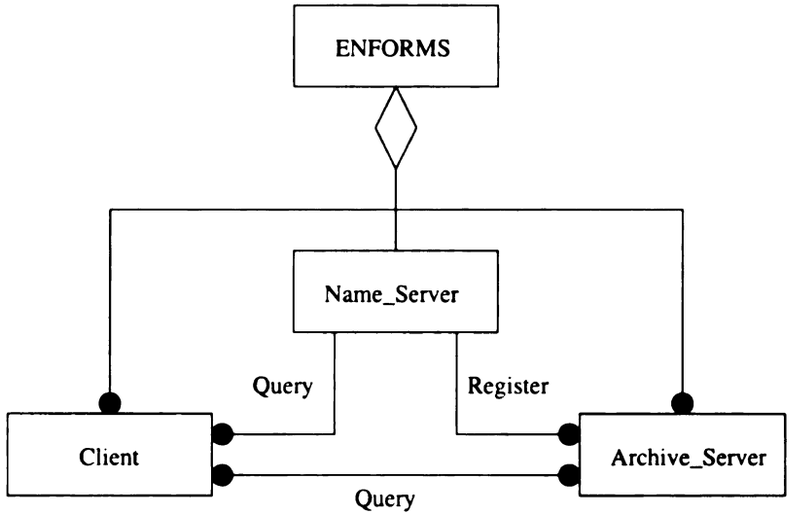

Figure 9.11: The refined object model for ENFORMS: an overview of the architecture

The execution of the ENFORMS system is initiated by activating the name server. The archive servers are then typically activated, followed by any number of clients, although the system does not preclude the possibility of activating the clients before the archive servers. During the normal operation of the system, archive servers and clients may be started or stopped at any time, while the name server remains continuously active.

9.5.1 Design models for individual objects

The object, functional, and dynamic models for *Name_Server* are given in Figure 9.12. Services *Register* and *GetTable* are provided by *Name_Server* for external services. The *Register* service accepts registration requests from *Archive Servers* and puts

the corresponding records into *Server Table.* Service *GetTable* returns the current *Server_Table* upon a request from *Client.*

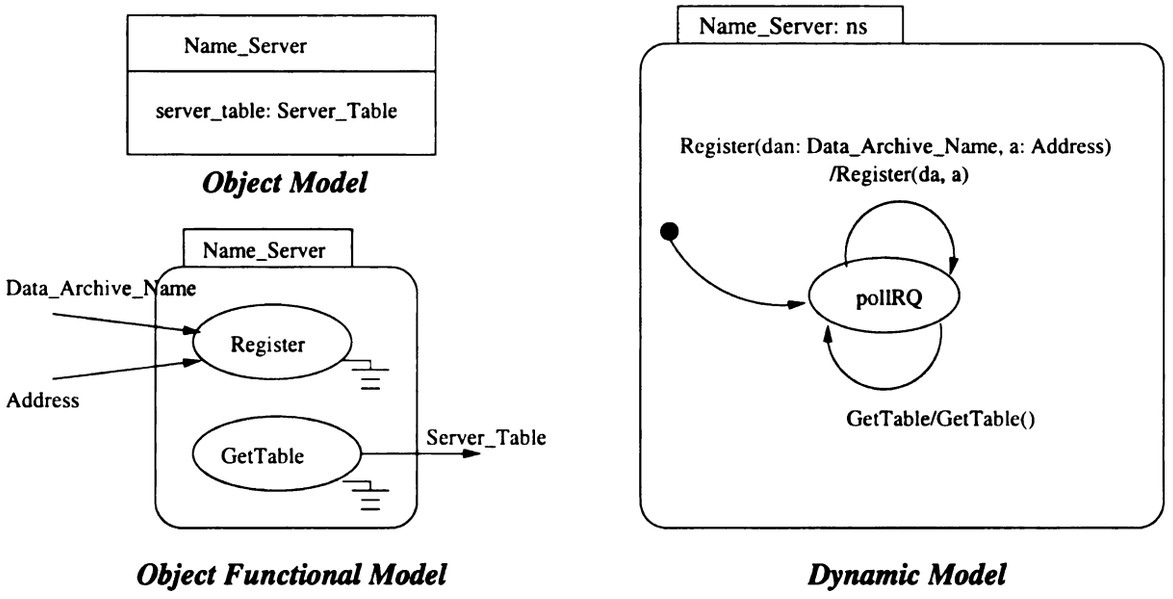

Figure 9.12: The design models for *Name_Server*

The formal specification automatically generated from the design models of *Name_Server* is shown in Figure 9.13. The formalization rules used to generate the formal specification include **OM1**, **OM2**, **OM10**, **DFR-1**, **DFR-2**, **DFR-3**, **DFR-4**, **DFR-6**, **DFR-7**, **DFR-8**, **DFR-11**, **DFR-12**, **DFR-13**, **FFR-1**, **FFR-2**, **FFR-3**, and **FFR-4**. The formal specification in Figure 9.14 extends the specification in Figure 9.13 with well-defined algebraic specifications, supplied by the specifiers according to the guidelines given in Chapter 6, that describe the properties of *Name_Server* and pre- and postconditions that capture the functionalities of the services.

The object, functional, and dynamic models for *Archive_Server* are given in Figure 9.15. Service *Query* is provided by *Archive_Server.* During the initialization of

```
specification Name_Server [Register, GetTable] (ns: Name_Server) : noexit

(* Register:    dan: Data_Archive_Name, a: Address -> Void *)
(* GetTable:     -> Server_Table *)

library
    ENFORMS, Boolean
endlib

type Name_Server is ENFORMS, Boolean
    sorts
        Name_Server, Server_Table, Data_Archive_Name, Address, Void
    opns
        undef_Name_Server : -> Name_Server
        _ eq _          : Name_Server, Name_Server -> Bool
        undef_Server_Table : -> Server_Table
        _ eq _          : Server_Table, Server_Table -> Bool
        undef_Data_Archive_Name : -> Data_Archive_Name
        _ eq _          : Data_Archive_Name, Data_Archive_Name -> Bool
        undef_Address    : -> Address
        _ eq _          : Address, Address -> Bool

        Register : Data_Archive_Name, Address -> Void
        GetTable : -> Server_Table

        Name_Server    : Server_Table -> Name_Server
        getTable    : Name_Server -> Server_Table
    eqns
        forall st: Server_Table
        ofsort Server_Table
            getTable (Name_Server(st)) = st;
endtype

behavior
    pollRQ [Register, GetTable] (ns)

where

process pollRQ [Register, GetTable] (ns: Name_Server) : noexit :=
    Register ? dan: Data_Archive_Name ? a: Address; Register !Register(dan, a);
        pollRQ[Register, GetTable] (ns)
    []
    GetTable; GetTable ! GetTable; pollRQ [Register, GetTable] (ns)
endproc
endspec
```

Figure 9.13: The automatically generated formal specification for *Name_Server*

```
specification Name_Server [Register, GetTable] (ns: Name_Server) : noexit

(* Register:    dan: Data_Archive_Name, a: Address -> Void *)
(*    ensures ns' = Name_Server(insert(getTable(ns^), dan, a)) *)
(* GetTable:     -> Server_Table *)
(*    ensures result = getTable(ns) *)

library
    ENFORMS, Boolean
endlib

type Name_Server is ENFORMS, Boolean
    sorts
        Name_Server, Server_Table, Data_Archive_Name, Address
    opns
        ......
        empty        : -> Server_Table
        insert        : Server_Table, Data_Archive_Name, Address -> Server_Table
        getAddress    : Server_Table, Data_Archive_Name -> Address
        delete        : Server_Table, Data_Archive_Name -> Server_Table
    eqns
        forall st: Server_Table, dan1, dan: Data_Archive_Name, a: Address
        ofsort Address
            getAddress (empty, dan) = undef_Address;
            dan1 eq dan => getAddress (insert(st, dan1, a), dan) = a;
            not(dan1 eq dan) =>
                getAddress (insert(st,dan1,a),dan) = getAddress (st,dan);
        ofsort Server_Table
            delete (empty, dan) = empty;
            dan1 eq dan => delete (insert(st, dan1, a), dan) = st;
            not(dan1 eq dan) =>
                delete (insert(st,dan1,a),dan) = insert(delete(st,dan),dan1,a);
            getTable (Name_Server(st)) = st;
endtype

behavior
......
endspec
```

Figure 9.14: The formal specification for *Name_Server* with algebraic specifications and pre/post-conditions

an *Archive_Server*, it registers its name and address to a *Name Server*, then enters the *pollRQ* state to wait for query requests from *Clients*. The *Query* service accepts query requests from *Clients*, performs the corresponding query, and returns the result of the query.

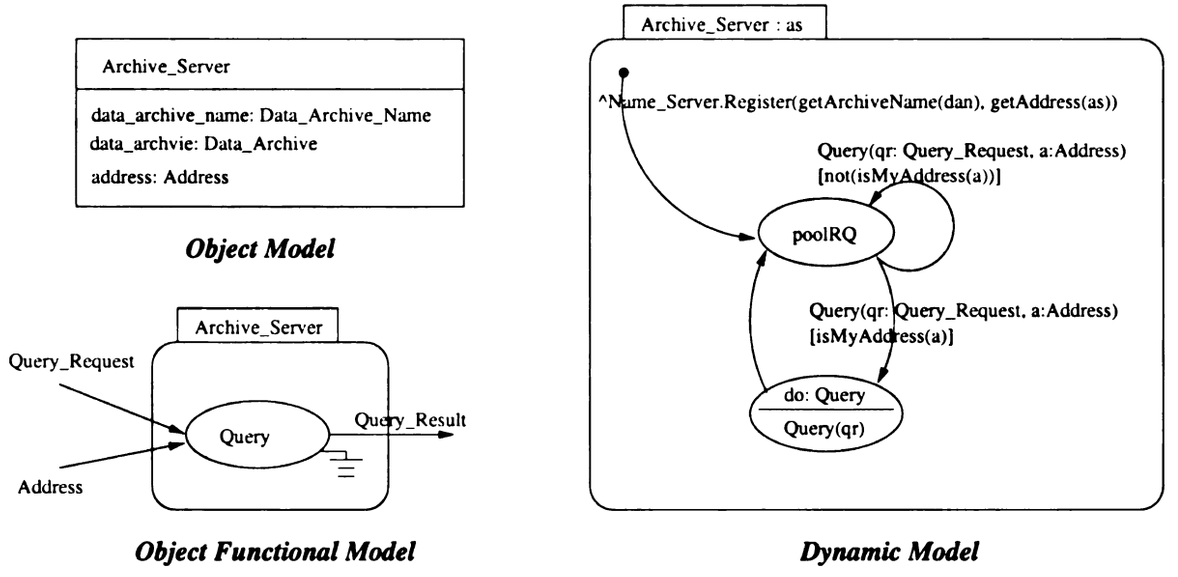

Figure 9.15: The design models for *Archive_Server*

The formal specification automatically generated from the design models of *Archive_Server* is shown in Figures 9.16 and 9.17. The formalization rules used to generate the formal specification include **OM1**, **OM2**, **OM10**, **DFR-1**, **DFR-2**, **DFR-3**, **DFR-4**, **DFR-6**, **DFR-7**, **DFR-8**, **DFR-9**, **DFR-11**, **DFR-12**, **DFR-13**, **FFR-1**, **FFR-2**, **FFR-3**, and **FFR-4**. The formal specification in Figure 9.18 extends the specification with well-defined algebraic specifications, supplied by the specifiers according to the guidelines given in Chapter 6, that describes the properties

of *Archive_Server* and pre- and postconditions that captures the functionalities of the services.

```
specification Archive_Server[Query,Name_Server_Register]
                (as:Archive_Server):noexit
(* Query:    qr: Query_Request, a: Address -> Query_Result *)
library
    ENFORMS, Boolean
endlib
type Archive_Server is ENFORMS, Boolean
    sorts
        Archive_Server, Data_Archive_Name,
        Data_Archive, Address, Query_Request, Query_Result
    opns
        undef_Archive_Server : -> Archive_Server
        _ eq _ : Archive_Server, Archive_Server -> Bool
        undef_Data_Archive_Name : -> Data_Archive_Name
        _ eq _ : Data_Archive_Name, Data_Archive_Name -> Bool
        undef_Data_Archive : -> Data_Archive
        _ eq _ : Data_Archive, Data_Archive -> Bool
        undef_Address : -> Address
        _ eq _ : Address, Address -> Bool
        undef_Query_Request : -> Query_Request
        _ eq _ : Query_Request, Query_Request -> Bool
        undef_Query_Result : -> Query_Result
        _ eq _ : Query_Result, Query_Result -> Bool
        Query : Query_Request, Address -> Query_Result
        isMyAddress : Address, Archive_Server -> Bool
        Archive_Server : Data_Archive_Name, Data_Archive, Address -> Archive_Server
        getArchive    : Archive_Server -> Data_Archive
        getAddress    : Archive_Server -> Address
        getArchiveName    : Archive_Server -> Data_Archive_Name
    eqns
        forall dan: Data_Archive_Name, da: Data_Archive, a: Address,
               as: Archive_Server, qr: Query_Request
        ofsort Data_Archive_Name
            getArchiveName (Archive_Server(dan, da, a)) = dan;
        ofsort Data_Archive
            getArchive (Archive_Server(dan, da, a)) = da;
        ofsort Address
            getAddress (Archive_Server(dan, da, a)) = a;
endtype
```

Figure 9.16: The automatically generated formal specification for *Archive Server* (1)

```
behavior
    Name_Server_Register !getArchiveName(as) !getAddress(as);
        pollRQ [Query, Name_Server_Register] (as)
where
process pollRQ [Query, Name_Server_Register] (as: Archive_Server) : noexit :=
    Query ? qr: Query_Request ? a: Address; (
    [isMyAddress(a, as)] -> Query [Query, Name_Server_Register] (as, qr, a)
    []
    [not(isMyAddress(a, as))] -> pollRQ [Query, Name_Server_Register] (as))
endproc
process Query[Query,Name_Server_Register]
        (as:Archive_Server,qr:Query_Request,a: Address):noexit:=
    i; Query ! Query (qr, a); pollRQ [Query, Name_Server_Register] (as)
endproc
endspec
```

Figure 9.17: The automatically generated formal specification for *Archive Server* (2)

The object, functional, and dynamic models for *Client* are given in Figure 9.19. Service *Query* is provided by *Client.* During the initialization time, a client retrieves a *Server Table* that contains the current available archive servers from the name server, then enters the *idle* state to wait for query requests from ENFORMS GUI. Once a query request occurs, the client looks up the server table, issues the query request to the corresponding archive server, waits for the query result from the archive, then returns the result of the query.

The formal specification automatically generated from the *Client* design models is shown in Figure 9.20. The formalization rules used to generate the formal specification include **OM1**, **OM2**, **OM10**, **DFR-1**, **DFR-2**, **DFR-3**, **DFR-4**, **DFR-5**, **DFR-8**, **DFR-11**, **DFR-12**, **DFR-13**, **FFR-1**, **FFR-2**, **FFR-3**, and **FFR-4**. The formal specification in Figure 9.22 extends the specification in Figures 9.20 and 9.21 with well-defined algebraic specifications, supplied by the specifiers according to the

```
specification Archive_Server [Query, Name_Server_Register]
              (as: Archive_Server) : noexit

(* Query:    qr: Query_Request, a: Address -> Query_Result *)
(*    ensures result = Query (qr, getArchive (as)) *)

library
    ENFORMS, Boolean
endlib

type Archive_Server is ENFORMS, Boolean
    sorts
        Data_Archive_Name, Data_Archive, Address,
        Archive_Server, Query_Request, Query_Result
    opns
        ......
    eqns
        ......
        ofsort Bool
            isMyAddress (a, as) = a eq getAddress(as);
            isValid (valid_Query_Request) = true;
        ofsort Query_Result
            isValid (qr) => Query(qr, da) = valid_Query_Result;
            not(isValid(qr)) => Query(qr, da) = undef_Query_Result;
endtype

behavior
......
endspec
```

Figure 9.18: The formal specification for *Archive_Server* with algebraic specifications and pre/post-conditions

guidelines given in Chapter 6, that describes the properties of *Client* and pre- and postconditions that capture the functionalities of the services.

Similar to what we have done to the system level design model of ENFORMS (see Figure 9.2), simulation and analyses can also be performed by manipulating the derived formal specifications for the objects introduced above.

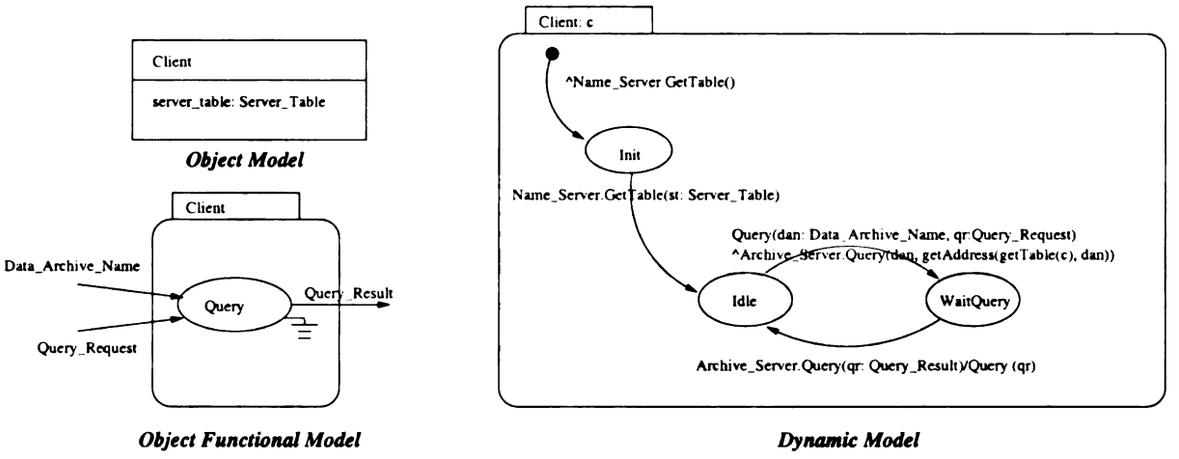

Figure 9.19: The design models for *Client*

9.5.2 The SRFM and refined dynamic model for ENFORMS

Given the design models of the aggregate objects *Client*, *Name Server*, and *Archive_Server*, the services of ENFORMS at the system level can be refined in terms of the services provided by the aggregate objects. Because the case study focuses on the distributed query feature of ENFORMS, the SRFM of service *Retrieve* of ENFORMS is given in Figure 9.23. The formalization rules used to generate the formal specification include **FFR-10**, **FFR-11**, **FFR-13**, and **FFR-14**.

