



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

UNIVERSITY OF ALBERTA

**Specification Methods for Distributed Software Design: Issues and
Approaches**

by

Narendra Ravi



A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of **Master of Science**

Department of Computing Science

Edmonton, Alberta
Spring 1992



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-73150-8

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: **Narendra Ravi**

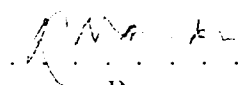
TITLE OF THESIS: **Specification Methods for Distributed Software Design:
Issues and Approaches**

DEGREE: **Master of Science**

YEAR THIS DEGREE GRANTED: **1992**

Permission is hereby granted to the UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication rights and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

(Signed) 

Permanent Address:
Plot 32, G. K. Colony
Secunderabad, A.P.
India - 500 594

Date: **Apr. 22, 1992**

Asatomaā Sadgamaya
Tamasomaā Jyotirgamaya
Mryutyorma Amritangamaya

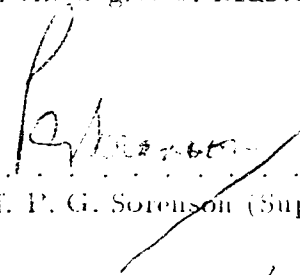
from Untruth unto Truth
from Darkness unto Light
from Death unto Life Divine

• *The Rig Veda*

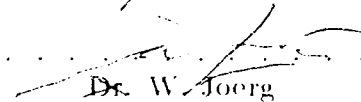
UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

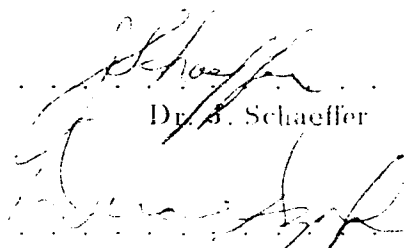
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Specification Methods for Distributed Software Design: Issues and Approaches** submitted by **Narendra Ravi** in partial fulfillment of the requirements for the degree of **Master of Science**.



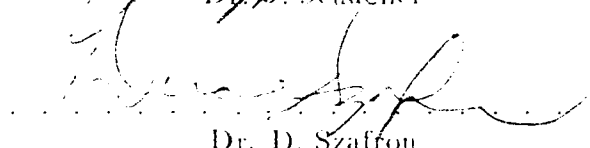
Prof. P. G. Sorenson (Supervisor)



Dr. W. Joerg



Dr. B. Schaeffer



Dr. D. Szaffron

Date: April 23, 1992

To my parents

Abstract

In this thesis we characterize the features of design specification languages for distributed software systems development. Concurrency is a design issue and therefore must be modeled during this stage of software development. Since distributed software is more complex than sequential software structurally, as well as behaviorally, correctness is an important issue. Thus a formal model is required for the development of robust software systems. We choose the *Incremental Transformation* (**IncTr**) model based on the approach proposed in [Lehmann et al., 1984] as the basis for distributed software development. The **IncTr** model provides a formal, step-wise refinement approach to the specification, verification and implementation of software systems.

To effectively apply this model to distributed software design we need an evolvable specification language with a consistent notation and associated semantics. To identify the features of such a language, we survey four groups of formal methods — *algebraic*, *logic-based*, *graph-based* and *object-based*, for specifying and verifying distributed systems, classified according to their underlying formalism. We develop criteria for evaluating these methods and compare them. We then propose a framework for developing design specification languages that defines an appropriate representation scheme for the **IncTr** model. Since tools are important components of a computer-based software engineering system, we also discuss the tool support required for the specification languages that could be developed based on the proposed framework.

Acknowledgments

I would like to thank Paul Sorenson, my supervisor, for his constant guidance, encouragement and support. Paul waded through many thesis drafts and was instrumental in weeding out a number of errors.

Randy Goebel taught me how to do research and I thank him for that. Paul Sorenson made sure I learnt it right. Thanks are due to my thesis committee members for their insightful comments and patience in reading two versions of my thesis.

A number of my friends made my stay in Edmonton an enjoyable experience. I would like to thank Aditya Ghose for many intellectually stimulating discussions, Manjunatha Bhat for moral support, Viral Maniar for enlivening the atmosphere during the long winters, and many other members of the India Students Association for relieving the monotony of academic work.

Without the encouragement and affection I received from my parents, I would never have made it this far. Thanks to my wife, Kavita, for her companionship during the last stages of my thesis.

Contents

1	Introduction	1
1.1	Related Work	2
1.2	Definitions	3
1.2.1	Scope of this Research: Design Specifications	4
1.3	Formal Methods: Motivation	4
1.4	Issues in Developing Distributed Systems	5
1.4.1	Specification of Distributed Systems	6
1.5	Structure of the Thesis	7
2	Issues in Formal Specifications	9
2.1	Riding the Elevators: An Example of a Reactive System of Processes . .	9
2.1.1	Problem Specification	9
2.1.2	Possible Solutions	10
2.2	Formal Issues in Specification	10
2.3	Mathematical Models and Theories: Their Relation to Specifications . .	11
2.3.1	Software Design by Incremental Transformations: IncTr Model . .	12
2.4	Properties of Concurrent Programs	14
2.5	Chapter Summary	17
3	Criteria for Evaluating Specification Methods	18
3.1	Characteristics of a Specification Formalism	18
3.2	Criteria for Comparison	20

3.2.1	Riding the Elevators: Comparison Criteria	22
3.3	Chapter Summary	22
4	Algebraic Methods	23
4.1	Lotos: A CCS-based Formalism with Abstract Data Types	23
4.1.1	Processes in Lotos	24
4.1.2	Process Definitions and Specifications	24
4.1.3	Combinators and Behavior Expressions	25
4.1.4	Verification	28
4.2	Example: Riding the Elevators	28
4.3	An Analysis of Lotos	31
4.3.1	Drawbacks of Lotos	33
4.4	Chapter Summary	33
5	Logic-based Methods	35
5.1	Introduction	35
5.2	The UNITY approach to Development of Parallel Program Specifications	35
5.2.1	UNITY theory	37
5.2.2	The UNITY Model	39
5.2.3	UNITY Program Notation	41
5.2.4	The UNITY Proof Rules	43
5.3	Example: Riding the Elevators	46
5.4	Analysis of UNITY	48
5.4.1	Drawbacks of UNITY	49
5.4.2	Unity and Temporal Logic	49
5.5	Chapter Summary and UNITY Extensions	50
6	Graph-based Methods	52
6.1	Statecharts: Extending State Transition Diagrams	52
6.1.1	Higraphs: Visual Formalisms	53

6.1.2	Statecharts: Features	54
6.1.3	STATEMATE: An Environment for Developing Distributed Systems	58
6.2	Example: Riding the Elevators	61
6.3	Analysis of Statecharts	63
6.3.1	Statecharts and Graph Grammars	66
6.4	Chapter Summary	69
7	Object-based Methods	71
7.1	Introduction	71
7.2	Object-based Design Methods	72
7.2.1	Object and Classes	72
7.2.2	Object-based Design	73
7.2.3	Object-based development and Concurrency	74
7.3	Actors: A Model for Distributed Computation	74
7.3.1	Components of an Actor System	75
7.3.2	The Actor Model	77
7.3.3	Properties of Actor Programs	78
7.4	Example: Riding the Elevators	81
7.4.1	A Discussion of Actors	84
7.4.2	Drawbacks	87
7.5	Chapter Summary	87
8	Capabilities of Specification Methods: An Evaluation	89
8.1	The Evaluation	89
8.1.1	Expressiveness: Data and Process Structures	90
8.1.2	Expressiveness: Specifying Constraints	90
8.1.3	Expressiveness: Specifying and Recording Transitions	92
8.1.4	Expressiveness: Concurrency and Communication Structures	92
8.1.5	Modularity and Compositionality	93

8.1.6	Verifiability: Reasoning with Time	94
8.1.7	Verifiability: Specifying and Verifying Properties	95
8.1.8	Semantics of Specification Formalisms	96
8.1.9	A Discussion of the Evaluation	97
8.2	Chapter Summary	99
9	A Proposed Framework for Design Representation Languages	101
9.1	Introduction	101
9.2	Specification Formalisms	101
9.2.1	Tools Relating Requirements to Designs	102
9.2.2	Design Specification Language Notation Considerations	103
9.2.3	Tools for Verifying Specifications	107
9.2.4	Tools Relating Designs to Implementations	109
9.2.5	Tool Support for Formal Reasoning	110
9.3	Research Directions	112
9.4	Summary	114
	Bibliography	115
A	A Formal View of Specifications in IncTr	121
A.1	Properties of Specifications	122
A.1.1	Consistency	122
A.1.2	Soundness and Completeness	122
A.1.3	Complete Specifications	123
A.1.4	Tightness	123
A.1.5	Extension	123
A.1.6	Systems of Specifications	124
A.1.7	Parameterization	124

List of Figures

4.1	Structure of Lotos specifications and process definitions.	25
4.2	A template of a Lotos data type specification.	26
4.3	A top-level specification of the elevator system.	29
4.4	Data type specifications for the elevator number.	30
4.5	Process specification of the button object.	31
4.6	Process specification of the elevator object.	32
5.1	The outline of specification of the elevator system in UNITY.	47
6.1	Hypergraphs.	54
6.2	A simple Statechart.	55
6.3	(a) History arrow, if entering for the first time, it enters C. (b) History connective applicable to the lowest level; in this case E.	57
6.4	(a) Implementing a condition represented by the predicate p. (b) Imple- menting a condition using a conditional connective.	57
6.5	(a) Implementing a selection among three events. (b) Implementing a selection among three events using a selection connective and a history input.	58
6.6	A high-level representation of the Elevator System using Statecharts. . .	62
6.7	The elevator subsystem.	63
6.8	State changes in the up state of an elevator i.	64
6.9	State changes in the Up_Button at floor i.	65
7.1	A template of an actor definition for an elevator behavior.	82
7.2	An actor definition for the main actor: <i>create-elevators</i>	84

7.3	A template of an actor definition for a button.	85
9.1	Essential features of a formal design specification language and their relationships.	105
9.2	Module interface definitions.	105
9.3	Module definitions.	106
9.4	Constraints, assertions and exceptions.	108

List of Tables

6.1	Some common events and conditions, and corresponding action sequences for statecharts.	60
8.1	Specification methods and associated representation structures: Data and process structures.	91
8.2	Specification methods and associated representation structures: Specifying constraints.	91
8.3	Specification methods and associated representation structures: Representing state transitions.	92
8.4	Specification methods and associated representation structures: Representing concurrent structures.	93
8.5	Specification methods and associated representation structures: Representing modules.	95
8.6	Specification methods and associated representation structures: Reasoning with time.	95
8.7	Specification methods and associated verification support facilities: Specifying and verifying properties.	96
8.8	Specification methods and associated verification support facilities: Semantics of specifications.	97

Chapter 1

Introduction

In this thesis we present a characterization of formalisms currently under study for specifying designs of distributed systems. We devise criteria for analyzing these formalisms and the associated specification languages. Later, we present an imperative framework that defines guidelines for the development of specification languages for distributed software systems.