The SRFM shows how ENFORMS makes use of the *Query* service supplied by the *Client* object in the client-server design architecture to implement the *Retrieve* functionality. The examination of services *Retrieve* and *Query* reveals that the input data flows of the two are incompatible: the input data flow for *Retrieve* is *Retrieve_Request* while those for *Query* are *Data_Archive_Name* and *Query_Request*. This leads to the data flow refinement of *Retrieve_Request* in the SRFM. The graphical notation *splitter*

```
specification Client [Query, Name_Server_GetTable, Archive_Server_Query]
              (c: Client) : noexit

(* Query:  qr: Query_Request, dan: Nata_Archive_Name -> Query_Result *)

library
    ENFORMS, Boolean
endlib

type Archive_Server is ENFORMS, Boolean
    sorts
        Data_Archive_Name, Address, Query_Request,
        Query_Result, Server_Table, Client
    opns
        undef_Data_Archive_Name : -> Data_Archive_Name
        _ eq _ : Data_Archive_Name, Data_Archive_Name -> Bool
        undef_Address : -> Address
        _ eq _ : Address, Address -> Bool
        undef_Query_Request : -> Query_Request
        _ eq _ : Query_Request, Query_Request -> Bool
        undef_Query_Result : -> Query_Result
        _ eq _ : Query_Result, Query_Result -> Bool
        undef_Server_Table : -> Server_Table
        _ eq _ : Server_Table, Server_Table -> Bool
        undef_Client : -> Client
        _ eq _ : Client, Client -> Bool

        Client    : Server_Table -> Client
        getTable    : Client -> Server_Table
    eqns
        forall st: Server_Table
        ofsort Server_Table
            getTable (Client(st)) = st;
endtype
```

Figure 9.20: The automatically generated formal specification for *Client*

```
behavior

    Name_Server_GetTable;
        Init [Query, Name_Server_GetTable, Archive_Server_Query] (c)

where

process Init [Query, Name_Server_GetTable, Archive_Server_Query]
             (c: Client) : noexit :=
    Name_Server_GetTable ? st: Server_Table;
        Idle [Query, Name_Server_GetTable, Archive_Server_Query] (c)
endproc

process Idle [Query, Name_Server_GetTable, Archive_Server_Query]
             (c: Client) : noexit :=
    Query ? dan: Data_Archive_Name ? qr: Query_Request;
    Archive_Server_Query ! dan ! getAddress(getTable(c), dan);
    WaitQuery [Query, Name_Server_GetTable, Archive_Server_Query] (c)
endproc

process WaitQuery [Query, Name_Server_GetTable, Archive_Server_Query]
        (c: Client) : noexit :=
    Archive_Server_Query ? qr: Query_Result; Query ! qr;
        Idle [Query, Name_Server_GetTable, Archive_Server_Query] (c)
endproc
endspec
```

Figure 9.21: The automatically generated formal specification for *Client*

(the solid box) splits *Retrieve_Request* into *Data_Archive_Name* and *Query_Request*. The SRFM also indicates that the output of *Query* is directly used as the output of *Retrieve*. The formal specification automatically generated from the SRFM is given in Figure 9.24.

In the formal specification further refined (by a developer) for ENFORMS, data sort *Retrieve_Request* is refined as a tuple of sorts *Data_Archive_Name* and *Query_Request*. This refinement is specified in terms of newly introduced operations *Retrieve_Request*, *get_data_archive_name*, *get_query_request*, and their corresponding

```
specification Client [Query, Name_Server_GetTable, Archive_Server_Query]
              (c: Client) : noexit

(* Query:  qr: Query_Request, dan: Nata_Archive_Name -> Query_Result *)

library
    ENFORMS
endlib

type Client is ENFORMS
    sorts
        Data_Archive_Name, Address, Query_Request,
        Query_Result, Server_Table, Client
    opns
        ......
        empty         : -> Server_Table
        insert         : Server_Table, Data_Archive_Name, Address -> Server_Table
        getAddress     : Server_Table, Data_Archive_Name -> Address
        delete         : Server_Table, Data_Archive_Name -> Server_Table
    eqns
        forall st: Server_Table, dan1, dan: Data_Archive_Name, a: Address
        ofsort Address
            getAddress (empty, dan) = undef_Address;
            dan1 eq dan => getAddress (insert(st, dan1, a), dan) = a;
            not(dan1 eq dan) =>
                getAddress (insert(st, dan1, a), dan) = getAddress (st, dan);

        ofsort Server_Table
            delete (empty, dan) = empty;
            dan1 eq dan => delete (insert(st, dan1, a), dan) = st;
            not(dan1 eq dan) =>
                 delete(insert(st,dan1,a),dan) = insert(delete(st,dan),dan1,a);
            getTable (Client(st)) = st;
endtype

behavior
......
endspec
```

Figure 9.22: The formal specification for *Client* with algebraic specifications and pre/post-conditions

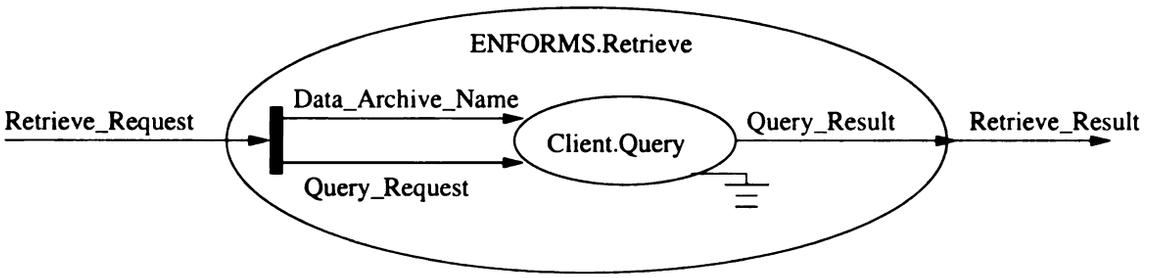

Figure 9.23: The SRFM for *Retrieve* service of ENFORMS

```
type ENFORMS_REF is ENFORMS, Name_Server, Archive_Server, Client
     opns
         Retrieve_Request : Data_Archive_Name, Query_Request -> Retrieve_Request
         get_data_archive_name : Retrieve_Request -> Data_Archive_Name
         get_query_request : Retrieve_Request -> Query_Request
         Retrieve_Result2Query_Result : Retrieve_Result -> Query_Result
         Query_Result2Retrieve_Result : Query_Result -> Retrieve_Result
     eqns
         forall dan, dan1, dan2: Data_Archive_Name, qr: Query_Request,
                rr: Retrieve_Result, qr1: Query_Result
         ofsort Data_Archive_Name
             get_data_archive_name (Retrieve_Request(dan, qr)) = dan;
         ofsort Query_Request
             get_query_request (Retrieve_Request(dan, qr)) = qr;
         ofsort Retrieve_Result
             Query_Result2Retrieve_Result(Retrieve_Result2Query_Result(rr)) = rr;
         ofsort Query_Result
             Retrieve_Result2Query_Result(Query_Result2Retrieve_Result(qr1)) = qr1;
endtype
```

Figure 9.24: The automatically generated formal specification for the SRFM of *Retrieve*

equations. The formalization rules **FFR-14** and **FFR-15** are given in Chapter 6. Operations *Retrieve_Result2Query_Result* and *Query_Result2Retrieve_Result* together with the related equations specify that sorts *Query_Result* and *Retrieve_Result* are convertible to one another.

Based upon the SRFM given in Figure 9.23, the dynamic model of ENFORMS is also refined in order to reflect the design that the *Retrieve* service of ENFORMS is realized in terms of the *Query* services supplied by *Client.* Figure 9.25 gives the refined dynamic model of ENFORMS. Since the object models describe the primary static structure of a system, they are not affected by the SRFMs.

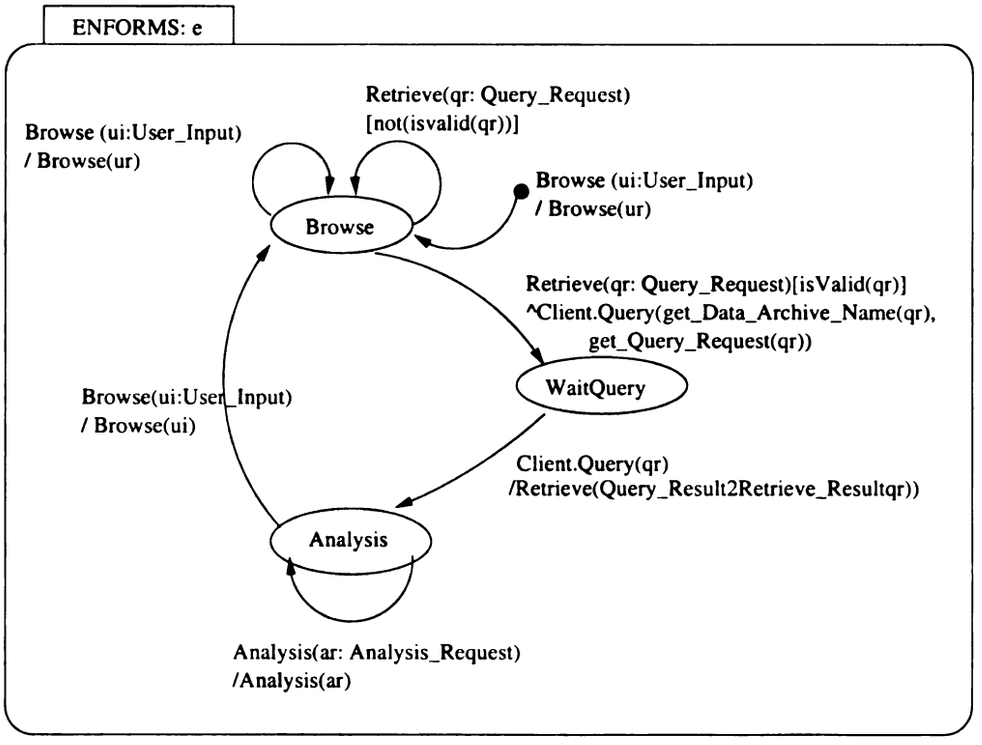

Figure 9.25: The refined design model for ENFORMS

In the refined dynamic model, a state, *WaitQuery*, is added to reflect the design that the *Retrieve* service of ENFORMS is realized in terms of the *Query* services supplied by *Client*. The changed part of the model specifies that: (1) once a *Retrieve* request occurs, the input argument is processed to split into *Data_Archive_Name* and *Query_Request*, (2) the request is redirected to a *Client* object for a *Query* service, (3) the system enters *WaitQuery* state to wait for the query result from the *Client*, and (4) after getting the query result from the *Client*, the system returns the converted result to the end user. The formal specification automatically generated from the refined dynamic model is shown in Figure 9.26. The formalization rules used to generate the formal specification include **DFR-1**, **DFR-2**, **DFR-3**, **DFR-4**, **DFR-5**, **DFR-6**, **DFR-7**, **DFR-8**, **DFR-11**, **DFR-12**, and **DFR-13**.

```
process Browse [Browse, Retrieve, Analysis, Client_Query] (e: ENFORMS) : noexit :=
    Browse ? ui: User_Input; Browse ! Browse (ui, getIndices(e));
        Browse[Browse, Retrieve, Analysis, Client_Query] (e)
    []
    Retrieve ? rr: Retrieve_Request; (
        [isValid(rr)] ->
            Client_Query ! get_data_archive_name (rr) ! get_query_request (rr);
            WaitQuery [Browse, Retrieve, Analysis, Client_Query] (e)
        []
        [not(isValid(rr))] ->
            Browse[Browse, Retrieve, Analysis, Client_Query] (e)
    )
endproc

process WaitQuery[Browse,Retrieve,Analysis,Client_Query](e:ENFORMS):noexit:=
    Client_Query ? rr: Query_Result; Retrieve ! Query_Result2Retrieve_Result (rr);
        Analysis [Browse, Retrieve, Analysis, Client_Query] (e)
endproc
```

Figure 9.26: The automatically generated formal specification from the refined dynamic model of ENFORMS

9.5.3 Composing the objects in parallel

The analysis of the formal specifications of individual objects enables designers to detect design flaws brought by inter-model inconsistency and incorrect behavior modeling for individual objects. Since the individual objects are designed to coordinate with each other in order to implement the functionalities introduced at a higher level of abstraction, it is necessary to compose the dynamic models of the concurrent objects and check if the objects, together, can realize the necessary functionalities (as depicted by the dynamic model in Figure 9.27).

For this case study, the analysis is focused on checking whether the objects designed in the client-server architectural style will realize the distributed query transparently. Figure 9.27 gives a state diagram that composes the dynamic models of ENFORMS, *Client*, *Name_Server*, and *Archive_Server* in parallel.

In the state diagram, the parameters of the distinguished sorts (*Name_Server*, *Client*, and *Archive_Server*) for the objects are instantiated. Both the *Client* and *Name_Server* objects are given *empty* server tables. Several data archives to which ENFORMS must provide access are described in [112]. Two of the data archives are used to instantiate the two *Archive_Server* objects.

STORET Ambient Water Quality Data: STORET archives data resulting from the chemical analysis of water samples taken periodically from selected sites. Samples are not necessarily taken at regular intervals for all sites. For each sample, STORET gives a site description (latitude/longitude or landmarks) where the sample was taken, date and time of sample, depth of sample, and an analysis of the sample (water temperature, nutrient levels, dissolved oxygen, etc.). The STORET archives are available via the EPA mainframe.

Permit Compliance System (PCS) Data: PCS contains data on permitted surface water discharge sites, such as storm drains and waste discharge pipes.

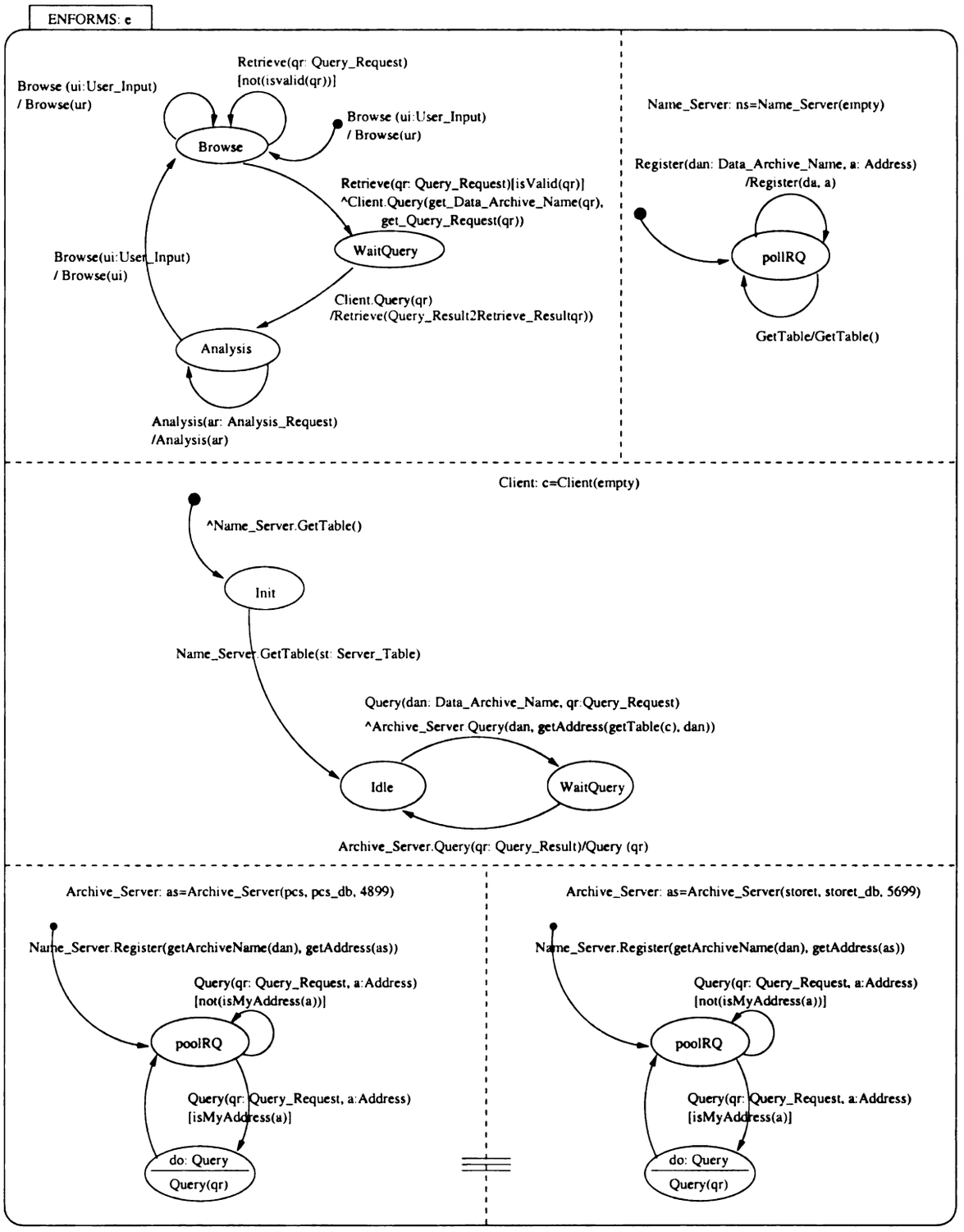

Figure 9.27: The dynamic models composed in parallel

> For each permit, an ID code for the discharging facility is given. The type of discharge is described and the limits on the discharges are specified. The PCS data is available via the EPA mainframe, and from Great Lakes Envirofacts.

Since the design is at a very high level of abstraction, it is neither necessary nor realistic to describe the two data archives in detail in terms of algebraic specifications. Instead, we introduce constants *storet* and *pcs* to represent the archive names, and *storet_db* and *pcs_db* to represent the archives for *STORET* and *PCS*, respectively. Similarly, instead of developing a complete theory that describes the addresses for the *Archive Server* objects, constants *4899* and *5699* of sort *Address* are randomly chosen to refer to the two *Archive_Server* objects.

Based upon the formalization rules **DFR-18** and **DFR-19** given in Chapter 5, algebraic specifications can be automatically generated to describe the newly introduced constants. The ACT ONE specification in Figure 9.28 specifies the constants in terms of *nullary* operations; it also asserts that every constant is equal to itself but distinguished from other constants of the same sort.

9.6 Design Analysis and Refinement

This section describes the analysis of diagrams via their formal specifications. It also discusses the refinement of the models based on errors detected during analysis.