The aim of this thesis is to compare different formalisms with the view of integrating the useful and compatible features of these formalisms in a comprehensive framework for specification and verification of distributed systems. As far as we know, there is no research of a similar nature or proportions to date. In this thesis we compare four different formalisms for specifying distributed systems: *algebraic* methods, *logic-based* methods, *graph-based* methods and *object-based* methods.

This thesis studies these methods based on an incremental approach to software development. We use the *Incremental Transformation* (referred to as **IncTr** in future) model [Lehmann et al., 1984] as the basis. The reason for choosing this model is that it defines the properties of a specification succinctly. We discuss this model formally in Section 2.3.1. In Chapter 9 we also discuss the semantics associated with the transformations of specification in terms of the tool support required.

In this chapter, we present other studies that compare different specification formalisms. After defining a few key terms we describe the scope of this thesis. We present arguments in favor of formal approaches to software development, and discuss some issues in the specification and development of distributed systems.

1.1 Related Work

In the last few years there has been a distinctive shift in the software engineering methodology. Formal methods are being emphasized. In addition, a few programming paradigms have become popular. In such a situation, there has been no recent work which summarizes the new ideas and puts them in perspective with respect to each other.

[Yau and Tsai, 1986] summarizes a number of traditional design methods. In this paper the authors survey architectural and detail design techniques. They also give a short description of designing distributed software based on petri nets. [Kelly, 1987] compares four methods for design of real-time systems. This short analysis concentrates on two object-based methods and two traditional methods of detail design. Some of the comparison criteria are: *foundations, semantic soundness, design verification and validation methods, maintainability, concurrent processing and communications, performance design, history of use, reusability, training needs, degree of automatic support, degree of support for other phases of the lifecycle, language considerations and graphical notation*. The paper defines a framework for comparing design methods, provides a method for choosing a design method and compares the features of four methods for real-time systems design.

Most other work has focussed on combining complimentary characteristics of two different methods. For example, object-based methods have been augmented with temporal logic [Diaz-Gonzalez and Urban, 1988], abstract data types [Bergstra and Klop, 1984], graphical notations [Bear et al., 1990; Ackroyd and Daum, 1991], functional refinement [Jalote, 1989], and horn clauses type logic programming [Vermeir, 1986].

A number of papers describe work that combines petri nets (augmented in some form or the other) with process theoretic methods [Best, 1984; Olderog, 1987; Shields, 1987; Gorrieri and Montanari, 1990; Boudol, 1990; Baldassari et al., 1988]. There is little work that compares/contrasts these two approaches. In fact these papers indirectly compare the methods they combine by introducing those features that are missing in one method from the second method.

One recent paper [Boudol and Larsen, 1990] attempts to find the relation between

the graph models of specification and labelled transition systems. This paper applies three criteria in comparing the two methods: *expressivity*, *modularity* and *refinement*. The paper then describes a system of transformations that allow a characterization of representations in labelled transition systems as graphical specifications and vice-versa.

1.2 Definitions

In this section, we define and discuss the usage of three important terms that are used in this thesis.

Definition 1.1 (Distributed System) *A distributed system is a collection of non-centralized, interacting computational elements.*

These interacting elements are generally connected in a computer network. This definition excludes all centralized parallel systems, but not the distributed multiprocessor variety. Examples of distributed systems include *reactive systems*, *networks of processes*, *concurrent, cooperating systems*, and *communicating systems*.

Definition 1.2 (Concurrency) *Concurrency is the ability to execute more than one task at the same time.*

Concurrency is an abstract concept. Concurrency at one level of detail need not imply concurrency at a different level. For example, we may have a number of concurrent processes, but the operating system running on a sequential machine may propagate such a view of the application. We take the view that a concurrent system has *apparent* concurrent activity at the highest level.

Definition 1.3 (Parallelism) *Parallelism is a restricted form of concurrency, wherein the concurrent activity is actual rather than apparent.*

We shall use “specification” to mean design specification in this thesis. We will use “requirements” for requirements specifications.

1.2.1 Scope of this Research: Design Specifications

In this thesis we focus on the design specification and its interaction with the preceding and succeeding phases assuming the following phases of the software development process:

- Requirements specification,
- Design specification, and
- Development and Verification

The scope of the current thesis does not cover other activities like testing, maintenance, etc. The problem definition activity is assumed to be a preliminary part of the conceptual modeling activity. Requirements specification develops a conceptual model of the software system.

Design specifications must bridge the gap between requirements specifications and implementations. Requirements specifications describe a system as a set of *what is needed* facilities embellished with constraints to specify any exceptions. Design specifications must narrow this definition of a system so that the requirements of a software system may be realized on a given group of resources.

Some researchers hold the view that the equation $Design = Requirements + More Constraints$ is too simplistic. For example, data definitions in requirements are abstract. However, design specifications cannot simply add to the set of constraints, but in most cases have to resort to better representation strategies to define data. In the case of distributed systems, the design phase should also be able to add information on partitioning the software system for allocation to the distributed resources. Usually this information is embedded in the description of software structures that allow resource sharing.

1.3 Formal Methods: Motivation

In this section we present arguments in favor of a formal method for specification and design of distributed software systems.

- Formal methods introduce precision in the specification of software due the mathematical nature of specifications.
- Formal methods provide a rigorous, mathematical basis for verification of properties of software [Ghezzi et al., 1991].
- Most structured techniques of software development begin with a model of the system under development. Unfortunately, the large number of possible models that exist make choosing a good model difficult. The choice is largely made on the experience and may not always be the best suited for that application.

A formal model is a collection of *concepts*, *axioms*, and *proof rules*. The proof rules generate new predicates from the axioms and existing proof rules. The choice of a model is based on the theory represented by the axioms and proof rules. The models at higher levels of description evolve from the basic model.

- The uniformity in the theory of software development allows one to define refinements as extensions to the model. Thus transformations between different representations of the same model are feasible.¹
- Since the definitions of the components of software are mathematical in nature, they are abstract enough to encourage reuse. Reuse can occur for both designs and code. This results in significant reductions in the costs of software development and maintenance.
- Lack of information about the global state is an issue in implementing distributed systems. In a formal approach, the ability to verify aspects of system behavior allows us to ignore most problems associated with lack of global state at the implementation stage.

1.4 Issues in Developing Distributed Systems

Three factors make distributed programming more difficult than sequential programming [Bal et al., 1989].

- The use of multiple processors.

¹This, of course, increases the complexity of the software tools to be used in the development processes. Nevertheless, the advantages of formal methods are numerous and thus justify the effort put in to develop appropriate languages and tools for transformational development.

- The need for cooperation among processors.
- The potential for partial failure.

Ideally, all three factors must be addressed in an environment for distributed programming. This entails providing **programming primitives** and **structures** for *parallelism*, for *communication* (as a means of cooperation among processors), and for handling *partial failures*. In addition, distributed software designers have the following issues also to contend with.

- maintaining a *dynamic process structure*, thereby providing a more efficient sharing of resources,
- keeping *communication costs* low,
- *allocation* and *synchronized* usage of *shared* and/or *duplicated* resources,
- mapping of *conceptual entities* to *physical units*,
- *decomposition* of a computational task for distributed processing,
- ability to *extend* the application on a (possibly enlarged) distributed system, and
- *availability* of the application versus availability of the systems it is running on.

The design process is compounded by the availability of a large number of methods of implementing distributed computation. A list of these with appropriate references to literature is given in [Bal et al., 1989].

1.4.1 Specification of Distributed Systems

Given the above issues in distributed programming, applying a formal method to the design of concurrent systems becomes a formidable task. This is complicated further due to the lack of a widely acceptable theory of concurrency. Representing the concurrent behavior of the components of a distributed application is difficult, as imposing a strict ordering on the occurrence of events would mean considering permutations of all possible sequences of events. But the advantages of achieving well-behaved and easily maintainable software outweigh the cost of developing the application in a formal manner.

The specification of a distributed system entails a formal description of the system as a composition of subsystems including their interfaces, and the (internal and external) behavior of these subsystems. Translating the requirements into such design specifications involves a careful definition of the components and their relationships. Therefore, the design representation should be abstract enough to capture the behavior of the application without emphasizing a particular approach to implementation. The choice of a model, and a method for distributed programming would be made at the later stages of design.

In addition to the specification of the behavior and the structure of a distributed system, we are also interested in identifying and establishing desirable properties² of the system. This is because these properties provide a computationally feasible mechanism to ensure that a specification behaves as it is supposed to. We provide a formal introduction to the various properties of a concurrent program in Chapter 2.

1.5 Structure of the Thesis

In the remainder of the thesis we will discuss a number of formal methods, classified into the four categories: *algebraic*, *logic-based*, *graph-based* and *object-based* methods.

A common example involving the specification of an elevator system will be used to illustrate and compare the different approaches these four major methods advocate. We present this example informally in Chapter 2. In this chapter, we also present issues in formal specification and development of concurrent programs. We also discuss the **IncTr** model formally. We discuss the properties of concurrent specifications under this framework. In Appendix A we present a formal view of specifications and their properties in the **IncTr** model.

In Chapter 3 we discuss the properties of a formal specification language. Based on these properties, we develop criteria for comparing the four groups of specification methods, which we do in the next four chapters.

Chapter 4 describes Lotos, a process algebra language, which combines the advantages of abstract data type theory with process algebras. Lotos is based on CCS (and to

²These properties are derived from the behavioral requirements definitions.

some extent on CSP). We then analyze specifications in Lotos and discuss its drawbacks.

Chapter 5 describes the UNITY approach to parallel program design. UNITY extends Floyd/Hoare method of program design to parallel programs. We use UNITY to provide an outline specification of the elevator system (introduced in Chapter 2) and analyze the method. This method is then compared with the use of temporal logic for specification of concurrent systems.

Chapter 6 describes statecharts — a method based on extended hypergraphs. We present STATEMATE — a tool that uses statecharts for the specification and development of reactive systems. After an analysis of this method, we compare this method with use of graph grammars for specification and analysis of distributed systems.

Object-oriented methods are described in Chapter 7. We briefly discuss the basic issues in object-based development and concurrency. Actors, a distributed computation model, is presented. We use actors to model the elevator system and highlight the features of a simple actor language. We then present an analysis of the actors method.

Chapter 8 presents an evaluation of the different groups of specification formalisms. This evaluation is based on the criteria defined in Chapter 3. We also present a brief discussion on the evaluation.

Chapter 9 discusses issues raised in the earlier chapters and proposes a framework for developing design specification languages for distributed systems. We also discuss the tool support required for a formal development environment during the transformation process of the software development. Chapter 9 also lists some possible areas for future research and development of a formal, interoperable environment for specification and development of distributed software.

Chapter 2

Issues in Formal Specifications

In this chapter we describe an example of a reactive system. We choose a reactive system as the example as it represents a large class of distributed systems. We then relate various issues for developing distributed software, as identified in Section 1.1, to the problem of specifying them. We then formalize these issues using the **IncTr** model for software. We use the example where ever necessary to illustrate the properties. We then discuss other issues related to semantics of concurrency.

2.1 Riding the Elevators: An Example of a Reactive System of Processes

In this section, we present an elevator system and methods for specifying it. We also discuss the type of verification arguments that we will present when specifying this system using the four different methods we are studying.

2.1.1 Problem Specification

An elevator system is composed of n elevators moving between m floors. Each elevator has a set of buttons, one for each floor. An elevator visits a floor when the corresponding button has been pressed. Each floor has two buttons, one for each direction. When a button is depressed, an elevator is called to the floor. Pressing a button illuminates it. An elevator remains at a floor if it has no request to service. The elevator system will meet the following requirements.

- Eventually all floors must be serviced and must be given equal priority.
- The requests within an elevator are serviced sequentially in the direction of travel. All such requests must be serviced eventually.
- When an elevator visits a floor, the illumination on the button(s) will be cancelled.

The system must be described with respect to the requirements placed on the behavior of the system¹. There are two main objects: *Elevators* and *Floors*. With this example, we attempt to demonstrate the ability of a method to represent and reason with the specifications.

2.1.2 Possible Solutions

This is a classical example of an incomplete specification of the problem. This system can be specified in three different ways.

1. The elevator system has a centralized control and the elevators get information about the requests they service via this coordinator.
2. The control is distributed among all elevators, which service requests as and when they arrive. The elevators race against each other to satisfy passenger requests.
3. The control is distributed, the elevators cooperate with each other in servicing the passenger requests.

We attempt to specify the system where the elevators cooperate while providing the service in a distributed control environment.

2.2 Formal Issues in Specification

In this section we elaborate on the issues presented in Section 1.4. We reiterate that a notation for specifying distributed applications must have a high degree of expressiveness and generality. To establish properties of a specification, we need a logic and its

¹This example has been adopted from its initial publication in a number of papers in the Fourth International Workshop of Software Specification and Design, 1987.

associated proof system which acts in conjunction with the representational structures. From the discussion in Section 1.4, we conclude that specification structures should

- be based on a sound semantic theory,
- be able to model a distributed application so that the resulting specification will be easily transformable into implementation structures,
- be able to represent a distributed application in terms of abstract entities so that further reuse of these descriptions for new entities is made easier,
- represent explicit concurrency², and
- contain assertional statements to
 - allow reasoning about the scope and significance of specifications, and
 - allow establishing desirable properties of specifications.

We use the **IncTr** model as a basis and formalize these conclusions in the next section.

2.3 Mathematical Models and Theories: Their Relation to Specifications

It is well known from formal language theory that a language is a collection of strings of symbols. These symbols belong to a set called the *alphabet*. *Syntax* of such a language is the set of rules that specify which strings belong to the language.

A *formal system* is a set of statements which are valid in some language [Björner and Jones, 1982]. For example, a correct program (in any language) is a formal system. *Semantics* define the meaning of a formal system. In the terminology of first order logic, the syntax is the set of well-formed formulas, and the semantics are the interpretations of these formulae. A few other approaches to formal specification are briefly described in [Lucas, 1982].

²Concurrency is a design issue, and therefore is represented during specification of designs, rather than specification of requirements.

A *formal system* can be defined to be a pair $\langle L, C_n \rangle$, where L is the language, and C_n is the *consequence closure* operator defining the deductive rules for the formal system. This operator varies for different formal systems.

Thus, a requirements document, which is the basis for specifications, is a *theory*. A specification is one *model* which satisfies this theory. A specification consists of several statements, called *theorems*, each describing a behavior of the model. At a different level, the implementation is again a model, whereas the specification, which is a proposal for implementation, is the theory which the implementation satisfies. We present the **IncTr** model of software design below. We then discuss the properties of specifications. We borrow the formal notation from [Lehmann et al., 1984].

2.3.1 Software Design by Incremental Transformations: IncTr Model

In the **IncTr** model, the design process starts with the requirements which is refined in a number of steps. The design process has an initial linguistic system (requirements specification language) and a target linguistic system (implementation language). The aim of the design process is to achieve a representation in the target linguistic system by a series of refinements³ applied to a specification in the initial linguistic system. A design specification language provides structures to represent all requirements information, and additional structures for information that will help in the transformation to an implementation language.

In the **IncTr** model, a software system is considered to be a set of formulas. These formulas are written in a formal system $\langle L, C_n \rangle$. The initial specification is a model of some abstract theory (requirements), the subsequent specifications are derived (refined) models of the initial specification. A specification at a level k is a theory for a specification at level $k + 1$. The process of software development is analogous to the process of deriving a minimal model in a logic system.

In the design process, refinements to a model are done by adding new information. This process of refinement is not necessarily monotonic. We may need to backtrack and redo the refinement if the generated model is not satisfactory. The representations at

³We use “refinement” and “refication” to mean the same thing. We use “transformation” to mean a series of refinements resulting in a change in the linguistic system.

different levels of refinement can be considered as different levels of abstractions.

The most important property of a specification is its *consistency* (with respect to a previous specification). Another important property is *completeness*. Completeness is not as crucial a property as consistency. Consistency requirement of a specification can not be relaxed, but completeness requirement is, in most cases, relaxed. Incompleteness allows the designer to ignore some concerns that are not important.

The initial (requirements) specification is assumed to be consistent. Various properties that define verifiable characteristics of the system at that representation are assumed to be known. Such a representation is said to be *implementable*, i.e., there exists a linguistic system in which a valid model can be constructed for this representation. Given the initial and target linguistic systems and the initial representation, the design process involves defining a set of verifiable criteria of representation correctness in an intermediate linguistic system. These criteria determine if a transformation is *correct* or not. By proving that these criteria are met, we satisfy a *proof obligation* in the design step.

In the process of refinement, transforming the specification in a formal system at level i to a representation in the next formal system at level $i + 1$ can be done in two ways. Everything derivable from the rules C_{n_i} is established as a consequence of the rules $C_{n_{i+1}}$. This is termed as *general reification*. The designer may choose to concentrate only on certain parts of the representation for refinement, or finds that some parts of the formal system are not useful in the current framework. This type of refinement is termed as *selective reification*⁴.

The refinement process normally involves changing the deductive rules or adding new deductive rules to the formal system. Such a change must be consistent before the formal system can be used to generate a representation. In Appendix A, we present a more formal view of specifications in the **IncTr** model, and define properties of specifications that form *proof obligations* in the design step.

⁴Selective reification allows the designer to ignore certain details, thus he loses global control over the design process. The selective reification process corresponds to the well-known technique of *separation of concerns* in structured programming.

2.4 Properties of Concurrent Programs

The properties of specifications described in Section 2.3.1 are important and desirable. Algorithms for computing these properties (e.g., completeness) are not only expensive, but hard to implement. Algorithms for some very simple problems in distributed computation (questions about deadlock and serializability, and communication scheduling and testing) have unknown complexities [Johnson, 1983; Johnson, 1984]. But fortunately, all properties of execution of concurrent programs can be expressed in terms of three classes of properties: *safety*, *liveness* and *fairness* [Schneider and Andrews, 1986]⁵. Of the three, *safety* and *liveness* are more important.

To get a better understanding of these properties, we define two temporal operators: *always* and *eventually*. These are defined as follows.

$\Box A$ — Eventually : From a point in time event A is true at some later point.

$\Diamond A$ — Always : From a point in time event A is always true.

\Box and \Diamond are duals, whose relationship is described by

$$\Diamond A \equiv \neg \Box \neg A .$$

Now we examine the safety and liveness properties a little closely below. We also define fairness informally. We shall see more examples of these properties in Chapter 5.

Safety

A safety property states that all finite prefixes of a (possibly infinite) computation sequence must satisfy some requirement. Such properties can be expressed by formulas of the form $\Diamond p$ for some predicate p . Disjunction and conjunction of safety properties are also safety properties.

Suppose the concurrent program is to implement some form of synchronization, say *mutual exclusion*, then the assertion “at most one process is in its critical section” is an example of a safety property. If some property P is true if a program is in its proper initial state (given by *Init*) then the safety property that must be proved is

$$\models \text{Init} \rightarrow \Diamond P$$

⁵Actually [Schneider and Andrews, 1986] mentions only *safety* and *liveness*, as fairness properties can be represented as safety and/or liveness properties.

which states that P must be true if $Init$ can be derived. The most important type of safety property one uses about concurrent programs is *invariance*. These properties have the form that P is a predicate that is invariant if $\models P \rightarrow \Box P$.

Intuitively, safety properties can be characterized as stating that *nothing bad will happen* (or something is always true). Once the property is violated, then the design of the distributed application will have to be changed so that in the execution of the new implementation, the safety property will hold. Being able to define such predicates and proving them to be true, is basic to verification using logic. Below, we give an example of a safety property in the elevator problem.

Example 2.1 Let a queue `floors` , denote the requests to be serviced by an elevator. Then a safety property *if the list is in waiting state, then floors is empty*, to be deduced is

$$\models (\text{elevator}_i = \text{waiting}) \rightarrow \neg \text{isempty}(\text{floors}_i)$$

Liveness

Liveness properties complement safety properties by requiring that certain predicate(s) hold at least once, infinitely many times, or continuously from a certain point in time. Alpern and Schneider [Alpern and Schneider, 1985] give formal syntactic and semantic definitions of liveness properties.