9.6.1 Design flaw detected in models

Chapter 5 introduces *name binding* as the mechanism that synchronizes the dynamic models of the concurrent objects. During the process that synchronized the dy-

```
type ENFORMS_REF is ENFORMS, Name_Server, Archive_Server, Client
    opns
        pcs     : -> Data_Archive_Name
        pcs_db     : -> Data_Archive
        storet     : -> Data_Archive_Name
        storet_db : -> Data_Archive
        4899     : -> Address
        5699     : -> Address
    eqns
        ofsort Bool
            pcs eq pcs = True;
            storet eq storet = True;
            undef_Data_Archive_Name eq undef_Data_Archive_Name = True;
            pcs eq undef_Data_Archive_Name = False;
            undef_Data_Archive_Name eq pcs = False;
            storet eq undef_Data_Archive_Name = False;
            undef_Data_Archive_Name eq storet = False;
            pcs eq storet = False;
            storet eq pcs = False;
            pcs_db eq pcs_db = True;
            storet_db eq storet_db = True;
            undef_Data_Archive eq undef_Data_Archive = True;
            pcs_db eq undef_Data_Archive = False;
            undef_Data_Archive eq pcs_db = False;
            storet_db eq undef_Data_Archive = False;
            undef_Data_Archive eq storet_db = False;
            pcs_db eq storet_db = False;
            storet_db eq pcs_db = False;
            4899 eq 4899 = True;
            5699 eq 5699 = True;
            undef_Address eq undef_Address = True;
            4899 eq undef_Address = False;
            undef_Address eq 4899 = False;
            5699 eq undef_Address = False;
            undef_Address eq 5699 = False;
```

Figure 9.28: The automatically generated formal specification that describes the constants in the refined models of ENFORMS

namic models through name binding, we encountered a problem. Since a *Client* object can communicate with multiple *Archive_Server* objects, the service request *Archive_Server.Query* of the *Client* object shall be bound with the *Query* services provided by both of the *Archive Server* objects. Because the two *Archive_Server* objects are interleaved with one another (represented by the three parallel lines on their *border*), given the semantics of LOTOS, either of them can respond to a *Query* request from the *Client* object once they are bound. Thus nondeterminism results. Since the nondeterminism is not a feature of the intended design, it must be eliminated.

A further investigation reveals that there is a modeling flaw in our design. The distributed system is designed in terms of a client-server architecture. An underlying communication mechanism between clients and servers is assumed in the system design. However, the assumption is not modeled in the design, thus resulting in the nondeterminism during the communication between clients and servers. In order to eliminate the nondeterminism, the design models need to be refined in order to incorporate the assumed communication support.

The *Channel* object class. Figure 9.29 shows a refined object model that introduces a *Channel* object class to facilitate the communication between *Client* objects and *Archive_Server* objects.

The modeling of *Channel* is relatively simple, because we do not intend to model the entire *operating system* and *networking* support in this case study. Figure 9.30 gives the three OMT models for *Channel*. The *Channel* class is modeled exclusively to be used by the *Client* object to communicate with the two *Name_Server* objects.

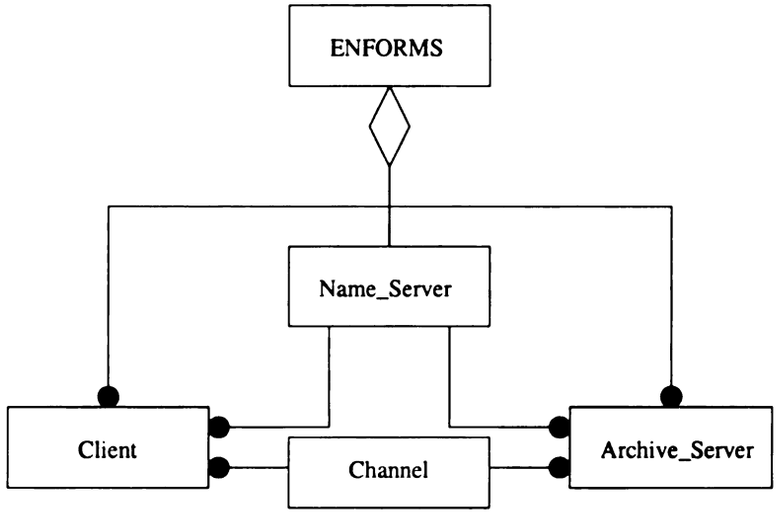

Figure 9.29: The refined object model for ENFORMS with *Channel* object

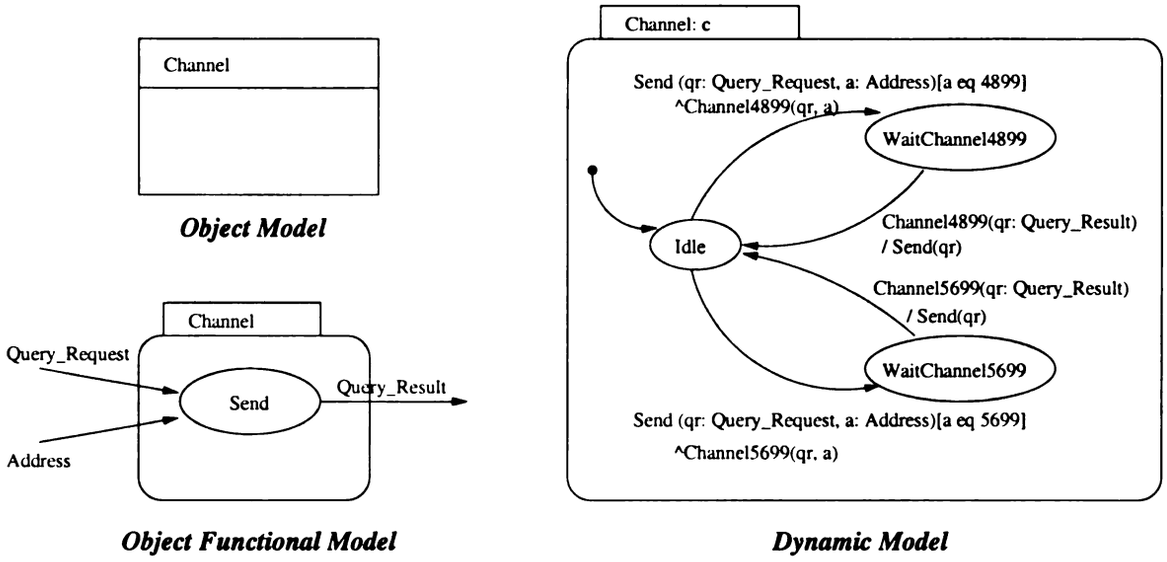

Figure 9.30: The design models for *channel*

A *Channel* object provides a *Send* service that accepts *Data_Archive_Name* and *Address* and returns *Query_Result*. A *Channel* object is designed to redirect a *Query* request from a *Client* object to the corresponding *Archive Server* object whose address is specified in the request, wait for the query result from the *Archive_Server* object, and then deliver the query result back to the requesting *Client* object. The formal specification automatically generated from the OMT models of *Channel* is contained in Figure 9.31. The formalization rules used to generate the formal specification include **OM1**, **OM2**, **OM10**, **DFR-1**, **DFR-2**, **DFR-3**, **DFR-4**, **DFR-5**, **DFR-8**, **DFR-11**, **DFR-12**, **DFR-13**, **FFR-1**, **FFR-2**, **FFR-3**, and **FFR-4**.

The revised state diagram. The revised state diagram that composes the dynamic models of individual objects in parallel is given in Figure 9.32. In the revised diagram, a dynamic model of a *Channel* object is incorporated. The dynamic models are also synchronized through name binding.

In the state diagram, the dynamic models of the individual objects are synchronized as follows:

- The two *Archive_Server* objects are interleaved.
- The *Client_Query* service request from ENFORMS's front-end is synchronized with the *Query* service provided by the *Client* object by sharing name *Query*.
- The *Archive_Server.Query* service request from the *Client* object is synchronized with the *Send* service provided by the *Channel* object by sharing the actual name *Archive_Query*.
- The *Name_Server.GetTable* service request from the *Client* object is synchronized with the *GetTable* service provided by the *Name_Server* object by sharing the actual name *GetTable*.
- The *Name_Server.Register* service requests from the two *Archive_Server* objects are synchronized with the *Register* service provided by the *Name_Server* object by sharing the actual name *Register*.

```
specification Channel [Send, Channel4899, Channel5699] (c: Channel) :
noexit

typedef Channel is ENFORMS, Channel
    sorts
        Channel
endtype

behavior
    Idle [Send, Channel4899, Channel5699] (c)

where

process Idle [Send, Channel4899, Channel5699] (c: Channel): noexit :=
    Send ? qr: Query_Request ? a: Address; (
    [a eq 4899] -> Channel4899!qr !a;
        WaitChannel4899 [Send, Channel4899, Channel5699] (c)
    []
    [a eq 5699] -> Channel5699!qr !a;
        WaitChannel5699 [Send, Channel4899, Channel5699] (c)
    )
endproc

process WaitChannel4899 [Send, Channel4899, Channel5699] (c: Channel): noexit :=
    Channel4899 ?qr:Query_Result; Send!qr;
        Idle[Send, Channel4899, Channel5699] (c)
endproc

process WaitChannel5699 [Send, Channel4899, Channel5699] (c: Channel): noexit :=
    Channel5699 ?qr:Query_Result; Send!qr;
        Idle[Send, Channel4899, Channel5699] (c)
endproc
endspec
```

Figure 9.31: The automatically generated formal specification from the design models of *Channel*

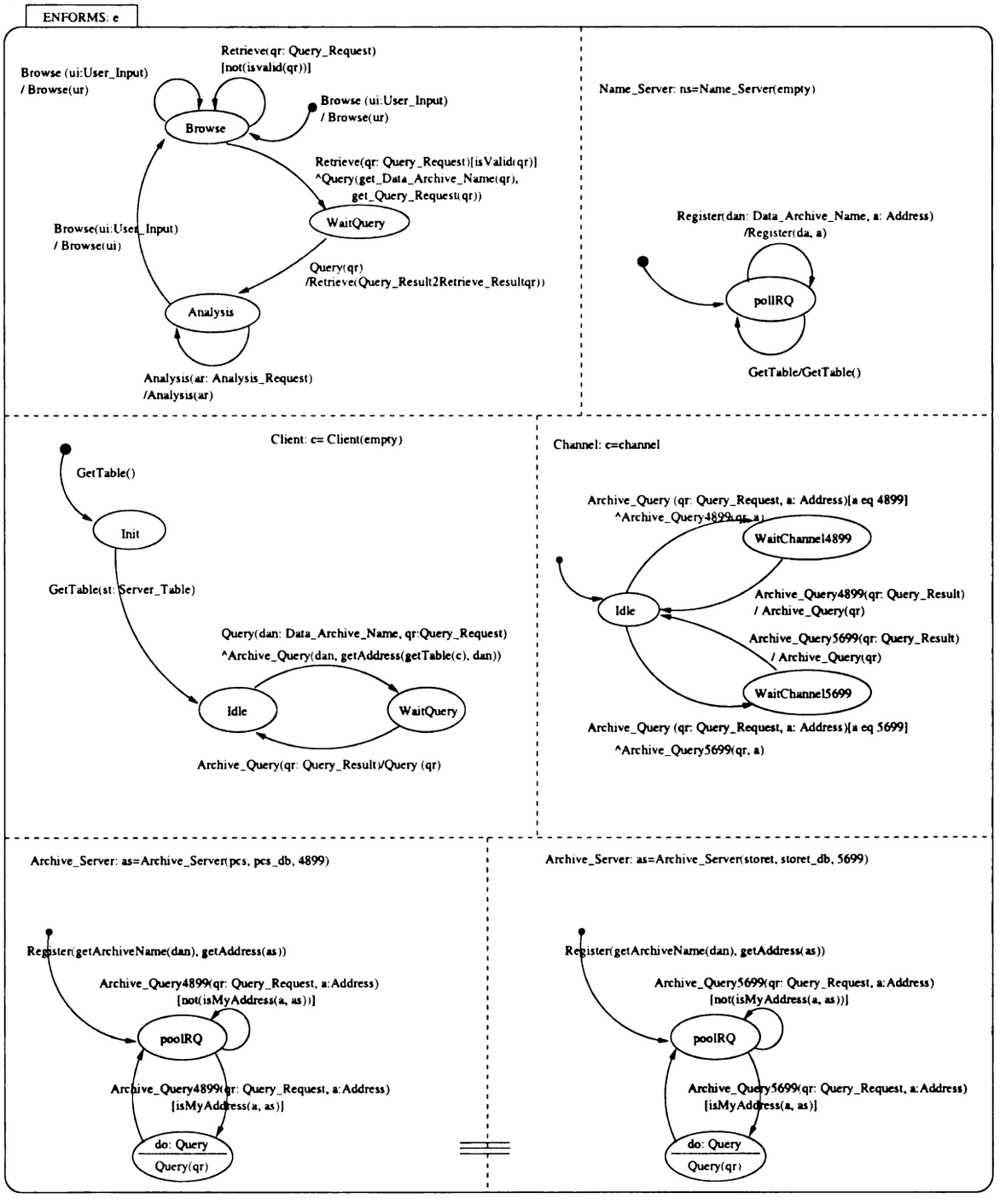

Figure 9.32: The revised design models (with a *Channel* object) composed in parallel

- The *Channel4899* service request from the *Channel* object is synchronized with the *Query* service provided by the *Archive_Server* object, whose *Data_Archive_Name* is *pcs* and *Address* is *4899*, by sharing the actual name *Archive_Query4899*.
- The *Channel5699* service request from the *Channel* object is synchronized with the *Query* service provided by the *Archive_Server* object, whose *Data_Archive_Name* is *pcs* and *Address* is *5699*, by sharing the actual name *Archive_Query5699*.

The formal specification in Figure 9.33 can be automatically generated from the state diagram (Figure 9.32) in which the behavior of individual objects are synchronized. The nature of process algebra specifications synchronizes the processes that specify the behavior of the individual objects through shared LOTOS gates. The formalization rules used to generate the formal specification include **DFR-19** and **DFR-20**.

9.6.2 Static analysis

The transcript in Figure 9.34 shows that the formal specification[1] passes the syntax checking and semantics analysis by LFE (a syntax analyzor) and LSA [64, 83], respectively. This indicates

- The formal specification, including both parts that are automatically generated and manually added, is syntactically correct.
- The algebraic specifications of individual objects are semantically consistent.
- The application of *name binding* through which the behavior of objects are synchronized does not result in inconsistency.

[1]Appendix H contains the formal specification automatically generated from the SRFM (Figure 9.23 and 9.24), the refined dynamic model of ENFORMS (Figure 9.25 and 9.26), and the state diagram that synchronizes behavior of individual objects (Figure 9.32 and 9.33).

```
specification ENFORMS_REF [Browse, Retrieve, Analysis, Query, GetTable,
    Archive_Query, Register, Archive_Query4899, Archive_Query5699]
    (e: ENFORMS): noexit

(* Browse:    ui: User_Input -> Retrieve_Request *)
(*    ensures    result = Browse (ui, getIndices(e)) *)
(* Retrieve:    rr: Retrieve_Request -> Retrieve_Result *)
(*    requires isValid(rr)    *)
(*    ensures result = Retrieve (rr, getArchives(e)) *)
(* Analysis:    ar: Analysis_Request -> Analysis_Result *)

library
    ENFORMS, Name_Server, Archive_Server, Client, Boolean, Channel
endlib
......
behavior

    (Browse ? ui: User_Input;
     Browse ! Browse (ui, getIndices(e));
     Browse [Browse, Retrieve, Analysis, Query] (e)
    |[Query]|
    (Client [Query, GetTable, Archive_Query] (Client(empty))
    |[GetTable,Archive_Query]|
    (Name_Server [Register, GetTable] (Name_Server(empty))
    |[Register]|
    (Channel[Archive_Query, Archive_Query4899, Archive_Query5699] (channel)
    |[Archive_Query4899,Archive_Query5699]|
    (Archive_Server [Archive_Query4899,Register]
         (Archive_Server(pcs,pcs_db,4899))
     |||
     Archive_Server [Archive_Query5699,Register]
         (Archive_Server(storet, storet_db, 5699))
    )))))

where
......
endspec
```

Figure 9.33: The automatically generated formal specification that synchronizes the behavior of individual objects

```
<99 aynow:~  >topo enforms -v -syntax
TOPO_3R6 (Mon Jan 23 15:19:15 MET 1995) /usr/local/LOTOS/Tools/TOPO
syntax analysis ...
lfe   enforms.lot > Ta01755
mv Ta01755 enforms.lfe
<100 aynow:~  >topo enforms -v -s -l design
TOPO_3R6 (Mon Jan 23 15:19:15 MET 1995) /usr/local/LOTOS/Tools/TOPO
semantics analysis ...
lfe  enforms.lot > Ta01777
mv Ta01777 enforms.lfe
lsa   -l design -p enforms enforms.lfe
```

Figure 9.34: Syntax checking and semantics analysis of the refined *enforms* formal specification

The expansion transformations provided by LOLA [84, 85, 86, 87] produces a compressed version of an Extended Finite State Machine (EFSM) from a given LOTOS behavior specification. The behavior of the generated EFSM is equivalent to the original LOTOS specification. The effect of an expansion is the removal of the most complex LOTOS operations (e.g., parallel operators) from the specification, producing an equivalent specification in terms of *action prefix*, *behavior choice*, *guards*, and *choice*, etc. This transformation can be used for state exploration, deadlock detection, deriving efficient implementations, etc.

Because the dynamic models of the individual objects are synchronized by parallel operators, we can use the expansion transformation to check if the behavior of the synchronized objects can be expressed in terms of an equivalent EFSM without parallel operators. This transformation can be considered a static analysis of state exploration. A deadlock detected by the transformation suggests that the behavior of objects has a synchronization problem. Figure 9.35 shows the transcript of such

an expansion. The expansion transformation explored 191 states and generated 373 transitions. The 4 deadlocks detected by the expansion transformation suggests that there is a problem with behavior synchronization among the communicating objects. We need to investigate the EFSM generated by the expansion transformation to look into the deadlock problem.

```
expand  -1
 Rewriting expressions in the specification.
 Rewriting done.
 Analysing unguarded conditions.
 Analysis done.

    Analysed states       = 191
    Generated transitions = 373
    Duplicated states     = 183
    Deadlocks             = 4

    Removing Parameters.
```

Figure 9.35: Using expansion transformation to detect synchronization error

Figure 9.36 shows a part of the EFSM generated from the LOTOS specification given in Appendix H. The four processes in Figure 9.36 leads the behavior of the EFSM to deadlocks. A further investigation of the four processes reveals that the deadlocks are caused by the synchronization between the *Client* and *Channel* objects. When the *Client* object raises a *Archive_Query* service request with constant *undef_Address* as the argument for *Address* to the Channel object, the *Channel* object enters a deadlock because the dynamic model of *Channel* (see Figure 9.30) did not specify how to react for address other than *4899* and *5699*.

```
......
process duplicate3 [browse,retrieve,analysis,query,gettable,archive_query,register,
        archive_query4899,archive_query5699] : noexit :=
   register  ! storet ! 5699;
   stop
endproc
......
process duplicate4 [browse,retrieve,analysis,query,gettable,archive_query,register,
        archive_query4899,archive_query5699] : noexit :=
   register  ! pcs ! 4899;
   stop
endproc
......
process duplicate9 [browse,retrieve,analysis,query,gettable,archive_query,register,
        archive_query4899,archive_query5699] (rr_92:retrieve_request) : noexit :=
   archive_query  ! get_query_request(rr_92) ! undef_address;
   stop
endproc
......
process duplicate17 [browse,retrieve,analysis,query,gettable,archive_query,register,
        archive_query4899,archive_query5699] (rr_92:retrieve_request) : noexit :=
   archive_query  ! get_query_request(rr_92) ! undef_address;
   stop
endproc
......
```

Figure 9.36: A part of the EFSM transformed from the LOTOS specification in Appendix H

Two approaches are available to eliminate the deadlock: (1) further refine the dynamic model for the *Channel* object to handle *Send* request with addresses other than *4899* and *5699*, or (2) add a guarding condition to the dynamic model for the *Client* object to ensure that the *Address* argument is valid before issuing the request for the service provided by the *Channel* object. Since the *Client* object is our focus of design, we choose the latter approach. Figure 9.37 shows a refined dynamic model for the *Client* object class. A guarding condition is associated with the *Query* service. If the argument of sort *Address* is not valid, constant *undef_Query_Result* is returned

to the requesting object, otherwise, the *Archive_Query* service of the *Channel* object will be requested to pass the request to the corresponding *Archive Server* object.