Basic liveness properties are those that are expressible by

$$\Box P, \Box \Diamond P \text{ or } \Diamond \Box P$$

where P is a predicate which describes an assertion over finite prefixes of sequences. The first formula states that P will be true at some point of time in the future. The second formula states that P will be true many times after some point of time in the future. The third formula states that eventually at some point of time P will be true at least once. Another common form of liveness property is $P \leadsto Q$, where P and Q are predicates. If we think of P as a predicate which asserts that a process is waiting

for service, and Q is a process receiving service, then we have stated a liveness property that no process starves.

Intuitively, a liveness property stipulates that *(eventually) something good will happen* (or something will eventually be true). A specification describing *liveness* properties asserts that a state function **must** change. A liveness property may become true any-time before the end of the program.

Example 2.2 The statement – *All floors will eventually be serviced* is a liveness property. This can be defined as a composition of the safety property in Example 2.1. If P_i is the safety property associated with each elevator i in Example 2.1, then the liveness property is represented as a composition

$$\models \forall i \Box P_i$$

Fairness

In nondeterministic computation the choice of a computation sequence is made from sets of alternate sequences. The problem of fairness is to ensure that a choice is made such that none of these choices are postponed forever. For example, in a time-sharing operating system, the choice of which process should be allotted the next processor time-slice is a fairness problem. Another family of fairness problems concern avoiding starvation of processes. For example, when using semaphores for synchronization in a correct manner, no process will wait forever while other processes advance.

All these fairness problems arise from a more basic problem; that of termination of a certain event or state. In such a form, fairness properties tend to be expressed as liveness properties. For example, the fairness property – *a program will always produce the correct answer*, can be represented using one safety and one liveness property. The safety property is that *the program never terminates with the wrong answer*, and the liveness property is *the program eventually terminates*.

Example 2.3 The statement –*all floors must be serviced with equal priority* is a fairness statement. This can be expressed as a composition of a safety and a liveness property.

The safety property is *if a two floors are not serviced, they will be serviced in the direction of travel*. The liveness property is *all floors will eventually be serviced*.

The formulation of the safety property is given below.

$$\models \forall i \forall j, k (\text{floor}_{ij} > \text{floor}_{ik}) \wedge (\text{direction}_i = \text{up}) \rightarrow \Diamond \text{service}_i(\text{floor}_{ik})$$

Let P be the safety property given above. Let Q be the liveness property defined in Example 2.2. The fairness property is represented as

$$\models P \wedge Q$$

A comprehensive account of the fairness issues in concurrent software development are presented in [Francez, 1986].

2.5 Chapter Summary

In this chapter we have addressed various issues in the use of formal methods for software development. We described a formal model **IncTr** for software development. Some of the important issues in software development using this model are:

- getting a requirements specification,
- identifying the initial and target formal systems,
- generating a set of criteria to verify correctness of transformations,
- defining properties that have to be satisfied by the specification, etc.

We also defined various properties of specifications of concurrent systems in this model. These properties allow a specification to be verified with respect to a previous representation.

Chapter 3

Criteria for Evaluating Specification Methods

In this chapter we develop criteria that will be used in the ensuing chapters for comparing specification methods. These criteria will be developed with the objective of determining an ideal framework for specifying distributed systems. Before presenting these criteria, we examine the characteristics desirable in a specification formalism for distributed systems.

3.1 Characteristics of a Specification Formalism

In addition to the ability to describe the distributed software structures (Section 1.4), a specification formalism must have four main characteristics listed below.

- *Expressiveness:* Expressive power of a specification language involves both the structural (syntactic) facilities and the reasoning capabilities.
- *Visibility:* The specifications should have no implicit information. Though partial specifications can exist during the design of software, they must also be visible, and explicitly state all constraints, data, and processes. No assumptions must be made about incomplete specifications, except that parts of the specification do not exist as yet.
- *Compositionality:* Defining systems in terms of compositions of subsystems (or modules) is important to comprehend a large specification.

- *Verification*: Availability of a set of proof rules to deduce desirable properties in an application generates confidence in the development process.

These characteristics translate to two basic components of a specification formalism: *notation* and *logic*. The notational component of a specification formalism can be represented as the following facilities.

- Facilities for data and process abstraction.
- Facilities to specify traces, which are event histories.
- Facilities to encapsulate data and related processes: *modularity*.
- Facilities to specify interfaces between modules.
- Facilities to specify nondeterminism and concurrency (communication and synchronization primitives).
- Facilities to specify assertions. Assertions are used to specify constraints on behavior and properties of the distributed system.
- Facilities to evolve a specification, successively refining it to add sufficient detail to allow an easy transformation to an implementation.

The logical component of the specification formalism will provide the specification system with a mechanism of verifying properties of the specifications. If verification mechanisms are to be provided in a consistent manner, the specification language must be based on a semantic theory. Usually, the verification mechanisms are embedded in a proof system [Goguen, 1981] which must have the following facilities.

- Ability to record recorded traces (event histories), and reason about them [Bartussek and Parnas, 1978].
- Ability to reason with partial specifications.
- Ability to deduce desirable properties of an application, for example, statements about performance goals, and behavioral constraints imposed on the system¹.

¹Though one would like to have a formal system generate such desirable properties, as of today, it still is up to the designer to specify these properties.

Such a proof system will consist of tools to check various behavioral aspects of a specification. For example, the proof system should be able to deduce that the behavior of a set of modules is consistent with the design constraints imposed on the modules. Also, the specification of the overall behavior of the module must be consistent with the combined specifications of the behavior of its submodules. Examples of behavior that can be checked are avoidance of deadlock and verifying mutual exclusion of two processes. The advantages of having a proof system for verifying such characteristics of distributed software is an important motivation for formal methods [Goguen, 1981].

A specification formalism should be able to represent and reason about an application independent of the type of the underlying hardware structure, and add implementation specific detail only after verifying the consistency of the specifications and after deciding on the underlying hardware.

3.2 Criteria for Comparison

We develop criteria for comparing specification methods based on the properties discussed in the previous section. We separate the notational and logical components of the criteria as well. The criteria for comparing notations are listed below.

1. Expressiveness

- Data and process abstraction: Abstraction of data is best provided by abstract data types. The mechanism of abstraction could be objects, processes, clauses or graphs.
- Structures for defining Constraints on data and processes: Like set expressions and logic, procedures, graphs. Data reification is possible only through assertional predicates on data and process objects.
- Structures for specifying transitions: Like state transition descriptions using traces, graphs or graph grammars. Reasoning about the behavior specifications can only be done with specification of state transitions.
- Constructs to represent concurrency and communication structures. Distributed systems are nondeterministic and this is caused by the concurrency

and communication aspects. Mechanisms to specify these constructs help reason about concurrent components in the distributed system.

2. Compositionality and Modularity: Large software development needs more structuring facilities than is normally provided by simple data encapsulation mechanisms supported by rules defining correct compositions.

The necessary support at the proof system involves a number of capabilities. The main mechanism needed is the ability to satisfy specified proof obligations. This can be expressed as a logical system to represent and reason² about

- properties of programs, like preconditions, postconditions, invariants and constraints on state transitions and processes in objects, and
- time and time intervals.

In addition, a verification system must have proper **semantics** associated with it. In short, the following list summarizes the facilities needed as a part of the proof system for verification of designs, and subsequent implementations.

- A proof system component to check consistency of specifications, for soundness and completeness.
- A proof system component to verify assertions in a specification. This gives the ability to prove partial correctness and a number of safety and liveness assertions.
- A proof system component to reason about time, and validity of processes in a given state. This gives the ability to verify assertions about the state of a system.
- A facility to record transitions as defined by the specification. This aids in proving that an implementation meets a specification if the transitions recorded as a part of the specification are observationally equivalent to those recorded as a part of the implementation.

²The ability to represent the designs as graphs will go a long way in verification and reuse. Graphs can be processed in an efficient manner, and have the ability to represent abstraction better than procedural representations. Verification mechanisms for graphs are simpler as a result. Not many specification formalisms have this facility.

3.2.1 Riding the Elevators: Comparison Criteria

With this example, we expect to demonstrate that the methods we analyze in this thesis have or do not have the following features.

- Expressivity.
- Compositionality.
- Verifiability.

For a method it may be difficult to determine if a verification tool is easy to build. We will indicate the availability of a logical framework for building such a tool.

3.3 Chapter Summary

In this chapter we have defined a set of criteria that is useful in evaluating representational and logical reasoning aspects of specification languages for distributed systems design. We believe that these criteria form the basis for a framework for designing specification languages for distributed software design, and we will propose such a framework in Chapter 9.

Chapter 4

Algebraic Methods

Algebraic models for specification focus on the specification and verification of various processes in a software system and, as a consequence, these process models are appropriate candidates for specification and verification of concurrent (and distributed) software systems.

In this chapter we describe Lotos, a language based on the process models of calculus of communicating systems (CCS) [Milner, 1980; Milner, 1983] and communicating sequential processes (CSP) [Hoare, 1985]. Lotos also incorporates representation structures for abstract data types [Goguen et al., 1977; Guttag and Horning, 1978]. We then use Lotos to model the elevators problem. The aim of such an exercise is to highlight the features of Lotos.

4.1 Lotos: A CCS-based Formalism with Abstract Data Types

Lotos (Language of Temporal Ordering Specification) is one of the three¹ formal description techniques developed within ISO [ISO, 1987] framework for formal specification and verification of ISO standards and open distributed systems. Lotos specifies concurrent systems by defining the temporal relations among interactions that form the externally observable behavior of a system.

The process behaviors and interactions are modeled using concepts from CCS, and

¹Estelle and SDL are the other two.

parallelism and concurrency are modeled using notions from CSP. Another aspect of Lotos deals with description of data structures, and expressions. This component is based on a formal theory of abstract data types: the *initial algebra* approach [Goguen et al., 1977].

In this section, we first give an overview of processes in Lotos. We then present templates that describe process definitions and specifications in Lotos. We briefly present structures for data type definitions. We then describe Lotos constructs that define operations on processes. These operations allow descriptions of process behavior.

4.1.1 Processes in Lotos

A distributed system is seen as a single *process*, consisting of several subprocesses. A Lotos specification consists of a hierarchical ordering of the definitions of these processes. A process performs two types of actions, those that are internal to the process, and those that involve interaction with its *environment*. The environment of a process P, in a system S, is a collection of processes of S that interact with P. Such interactions are based on synchronization, where each interaction is an *event*. Events form the central notion of activity in Lotos descriptions. Every event is an observable action, and is atomic; i.e., all processes involved in an event have the same view of the event. Either an event has taken place for all the processes involved, or the event cannot have taken place at all. The process definition of P specifies behavior, by defining sequences of observable actions. Each process has associated ports called *gates*.

Two interacting processes perform observable actions with respect to each other. This complementary notion is similar to the notion of *partner ports* in CCS [Milner, 1989]. If the two processes are coupled only at certain gates, then they must synchronize at those gates only. A process representing two interacting processes has the ability to hide the interacting gates from the external view. This provides for localizing control and hence easier verification of behavior.

4.1.2 Process Definitions and Specifications

A process definition defines the name of a process, the gates it uses, its component processes and the constraints on the behavior of the component processes. A process

specification defines a set of processes that interact with each other, the gates used, the type definitions that represent the interface data structures, the behavior and constraints on processes and the process definitions. Typical structures of a Lotos specification and a process definition is given in Figure 4.1.

```

specification  $\langle \text{spec-id} \rangle [ \langle \text{gate-id-list} \rangle ] ( \text{parameter-list} ) : \text{functionality}$ 
     $\text{type-definitions}$ 
    behavior
     $\text{behavior-expression}$ 
    where
     $\text{type-definitions}$ 
     $\text{process-definitions}$ 
endspec

process  $\langle \text{proc-id} \rangle [ \langle \text{gate-id-list} \rangle ] ( \text{parameter-list} ) : \text{functionality} :=$ 
     $\text{behavior-expression}$ 
    where
     $\text{type-definitions}$ 
     $\text{process-definitions}$ 
endproc

```

Figure 4.1: Structure of Lotos specifications and process definitions.

A Lotos specification is enriched by the algebraic methods of abstract data type specification, parameterization of data types, user-defined data types, type renaming and the notion of equality². A Lotos data type specification is described in Figure 4.2. This is one example of a data type specification. There are other possible templates, an example of which will be presented when discussing the elevator problem.

4.1.3 Combinators and Behavior Expressions

A process definition specifies process behaviors and constraints on the behavior by means of *behavior expressions*. A behavior expression defines an ordering on events. Behavior expressions are formed by applying the following combinators to Lotos events and processes. Most of these combinators were adapted from CCS. Some extensions were borrowed from CSP.

²These concepts are based on heterogeneous algebras [Goguen et al., 1977].

```

type { ADTName } is
  with sorts
    { Sorts }
  opns
    { Functions }
  eqns
    forall
      { Variables }
    ofsort
      { Equations }
endtype

```

Figure 4.2: A template of a Lotos data type specification.

Inaction: A completely inactive process is represented by **stop**.

Action Prefix: The action prefix combinator $:$ produces another behavior expression from an existing one by prefixing it with an event. This is similar to the action prefix combinator \cdot of CCS. This combinator implies sequentiality. For example, $a : \mathbf{stop}$ describes the behavior of a process participating in an event a . If event a occurs, then the process exhibits the behavior **stop**.

Choice: The choice combinator $[]$ simulates the nondeterministic choice operator $+$ of CCS. If B_1 and B_2 are two behavior expressions, then $B_1 [] B_2$ denotes a nondeterministic process that behaves like either B_1 or B_2 .

Parallel Composition: Lotos provides a wider scope of parallel composition than CCS. Lotos borrows these extensions from CSP. There are three variants.

- *General Case:* $B_1 [g_1, \dots, g_n] B_2$. Let $S = [g_1, \dots, g_n]$, be a set of user-defined *synchronization* gates and B_1 and B_2 be two processes. Then the expression $B_1 [S] B_2$ defines a process that is able to perform any action that either B_1 or B_2 can perform at some gate g_k , where $g_k \notin S$. Such a process can also perform any action that B_1 and B_2 can perform at some gate g_i in S . If two processes P_1 and P_2 have a common gate, then the processes interact, otherwise the processes perform unobservable actions.

- *Interleaving:* $B_1 ||| B_2$. When the set of synchronization gates, S , is empty, then either of the two processes perform, depending on which is ready to do so, or both interleave actions, as there is no dependency between these two processes.
- *Complete Synchronization:* $B_1 || B_2$. When the set of synchronization gates, S , is the set of all gates, then processes interact synchronously.

Hiding: This method allows some processes to be defined as unobservable. This has the advantage of abstraction, similar to the concept of function localization to related data in object-oriented programming languages. The expression

$$\mathbf{hide} \langle gate_X \rangle \text{ in } behavior_A [gate_list] \mid [gate_X] \mid behavior_B [gate_list]$$

makes the synchronization between the two processes *behavior A* and *behavior B* at *gate_X* unobservable. This facility is used to exclude interference from the environment.

Process Instantiation: A process is instantiated by associating a process name with the list of *actual* gates. These instantiations occur in behavior expressions of other processes definitions, or in the same process definition (in which case, it is recursion). Semantics of gates and relabelling occur as in most programming languages. For example, in

```
process buffer[in, out] :=
  in; out; buffer [in, out]
endproc
```

buffer process is defined as a continuous process having actions *in; out; ...*. The *buffer* process is instantiated within itself to define an infinite sequence.

Termination: A successful termination is denoted by a process **exit**.

Sequential Composition: Sequential composition of two behavior expressions, B_1 and B_2 is denoted by $B_1 \gg B_2$. Informally, this implies that when B_1 terminates successfully, execution of B_2 is enabled.

Disabling: A process can disable another process. In the notation $B_1 \{> B_2$, B_2 disables B_1 . This provides control in the specification for application generated interrupt handling.

4.1.4 Verification

Verification of Lotos specifications is based on the CCS notions of *observational equivalence* and *bisimulation*. The idea of *observational equivalence* is that the behavior of a process is determined by the manner in which it interacts with the external observers (environment), and two processes are considered as equivalent whenever we cannot distinguish between them by *external observations*. This concept also forms a basis for the semantics of a representation in Lotos (as in CCS). Given a behavior expression, the specification can be represented as a tree structure. Instead of the tree representing the semantics of the behavior expression, we form equivalence classes of trees, and conclude that two expressions are equivalent if the corresponding tree representations fall in the same equivalence class.

Bisimulation is a property of two behavior expressions that determines the equivalence of their observable sequences. There are tools that help establish this property. One such tool described in [Fernandez and Mournier, 1991] uses the method of forming equivalence classes of tree representations.

A number of implementations and software tool sets have been developed for Lotos. These include graph-based Lotos editors, data type verifiers, process and data type specification templates, libraries of abstract data types, and specifications analyzers and simulators. A few of these tools are discussed in [Turner, 1988].

4.2 Example: Riding the Elevators

We present a design for the elevator example as a set of decompositions from a top-level description. The elevator system has two main groups of processes. The processes at each floor and the processes in each elevator. Since these processes have similar behavior expressions, we shall denote the process description by a subscript. The top-level specification of the elevator system is given in Figure 4.3.

There are two types of data variables: *integer* and *queue* of integers. In fact, the data types are natural numbers. In Lotos, the data type specifications are defined as in Figure 4.4. In this specification, **Queue** is a parameterized data type and is used to define specific types of queues.

```

specification { Elevator-system [N,M]: noexit
  type NaturalNumbers is
    ...
  endtype
  type NaturalNumberQueue is
    ...
  endtype
  behavior
    process floor-button [I]          // The button behavior
      ...
    endproc
    process elevator [J]             // The elevator behavior
      ...
    endproc
  where
    ...
endspec

```

Figure 4.3: A top-level specification of the elevator system.

We now give the basic outline of the two main processes. The design is based on a broadcast of messages among the buttons on floors and elevators. It is interesting to note that in the specification of the elevator system, we have specified the system to be a continuously operating one. This can be specified in a different manner by augmenting the **noexit** statement with exception conditions.

The button process will have boolean variables indicating whether the floor has been serviced or not and whether the light is illuminated or not. We have not specified a number of other processes – for example, the process components where the status is maintained, or the cancellation of the request after it is serviced. The button process is sketched in Figure 4.5. It is important to note that the specification describes N elevator processes and M button processes. Also, we have not described the button processes associated with each elevator. This extension would not cause any more features of Lotos to be used.

The elevator process addresses a number of issues of the system specification, including sending an off signal after a floor is serviced, adding a floorid to its service list, changing its status after it receives a service request and maintaining its status over a

```

type NaturalNumbers is
  sorts nat
  opns
    0      :  $\rightarrow$  nat
    succ   : nat  $\rightarrow$  nat
endtype

type Queue is
  formalsorts element
  formalopns e0 : element
  sorts queue
  opns
    create :  $\rightarrow$  queue
    add : element, queue  $\rightarrow$  queue
    first : queue  $\rightarrow$  element
  eqns
    forall x, y : element, z : queue
    ofsort element
      first (create) = e0 ;
      first (add(x, create)) = x;
      first (add(x, add(y, z))) = first (add(y, z)) ;
endtype

type NaturalNumberQueue is
  Queue actualizedby NaturalNumbers using
  sortnames nat for element
  opnnames 0 for e0
endtype

```

Figure 4.4: Data type specifications for the elevator number.

```

process button [l]:noexit := (l ?floorid:NaturalNumbers)
  ...
  |||
  on[floorid] and not entered[floorid] >> !floorid ;
                                     // If the floor has not been serviced, broadcast
                                     // floorid so it may be serviced
  ||| serviced[floorid] >> off[floorid]
                                     // if the floor has been serviced then switch the
                                     // button on that floor off

  // Note that both the above process descriptions have sequential components
  ...
  where
    data type definitions and other process definitions
endproc

```

Figure 4.5: Process specification of the button object.

period of activity. This process is defined in Figure 4.6.

We could have implemented a cost-based behavioral model of the process where each elevator decides whose cost is the least for a particular request and adds the request to its service list. The elevator also broadcasts a message indicating such an action. This modification is simple and can be accomplished with ease.

4.3 An Analysis of Lotos

• Notational Features

- *Unit of computation:* The basic unit of computation is a process.
- *Data abstraction:* Each process description totally encapsulates associated processes and data type definitions. Data abstraction is based on well founded principles of abstract data types and are not provided via procedural abstraction mechanisms.
- *Modularity and Compositionality:* Modularity is provided by specification definitions. These specification definitions encapsulate process descriptions and associated type descriptions. Compositional specifications are generated

```

process elevator[J,floorid]:noexit :=
  (J ?Eid:NaturalNumbers || floorid ?fid:NaturalNumbers, serviceQ:NaturalNumberQueue)
  ...
  |||                                     // all these processes interleave
  enter[fid]                             // the floor fid is serviced
  |||
  !serviced[fid] >> off[fid]             // Concurrently send an off signal to the corresponding button p
  |||
  fid and add(fid,serviceQ[Eid]) [> waiting[Eid]    // Add a service request
                                          // from a floor fid to the current elevator Eid and
                                          // interrupt the waiting process for that elevator
  |||
  ...
where
  data type definitions and other process definitions
endproc

```

Figure 4.6: Process specification of the elevator object.

by the use of parallel, sequential, choice, and disrupt combinators.