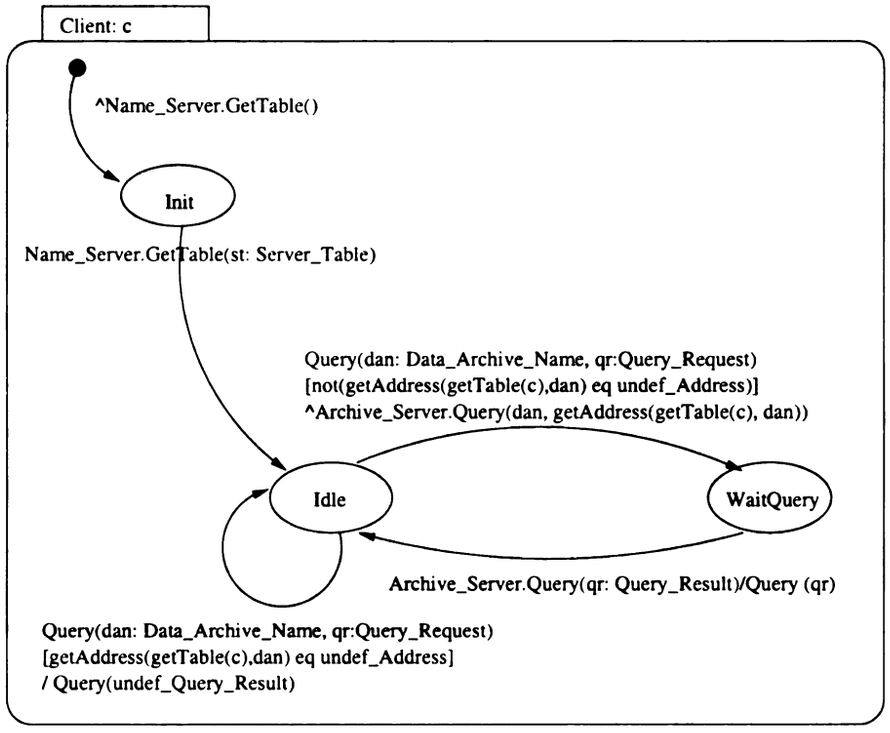

Figure 9.37: The refined dynamic model of *Client*

The changes made in the refined dynamic model only affect the specification of the *Idle* state of *Client*. Figure 9.38 contains the automatically generated formal specification for the refined *Client* object. The formalization rules used to generate the formal specification include **DFR-1**, **DFR-2**, **DFR-3**, **DFR-4**, **DFR-5**, **DFR-6**, **DFR-7**, **DFR-11**, **DFR-12**, and **DFR-13**.

After the changed specification for the modified dynamic model of the *Client* object class is extended to the formal specification in Figure 9.33, another expansion transformation is conducted to perform a static analysis of the synchronization. The

```
process Idle [Query, Name_Server_GetTable, Archive_Server_Query]
        (c: Client) : noexit :=
    Query ? dan: Data_Archive_Name ? qr: Query_Request;
    ([getAddress(getTable(c),dan) eq undef_Address]
        -> Query ! undef_Query_Result;
           Idle [Query, Name_Server_GetTable, Archive_Server_Query] (c)
    []
    [not(getAddress(getTable(c),dan) eq undef_Address)]
        -> Archive_Server_Query ! qr ! getAddress(getTable(c), dan);
           WaitQuery [Query, Name_Server_GetTable, Archive_Server_Query] (c))
endproc
```

Figure 9.38: The formal specification for the refined *Idle* state of *Client*

transcript of the expansion transformation is given in Figure 9.39. No deadlock is detected after the behavior of the *Client* class is refined.

```
expand  -1
 Rewriting expressions in the specification.
 Rewriting done.
 Analysing unguarded conditions.
 Analysis done.

    Analysed states       = 205
    Generated transitions = 415
    Duplicated states     = 211
    Deadlocks             = 0

    Removing Parameters.
```

Figure 9.39: The transcript of expansion transformation after the behavior of the *Client* is refined

9.6.3 Dynamic analysis

Figure 9.8 contains a test process that dynamically analyzes the behavior of ENFORMS before refinement. We can check the *testing equivalence* [94][2] between the behavior of ENFORMS before and after the refinement by applying the given test process to the refined behavior specification. Since new constants are introduced during the refinement process, the test process is also adapted to reflect the change. In the test process of Figure 9.40, the two **valid_Retrieve_Request** constants are substituted by **retrieve_request (pcs, valid_query_request)** and **retrieve_request (storet, valid_query_request)**.

```
process test [Browse, Retrieve, Register, GetTable, success] : noexit :=
    Browse ! valid_User_Input;
    Browse ? rr: Retrieve_Request;
    Retrieve ! retrieve_request (pcs, valid_query_request);
    Retrieve ? rr: Retrieve_Request;
    ([isValid(rr)] -> Browse ! valid_User_Input;
                      Browse ? rr: Retrieve_Result;
                      Retrieve ! retrieve_request (storet, valid_query_request);
                      Retrieve ? qr: Retrieve_Result;
                      ([isValid(rr)] -> success; stop))
endproc
```

Figure 9.40: The test process for the refined formal specification of ENFORMS

Because this case study focuses on checking whether the objects designed in the client-server architectural style will realize the distributed query transparently, the services, other than *Browse*, *Retrieve*, and *Analysis*, through which the individual object communicate and synchronize with each other are hidden from external access.

[2]Two processes are considered testing equivalent to each other if no difference of their behavior can be detected from external testing.

This is described in terms of LOTOS gate hiding construct by hiding the corresponding gates of the behavior specification given in Figure 9.33. The modified behavior specification is shown in Figure 9.41.

```
......
behavior

hide Query, GetTable, Archive_Query, Register,
     Archive_Query4899, Archive_Query5699 in
    (Browse ? ui: User_Input;
     Browse ! Browse (ui, getIndices(e));
     Browse [Browse, Retrieve, Analysis, Query] (e)
    |[Query]|
    (Client [Query, GetTable, Archive_Query] (Client(empty))
    |[GetTable,Archive_Query]|
    (Name_Server [Register, GetTable] (Name_Server(empty))
    |[Register]|
    (Channel[Archive_Query, Archive_Query4899, Archive_Query5699] (channel)
    |[Archive_Query4899,Archive_Query5699]|
    (Archive_Server[Archive_Query4899,Register](Archive_Server(pcs,pcs_db,4899))
     |||
     Archive_Server[Archive_Query5699,Register]
          (Archive_Server(storet, storet_db, 5699)))))))

where
......
```

Figure 9.41: Communication among individual objects are hidden

If the test process in Figure 9.40 can always reach the *success* event for the behavior specification given in Figure 9.41, then we can assert that the behavior of ENFORMS before and after refinement preserve testing equivalence for the given test. Unfortunately, the transcript generated by *TestExpand* in Figure 9.42 shows that no *success* event is reached during the exhaustive test. Instead, 30 deadlocks are detected and the test is rejected.

```
lola> test -1 success test -y
testexpand -1 success test -y

  Composing behaviour and test :

     test [browse,retrieve,register,gettable,success]
   |[browse,retrieve,analysis,gettable,register]|
     (hide query,gettable,archive_query,register,
           archive_query4899,archive_query5699 in
        browse  ? ui_27:user_input;
        browse  ! browse(ui_27,getindices(e_26));
        browse [browse,retrieve,analysis,query] (e_26)
      |[query]|
          client [query,gettable,archive_query] (client(empty))
        |[get-table,archive_query]|
            name_server [register,gettable] (name_server(empty))
          |[register]|
              channel[archive_query,archive_query4899,archive_query5699] (channel)
            |[archive_query4899,archive_query5699]|
                archive_server [archive_query4899,register]
                     (archive_server(pcs,pcs_db,4899))
              |||
                archive_server [archive_query5699,register]
                     (archive_server(storet,storet_db,5699))
     )

    Analysed states       = 91
    Generated transitions = 90
    Duplicated states     = 0
    Deadlocks             = 30

    Process Test = test
    Test result  = REJECT.

                   successes = 0
                       stops = 30
                       exits = 0
                cuts by depth = 0
```

Figure 9.42: The test process is rejected by the refined behavior specification of ENFORMS

Reviewing the log file generated by *TestExpand* reveals that the deadlock is caused by specification incompleteness. The SRFM in Figure 9.23 refines data sort *Retrieve_Request* in terms of *Data_Archive_Name* and *Query_Request* and introduces sort conversion operations for *Query_Result* and *Retrieve_Result*. However, the Boolean operation *isValid* operation for both *Retrieve_Request* and *Retrieve_Result* sorts are not further specified to cover the newly introduced operations. In the behavior specification, the *isValid* operation of *Retrieve_Request* is used in *Client* to check if the input is valid. In the test process the *isValid* of *Retrieve_Result* is issued to check whether the returned result is valid. Therefore, the less specified Boolean operations result in deadlocks during the execution of *TestExpand*. The operations and equations are added to the algebraic specification of the refined ENFORMS to cover the data refinement of Figure 9.23. The specification in Figure 9.43 shows the added operations and equations.

```
type ENFORMS_REF is ENFORMS, Name_Server, Archive_Server, Client, Channel
    opns
        ......
        isValid : Data_Archive_Name -> Bool
        isValid : Query_Result -> Bool
    eqns
        forall qr: Query_Request, dan: Data_Archive_Name, qr1: Query_Result
            isValid (qr) = not(qr eq undef_Query_Request);
            isValid (dan) = not(dan eq undef_Data_Archive_Name);
            isValid (Retrieve_Request(dan, qr)) = isValid (dan) and isValid (qr);
            isValid (qr1) = not (qr1 eq undef_Query_Result);
            isValid (Query_Result2Retrieve_Result(qr1)) = isValid(qr1);
    ......
endtype
```

Figure 9.43: The added operations and equations that are resulted by the SRFM of Figure 9.23

After adding the necessary operations and equations into the refined algebraic specification of ENFORMS, another round of exhaustive testing is executed. The transcript of the test is shown in Figure 9.44. Fortunately, there are two execution paths that led to the *success* event during the exhaustive exploration test. However, the 80 deadlocks detected by the test are still discouraging. A further analysis is necessary in order to determine the cause of the deadlocks.

```
lola> test -1 success test -y
testexpand -1 success test -s -y

 Rewriting expressions in the specification.
 Rewriting done.
 Analysing unguarded conditions.
 Analysis done.

  Composing behaviour and test :
  ......

    Analysed states       = 293
    Generated transitions = 294
    Duplicated states     = 0
    Deadlocks             = 80

    Process Test = test
    Test result  = MAY PASS.

    82 executions analysed:

                  successes = 2
                      stops = 80
                      exits = 0
              cuts by depth = 0
```

Figure 9.44: The transcript of *TestExpand* test after necessary operations and equations are added

---

ENFORMS is designed in terms of a client-server architectural style that reflects and utilizes the distributed nature of the system. The execution of the ENFORMS

system is initiated by activating the name server. The archive servers are then typically activated, followed by any number of clients, although the system does not preclude the possibility of activating the clients before the archive servers. Therefore, it is very likely that a client starts and accepts a service request before the corresponding archive server is ready to serve. Under this circumstance, the client is not able to return a valid query result. Our investigation of the log file generated by *TestExpand* identifies a large number of these cases that have led to deadlocks. However, under some circumstances, even when the two *Archive_Server* objects are started and registered before the *Client* object accepts a *Query* request, deadlock still appears. A stepwise interactive simulation that recovers the path to deadlock reveals that the deadlock is caused by the current design that the *Client* object only acquires a *Server_Table* during its initialization time but not afterwards. If the following sequence of events occurs, a deadlock will be produced.

1. The *Client* object starts and acquires an *empty* server table from the *Name_Server* object.
2. The two *Archive_Server* objects are started and registered.
3. The *Client* object accepts a *Query* request from the test process.
4. The *Client* returns constant *undef_Query_Result* because there is no available archive server.
5. The returned result leads the test process into a deadlock.

The above analysis indicates that a copy of an obsolete server table possessed by the *Client* object will fail the test. There are many means of design that can handle this problem in the context of a distributed system. The straightforward but not necessarily optimal approach used in the case study is to let the *Client* object acquire

a copy of an up-to-date server table from the *Name Sever* object before it attempts to request services from *Archive_Server* objects. The refined dynamic model for the *Client* object class is given in Figure 9.45. The formalization rules used to generate the formal specification include **DFR-1**, **DFR-2**, **DFR-3**, **DFR-4**, **DFR-5**, **DFR-6**, **DFR-7**, **DFR-11**, **DFR-12**, and **DFR-13**.

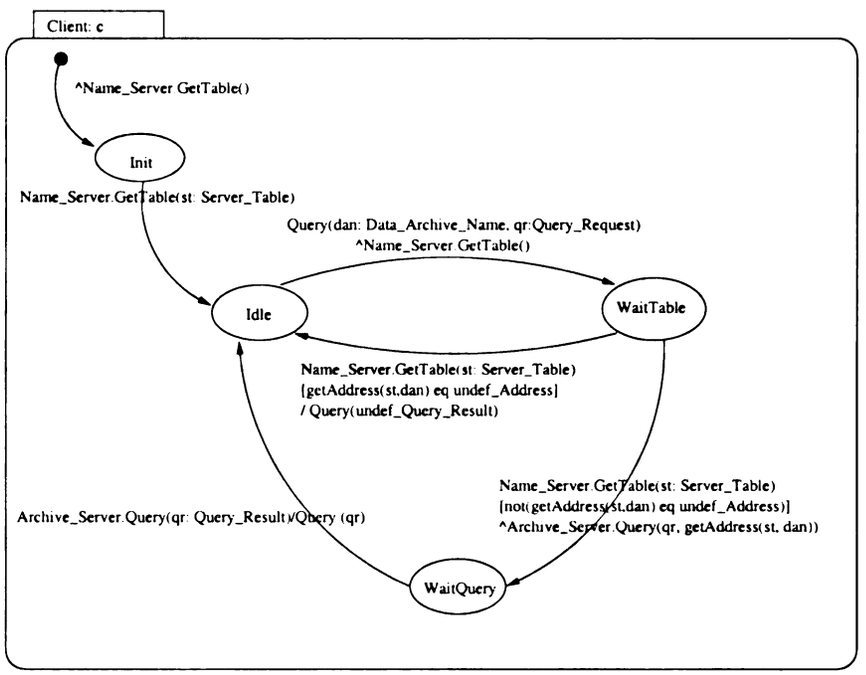

Figure 9.45: The further refined dynamic model of *Client*

In the refined dynamic model, a new state, *WaitTable*, with associated transitions are introduced, and the transition that leaves from *Idle* state given a *Query* service request is modified. The formal specification automatically generated from the changed parts of the dynamic model is given in Figure 9.46.

Figure 9.47 shows the execution of the *TextExpand* after the design of the *Client* object class is refined. In all, there are 802 states analyzed and 913 transitions

```
process Idle [Query, Name_Server_GetTable, Archive_Server_Query]
             (c: Client) : noexit :=
    Query ? dan: Data_Archive_Name ? qr: Query_Request;
    Name_Server_GetTable;
    WaitTable [Query, Name_Server_GetTable, Archive_Server_Query] (c, qr, dan)
endproc

process WaitTable [Query, Name_Server_GetTable, Archive_Server_Query]
        (c: Client, qr: Query_Request, dan: Data_Archive_Name) : noexit :=
    Name_Server_GetTable ? st: Server_Table;
    ([getAddress(st,dan) eq undef_Address] ->
        Query ! undef_Query_Result;
            Idle [Query, Name_Server_GetTable, Archive_Server_Query] (c)
    []
    [not(getAddress(st,dan) eq undef_Address)] ->
        Archive_Server_Query ! qr ! getAddress(st, dan);
        WaitQuery [Query, Name_Server_GetTable, Archive_Server_Query] (c))
endproc
```

Figure 9.46: The formal specification for the refined *Idle* state of *Client*

generated during the exhaustive test. Among the 160 execution paths, 112 reach the *success* event, 48 result in deadlock. The designed ENFORMS system **may** pass the test process given in Figure 9.40.

Our further study shows that all 48 deadlocks can be attributed to the nature of distributed components. That is, the client cannot be guaranteed that all archive servers will be active upon enter into ENFORMS. In this context, we can assert that *the client-server design of* ENFORMS *realizes the retrieve function over a distributed data archives transparently.*

```
lola> test -1 success test -y
testexpand -1 success test -s -y

 Rewriting expressions in the specification.
 Rewriting done.
 Analysing unguarded conditions.
 Analysis done.

  Composing behaviour and test :
  ......

    Analysed states       = 802
    Generated transitions = 913
    Duplicated states     = 0
    Deadlocks             = 48

    Process Test = test
    Test result  = MAY PASS.

    160 executions analysed:

                  successes = 112
                      stops = 48
                      exits = 0
               cuts by depth = 0
```

Figure 9.47: The result of *TestExpand* after the behavior of the *Client* object class is refined

9.7 Summary

In this chapter, through the application of the design process and the formalization rules to a portion of ENFORMS, we have determined the following:

- The formalization and integration rules for OMT models can help the designers to derive more precise and integrated descriptions of design.
- The proposed design process facilitates the refinement of design.
- Based on the formal specifications derived from OMT models, analysis can be performed to check design consistency and to facilitate understanding and communication about the design.

Four problems were detected during the refinement process through various analysis techniques:

- An underlying communication mechanism between clients and servers is assumed in the system design. However, this assumed support was not modeled in the design, and resulted in nondeterminism during the communication between clients and servers.
- A *Client* object may issue invalid requests to *Archive_Servers* and result in deadlock condition.
- Data refinement in the SRFM of ENFORMS brought in incompleteness to algebraic specification. The corresponding operations in the algebraic specifications are obligated to be further refined.
- An obsolete copy of a server table held by the *Client* object will cause a deadlock condition.

Chapter 10

Conclusions and Future Investigations

Formal specifications are gaining increasing attention as a means to rigorously document requirements and design information since the well-defined notations are amenable to automated processing for numerous analysis tasks [2], including verification of the correctness of resulting systems. However, attempting to construct a formal specification directly from an informal, high-level requirements document can be challenging. Formal descriptions potentially involve considerable syntactic details and may require careful planning and organization on the part of the specifier in order to develop modular specifications.

A complementary approach to describing requirements is the use of graphical modeling notations. The intuitive diagrammatic notations are easy to understand and facilitate model construction. However, the lack of formal semantics of graphical notations deprives designers of analysis techniques that can help designers to de-

tect and eliminate design errors during the early stages of the software development process.