- *Constraints*: Constraints are defined by behavioral expressions which specify behavioral constraints on the processes. The rich set of combinators help specify constraints on processes by imposing an ordering on them.
- *Transitions*: Transitions are recorded as a sequence of instantiated process/behavioral expressions. The transitions are characterized by the use of choice, disrupt and action combinators.
- *Concurrent structures*: Lotos provides a rich set of concurrency specification structures. The communication structures are similar to the channel specifiers in CSP, and concurrency structures are similar to those provided in CCS.

• Semantic Features

- *Defining assertions*: Assertions in Lotos are a part of the specification of the behavior of processes. These assertions allow one to prove the equivalence of two (or more) specifications. This equivalence can be associated with refinements of specifications.
- *Reasoning with time*: The temporal ordering imposed on the behavior of

processes is not sufficient to specify a total ordering on the system. The reasoning mechanisms are only able to infer timing relations between the collections of partial orders.

- *Defining provable properties:* Properties of Lotos specifications embedded in the specification of processes and higher-level specifications. Properties can be asserted as true or false within each process. This allows proving properties at different levels of detail within the same specification.

4.3.1 Drawbacks of Lotos

The Lotos language is fairly complex and requires a lot of time to gain proficiency. The lack of adequate tools to support development in Lotos makes the task of evaluating such a language more difficult. The use of labelled event structures to define the semantics of Lotos makes it difficult to specify true concurrency, where the events are ordered using a collection of partial orders.

One of the major drawbacks is that all the notational features are integrated in one major specification notation. Building tools for this specification formalism that provide an extensive set of services would appear to be a difficult task. In spite of this a number tools have been built [Usenet, 1991].

4.4 Chapter Summary

We have described the Lotos specification language. The language combines the best of process algebras and abstract data type specifications. The language has a rich set of combinators (operators) and is well-suited for development of distributed systems.

We also have presented an example of a small reactive system in Lotos. We can conclude that Lotos is a fairly complete system for specifying distributed systems in a compositional and modular manner.

Labelled transition systems provide the operational semantics for Lotos. There are a few other algebraic methods which attempt to provide a different approach to specifying concurrent systems. [Kaplan, 1989] provides one approach to algebraic specification of concurrent systems. This approach attempts to define concurrency within the frame-

work of initial algebra semantics. The method tries to explain the labelled transition systems semantics using a category-theoretic approach.

[Stark, 1989] defines concurrent transition systems to represent the semantics of concurrency. Concurrent transition systems extend the labelled transition system semantics by using a concurrent alphabet. The resulting structures studied are the *trace automata*. The method provides procedures to define morphisms on the class of concurrent transition systems, thus forming a category. Properties of these categories are also defined.

Chapter 5

Logic-based Methods

5.1 Introduction

We will present an important logic-based method and its system of transformation and reasoning: UNITY. UNITY (short for **U**nbounded, **N**ondeterministic, **I**terative **T**ransformations) is a small language, with associated proof rules to derive invariant, correctness, and termination properties of parallel programs. This method tries to provide an axiomatic basis for parallel program design, in some ways analogous to the Floyd/Hoare axioms for sequential programs.

We first present the background to UNITY's development. The important concepts of UNITY are then highlighted followed by a description of the UNITY model of program transformations. After a brief section on the structure of UNITY programs, we present the rules to specify and derive properties of concurrent programs. We use UNITY to model the behavior of the elevator system. After summarizing the features of UNITY, we present the drawbacks of UNITY and compare it with the use of temporal logic for specification of concurrent programs.

5.2 The UNITY approach to Development of Parallel Program Specifications

The UNITY approach preserves the Hoare-style assertions, which were later popularized by Dijkstra as precondition, postcondition and invariant specifications for sequential

programs [Dijkstra, 1976]. UNITY has a small language, a computational model, and a logic for associated proof methods. The UNITY method also provides rules for refinement of specifications. In the preface of [Chandy and Misra, 1988] the authors explain the choices they made in developing a theory of parallel program development. Briefly the choices made are as follows.

- *Designs of programs rather than programs:* The authors believe that a design is more important than the code, and to work with designs one needs a model of computation, a notation to specify designs, and its associated proof rules. The emphasis is on correct program designs.
- *Foundation of programming instead of a taxonomical analysis of machines, methods, programming styles or applications:* The authors believe that program development is independent of the underlying architecture and mapping to an architecture should be done after a correct program is developed. They do not believe that the architecture should determine the method of program development.
- *Formal instead of informal notation to describe programs:* The belief is that a mathematical framework will bring discipline to the task of designing and reasoning about designs.
- *Reasoning about static aspects rather than the unfolding computations (called operational reasoning) of a program:* The authors reason that:
 - Operational arguments are more error-prone.
 - Operational arguments are not very convincing.
 - Operational arguments are longer.

The authors believe that the essence of programming is to make a series of stepwise refinements to the designs. In support of their approach the authors present examples from operating system problems, termination detection in asynchronous programs, garbage collection, conflict resolution in parallel systems, combinatorial problems, matrix problems, and communication protocols.

5.2.1 UNITY theory

The UNITY theory is based on the premise that program designs should be independent of architecture upon which they are to be implemented. If the problem specifies the architecture, the refinement process is continued until an efficient representation is obtained for the target architecture.

The UNITY theory addresses issues relating to *nondeterminism*, *control flow*, *synchrony and asynchrony*, *states* and *assignments*, *proof systems* supporting program development by *stepwise refinement of designs*, and *decoupling of correctness* (of programs) from *complexity* (due to architectures). We present UNITY's approach to these issues below.

Nondeterminism

Nondeterminism is caused by the lack of information necessary to make a choice between many options available at any point. For example, in a Horn clause logic program, several predicates can match a given goal. The choice of a predicate for execution is arbitrary, i.e., nondeterministic.

Abstracting program design to a very high level will cause a lack of detail, and may cause nondeterminism. UNITY supports this view of program designs. Nondeterminism also implies that program designs are simpler due to the absence of detail, and such designs in UNITY can be optimized by limiting the executions to the domain of a given architecture. UNITY provides rules to add information to a high-level solution so the final specification will be a program that can be executed efficiently on the target architecture.

Absence of Control Flow

Control flow is a characteristic of sequential computation, but not of concurrent computation. UNITY designs, during the initial stages, are devoid of any concern related to control flow. For example, two object classes communicate with each other using a well-defined interface. The interface definition is an acceptable concept in a program

design, but the control sequence defining the order of computations in the methods of the two interacting objects will only limit the generality of the design.

Synchrony and Asynchrony

Chandy and Misra intend UNITY to be useful for a variety of architectures. SIMD machines are characterized by the synchronous behavior of processing elements, whereas the processes in a distributed network are asynchronous. Both these characteristics are basic to concurrency, and hence are supported by UNITY.

States and Assignments

State-transition systems are important when modeling concurrent systems. To develop program designs one needs representational elements for these concepts. A simple element is the inclusion of assignments in the set of notational constructs and use variables to store representations.

But the assignment statement must not form the basis of representation and computation. This introduces the concept of control flow into a specification which should be avoided. This can be done if a non-imperative approach, as in functional languages, is used in defining the assignment. Such a definition of the assignment can be augmented by allowing *multiple assignments* in a single statement. This introduces simultaneity in processing assignment statements.

Proving properties from Program Designs

UNITY provides mechanisms to prove properties of program designs. Such a feature is helpful when proving that incrementally developed program designs are correctly derived refinements of specifications at the previous steps in the process of design.

Separation of Correctness and Complexity

UNITY separates correctness concerns from complexity issues. Correctness concerns are addressed as a part of the program design process. Complexity is an issue that comes up during mapping a design to a particular architecture. This is when one is concerned with the method of program execution.

5.2.2 The UNITY Model

A UNITY program consists of a declaration of variables, a specification of their initial values, and a set of multiple-assignment statements. A UNITY program executes forever after starting from a state satisfying the initial condition. Assignment statements are selected in a nondeterministic manner. The selection is a “fair” process and therefore every assignment statement is selected infinitely often.

Separating Programs and Implementations in UNITY

A UNITY program describes *what* should be done in the form of initial state and state transformations. It doesn’t specify *when*, *how*, and on *which processor* an assignment should be executed, and *when* and *how* a program execution may halt. These are concerns at the mapping stage.

The Process of Mapping Program Designs to Architectures

The architectures that will be discussed are von Neumann architecture, synchronous shared-memory multiprocessors, and asynchronous shared-memory multiprocessors.

A mapping to a von Neumann architecture specifies the schedule of assignments and the condition(s) for which the program execution terminates. The execution schedule can be represented as a finite sequential list of assignments in which each assignment in the program appears at least once (to satisfy the “fairness” condition). Another “fairness” concern, that of representing a terminating program, is to view the execution schedule as a finite prefix of the sequence of (infinite number of) assignments.

If the execution of any instruction of the program in a state S will cause no change in the state, then this state is called the *fixed point*. A predicate FP characterizes the *fixed point* and is a conjunction of the equations that result when the assignment operator $:=$ is replaced by an equality $=$ operator. The predicate FP is true only if the left and right sides of the equality are identical. Stopping at a fixed point implements termination in a UNITY program. A *stable prop.* ϕ of a program is one which always remains true, once it becomes true. Thus FP is a stable property.

In a synchronous shared-memory multiprocessor, a fixed number of identical processors share memory, and a clock. Every processor executes one instruction. A mapping is easily represented as the execution of each component of a multiple assignment statement.

In an asynchronous shared-memory multiprocessor, identical processors share memory, but not the clock. Simultaneous references to the same memory location results in accesses to memory in some nondeterministic manner. Mapping to such an architecture can be specified by partitioning the statements of the program amongst the processors. Such a partitioning needs to be “fair”. Such an architecture would not allow two instructions that modify the same memory location to execute concurrently. This represents a fair interleaving of the two instructions’ individual executions.