In order to take advantages of both the intuitive diagrammatic and rigorous formal techniques, the integration of the two approaches is clearly motivated. Therefore, the major motivation for our research was to provide developers with a means to take advantage of the benefits of an easy to use, graphical-based modeling approach with the advantages afforded by formal approaches to software development, all within the context of facilitating technology transfer [113].

In this dissertation, we refined and introduced formalisms to the three complementary OMT models: object, functional, and dynamic models. An integration of the three models is achieved in terms of the underlying formal semantics. Based upon the formalization and integration, rigorous analyses can be applied to the formal specifications to perform specification analysis. In addition, a design process that employs the formalization, integration, and analysis techniques is proposed. A case study that applied the proposed formalization rules and design process to an industrial project is also discussed.

10.1 Summary of Contributions

There are three primary objectives for this research.

- Improve and facilitate the software development process for software engineers, with particular emphasis on the early stages of software development.
- Provide a modeling and specification technique that integrates informal and formal techniques, thereby enabling rigorous analyses in the early stages of development.

- Exploit and integrate existing formal methods technologies.

In order to help practicing software engineers, we solicited input from numerous industrial organizations and contacts to determine what graphical, object-oriented modeling techniques were commonly used. OMT was determined to be one of the most widely used object-oriented modeling techniques, but it suffered from a lack of ability on the part of software developers to determine ambiguity and incompleteness problems in the early stages of development. In addition, OMT contains simple, graphical models that are also used by other object-oriented modeling notations. Therefore, another advantage of this research is that any formalizations that we develop are not specific to OMT.

Accordingly, the main objective of the formalization rules is to (1) introduce formal semantics to the graphical notations, and (2) integrate the models in terms of the underlying formal semantics. These rules also make the models amenable to automatic formal specification generation. Given an integrated, single formal specification for each object, automated specification analysis can be performed.

From the formal methods community, we identified Larch and LOTOS to be commonly used specification languages with rich tool support. In addition, the approach proposed by Larch to describe design entities in terms of a two-tiered language is also incorporated in the formal specification technique of this research. The large number of available LOTOS tools greatly enhance the analyses applicable to the formal specifications. The currently available specification analysis techniques include (1) consistency checking among the three models of a single object, (2) behavior

simulation and analysis for single objects, (3) synchronization analysis for multiple communicating objects, and (4) specification simulation and debugging for communicating objects.

The contribution of this dissertation research is four-fold.

- **Formalization of Dynamic Model.**

  Nineteen formalization rules have been developed to formalize the dynamic models in terms of LOTOS process algebra specifications. The formalization of the state diagrams is defined within the context of the object model formalization [27]. The formalization enables the precise specification of the behavior of objects and the simulation of system behavior through executable specifications.

- **Formalization of Functional Model.**

  Two functional models, Object Functional Model (OFM) and Service Refinement Functional Model (SRFM), are introduced in order to integrate the functional models into object-oriented technology. Two guidelines for deriving algebraic specifications are given. Sixteen formalization rules have been introduced to formalize the functional models in terms of algebraic and predicate specifications. The pre- and postconditions of services enable symbolic execution of the high-level design model thus providing a means to perform simulation, verification, and validation during the early phases of software development.

- **Integration of the Three Complementary Models.**

  The integration of the three complementary models is three-fold. First, the functional and dynamic models are derived in the context of object models. Third, the integration is achieved by composing the dynamic models and SRFMs hierarchically according to the system structure specified in the object model. The integration and formalization of the three models enable designers to perform analysis tasks by using the derived formal specifications.

- **Process for Model Construction, Specification, and Refinement.**

  A design process has been developed to facilitate the development of formal design specifications in parallel with the development of OMT's semi-formal, graphical models. The rigorous mathematical foundation of formal specifications provides a more precise means to describe the design thus avoiding ambiguities. The symbolic execution of the design specifications can help designers better understand the design as well as facilitate the communication among designers and even with the customers. Specification analysis can be performed during model refinement of the design process to detect and eliminate design flaws during earlier stages of software development.

10.2 Impact of Research and Future Investigations

The formal description technique of this research is unique. It provides a means to use an integrated, single formal specification framework to describe different aspects of an object, yet the descriptions of the different aspects are integrated rather than isolated. The graphical front-end (that is, the OMT models) of the technique makes it more user-friendly and easy to use for general software designers. The proposed design process provides a guideline for designers to systematically utilize this formal technique during the design phase in a stepwise, incremental fashion. The specification analysis enabled by the formal technique can help designers to detect design flaws during earlier stages of software development. Although OMT was chosen as the graphical front-end, this back-end formal technique can also be applied to other graphical notations with some straightforward modifications.

Several avenues of research appear promising for future investigations.

- UML has been adopted as a standard notation for object-oriented modeling [43, 44, 45, 46]. Given the fact that OMT is one of the original graphical notations from which UML evolve, our back-end formal technique can be directly extended to UML notation. However, in UML, the services and functionalities are not explicitly modeled, no counterparts for OFMs and SRFMs of our notations are identified. More effort is necessary to address this issue.

- The results of the case study performed in this dissertation are quite encouraging. Four design flaws were detected by the formal technique in our case study.

However, in order to gain more empirical experience and to refine and fine tune the proposed techniques, more case studies should be conducted.

- The research focused on applying formal techniques during the design phase of software development. Given that the Larch LIL is specific to programming languages and LOTOS can produce prototype source code in C and Ada, the research investigations can be extended to the implementation phase. Based on the framework of the current formal techniques, automatic or semi-automatic, computer assisted source code generation is a potentially promising research topic.

- Before the proposed technique can be accessible to general software developers, tools that support graphical model construction, formal specification generation, and corresponding maintenance facilities must be developed first. Without tool support, the advantages of the techniques will be more difficult to exploit. The case study required the manual generation of all of the formal specifications according to proposed rules, an extremely tedious and cumbersome endeavor.

- The description of the signatures and pre- and postconditions of services provided by objects is not directly supported by LOTOS. Thus the descriptions, originated from Larch's LIL technique, are enclosed as annotations currently. In order to fully exploit our formal technique, it is necessary to extend LOTOS to include facilities with the similar expressiveness of Larch's LIL language. The recently proposed ELOTOS [80] has stronger expressiveness power, further research is also needed to investigate how to use ELOTOS in our formal descrip-

tions. However, the drawback is that currently there is limited tool support for ELOTOS.

- Although intuitive graphical notations are used to develop the formal models, the textual presentation of the results of formal analysis may be difficult for most designer with limited background in mathematics and formal methods. Further research and investigation on how to visually present the results of the formal analysis in an intuitive fashion is necessary.

- Although specification and modeling errors can be relatively easy to detect with existing tools, the current tool support of LOTOS does not have a friendly environment for specification debugging. Better tool support for debugging is also desirable to facilitate a designer's ability to identify and locate the cause of design errors.

Bibliography

Bibliography

[1] W. W. Gibbs, "Software's chronic crisis," *Scientific American*, pp. 86–95, September 1994.

[2] J. M. Wing, "A Specifier's Introduction to Formal Methods," *IEEE Computer*, vol. 23, pp. 8–24, September 1990.

[3] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey: Prentice Hall, 1991.

[4] J. D. Ullman, *Principles of Database Systems.* Computer Science Press, 1983.

[5] M. D. Fraser, K. Kumar, and V. K. Vaishnavi, "Strategies for incorporating formal specifications," *Communications of the ACM*, pp. 74–86, October 1994.

[6] R. S. Pressman, *Software Engineering: A Practitioner's Approach.* McGraw Hill, fourth ed., 1997.

[7] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development, The Fusion Method.* Object-Oriented Series, Prentice Hall, 1 ed., 1994.

[8] D. Harel, A. Pneuli, J. P. Schmidt, and R. Sherman, "On the formal semantics of statecharts," in *Proceedings of the 2nd IEEE symposium on Logic in Computer Science*, pp. 54–64, 1987.

[9] J. V. Guttag and J. J. Horing, *Larch: Language and Tools for Formal Specification.* New York, New York: Springer-Verlag, 1993.

[10] S. Garland and J. Guttag, "A guide to LP, the Larch prover," Technical Report TR 82, DEC SRC, December 1991.

[11] P. van Eijk, "The design of a simulator tool," in *The Formal Description Technique LOTOS* (P. van Eijk, C. Vissers, and M. Diaz, eds.), pp. 351–390, Elsevier Science Publishers B.V., 1989.

[12] A. Fantechi, S. Gnesi, and C. Laneve, "An expressive temporal logic for basic LOTOS," in *Formal Description Techniques, II* (S. Vuong, ed.), pp. 261–276, Elsevier Science Publishers B.V., 1990.

[13] C. B. Jones, *Systematic Software Development Using VDM.* Prentice Hall International Series in Computer Science, Prentice Hall International (UK) Ltd., second ed., 1990.

[14] J. Wing, "Role of formal specifications (discussion group symmary in NRL invitational workshop on testing and proving, 1986)," vol. 11, pp. 65–67, Oct. 1986.

[15] D. Gries, *The Science of Programming.* Springer-Verlag, 1981.

[16] J. Bergstra, J. Heering, and P. Klint, *Algebraic Specification.* Addison Wesley, 1989.

[17] S. Fickas, "Automating the transformational development of software," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 1268–1277, November 1985.

[18] J. Stoy, *The Scott-Strachey Approach to Programming*, ch. Denotational Semantics. Cambridge: MIT Press, 1977.

[19] J. V. Guttag, J. J. Horning, K. J. S.J. Garland, A. Modet, and J. Wing, *Larch: Languages and tools for formal specification.* Texts and Monographs in Computer Science, Springer-Verlag, 1993.

[20] J. Guttag and J. Horning, "Introduction to LCL, a Larch/C interface language," tech. rep., Digital Equipment Corporation, Systems Research Center, July 29 1991.

[21] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS," in *The Formal Description Technique LOTOS* (P. H. J. V. Eijk, C. A. Vissers, and M. Diaz, eds.), pp. 23–73, North-Holland, 1989.

[22] R. Milner, "A calculus of communicating systems," in *Lecture Notes in Computer Science 92*, Springer-Verlag, 1980.

[23] C. Hoare, *Communicating Sequential Processes.* Englewood Cliffs, New Jersey: Prentice Hall International, 1985.

[24] J. Bergstra and J. W. Klop, "Process algebra for synchronous communication," in *Information and Control 60*, pp. 109–137, 1984.

[25] K. J. Turner, ed., *Using Formal Description Techniques — An Introduction to ESTELLE, LOTOS, and SDL.* Wiley, 1993.

[26] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification.* Berlin: Springer-Verlag, 1985.

[27] R. H. Bourdeau and B. H. C. Cheng, "A formal semantics of object models," *IEEE Transactions on Software Engineering*, vol. 21, pp. 799–821, October 1995.

[28] M. Wirsing, "Algebraic specification," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), ch. 13, Amsterdam: Elsevier Science Publishers and MIT Press, 1990.

[29] B. H. C. Cheng, E. Y. Wang, and R. H. Bourdeau, "A graphical environment for formally developing object-oriented software," in *Proc. of IEEE 6th International Conference on Tools with Artificial Intelligence*, November 1994.

[30] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas, "TROLL–a language for object-oriented specification of information systems," *ACM Transactions on Information Systems*, 1996.

[31] A. M. D. Moreira, *Rigorous Object-Oriented Analysis.* PhD thesis, Department of Computing Science and Mathematics, University of Stirling, August 1994.

[32] R. Jungclaus, "Modeling of dynamic object systems – a logic-based approach," in *Advanced Studies in Computer Science*, Vieweg Verlag, 1993.

[33] H. Ehrig and B. Maher, *Fundamentals of Algebraic Specification I: Equations and Initial Semantics.* Springer-Verlag, 1985.

[34] C. Hoare, "Communicating sequential processes," *Communications of the ACM*, pp. 666–677, August 1978.

[35] R. Milner, *A Calculus of Communicating Systems.* Springer-Verlag, 1990.

[36] E. A. Emerson, "Temporal and model logic," in *Formal Models and Semantics* (J. van Leeuwen, ed.), pp. "995–1072", Elsevier Science Publishers B.B., 1990.

[37] G. Engles, M. Gogolla, U. Hohenstein, K. Hulsmann, R. Lohr-Richter, G. Saake, and H. E. Ehrich, "Conceptual modeling of database applications using an extended ER model," *Data Knowledge Engineering*, vol. 9, no. 2, pp. "157–204", 1992.

[38] A. M. D. Moreira and R. G. Clark, "Rigorous object-oriented analysis,," in *International Symposium on Object-Oriented Methodologies and Systems (ISOOMS)*, pp. 65–78, Springer-Verlag, September 1994.

[39] A. M. D. Moreira and R. G. Clark, "Adding rigour to object-oriented analysis," *Software Engineering Journal*, vol. 11, no. 5, pp. 270–280, 1996.

[40] P. Coad and E. Yourdon, *Object-Oriented Analysis.* Englewood, New Jersey: Yourdon Press, Prentice Hall, 1990.

[41] G. Booch, *Object-oriented Design with Applications.* Redwood City, California: Benjamin/Cummings Pub. Co., 1991.

[42] K. Beck and W. Cunningham, "A laboratory for teaching object-oriented thinking," in *ACM OOPSLA'89 Conference Proceedings*, 1989.

[43] Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951, *UML Summary*, 1.1 ed., September 1997.

[44] Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951, *UML Semantics*, 1.1 ed., September 1997.

[45] Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951, *Notation Guide*, 1.1 ed., September 1997.

[46] M. Fowler and K. Scott, *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley, 1997.

[47] Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951, *Object Constraint Language Specification*, 1.1 ed., September 1997.

[48] Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951, *Rational Objectory Process 4.1 - The UML Process*, 4.1 ed., September 1997.

[49] G. Booch, *Object-Oriented Analysis and Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., second ed., 1994.

[50] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1994.

[51] B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object-Oriented Modeling.* Wiley professional computing, New York: Wiley & Sons, 1994. 525p.

[52] B. Selic, G. Gullekson, J. McGee, and I. Engelberg, "ROOM: An object-oriented methodology for developing real-time systems," in *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering* (G. Forte, N. H. Madhavji, and H. A. Muller, eds.), (10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1264), pp. 230–240, IEEE, IEEE Computer Society Press, July 1992.

[53] F. Polack, M. Whiston, and K. Mander, "The SAZ project: Integrating SSADM and Z," in *First International Symposium of Formal Methods Europe* (J. Woodcock and P. Larsen, eds.), (Denmark), pp. 540–557, Springer-Verlag, April 1993.

[54] L. Semmens and P. Allen, "Using Yourdon and Z: an approach to formal specification," in *Proc. of Fifth Annual Z user Group Meeting*, (Oxford), Springer-Verlag, December 1991.

[55] R. B. France, "Semantically extended data flow diagrams: A formal specification tool," *IEEE Transactions on Software Engineering*, vol. 18, pp. 329–346, April 1992.

[56] E. Astesiano, A. Giovini, and G. Reggio, "Data in a concurrent environment," in *International Conference on Concurrency*, (New York), Springer-Verlag, 1988.

[57] E. Astesiano and G. Reggio, "SMoLCS-driven concurrent calculi," in *TAPSOFT'87*, (New York), Springer-Verlag, 1987.

[58] P. Lucas, "VDM: Origins, hopes, and achievements," in *VDM-Europe Symposium 1987: A Formal Method at Work* (D. Bjorner, C. Jones, M. M. an Airchinnigh, and E. Neuhold, eds.), pp. 1–18, March 1987.

[59] C. B. Jones, *Systematic Software Development using VDM.* Prentice Hall, 1990.

[60] E. W. Dijkstra, *A Descipline of Programming.* Prentice-Hall, 1976.

[61] R. J. R. Back, *Correctness Preserving Program Refinements: Proof Theory and Applications.* Mathematical Centre, 1980.

[62] E. Y. Wang, H. A. Richter, and B. H. C. Cheng, "Formalizing and integrating the dynamic model within OMT," in *IEEE Proceedings of the 19th International Conference on Software Engineering*, (Boston, MA), pp. 45–55, IEEE, May 1997.

[63] E. Y. Wang, H. A. Richter, and B. H. C. Cheng, "Formalizing and integrating the dynamic model within OMT," Tech. Rep. MSU-CPS-96-13, Michigan State University, March 1996. Submitted to IEEE Transactions of Software Engineering, under revision.

[64] A. Azcorra, J. Quemada, and J. Mañas, "LOTOS tool survey."

[65] P. van Eijk, "Tools for LOTOS specification style transformation," in *Formal Description Techniques, II* (S. Vuong, ed.), pp. 43–51, Elsevier Science Publishers B.V., 1990.

[66] J. A. Mañas, T. de Miguel, J. Salvachúa, and A. Azcorra, "Tool support to implement LOTOS formal specifications," *Computer Networks and ISDN Systems*, vol. 25, pp. 815–839, 1993.

[67] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, July 1987.

[68] N. G. Leveson, M. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements specification for process-control systems," *IEEE Transactions on Software Engineering*, vol. 20, pp. 684–707, September 1994.

[69] Rational Software Corporation, Santa Clara, CA 95051-0951, *UML Notation Guide*, 1.0 ed., January 1997.

[70] Rational Software Corporation, Santa Clara, CA 95051-0951, *UML Semantics*, 1.0 ed., January 1997.

[71] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structed design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[72] E. N. Yourdon and L. L. Constantine, *Structured Design.* Yourdon Press, 1978.

[73] G. Booch, "What is and what isn't object-oriented design?," *American Programmer*, vol. 2, no. 7-8, pp. 14–21, 1989.

[74] D. D. Champeaux, "Object-oriented analysis and top-down software development," in *European Conference Object-Oriented Programming, Lecture Notes in Computer Science*, pp. 360–376, Springer-Verlag, 1991.

[75] E. Y. Wang and B. H. C. Cheng, "Formalizing and integrating the functional model into object-oriented design," in *Tenth International Conference on Software Engineering and Knowledge Engineering*, (San Francisco Bay, USA), June 1998.

[76] P. Ward and S. Mellor, *Structured Development for Real-Time Systems.* New York: Yourdon Press, 1985.

[77] D. Hatley and I. Pirbhai, *Strategies for Real Time System Specification.* New York: Dorset House Publishing, 1987.

[78] D. E. Monarchi and G. I. Puhr, "A research typology for object-oriented analysis and design," *Communications of the ACM*, vol. 35, pp. 35–47, September 1992.

[79] R. G. Fichman and C. F. Kemerer, "Object-oriented and conventional analysis and design methodologies," *IEEEE Computer*, pp. 22–39, October 1992.

[80] E. ot LOTOS, "Working draft on enhancements to LOTOS." Source: ISO/IEC JTC1/SC21/WG7.

[81] N. Shankar, S. Owre, and J. Rushby, "The PVS proof checker," reference manual, March 1993.

[82] A. Azcorra, J. Quemada, and J. Manas, "LOTOS tool survey," Tech. Rep. R/1.2.3-UPM/1, ETSI Telecomunicacion Ciudad Universitaria s/n, E-28040 Madcid, Spain, 1991.

[83] J. A. Mañas, T. de Miguel, T. Robles, J. Salvachúa, G. Huecas, and M. Veiga, "TOPO: Quick reference: Front-end – version 3R1."