Modeling Programming Language Structures and Software Architectures

Let us take the example of a program fragment that imposes a sequential order on the execution. A statement of the form **await** B **do** S in an asynchronous shared-variable program is encoded in UNITY as: a statement that does not change the value of any variable if B is false, otherwise has the same effect as S . A corresponding representation can be designed for Petri nets, which are asynchronous in nature.

Message passing is implemented with the channels represented by variables. Sending a message is analogous to adding the message to the end of a message sequence, and receiving a message is analogous to removing the message at the head of the sequence.

UNITY provides a framework for sequencing instructions, though it does not have the facility to control the execution sequence of statements of the program.

5.2.3 UNITY Program Notation

<i>program</i>	—	Program	<i>program-name</i>
		declare	<i>declare-section</i>
		always	<i>always-section</i>
		initially	<i>initially-section</i>
		assign	<i>assign-section</i>
		end	

A *program-name* is a string of characters. The *declare-section* defines variables in a manner similar to PASCAL. Basic data types are integers and boolean. Arrays and sets of basic data types form the more complex data types.

Assign-section

The *Assign-section* defines the statements of the program. An assignment statement can have enumerated assignments or quantified assignments. Logical implication is a valid operator in UNITY programs.

Examples

Multiple assignments : $x, y, z := 1, 2, 3$ is the same as

$$x := 1 \parallel y := 2 \parallel z := 3 .$$

Arrays are assigned values by the use of quantification.

$$\langle \parallel i : 0 \leq i \leq N :: A[i] := B[i] \rangle .$$

The switch-case conditions given by

$$x = \begin{cases} y & \text{if } y > 0 \\ z + y & \text{otherwise} \end{cases}$$

is represented in UNITY as

$$x := \begin{cases} y & \text{if } y > 0 \\ z + y & \text{if } y \leq 0 \end{cases} \sim$$

where \sim is used to separate the two cases. Enumerated assignments are represented by

$$x, y := y, x$$

for exchanging x , and y .

An example of quantified assignments is an operation on an array of elements. An identity matrix $I[1..N][1..N]$ defined by

$$\langle \forall i, j : 1 \leq i \leq N \wedge 1 \leq j \leq N :: I[i, j] := 0 \text{ if } i \neq j \sim 1 \text{ if } i = j \rangle .$$

Initially-section

The *initially-section* defines initial values of variables used in a UNITY program. The syntax of a statement in the *initially-section* is the same as a statement in the *assignment-section* except that in the $:=$ symbol is replaced by the $=$ symbol. The *initially-section* essentially defines a non-circular set of equations. This definition satisfies the constraints that:

- a variable appears at most once on the left-hand side of an equation,
- there is an ordering of the equations such that any variable in a quantification is either a bound variable or a variable that appears on the left-hand side of an equation already defined, and
- the set of equations are completely ordered in such a manner that after the equations have been expanded over the quantifications, any variable appearing on the right-hand side of an equation, or in a subscript, appears in the left-hand side of an equation earlier in the ordering.

An example:

initially

$$\begin{aligned} & N = 3 \\ \&\& \quad \langle \&\& k : 0 \leq k < N :: A[N - k] = k \rangle \end{aligned}$$

The *initial* condition defined is generally the strongest predicate that holds initially. The predicate can be obtained by applying the following rules to the initial statements:

$\&\&$ and $\&\&$ are to be treated as a logical *and* (\wedge), and an initialization of the form

$$x = \text{exp}_0 \text{ if cond}_0 \text{ else } = \dots \text{ else } = \text{exp}_n \text{ if cond}_n$$

is to be treated as $(\text{cond}_0 \Rightarrow (x = \text{exp}_0)) \wedge \dots \wedge (\text{cond}_n \Rightarrow (x = \text{exp}_n))$.

Always-section

The *always-section* defines some variables as functions of other variables. The variable appearing in the left-hand side is termed a *transparent variable*, and does not appear in the left-hand side of any initialization or assignment. This section defines a set of *invariants* for the program, and if all variables used in the program are transparent, the program can be considered as a set of equations. Also each occurrence of such a variable can be replaced in a macro like fashion by its definition (this property is called *referential transparency*). Transparent variables are well suited for lazy evaluation.

The syntax of a statement in this section is similar to that of statements in the *initially-section*. An example of an invariant is, *Interest = (Principal * Time * Rate / 100)*.

5.2.4 The UNITY Proof Rules

UNITY logic is based on three logical operators: *unless*, *ensures* and *leads-to*. These operators describe temporal properties of a program. *unless* describes safety properties and *ensures* and *leads-to* describe liveness properties. In this section, we define these operators and some associated proof rules. These rules help derive and prove properties of UNITY programs.

The program fragment,

$$p \text{ unless } q$$

denotes that once predicate p is true, it remains true at least until q is not true. For example, a message is received only if it has been sent earlier. If *received* and *sent* are the two predicates, then

$$(\neg \text{received}) \text{ unless sent}.$$

The above statement does not comment on the future states of the program if p is not true and q is true at the time the statement was made. Invariant predicates can be defined in the same manner, thus deriving stable properties of programs. For example,

p *unless false* implies p remains true once it becomes true. An alternative method of saying the same thing is p *is stable*. If p is true in the initial state, and p is stable, then p is *invariant*.

The program fragment,

$$p \text{ ensures } q$$

implies that p *unless* q holds for the program and if p holds at any point in the execution of the program then q holds eventually. Put formally,

$$p \text{ ensures } q \equiv p \text{ unless } q \wedge (\exists \text{ statement } s :: \{p \wedge \neg q\} s \{q\}) .$$

After p becomes true, statement s executes, after which q becomes true.

The program fragment,

$$p \text{ leads-to } q, \quad \text{denoted as } p \multimap q ,$$

defines a condition where once p becomes true, q becomes true. The operator does not assume the persistence of the truth value of p for q to become true at a later point.

Rules for Inference

Some of the rules for inference and derivations are given. These rules are used in proving the adequacy of specifications (or otherwise). The method does not use transitions as a basis, but just the predicates and their corresponding statements. The first rule¹ provides a definition of *leads-to* relation. The second rule states that *leads-to* is transitive. The third rule states that if every predicate in a set *leads-to* q then their disjunction also *leads-to* q .

(basis)

$$\frac{p \text{ ensures } q}{p \multimap q}$$

(transitivity)

$$\frac{p \multimap q, q \multimap r}{p \multimap r}$$

¹The rules are interpreted in the following manner. For two predicates A and B ,

$$\frac{A}{B}$$

means infer B from A .

(disjunction)

$$\frac{\langle \forall m :: p.m \mapsto q \rangle}{\langle \exists m :: p.m \rangle \mapsto q}$$

Rules for *unless* are given below.

(consequent weakening)

$$\frac{p \text{ unless } r}{p \text{ unless } q, q \Rightarrow r}$$

(conjunction)

$$\frac{p \text{ unless } q, p' \text{ unless } q'}{p \wedge p' \text{ unless } (p \wedge q') \vee (p' \wedge q) \vee (q \wedge q')}$$

A number of other axioms allow reasoning about properties of programs. The **substitution axiom** states that it is possible to replace an invariant by a true value. Given $p \mapsto q \wedge I$, where I is the invariant, it suffices to derive $p \mapsto q$.

Program Compositions

Two programs P_1 and P_2 that can be executed concurrently are denoted by $P_1 \parallel P_2$. The set of statements of $P_1 \parallel P_2$ are the *union* of the statements of P_1 and P_2 . The *unless* property holds in $P_1 \parallel P_2$ if it holds in both P_1 and P_2 . The *ensures* property holds in $P_1 \parallel P_2$ if the corresponding *unless* property holds in both P_1 and P_2 and the *ensures* property holds in at least one of P_1 and P_2 .

This can be written as two rules:

$$\begin{aligned} p \text{ unless } q \text{ in } P_1 \parallel P_2 &\equiv p \text{ unless } q \text{ in } P_1 \wedge p \text{ unless } q \text{ in } P_2 \\ p \text{ ensures } q \text{ in } P_1 \parallel P_2 &\equiv (p \text{ unless } q \text{ in } P_1 \wedge p \text{ ensures } q \text{ in } P_2) \\ &\vee (p \text{ ensures } q \text{ in } P_1 \wedge p \text{ unless } q \text{ in } P_2) \end{aligned}$$

We can infer that

$$\frac{p \text{ unless } q \text{ in } P_1 \parallel P_2}{p \text{ unless } q \text{ in } P_1, p \text{ is stable in } P_2}$$

$$\frac{p \text{ ensures } q \text{ in } P_1 \parallel P_2}{p \text{ ensures } q \text{ in } P_1, p \text{ is stable in } P_2}$$

A Simple Example: Heap sort

In this example, a UNITY program for heap sort is presented. Initially, let $Y = X$ where X and Y are two arrays of integers. Let m be an integer variable with values ranging from 1 to N . A portion of the array Y is invariant because that portion has filled up its final values. Let v be the largest element in $Y[1 \dots m]$. Let big be an index such that $Y[big] = v$. The main action of the program is

Interchange $Y[m]$ and $Y[big]$ and concurrently decrement m if $m > 1$.

program HeapSort

declare big, v : integer ;

X, Y : array[1..N] of integer

always $v = \ll \max i : 1 \leq i \leq m :: Y[i] \gg$

$\square big = \ll \text{any } i : (1 \leq i \leq m) \wedge (v = Y[i] :: i) \gg$

initially $i : 1 \leq i \leq m :: Y[i] = X[i] \gg$

$\square m = N$

assign $Y[m], Y[big], m := Y[big], Y[m], m-1 \quad \text{if } m > 1$

end HeapSort

5.3 Example: Riding the Elevators

The elevator system has a straight forward implementation in UNITY notation. The use of a process based approach allows us to specify the system as a set of interacting processes. The implementation details of interaction can be specified after deciding the target architecture. The overall skeleton of the UNITY program is given in Figure 5.1.

We shall present additional features of UNITY using these examples as a part of the analysis of UNITY.