[84] J. Q. Vives, S. P. Gómez, and D. L. López, "LOLA: Quick reference version 3R6."

[85] J. quemada, A. Fernandez, and J. A. Manas, "LOLA: Design and verification of protocols using LOTOS," in *IBERIAN Conference on Data Communications*, (Lisbon), May 1987.

[86] J. Quemada, S. Pavon, and A. Fernandez, "State exploration by transformation with LOLA," in *Workshop on Automatic Verification Methods for Finite State Sy stems*, (Grenoble), June 1989.

[87] S. Pavon and M. Llamas, "The testing functionalities of LOLA," in *Formal Description Techniques* (J. Quemada, J. A. Manas, and E. Vazquez, eds.), vol. III, pp. 559–562, IFIP, Elsevier Science B.V. (North-Holland)., 1991.

[88] H. Eertink, *SMILE User Manual.* University of Twente, Department of Computer Science and Department of Electrical Engineering, P. O. Box 217, 7500 AE Enschede, The Netherlands.

[89] P. van Eijk and H. Eertink, "Desing of the LotosSphere symblic LOTOS simulator," in *Formal Description Techniques* (E. V. J. Quemada and J. A. Manas, eds.), pp. 577–580, Madrid, Spain: IFIP, North-Holland, 1990.

[90] T. L. Group, *XELUDO User Manual.* The LOTOS group, University of Ottawa, January 1994.

[91] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[92] T. Bolognesi and M. Caneve, "Equivalence verification: Theory, algorithems and a tool," in *The Formal Description Technique LOTOS* (P. van Eijk, C. Vissers, and M. Diaz, eds.), pp. 303–326, Elsevier Science Publishers B.V., 1989.

[93] E. Brinksma, "A theory for the derivation of tests," in *The Formal Description Technique LOTOS* (P. van Eijk, C. Vissers, and M. Diaz, eds.), pp. 235–248, Elsevier Science Publishers B.V., 1989.

[94] R. de Nicola and M. Hennessy, "Testing equivalences for processes," *Theoretical Computer Science*, vol. 34, pp. 83–133, November 1984.

[95] E. S. Taylor, *Interim Report On Engineering Design.* Massachusetts Institute of Technology, 1953.

[96] B. Myers, *Composite Structured Design.* Van Nostrand, 1978.

[97] S. A. Dart, R. J. Ellison, P. H. Feiler, and A. N. Habermann, "Software development environment," *IEEE Computer*, pp. 18–28, Novemember 1987.

[98] J. N. Brinkkemper, *Formalization of Information Systems Modelling.* PhD thesis, Katholieke Universiteit te Nijmegen, June 1990.

[99] S. R. Schach, *Software Engineering.* Richard D. Irwin Inc. and Aksen Associates Inc., 1990.

[100] D. Garlan and D. Perry, "Introduction to the special issue on software architecture," *IEEE Transaction on Software Engineering*, vol. 21, April 1995.

[101] D. C. Luckham, J. Vera, and S. Meldal, "Three concepts of system architecture," Tech. Rep. CSL-TR-95-674, Stanford University, Computer Science Department, July 1995.

[102] M. Shaw, R. DeLine, and G. Zelenik, "Abstractions and implementations for architectural connections," in *Proc. 3rd Intl. Conf. on Configurable Distributed Systems*, 1996.

[103] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch: Why reuse is so hard," *IEEE Software*, vol. 12, pp. 17–26, November 1995.

[104] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Pattern.* Addison Wesley, 1994.

[105] B. H. C. Cheng, Y. Chen, G. C. Gannod, W. McUmber, and E. Y. Wang, "Discussion of architecture style." Private communication, April 1995.

[106] I. Kant, *Critique of Pure Reason.* Hackett, Indianapolis, IN., 1996.

[107] R. H. Bourdeau, B. H. C. Cheng, and B. Pijanowski, "A regional information system for environmental data analysis," *Photogrammetric Engineering & Remote Sensing*, vol. 62, July 1996. Special issue on "Remote Sensing and Global Environmental Change".

[108] R. H. Bourdeau, B. Pijanowski, and B. H. C. Cheng, "A decision support system for regional environmental analysis," in *Proc. of 25th International Symposium on Remote Sensing and Global Environmental Change: Tools for Sustainable Development (Vol. II)*, (Graz, Austria), pp. 223–233, 1993.

[109] Great Lakes Committee, "Keeping it on the land: improving Great Lakes water quality by controlling soil erosion and sedimentation," *The Advisor*, vol. June/July, 1992.

[110] B. H. C. Cheng, R. H. Bourdeau, and G. C. Gannod, "The object-oriented development of a distributed multimedia environmental information system," in *Proc. of IEEE 6th International Conference on Software Engineering and Knowledge Engineering*, pp. 70–77, June 1994.

[111] J. L. Sharnowski, G. C. Gannod, and B. H. C. Cheng, "A distributed multimedia environmental information system," in *Proc. of IEEE Int. Conference on Multimedia and Computing Systems*, (Washington, DC), pp. 142–194, May 1995.

[112] R. H. Bourdeau, B. H. C. Cheng, and G. C. Gannod, "A requirements analysis report for a regional decision support system," Technical Report MSU-CPS-94-70, Michigan State University, Department of Computer Science, A714 Wells Hall, East Lansing, 48824, November 1994.

[113] D. Craigen, S. Gerhart, and T. Ralston, "Formal methods reality check: Industrial usage," *IEEE Transactions on Software Engineering*, vol. 21, pp. 90–98, February 1995.

Appendices

Appendix A

The Object Model Formalization Rules

The formalization rules for the object model are given in this Appendix.

Definition (Semantics of object models) :
Let $\mathcal{O}$ be an object model. Let R be a binary association in $\mathcal{O}$ relating objects from classes D_1 and D_2. The semantics of $\mathcal{O}$ is an algebraic specification satisfying the following data.

(OM1) Each class C in the object model $\mathcal{O}$ is denoted by a sort of the same name.

(OM2) For each class C, a sort *C-STATES* is introduced as well as two nullary functions given by

$$\mathit{undef}_C : \quad \rightarrow \mathit{C\text{-}STATES} \ , \quad \mathit{err}_C : \quad \rightarrow C \ .$$

(OM3) Each object-state s, for which a double-headed arrow leads from a class C to the oval containing s, is denoted by a function with signature $s : \quad \rightarrow \mathit{C\text{-}STATES}$, and for every pair of object-states s_1 and s_2, the axiom $s_1 \neq s_2$ is included.

(OM4) For each class C, a valuation function '$' is introduced with the signature

$$\$: C \rightarrow \mathit{C\text{-}STATES} \quad .$$

The valuation of the error object is added as an axiom:

$$\$(\mathit{err}_C) = \mathit{undef}_C \quad .$$

(OM5) If there is a double-headed arrow labeled a (to indicate an attribute), leading from a class C to a class D (which depicts an attribute a of C), then the function signature

$$a : C \rightarrow D \quad ,$$

is added to specification for class C.

(OM6) If the class D in rule (OM5) is an external class, then the trait for D is `included` by the specification for C. If D has no parameters, then the clause

```
includes
    CLASS-D
```

is added. If D has parameters $p_1, \ldots, p_k$, and there is a line connecting each p_i to a class q_i, then the following clause is added:

```
includes
    CLASS-p1, ..., CLASS-pk,
    CLASS-D ( q1 for p1, ..., qk for pk)
```

(OM7) Association R is denoted by the predicate

$$R : D_1, D_k \rightarrow BOOL \ .$$

(OM8) The endpoints of association R determine a set of axioms. Suppose the D_1-endpoint depicts a multiplicity of m and the D_2-endpoint depicts a multiplicity of n. Then the axioms are derived by the following steps:

1. Decompose the m-to-n association R into an m-to-1 and 1-to-n binary association,
2. Determine the second-order specifications, P_1 and P_2, of each of these associations using the basis schemata,
3. Calculate the "intersection", P, of the specifications P_1 and P_2,
4. Unfold and skolemize P, yielding a set of first-order axioms that are included in the trait for R.

(OM9) Error object constraints are introduced:

$$(\forall d_1 : D_1, d_2 : D_2 \ \bullet \ R(err_{D_1}, d_2) \wedge d_1 \neq err_{D_1} \Rightarrow \neg R(d_1, d_2)) \ ,$$

$$(\forall d_1 : D_1, d_2 : D_2 \ \bullet \ R(d_1, err_{D_2}) \wedge d_2 \neq err_{D_2} \Rightarrow \neg R(d_1, d_2)) \ .$$

(OM10) The attributes, in the form of variable-type pairs, of an object class are formalized as sorts and a tuple that maps attribute types to the distinguished sort that represents the object class.

For object O, given attributes $a_1 : A_1, a_2 : A_2, ..., a_n : A_n$ (where $a_1, a_2, ..., a_n$ are variables and $A_1, A_2, ..., A_n$ are types), create the following expression in LOTOS specification

```
typedef O is
    sorts
        O, A_1, A_2, ..., A_n
    opns
        O : A_1, A_2, ..., A_n --> O
        get_a_1 : O --> A_1
        get_a_2 : O --> A_2
        ......
        get_a_n : O --> A_n
    eqns
        forall a_1 : A_1, a_2 : A_2, ..., a_n : A_n
        ofsort A_1
            get_a_1(O(a_1, a_2, ..., a_n)) = a_1
        ofsort A_2
            get_a_2(O(a_1, a_2, ..., a_n)) = a_2
            ......
        ofsort A_n
            get_a_n(O(a_1, a_2, ..., a_n)) = a_n
endtype
```

Appendix B

The Dynamic Model Formalization Rules

The formalization rules for the dynamic model are given in this Appendix. Rules **DFR-1** to **DFR-13** are for simple state diagrams; rules **DFR-14** to **DFR-20** are for concurrent state diagrams.

DFR-1 The object states S are formalized as LOTOS processes.
For every state s, $s \in S$, create the following LOTOS expression
process s : noexit :=
endproc

DFR-2 Every process that formally specifies a state s is associated with a parameter x of the distinguished sort ϕ.
For every state s, $s \in S$, and distinguished sort $\phi \in \Phi$, create the following LOTOS expression
process s (x: ϕ): noexit :=
endproc

DFR-3 The events of an object together with the events to be triggered are formalized as the inter-object communication external to the object, which is a subset of Γ (set of the object operators). The events, Σ, and events to be triggered, Ξ, in a state diagram D are specified as a formal gate list $[\Sigma \cup \Xi]$ for the processes in S.
For every state s, $s \in S$, distinguished sort ϕ, events Σ, and events to be triggered Ξ ($\phi \in \Phi, \Sigma \subseteq \Gamma$), create the following expression
process s $[\Sigma \cup \Xi]$(x: ϕ): noexit :=
endproc

DFR-4 Attribute pairs $x_i : a_i$ (data items) associated with an event e are formalized as *variable declarations* $?x_i : a_i$ associated with a LOTOS gate e.
For every state s and event e ($s \in S, e \in \Sigma$), distinguished sort ϕ, events Σ, and events to be triggered Ξ ($\phi \in \Phi, \Sigma \subseteq \Gamma$), if the partial function φ maps (s, e) to pairs $x_1 : a_1, x_2 : a_2, ..., x_n : a_n$, a non-empty subset of $\Lambda \times \Phi$ ($\{(x_1, a_1), (x_2, a_2), ..., (x_n, a_n)\} \in 2^{\Lambda \times \Phi}$), create the following expression
process s $[\Sigma \cup \Xi]$(x: ϕ): noexit :=
e $?x_1 : a_1$, $?x_2 : a_2$, ..., $?x_n : a_n$;
endproc
if the partial function φ maps (s, e) to an empty set (no associated attribute for event e), create the following expression
process s $[\Sigma \cup \Xi]$(x: ϕ): noexit :=
e;
endproc

DFR-5 An event to be triggered, e', is formalized as a value declaration at a LOTOS gate.

For every state s, event e, and condition c ($s \in S, e \in \Sigma, c \in C$), distinguished sort ϕ, events Σ, and events to be triggered Ξ ($\phi \in \Phi, \Sigma \subseteq \Gamma$), if the partial function δ maps (s, e, c) to a triggerable event e', create the following expression

process s [$\Sigma \cup \Xi$](x: ϕ): noexit :=
  e' !y_1, !y_2, ..., !y_m;
endproc,

where $\{y_1, y_2, ..., y_m\}$, a subset of $\{x_1, x_2, ..., x_n, x\}$, is specified in the corresponding state diagrams.

DFR-6 A guarding condition c is formalized as a guarded expression [c].

For every state s, event e, and condition c ($s \in S$, $e \in \Sigma$, $c \in C$), distinguished sort ϕ, events Σ, and events to trigger Ξ ($\phi \in \Phi, \Sigma \subseteq \Gamma$), if the partial function φ maps (s, e) to pairs $x_1 : a_1, x_2 : a_2, ..., x_n : a_n$, a non-empty subset of $\Lambda \times \Phi$ ($\{(x_1, a_1), (x_2, a_2), ..., (x_n, a_n)\} \in 2^{\Lambda \times \Phi}$), a non-empty subset of sorts Φ ($\{a_1, a_2, ..., a_n\} \in 2^{\Phi}$), and $c \neq c_0$, create the following expression

process s [$\Sigma \cup \Xi$](x: ϕ): noexit :=
  e ?$x_1 : a_1$, ?$x_2 : a_2$, ..., ?$x_n : a_n$; ([c] → ...)
endproc,

where $x_1, x_2, ..., x_n$ are variables to hold the attributes associated with the event.

DFR-7 A newly introduced predicate in a guarding condition is formalized as an operation of *Boolean* in the algebraic specification section.

Given object D, for every state s, event e, condition c ($s \in S$, $e \in \Sigma$, $c \in C$), distinguished sort ϕ, and predicate p introduced in condition c, if $x_1, x_2, ..., x_n$ are the arguments for p of sorts $X_1, X_2, ..., X_n$, create the following expression

specification D [$\Sigma \cup \Xi$](x: ϕ): noexit
typedef ϕ is Boolean
  opns
    p : $X_1, X_2, ..., X_n \longrightarrow$ Bool
endtype
...
process s [$\Sigma \cup \Xi$](x: ϕ): noexit :=
...
endproc
endspec

DFR-8 An action a is formalized in terms of a value declaration !a by an operator reference at the corresponding gate e.

For every state s, event e, and condition c ($s \in S$, $e \in \Sigma$, $c \in C$), if the partial function φ maps tuple (s, e) to sorts $a_1, a_2, ..., a_n$ (a non-empty subset of sorts Φ ($\{a_1, a_2, ..., a_n\} \in 2^{\Phi}$)), $c \neq c_0$, and the partial function δ maps tuple (s, e) to an action a ($a \in \Delta$), create the following expression

process s [$\Sigma \cup \Xi$](x: ϕ): noexit :=
  e ?$x_1 : a_1$, ?$x_2 : a_2$, ..., ?$x_n : a_n$;
    ([c] → e !a(y_1, y_2, ..., y_m))
endproc,

where $\{y_1, y_2, ..., y_m\}$, a subset of $\{x_1, x_2, ..., x_n, x\}$, is specified in the corresponding state diagrams.

DFR-9 An activity a is formalized as a value declaration !a at the corresponding gate e preceded by a nondeterministic *internal* event i.

For every s, $s \in S$, distinguished sort ϕ, events Σ, events to be triggered Ξ ($\phi \in \Phi, \Sigma \subseteq \Gamma$), and arguements y_1, y_2, ..., y_n of sorts Y_1, Y_2, ..., Y_n for activity a, if the partial function λ

maps state s to an activity a ($a \in \Delta$), create the following expression

process s [$\Sigma \cup \Xi$](x: ϕ, $y_1 : Y_1$, $y_2 : Y_2$, ..., $y_n : Y_n$): noexit :=
  i; e !a(y_1, y_2, ..., y_n);
endproc

DFR-10 A transition can cause a state change if and only if the action that corresponds to a transition is defined as a modifier operator.

For every state s and event e ($s \in S, e \in \Sigma$), if the partial function ψ does not map tuple (s,e) to state s ($s \neq \psi(s,e)$), then the partial function δ maps tuple (s,e) to a modifier operator ($\delta(s,e) \in \Omega$); if the partial function δ maps tuple (s,e) to a non-modifier operator ($\delta(s,e) \in \Theta$), then the partial function ψ maps tuple (s,e) to state s ($s = \psi(s,e)$).

DFR-11 The state transitions are formalized as process instantiations.

For every state s and event e ($s \in S, e \in \Sigma$), distinguished sort ϕ, events Σ, and events to be triggered Ξ ($\phi \in \Phi, \Sigma \subseteq \Gamma$), if the partial function φ maps (s,e) to pairs $x_1 : a_1, x_2 : a_2, ..., x_n : a_n$, a non-empty subset of $\Lambda \times \Phi$ ($\{(x_1,a_1),(x_2,a_2),...,(x_n,a_n)\} \in 2^{\Lambda \times \Phi}$), the partial function δ maps tuple (s,e) to an action a ($a \in \Delta$), the partial function ψ maps tuple (s,e) to a state s' ($s' \in S$), and a is of sort ϕ, create the following expression

process s [$\Sigma \cup \Xi$](x: ϕ): noexit :=
  e ?$x_1 : a_1$, ?$x_2 : a_2$, ..., ?$x_n : a_n$;
  s'[$\Sigma \cup \Xi$](a(y_1, y_2, ..., y_m))
endproc

if a is not of sort ϕ, create the following expression

process s [$\Sigma \cup \Xi$](x: ϕ): noexit :=
  e ?$x_1 : a_1$, ?$x_2 : a_2$, ..., ?$x_n : a_n$;
  e !a(y_1, y_2, ..., y_m);s'[$\Sigma \cup \Xi$](x)
endproc

DFR-12 Multiple state transitions from a single state are formalized as process instantiations composed by a *choice* operator.

For every state s, events e_1 and e_2 ($s \in S$, $e_1, e_2 \in \Sigma$), distinguished sort ϕ, events Σ, and events to be triggered Ξ ($\phi \in \Phi, \Sigma \subseteq \Gamma$), if $e_1 \neq e_2$, the partial function φ maps tuple (s,e_1) to $\{x_{11} : a_{11}, x_{12} : a_{12}, ..., x_{1n} : a_{1n}\}$, (s,e_2) to $\{x_{21} : a_{21}, x_{22} : a_{22}, ..., x_{2j} : a_{2j}\}$ (where $\{(x_{11},a_{11}),)x_{12},a_{12}),...,(x_{1n},a_{1n})\}, \{(x_{21},a_{21}),(x_{22},a_{22}),...,(x_{2j},a_{2j})\} \in 2^{\Lambda \times \Phi}$), the partial function δ maps tuple (s,e_1) to a_1, (s,e_2) to a_2 ($a_1, a_2 \in \Delta$), and the partial function ψ maps tuple (s,e_1) to $s_{1'}$, (s,e_2) to $s_{2'}$, create the following expression[1]

process s [$\Sigma \cup \Xi$](x: ϕ): noexit :=
  e_1 ?$x_{11} : a_{11}$, ?$x_{12} : a_{12}$, ..., ?$x_{1n} : a_{1n}$;
  e_1 !a_1(y_{11}, y_{12}, ..., y_{1m}); $s_{1'}$[$\Sigma \cup \Xi$](x)
[]
  e_2 ?$x_{21} : a_{21}$, ?$x_{22} : a_{22}$, ..., ?$x_{2j} : a_{2j}$;
  e_2 !a_2(y_{21}, y_{22}, ..., y_{2k}); $s_{2'}$[$\Sigma \cup \Xi$](x)
endproc

DFR-13 A state diagram can be formalized as either a LOTOS *specification* or a LOTOS *process definition.*

For state diagram $D = (S, \Sigma, \Lambda, \Delta, \Phi, C, \Xi, \psi, \lambda, \delta, \varphi, \zeta, s_0, c_0)$, we can have either

process D [$\Sigma \cup \Xi$] (x: ϕ) :**noexit :=**
  s_0 [$\Sigma \cup \Xi$] (x)
where
  ⟨ algebraic specification ⟩
  ⟨ process definitions ⟩
endspec

OR

[1] The expression may differ according to DFR-10 depending on the sort of a_i.

specification D [$\Sigma \cup \Xi$] (x: ϕ) :**noexit**
⟨ algebraic specification ⟩
behaviour
s_0 [$\Sigma \cup \Xi$] (x)
where
⟨ process definitions ⟩
endspec,

where the first format is used if the process will be composed within other specifications to form more complex behavior specifications; the second format is used if the specification is a top level specification.

DFR-14 Every *aggregate* object is responsible for composing the behavior of its concurrent *aggregation* objects to form one behavior specification.

DFR-15 The behavior of every *aggregation* object must be composed by and only by its *aggregate* object.

DFR-16 Only when there is an association between two *aggregation* objects, can the *aggregate* object model the corresponding synchronization between the two objects.

DFR-17 In a state diagram that composes dynamic models of concurrent objects, the dynamic behaviors are synchronized through shared services.
The state diagram $D = (S, \Sigma, \Lambda, \Delta, \Phi, C, \Xi, \psi, \lambda, \delta, \varphi, \zeta, s_0, c_0)$ composed of
$D_1 = (S_1, \Sigma_1, \Lambda_1, \Delta_1, \Phi_1, C_1, \Xi_1, \psi_1, \lambda_1, \delta_1, \varphi_1, \zeta_1, s_{10}, c_0)$,
$D_2 = (S_2, \Sigma_2, \Lambda_2, \Delta_2, \Phi_2, C_2, \Xi_2, \psi_2, \lambda_2, \delta_2, \varphi_2, \zeta_2, s_{20}, c_0)$,
....., and $D_n = (S_n, \Sigma_n, \Lambda_n, \Delta_n, \Phi_n, C_n, \Xi_n, \psi_n, \lambda_n, \delta_n, \varphi_n, \zeta_n, s_{n0}, c_0)$ is formalized as:

specification D[$\Sigma_1 \cup \Sigma_2 \cup .... \cup \Sigma_n \cup \Sigma$](x:$\phi$,$x_1$:$\phi_1$,$x_2$ ϕ_2,...,x_n:ϕ_n): **noexit**
<algebraic specification>
behaviour
s_0 [$\Sigma \cup \Xi$] (x)
|[Υ_1]|
(D_1[$\Sigma_1 \cup \Xi_1$] (x_1)
|[Υ_2]|
(D_2[$\Sigma_2 \cup \Xi_2$] (x_2)
|[Υ_3]|
(.....
|[Υ_n]|
D_n[$\Sigma_n \cup \Xi_n$] (x_n)
)))
where
<process definitions>
endspec,

where $\Upsilon_1, \Upsilon_2, ..., \Upsilon_n$ are shared services.

DFR-18 In a state diagram, the interleaving dynamic models are grouped together in terms of LOTOS interleaving process algebras.
If state diagrams $D_1 = (S_1, \Sigma_1, \Lambda_1, \Delta_1, \Phi_1, C_1, \Xi_1, \psi_1, \lambda_1, \delta_1, \varphi_1, \zeta_1, s_{10}, c_0)$,
$D_2 = (S_2, \Sigma_2, \Lambda_2, \Delta_2, \Phi_2, C_2, \Xi_2, \psi_2, \lambda_2, \delta_2, \varphi_2, \zeta_2, s_{20}, c_0)$,
....., and $D_n = (S_n, \Sigma_n, \Lambda_n, \Delta_n, \Phi_n, C_n, \Xi_n, \psi_n, \lambda_n, \delta_n, \varphi_n, \zeta_n, s_{n0}, c_0)$ are interleaving dy-

namic models in a given state diagram, the following specification shall be generated:

behaviour

```
.......
(
    D1[Σ1 ∪ Ξ1] (x1)
    |||
    D2[Σ2 ∪ Ξ2] (x2)
    |||
    .....
    |||
    Dn[Σn ∪ Ξn] (xn)
)
.......
```

DFR-19 If c is a constant, of sort C, introduced by the instantiation of parameter x of distinguished sort O for object O in a state diagram, then create the following expression

type O is
  sorts
    .....
  opns
    $c : \longrightarrow C$
  ....
endtype

DFR-20 For all constants $c_1, c_2, ..., c_n$ of sort C introduced in object O, create the following expression

type O is Boolean
  sorts
    .....
  opns
    .....
  eqns
    ofsort C
      c_1 eq c_1 = True;
      c_2 eq c_2 = True;
      .....
      c_3 eq c_3 = True;
      c_1 eq c_2 = False;
      c_1 eq c_3 = False;
      .....
      c_1 eq c_n = False;
      .....
      c_{n-1} eq c_n = False;
endtype

Appendix C

The Functional Model Formalization Rules

The formalization rules for the functional model are given in this Appendix. Rules **FFR-1** to **FFR-11** are for simple OFMs; rules **FFR-12** to **FFR-16** are for SRFMs.

FFR-1 Each data item in an OFM is formalized as a sort in the corresponding algebraic specification.
For all the data items, $D_i(1 \leq i \leq n)$, in an OFM, create the following expression in the sort declaration part of an ACT ONE specification of Full LOTOS

sorts
$D\_1,\ D\_2,\ ...,\ D\_n$

FFR-2 For every sort, S, used in the object specification, introduce operator **undef_s:** $\rightarrow S$.

FFR-3 Each service in an OFM is formalized as a gate that is associated with input and output in the corresponding Full LOTOS specification.
For all the servicess, $S_i(1 \leq i \leq n)$, with inputs, $I_{i_j}(1 \leq i_j \leq |I_i|$ (I_i represents the permutation of the input data), and outputs, O_i, in the OFM of object $OBJECT$, create the following expression

specification $OBJECT$ [S_1, S_2, ..., S_n] **: noexit :=**
(* S_1 : $i_{1_1} : I_{1_1}$, $i_{1_2} : I_{1_2}$, ..., $i_{1_{|I_1|}} : I_{1_{|I_1|}}$ -> O_1 *)
(* S_2 : $i_{2_1} : I_{2_1}$, $i_{2_2} : I_{2_2}$, ..., $i_{2_{|I_2|}} : I_{2_{|I_2|}}$ -> O_2 *)
...
(* S_n : $i_{1_n} : I_{1_n}$, $i_{1_n} : I_{1_n}$, ..., $i_{n_{|I_n|}} : I_{n_{|I_n|}}$ -> O_n *)
.....
typedef $OBJECT$ **is**
.....
opns
S_1 : I_{1_1}, I_{1_2}, ..., $I_{1_{|I_1|}}$ -> O_1
S_2 : I_{2_1}, I_{2_2}, ..., $I_{2_{|I_2|}}$ -> O_2
.....
S_n : I_{1_n}, I_{1_n}, ..., $I_{n_{|I_n|}}$ -> O_n
.....
endtype endspec

FFR-4 The set of attributes of an object is formalized as a parameter of the distinguished sort for the corresponding Full LOTOS specification.
For object class O, and its distinguished sort ϕ and a set of services Σ, create the following

specification O [Σ] ($o : \phi$) **: noexit :=**
...
expression **endspec**

FFR-5 The pre-state of the attributes of an object is stored in a local variable when a process, representing a state in the dynamic model, is entered.
For object class O, and its distinguished sort ϕ and a set of services Σ, if p is a process, representing a state in the dynamic model, defined in the process algebras, then create the following expression that introduces local variable PRE to store values of attributes at the entrance of the process

```
specification O [Σ] (o : φ)  : noexit :=
...
        process p [Σ] (o : φ)  : noexit :=
                ( let PRE:= o  in
                ...
                )
        endproc
...
endspec
```

FFR-6 The guarding condition and the precondition that are associated with the action of a state transition are conjuncted and formalized in terms of a guarded expression in LOTOS process algebras.
Suppose event e (associated with arguments $x_1 : a_1, x_2 : a_2, ..., x_n : a_n$), condition c, action a[1] (associated with parameters y_1, y_2, ..., y_m, if event e directly refers to service a, then the y's and x's are identical too), together constitute a state transition from state s of an object with distinguished sort ϕ and services Σ, if p is the precondition for service/function a, create the following expression

```
process s [Σ](o: φ): noexit :=
        ● ?x1 : a1, ?x2 : a2, ..., ?xn : an ;
            ([c and p] → <a(y1, y2, ... ym)>)
endproc
```

FFR-7 The postcondition that is associated with the action of a state transition is formalized in terms of a guarded expression in LOTOS process algebras.
Suppose event e (associated with arguments $x_1 : a_1, x_2 : a_2, ..., x_n : a_n$), condition c, action a action a (associated with parameters y_1, y_2, ..., y_m, if event e directly refers to service a, then the y's and x's are identical too), together constitute a state transition from state s of an object with distinguished sort ϕ and services Σ, if p and q are the pre- and postconditions for service/function a, create the following expression

```
process s [Σ](o: φ): noexit :=
        ● ?x1 : a1, ?x2 : a2, ..., ?xn : an ;
            ([c and p] → <a(y1, y2, ... ym)>);
            ([q]→...)
endproc
```

FFR-8 The postcondition that is associated with the action of a state transition, in the form of $result = \langle expression \rangle$, is formalized in terms of a value declaration in LOTOS process algebras.
Suppose event e (associated with arguments $x_1 : a_1, x_2 : a_2, ..., x_n : a_n$), condition c, action a action a (associated with parameters y_1, y_2, ..., y_m, if event e directly refers to service a, then the y's and x's are identical too), together constitute a state transition from state s of an object with distinguished sort ϕ and services Σ, if q is an expression, in the form of $result = op\ (z_1, z_2, ..., z_l)$ (op is an operator defined in the corresponding algebraic specifications, z_i $(1 \leq i \leq l)$ is a subset of y_i $(1 \leq i \leq m)$ and attributes) within the postcondition for service/function a, then create the

[1] a refers to a service/function.

following expression

```
process s [Σ](o: φ): noexit :=
        e ?x1 : a1, ?x2 : a2, ..., ?xn : an;
            ([c and p] → e! op(z1, z2, ... zl))
endproc
```

FFR-9 The postcondition that is associated with the action of a state transition can be formalized in terms of a local variable declaration in LOTOS process algebras.

Suppose event e (associated with arguments $x_1 : a_1, x_2 : a_2, ..., x_n : a_n$), condition c, action a(associated with parameters y_1, y_2, ..., y_m, if event e directly refers to service a, then the y's and x's are identical too), together constitute a state transition from state s of an object with distinguished sort ϕ and services Σ, if Q is a set of expression, in the form of $o^{\wedge}.a_i = op_i\ (z_{i_1}, z_{i_2}, ..., z_{i_l})(1 \leq i \leq k)$ (op_i is an operator defined in the corresponding algebraic specifications, $z_{i_j}\ (1 \leq j \leq l)$ is a subset of $y_i\ (1 \leq i \leq m)$ and attributes) within the postcondition for service/function a, create the following expression

```
process s [Σ](o: φ): noexit :=
        e ?x1 : a1, ?x2 : a2, ..., ?xn : an;
            ([c and p] →
                (let POST.a1 := op1(z1_1, z1_2, ... z1_l),
                     POST.a2 := op2(z2_1, z2_2, ... z2_l),
                     ...,
                     POST.ak := opk(zk_1, zk_2, ... zk_l) in
                     ...
                )
            ...
            )
endproc
```

FFR-10 The refinement of object O, results in a refined algebraic specification, S_REF, that includes additional operators and axioms.

For every object O that is further refined into a set of aggregate objects O_1, O_2, ..., O_n, create the following expression

```
type O_REF is
    ...
endtype
```

FFR-11 In a refined algebraic specification for an object (or system), the algebraic specifications for the aggregate objects are included in order to make use of the services defined in the aggregate objects.

For every object O that is further refined into a set of aggregate objects O_1, O_2, ..., O_n, create the following expression

```
type O_REF is O1, O2, ..., On
    ...
endtype
```

FFR-12 Each internal function in an SRFM is declared along with services as annotations and operators of algebraic specification.

For all the internal functions, $F_i(1 \leq i \leq n)$, with inputs, $I_{i_j}(1 \leq i_j \leq |I_i|$ (I_i represents the permutation of the input data), and outputs, O_i, in the OFM of object $OBJECT$, create the

following expression

```
specification OBJECT [< service list>]]  : noexit :=
        (* F1 : i1_1 : I1_1, i1_2 : I1_2, ..., i1_|I1_1| : I1_|I1_1| -> O1 *)
        (* F2 : i2_1 : I2_1, i2_2 : I2_2, ..., i2_|I2_1| : I2_|I2_1| -> O2 *)
        ...
        (* Fn : in_1 : In_1, in_1 : In_1, ..., in_|In_1| : In_|In_1| -> On *)
        ......
typedef OBJECT is
        ......
        opns
                F1 : I1_1, I1_2, ..., I1_|I1_1| -> O1
                F2 : I2_1, I2_2, ..., I2_|I2_1| -> O2
                ......
                Fn : In_1, In_1, ..., In_|In_1| -> On
        ......
endtype endspec
```

FFR-13 If a data flow D is aggregated into a set of data flows, D_1, D_2, ..., D_n, then D is redefined/refined as a set of operations that constitute D from D_1, D_2, ..., D_n and single out D_1, D_2, ..., D_n from D.

For every data flow D that is aggregated into a set of data flows, D_1, D_2, ..., D_n, create the following expression

```
sorts
        D_1, D_2, ..., D_n
opns
        D : D_1, D_2, ..., D_n ⟶ D
        get_d_1 : D ⟶ D_1
        get_d_2 : D ⟶ D_2
        ......
        get_d_n : D ⟶ D_n
eqns
        forall d_1 : D_1, d_2 : D_2, ..., d_n : D_n
        ofsort D_1
                get_d_1(D(d_1, d_2, ..., d_n)) = d_1
        ofsort D_2
                get_d_2(D(d_1, d_2, ..., d_n)) = d_2
                ......
        ofsort D_n
                get_d_n(D(d_1, d_2, ..., d_n)) = d_n
```

FFR-14 All the data splittings for a data flow D together with D is redefined/refined as a record data type.

For data flow D and all the sets of data flows, S_1, S_2, ..., S_n, into which D is split, if data items D_1, D_2, ..., D_m cover all the data items in S_i $(1 \leq i \leq n)$ and do not contain redundant data

items, then create the following expression

```
sorts
      D_1, D_2, ..., D_n
opns
      D : D_1, D_2, ..., D_n ⟶ D
      get_d_1 : D ⟶ D_1
      get_d_2 : D ⟶ D_2
      ......
      get_d_n : D ⟶ D_n
eqns
      forall d_1 : D_1, d_2 : D_2, ..., d_n : D_n
      ofsort D_1
            get_d_1(D(d_1, d_2, ..., d_n)) = d_1
      ofsort D_2
            get_d_2(D(d_1, d_2, ..., d_n)) = d_2
            ......
      ofsort D_n
            get_d_n(D(d_1, d_2, ..., d_n)) = d_n
```

FFR-15 If a data flow D is duplicated to or selected from a set of data flows, D_1, D_2, ..., D_n, operators $D\_2\_D_1$, $D_1\_2\_D$, $D\_2\_D_2$, $D_2\_2\_D$, ..., $D\_2\_D_n$, and $D_n\_2\_D$, in the formats of

```
opns
      D_2_D_1: D → D_1
      D_2_D_2: D → D_2
      ...
      ...
      D_2_D_n: D → D_n
```

are introduced in the refined algebraic specification. In addition, the following axioms are used to specify the mutual convertibility:

```
eqns
      forall x: D
      ofsort D
            D_1_2_D (D_2_D_1 (x)) = x;
            D_2_2_D (D_2_D_2 (x)) = x;
            ....
            D_n_2_D (D_2_D_n (x)) = x;
```

FFR-16 The post- and preconditions of two adjacent services in a SRFM shall not form a contradiction. For any two services s_1 and s_2 in a SRFM, if (1) the output of s_1 serves as an input of s_2, (2) p_{s_1} and q_{s_1} are the pre- and post conditions for s_1, and (3) p_{s_2} is the precondition for s_2, create and prove the following expression

$(p_{s_1} \Rightarrow ((q_{s_1} \text{ and } p_{s_2}) \text{ eq false})) = \text{false}$

Appendix D

Complete LOTOS specification for *Disk Manager* Behavior

```
specification Disk_Manager[input,output,com,dec,ins,ret,delete] : noexit

library
     Boolean, NaturalNumber, HexString
endlib

type GenericData is NaturalNumber, HexString
endtype

type Data is GenericData renamedby
    sortnames
        D for HexString
        K for Nat
    opnnames
        sizeof for Length
endtype

behavior
hide com, dec, ins, ret, del in
  Storage[ins,ret,del] |[ins,ret,del]| idle[input,output,com,dec,ins,ret,del]
  |[com,dec]| compression[com,dec]

where
  process idle[input,output,com,dec,ins,ret,delete]:noexit:=
    input ?d:D ?k:K; com !d ; wait[input,output,com,dec,ins,ret,delete](k)
    []
    output ?k:K; ret!k; wait[input,output,com,dec,ins,ret,delete](k)
  endproc

  process wait[input,output,com,dec,ins,ret,delete](k:K):noexit:=
    com?d:D;ins!d!k;idle[input,output,com,dec,ins,ret,delete]
    []
    ret?d:D;dec!d;wait[input,output,com,dec,ins,ret,delete](k)
    []
    dec?d:D;output!d;idle[input,output,com,dec,ins,ret,delete]
  endproc
```

```
process Storage[insert,retrieve,delete]:noexit:=
     Empty_state[insert,retrieve,delete](new)

where
type Storage is Boolean, NaturalNumber, Data
     sorts
       S
     opns
       undef_d   : -> D
       undef_s   : -> S
       count     : S -> Nat
       insert    : S, D, K-> S
       retrieve  : S, K -> D
       delete    : S, K -> S
       _ eq _    : D, D -> Bool
       _ eq _    : K, K -> Bool
     eqns
         forall d: D, k, k1, k2: K, s: S
         ofsort Nat
              count(new) = 0;
              count(insert(s, e)) = Succ(0) + count(s);
         ofsort D
              retrieve(new, k) = undef_d;
              k1 eq k2 => retrieve(insert(s,d,k2),k1) = d;
              not(k1 eq k2) => retrieve(insert(s,d,k2),k1)=retrieve(s,k1);
         ofsort S
              delete(new, k) = new;
              k1 eq k2 => delete(insert(s,d,k2),k1)=s;
              not(k1 eq k2) =>delete(insert(s,d,k2),k1)=delete(s,k1);
endtype

     process Empty_State[insert,retrieve,delete](s:S):noexit:=
          insert ?d:D ?k:K; None_Empty_State[insert,retrieve,delete](insert(s,d,k))
     endproc

     process None_Empty_State[insert,retrieve,delete](s:S):noexit:=
          retrieve ?k:K; retrieve !retrieve(s,k);
              None_Empty_State[insert,retrieve,delete](s))
         []
          insert ?d:D ?k:K;
             None_Empty_State[insert,retrieve,delete](insert(s,d,k))
         []
          delete ?k:K; (
             [count(s) gt Succ(0)] ->
               None_Empty_State[insert,retrieve,delete](delete(s,k))
             []
             [count(s) eq Succ(0)] ->
               Empty_State[insert,retrieve,delete](delete(s,k)))
     endproc
endproc

process compression [compress, decompress] : noexit :=
```

```
  Idle_State [compress, decompress]

where
type Compression is NaturalNumber, Data
  opns
    compress    : D -> D
    decompress  : D -> D
  eqns
    forall x: D
    ofsort Bool
      sizeof(x) ge sizeof(compress(x)) = true;
      sizeof(x) le sizeof(decompress(x)) = true;
    ofsort D
      decompress(compress(x)) = x;
endtype

  process Idle_State [compress, decompress] : noexit :=
    compress ?data:D; Compress_State[compress,decompress](data)
    []
    decompress?data:D;Decompress_State[compress,decompress](data)
  endproc

  process Compress_State[compress,decompress](data:D):noexit :=
    i; compress !compress(data); Idle_State[compress,decompress]
  endproc

  process Decompress_State[compress,decompress](data:D):noexit :=
    i; decompress!decompress(data);Idle_State[compress,decompress]
  endproc
endproc
endspec
```

Appendix E

Instantiated LOTOS specification for *Disk Manager*

```
specification Disk_Manager [Input, Output] : noexit
     (* Input :D: Data, k: Key -> Void *)
     (*   requires d eq undef_d = false and k eq undef_k = false *)
     (*   modifies disk *)
     (*   ensures disk' eq input(disk^, d, k) = true *)
     (* Output : k:Key -> HexString *)
     (*   requires disk eq empty = false and k eq undef_k = false *)
     (*   ensures result = output (disk,k) *)

library
     Boolean, HexString, NaturalNumber
endlib

type Disk_Manager is Boolean, HexString, NaturalNumber
     sorts
          Disk
     opns
          empty   : -> Disk
          input   : Disk, HexString, Nat -> Disk
          output  : Disk, Nat -> HexString
          undef_d : -> HexString
          undef_k : -> Nat
     eqns
          forall disk: Disk, d,x: HexString, k, k1, k2: Nat, y: HexDigit
          ofsort Bool
            undef_d eq undef_d = true;
            undef_k eq undef_k = true;
            undef_k eq 0 = false;
            k ge 0 => succ(k) eq undef_k = false;
            Hex(y) eq undef_d = false;
            (y + x) eq undef_d = false;
          ofsort HexString
            output (empty, k) = undef_d;
            (k1 eq k2) => output(input(disk,d,k1),k2) = d;
            not(k1 eq k2) => output(input(disk,d,k1),k2) = output(disk,k2);
endtype
```

```
behavior
     Idle[Input, Output] (empty)
where
     process Idle [Input, Output] (disk: Disk) :noexit:=
          Input ?d:HexString ?k:Nat; Idle [Input, Output](input(disk,d,k))
          []
          Output ? k:Nat; Output ! output(disk, k); Idle [Input, Output](disk)
     endproc
endspec
```

Appendix F

LSL specifications

```
Disk (D, K): trait

        introduces
                empty   : -> Disk
                input   : Disk, D, K -> Disk
                output  : Disk, K -> D
                delete  : Disk, K -> Disk
                undef_d : -> D

        asserts
                Disk generated by empty, input
                \forall d: D, k, k1: K, disk: Disk
                output (empty, k) == undef_d;
                output (input(disk,d,k), k1) == if (k=k1) then d
                        else output(disk,k1);
                delete (empty, k) == empty;
                delete (input(disk,d,k), k1) == if (k=k1) then disk
                        else delete(disk,k1);


Storage (D, K): trait

    includes Natural

    introduces
         empty    : -> S
         count    : S -> N
         insert   : S, D, K -> S
         retrieve : S, K -> D
         delete   : S, K -> S
         undef_d  : -> D
         undef_s  : -> S

    asserts
         S generated by empty, insert
         \forall d, d1, d2: D, k, k1, k2: K, s, s1, s2: S
         count(empty) == 0;
         count(insert(s, d, k)) == succ(count(s));
         ~(insert(s,d,k) = empty);
```

```
        retrieve(empty, k) == undef_d;
        retrieve(insert(s, d, k2), k1) == if (k1 = k2) then d
              else retrieve(s, k1);
        delete(empty, k) == empty;
        delete(insert(s, d, k2), k1) == if (k1 = k2) then s
              else delete(s, k1);


Compressor (D): trait

        includes Natural

        introduces
                compress   : D -> D
                decompress : D -> D
                size       : D -> N

        asserts
                \forall d : D
                size (compress(d)) <= size(d);
                size (d) <= size(decompress(d));
                decompress(compress(d)) = d;


Disk_ref (D, K): trait

        includes Storage(D,K), Compressor(D)

        implies
                \forall s: S, n: N
                ~(s=empty) => ~(retrieve(s,n) = undef_d);
```

Appendix G

LOTOS specification for revised *Disk Manager*

```
specification Disk_Manager [input, output] : noexit

library
     Boolean, NaturalNumber
endlib

type Disk_Manager is
     sorts
          D, K
endtype

behavior

 hide com, dec, ins, ret, delete in
  Storage[ins, ret, delete]
  |[ins, ret]|
  idle[input, output, com, dec, ins, ret, delete]
  |[com, dec]|
  compression[com,dec]

where

  process idle [input, output, com, dec, ins, ret, delete] : noexit :=
    input ?d:D ?k:K; com !d !k; wait[input,output,com,dec,ins,ret,delete](k)
    []
    output ?k:K; ret !k; wait[input, output, com, dec, ins, ret, delete](k)
  endproc

  process wait [input, output, com, dec, ins, ret, delete] (k: K) : noexit :=
    com ?d:D; ins !d !k; idle [input, output, com, dec, ins, ret, delete]
    []
    ret ?d:D; dec !d; wait[input, output, com, dec, ins, ret, delete](k)
    []
    dec ?d:D; output !d; idle [input, output, com, dec, ins, ret, delete]
  endproc
```

```
process Storage [insert, retrieve, delete] : noexit :=

(*behaviour*)
     Empty [insert, retrieve, delete] (new)

where
type Storage is Boolean, NaturalNumber
     sorts
          S, D, K, E
     opns
         new       : -> S
         undef_d   : -> D
         undef_s   : -> S
         count     : S -> Nat
         insert    : S, D, K -> S
         retrieve  : S, K -> D
         delete    : S, K -> S
         _ eq _    : E, E -> Bool
         _ eq _    : D, D -> Bool
         _ eq _    : K, K -> Bool

     eqns
         forall d: D, k, k1, k2: K, s: S, e, e1, e2: E
        ofsort Nat
              count(new) = 0;
              count(insert(s, d, k)) = Succ(0) + count(s);
         ofsort D
              retrieve(new, k) = undef_d;
              k eq k1 => retrieve(insert(s,d,k), k1) = d;
              not(k eq k1) => retrieve(insert(s,d,k), k1) = retrieve(s,k1);
         ofsort S
              delete(new, k) = new;
              k1 eq k2 => delete(insert(s, d, k2), k1) = s;
              not(k1 eq k2) => delete(insert(s, d, k2), k1) = delete(s, k1);
endtype

     process Empty [insert, retrieve, delete] (s: S): noexit :=
          insert ? d : D ? k : K;
              None_Empty [insert, retrieve, delete] (insert(s, d, k))
     endproc

     process None_Empty [insert, retrieve, delete] (s: S): noexit :=
          retrieve ? k : K; retrieve ! retrieve (s, k);
              None_Empty [insert, retrieve, delete] (s)
         []
          insert ? d : D ? k : K;
              None_Empty [insert, retrieve, delete] (insert(s,d,k))
         []
          delete ? k : K; (
              [count(s) gt Succ(0)] ->
                  None_Empty [insert, retrieve, delete] (delete(s,k))
             []
              [count(s) eq Succ(0)] ->
                  Empty [insert, retrieve, delete] (delete(s,k))
```

```
          )
     endproc
endproc

process compression [compress, decompress] : noexit :=

(*library
    NaturalNumber
endlib*)

(*behavior*)
  Idle [compress, decompress]

where
type Compression is NaturalNumber
  sorts
    D
  opns
    compress    : D -> D
    decompress  : D -> D
    sizeof      : D -> Nat
  eqns
    forall x: D
    ofsort Bool
      sizeof(x) ge sizeof(compress(x)) = true;
      sizeof(x) le sizeof(decompress(x)) = true;
    ofsort D
      decompress(compress(x)) = x;
endtype

  process Idle [compress, decompress] : noexit :=
    compress ? data: D; Compress [compress, decompress] (data)
    []
    decompress ? data: D; Decompress [compress, decompress] (data)
  endproc

  process Compress [compress, decompress] (data: D) : noexit :=
    i; compress ! compress (data); Idle [compress, decompress]
  endproc

  process Decompress[compress, decompress] (data: D) : noexit :=
    i; decompress ! decompress (data); Idle [compress, decompress]
  endproc
endproc

endspec
```

Appendix H

Complete LOTOS specification of ENFORMS for static analysis

```
specification ENFORMS_REF [Browse, Retrieve, Analysis, Query, GetTable,
              Archive_Query, Register, Archive_Query4899,
              Archive_Query5699] (e: ENFORMS): noexit

(* Browse:    ui: User_Input -> Retrieve_Request *)
(*    ensures    result = Browse (ui, getIndices(e)) *)
(* Retrieve:    rr: Retrieve_Request -> Retrieve_Result *)
(*    requires isValid(rr)    *)
(*    ensures result = Retrieve (rr, getArchives(e)) *)
(* Analysis:    ar: Analysis_Request -> Analysis_Result *)

library
    ENFORMS, Name_Server, Archive_Server, Client, Boolean, Channel
endlib

type ENFORMS_REF is ENFORMS, Name_Server, Archive_Server, Client, Channel
    opns
        Retrieve_Request : Data_Archive_Name, Query_Request -> Retrieve_Request
        get_data_archive_name : Retrieve_Request -> Data_Archive_Name
        get_query_request : Retrieve_Request -> Query_Request
        Retrieve_Result2Query_Result : Retrieve_Result -> Query_Result
        Query_Result2Retrieve_Result : Query_Result -> Retrieve_Result
        pcs    : -> Data_Archive_Name
        pcs_db    : -> Data_Archive
        4899    : -> Address
        5699    : -> Address
        storet    : -> Data_Archive_Name
        storet_db : -> Data_Archive
        channel : -> Channel

    eqns
        forall dan, dan1, dan2: Data_Archive_Name, qr:
               Query_Request, rr: Retrieve_Result, qr1: Query_Result
        ofsort Data_Archive_Name
            get_data_archive_name (Retrieve_Request(dan, qr)) = dan;
```

```
        ofsort Query_Request
            get_query_request (Retrieve_Request(dan, qr)) = qr;
        ofsort Retrieve_Result
            Query_Result2Retrieve_Result(Retrieve_Result2Query_Result(rr)) = rr;
        ofsort Query_Result
            Retrieve_Result2Query_Result(Query_Result2Retrieve_Result(qr1)) = qr1;
        ofsort Bool
            pcs eq pcs = True;
            storet eq storet = True;
            undef_Data_Archive_Name eq undef_Data_Archive_Name = True;
            pcs eq undef_Data_Archive_Name = False;
            undef_Data_Archive_Name eq pcs = False;
            storet eq undef_Data_Archive_Name = False;
            undef_Data_Archive_Name eq storet = False;
            pcs eq storet = False;
            storet eq pcs = False;
            pcs_db eq pcs_db = True;
            storet_db eq storet_db = True;
            undef_Data_Archive eq undef_Data_Archive = True;
            pcs_db eq undef_Data_Archive = False;
            undef_Data_Archive eq pcs_db = False;
            storet_db eq undef_Data_Archive = False;
            undef_Data_Archive eq storet_db = False;
            pcs_db eq storet_db = False;
            storet_db eq pcs_db = False;
            4899 eq 4899 = True;
            5699 eq 5699 = True;
            undef_Address eq undef_Address = True;
            4899 eq undef_Address = False;
            undef_Address eq 4899 = False;
            5699 eq undef_Address = False;
            undef_Address eq 5699 = False;
endtype

behavior

    (Browse ? ui: User_Input;
     Browse ! Browse (ui, getIndices(e));
     Browse [Browse, Retrieve, Analysis, Query] (e)
    |[Query]|
    (Client [Query, GetTable, Archive_Query] (Client(empty))
    |[GetTable,Archive_Query]|
    (Name_Server [Register, GetTable] (Name_Server(empty))
    |[Register]|
    (Channel[Archive_Query, Archive_Query4899, Archive_Query5699] (channel)
    |[Archive_Query4899,Archive_Query5699]|
    (Archive_Server [Archive_Query4899, Register]
                    (Archive_Server(pcs, pcs_db, 4899))
     |||
     Archive_Server [Archive_Query5699, Register]
                    (Archive_Server(storet, storet_db, 5699)))))))

where
```

```
process Browse [Browse, Retrieve, Analysis, Client_Query](e:ENFORMS):noexit:=
    Browse ? ui: User_Input; Browse ! Browse (ui, getIndices(e));
        Browse[Browse, Retrieve, Analysis, Client_Query] (e)
    []
    Retrieve ? rr: Retrieve_Request; (
        [isValid(rr)] -> Client_Query !get_data_archive_name(rr)
                                      !get_query_request (rr);
            WaitQuery[Browse, Retrieve, Analysis, Client_Query](e)
        []
        [not(isValid(rr))] -> Browse[Browse,Retrieve,Analysis,Client_Query](e)
    )
endproc

process WaitQuery [Browse, Retrieve, Analysis, Client_Query] (e: ENFORMS)
        : noexit :=
    Client_Query ? rr: Query_Result; Retrieve ! Query_Result2Retrieve_Result (rr);
        Analysis [Browse, Retrieve, Analysis, Client_Query] (e)
endproc

process Analysis [Browse, Retrieve, Analysis, Client_Query] (e: ENFORMS)
        : noexit :=
    Browse ? ui: User_Input;
        Browse ! Browse (ui, getIndices(e));
        Browse[Browse, Retrieve, Analysis, Client_Query] (e)
    []
    Analysis ? ar: Analysis_Request;
        Analysis! Analysis (ar);
        Analysis[Browse, Retrieve, Analysis, Client_Query] (e)
endproc


(********************Channel*****************************)
process Channel [Send, Channel4899, Channel5699] (c: Channel) : noexit :=

    Idle [Send, Channel4899, Channel5699] (c)

where

process Idle [Send, Channel4899, Channel5699] (c: Channel): noexit :=
    Send ? qr: Query_Request ? a: Address; (
    [a eq 4899] -> Channel4899 ! qr ! a;
                   waitChannel4899 [Send, Channel4899, Channel5699] (c)
    []
    [a eq 5699] -> Channel5699 ! qr ! a;
                   waitChannel5699 [Send, Channel4899, Channel5699] (c)
    )
endproc

process waitChannel4899 [Send, Channel4899, Channel5699] (c: Channel): noexit :=
    Channel4899 ? qr: Query_Result; Send ! qr;
        Idle[Send, Channel4899, Channel5699] (c)
endproc

process waitChannel5699 [Send, Channel4899, Channel5699] (c: Channel): noexit :=
```

```
    Channel5699 ? qr: Query_Result; Send ! qr;
        Idle[Send, Channel4899, Channel5699] (c)
endproc
endproc

(*****************************Client*********************************)

process Client [Query, Name_Server_GetTable, Archive_Server_Query] (c: Client)
              : noexit :=

    Name_Server_GetTable;
         Init [Query, Name_Server_GetTable, Archive_Server_Query] (c)

where

process Init [Query, Name_Server_GetTable, Archive_Server_Query] (c: Client)
        : noexit :=
    (let PRE: Client = c in
    Name_Server_GetTable ? st: Server_Table;
    (let POST: Client = Client(st) in
        Idle [Query, Name_Server_GetTable, Archive_Server_Query] (POST))
    )
endproc

process Idle [Query, Name_Server_GetTable, Archive_Server_Query] (c: Client)
        : noexit :=
    Query ? dan: Data_Archive_Name ? qr: Query_Request;
    Archive_Server_Query ! qr ! getAddress(getTable(c), dan);
    WaitQuery [Query, Name_Server_GetTable, Archive_Server_Query] (c)
endproc

process WaitQuery [Query, Name_Server_GetTable, Archive_Server_Query] (c: Client)
        : noexit :=
    Archive_Server_Query ? qr: Query_Result;
        Query ! qr; Idle [Query, Name_Server_GetTable, Archive_Server_Query] (c)
endproc
endproc

(*********************Archive Server*********************************)

process Archive_Server [Query, Name_Server_Register] (as: Archive_Server)
        : noexit :=

    Name_Server_Register !getArchiveName(as) !getAddress(as);
        pollRQ [Query, Name_Server_Register] (as)

where

process pollRQ [Query, Name_Server_Register] (as: Archive_Server) : noexit :=
    Query ? qr: Query_Request ? a: Address; (
    [isMyAddress(a, as)] -> Query [Query, Name_Server_Register] (as, qr)
    []
    [not(isMyAddress(a, as))] -> pollRQ [Query, Name_Server_Register] (as))
endproc
```

```
process Query [Query, Name_Server_Register]
        (as: Archive_Server, qr: Query_Request) : noexit :=
    i; Query ! Query (qr, getArchive(as));
        pollRQ [Query, Name_Server_Register] (as)
endproc
endproc

(**********************Name_Server*******************************)

process Name_Server [Register, GetTable] (ns: Name_Server) : noexit :=

    pollRQ [Register, GetTable] (ns)

where

process pollRQ [Register, GetTable] (ns: Name_Server) : noexit :=
    (let PRE: Name_Server = ns in
    Register ? dan: Data_Archive_Name ? a: Address;
    ( let POST: Name_Server = Name_Server(insert(getTable(PRE), dan, a)) in
        pollRQ[Register, GetTable] (POST))
    []
    GetTable; GetTable ! getTable(ns); pollRQ [Register, GetTable] (ns)
    )
endproc
endproc
endspec
```