Program Elevator-system**declare**

N, M : integer ; *N elevators service M floors*
 service : array [1..N][1..M] of (served, notserved) ;
 buttons : array [1..2][1..M] of (on, off) ;
 ...

initially

Initialize the above variables to appropriate values
Such an initialization causes the definition of initial properties of the system
Initially all elevators are on floor 1

always

A simple invariant that has to be maintained is:
all elevators service first those requests in the direction they are travelling
 add (request, schedule) if (request > curr.floor \wedge direction [i] := up)
 \vee (request < curr.floor \wedge direction [i] := down)
 \neg add (request, schedule) if (request > curr.floor \wedge direction [i] := down)
 \vee (request < curr.floor \wedge direction [i] := up)

assign

The assign section first defines the behavior of the elevators
This behavior is defined as being concurrent with the behavior of the button

{
 [i, j ::
 on[up][i] \wedge \neg entered \Rightarrow broadcast(up, i)
 [serviced[dir, i] \Rightarrow off[dir, i]
 ...
 }
 }

The button behavior is defined
If a floor is not served, then the button number is added to the schedule of an elevator
A button is switched off if the floor for that direction has been serviced

[if button[dir][i] := notserved \Rightarrow schedule[j](i)
 [if service[j][i] := served \Rightarrow button[dir][i] := off
 ...

end

Figure 5.1: The outline of specification of the elevator system in UNITY.

5.4 Analysis of UNITY

- **Notational Features**

- *Unit of computation:* The basic unit of computation is a program statement in UNITY.
- *Data abstraction:* In UNITY, each set of related processes is described in a program. Other forms of data abstraction supported by UNITY include user-defined data types.
- *Modularity and Compositionality:* UNITY specifications are composed of a set of interacting programs. UNITY is a highly modular specification method. Composition of specifications in UNITY follows a uniform set of rules. These rules are the same for a single program and for a collection of interacting programs. In the above example, a number of programs like **serviced**, **broadcast** and **schedule** compose the UNITY specification for the elevator system.
- *Constraints:* Logical formulae specify constraints. These formulae define behavioral and structural constraints on the system specification. Logic provides a powerful mechanism for specifying constraints.
- *Transitions:* UNITY was not built with the dynamic analysis of programs in mind. UNITY does not support transitions.
- *Concurrent structures:* UNITY provides a rich set of concurrency and communication primitives. UNITY also provides a set of rules for transformation of such high-level primitives into architecture dependent code.

- **Semantic Features**

- *Defining assertions:* The logical framework provides a powerful mechanism for specification of assertions in UNITY. These assertions allow one to prove properties of the specification, such as safety and liveness properties.
- *Reasoning with time:* The temporal ordering imposed on the behavior of processes is not sufficient to specify a total ordering on the system. The reasoning mechanisms are only able to infer timing relations between the collections of partial orders.

- *Defining provable properties:* Specific properties can be inferred using the logic rules of UNITY. As yet no tool exists to realize this automatically, but the potential exists. For example, a few properties that can be defined for the example are as follows.
 - * The predicate **waiting** *unless* **request-exists** defines the state that an elevator is in the waiting state, and will remain so until there is some request for service.
 - * The predicate **visited[floor]** *leads-to* **button[floor] := off** defines a sequence of actions in the elevator system.
 - * A fairness property of the elevator can be expressed as **request-exists** *ensures* \neg **waiting**.

5.4.1 Drawbacks of UNITY

UNITY, though a simple language, does not provide a few basic facilities like the ability to specify transitions among processes and constraints on these transitions. The logical framework also makes building efficient tools for UNITY a difficult task. This is because pure logic does not allow efficient implementations [Hogger, 1984]. It is not possible to dynamically verify the designs generated by UNITY as it does not provide a framework for specifying transitions. The authors justify this choice by saying that dynamic analysis is error-prone. But the need for transitions and reasoning about the dynamic aspects of a specification is justified by the fact that most distributed software systems are generally underspecified. It is difficult to completely specify² most distributed systems, specially the reactive systems. The reason is that the types of interactions with the environment are not completely enumerable.

5.4.2 Unity and Temporal Logic

In this section, we attempt to show that UNITY is a very simple system and may be subsumed by other logic formalisms like temporal logic and transition systems. Such a

²There are other systems like system browsers, artificial intelligence programs and heuristics based programs which cannot be completely specified. This is because these programs interact with the environment in resulting in an unspecifiable type of data exchange.

formalization will help in providing a sound and complete³ framework for UNITY.

Most concurrent design systems are based on transition systems. Transition systems are normally characterized by a group of graphs, each with a starting control point. Temporal logic has been shown to be adequate for the specification of transition systems [Manna and Pnueli, 1983]. UNITY is a simpler transition system (actually a first-order transition system) whose underlying graph has exactly one root vertex. The execution of UNITY programs is constrained so that every transition occurs infinitely often on this graph.

Transition systems and temporal logic are closely related. Temporal logic is expressive enough to prove properties of transition systems. Compositional verification of properties is achieved by using transition logic. By this we mean that if the program fragments have certain properties, then a compositional verification will try and compose all these properties into one single property. Since transition systems have a number of graphs, one representing each program fragment, compositional verification is important.

UNITY has just one basic program, and hence is termed a *single-location-program*. Obviously, this is a simpler program than a full transition system which has many such locations of control. Therefore UNITY is easily modeled using transition and temporal logics. A formal presentation of this discussion is in [Gerth and Pnueli, 1989].

5.5 Chapter Summary and UNITY Extensions

The UNITY formalism attempts to provide a parallel version of Floyd/Hoare style programming logic. The method provides a logical basis for defining and proving temporal and invariant properties of programs. The specification of these properties has to be done by the programmer and there is no mechanism yet that allows the extraction of such properties from formal specifications of concurrent systems.

We present a brief description of the method and model the elevator problem in UNITY. We then analyze UNITY and compare it with temporal logic specifications.

³Temporal logic (at least the linear temporal logic) is claimed to be a complete formalism [Manna and Pnueli, 1988].

This comparison is restricted to the semantic aspects of UNITY, as temporal logic does not have a notation of its own.

[Sanders, 1989] provides an extension to UNITY logic and describes a method for refinement of logic specifications to implementation level programming structures. [Gribomont, 1990] extends UNITY with a transition system semantics and defines a transformational approach to formal concurrent systems development. Though the transition system semantics are algebraic, the programming semantics are refinements of UNITY semantics.

Logic proof systems provide a rigorous model for verification of concurrent systems. Tools for verification tasks must have proof systems that are able to reason about temporal properties, and temporal logic is best suited for that type of reasoning. There also exists a grammar-based approach to specification of temporal properties of concurrent systems [Manna and Pnueli, 1987]. Incorporating such techniques into tools would help automate the process of specifying temporal and invariant properties of software systems.

Chapter 6

Graph-based Methods

Concurrent computation cannot be represented using sequential structures. In a concurrent program, events take place in parallel and it is now recognized that such simultaneous computation is well represented using graph structures.

In this chapter, we will present a method called *statecharts*, which is based on a special class of graphs known as *hypergraphs*. This method provides a semantics based on extended state transition diagrams. We briefly present the basic concepts of this method and describe the various components of the statecharts visual language.

This method has been implemented as an environment for development of reactive systems known as STATEMATE. We present the method used in STATEMATE to compose specifications. A brief description of the associated tools is also presented, and we then use this formalism to model the elevators problem. Next an analysis statecharts is presented, followed by a short discussion of graph grammars as an underlying framework for graph-based computation. This discussion will highlight those features of graph grammars that are better represented than in statecharts.

6.1 Statecharts: Extending State Transition Diagrams

States and transitions adequately describe the behavior of complex concurrent systems. So state transition (ST) diagrams and the corresponding finite state machine model are a good choice to model formally such behavior. ST diagrams are directed graphs with nodes denoting states and edges denoting transitions. The edges are labelled with

triggering events and guards.

Such an arrangement is not adequate to represent large systems, as the number of states grows exponentially with the size of the system. Moreover, the description is not stratified, whereas most complex systems are hierarchical in nature. We should be able to group a related set of states into a larger state — *compositionality* and *refinement*, define abstractions over a set of transitions, and define a group of states as being concurrent and independent — *orthogonality*.

Statecharts extend the ST diagrams to model AND/OR decomposition of states, represent hierarchically structured systems, and provide a sense of abstraction when viewing software component descriptions [Harel, 1987b]. The following equation characterizes statecharts.

$$\begin{aligned} \text{Statecharts} = & \text{state-transition diagrams} + \text{depth} + \text{orthogonality} \\ & + \text{broadcast communication} \end{aligned}$$

In this section we present the graph specializations that form the basis for statecharts. These specializations are called *higraphs* which in turn are specializations of *hypergraphs*.

6.1.1 Higraphs: Visual Formalisms

A graph can be viewed as a set of nodes with a binary relation defined on the nodes. Restricting the type of relation will yield special classes of graphs. *Hypergraphs* are an extension of graphs where the relation on nodes is not binary and has no fixed arity. Hypergraphs have the ability to be more expressive in representing objects and their relationships. Figure 6.1 shows a representation of hypergraphs.

Euler diagrams provide a convenient method of representing logical propositions on collection of sets. They also define structural relationships between the sets. *Higraphs* [Harel, 1987a] are a hypergraph-based extension to an extended form of Euler diagrams. Euler diagrams are extended to represent the Cartesian product. In addition, the resulting nodes are connected using edges and hyperedges.

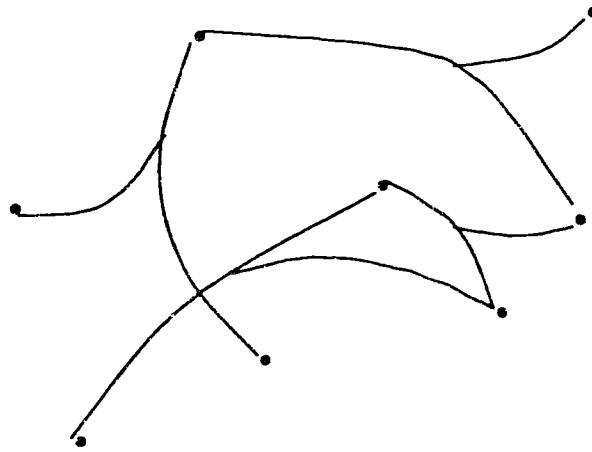


Figure 6.1: Hypergraphs.

6.1.2 Statecharts: Features

The higraph notation uses rounded rectilinear shapes to represent sets. Statecharts are an extension to higraphs where the rounded rectilinear shapes represent states and arrows represent transitions.

Statecharts extend ST diagrams in a number of ways. A statechart provides depth and hierarchy in describing the structure of a system. This allows abstractions to be built. Also statecharts model concurrency in a more natural way by providing facilities for local composition and not like ST diagrams where concurrency can only be modeled by its global states.

Figure 6.2 gives an example of a statecharts representation. The Cartesian product is defined as a *compos...*. The shape representing the set Z is a cartesian product of the shapes representing sets X and Y in Figure 6.2. Depth is represented by the insideness of the rectilinear shapes.

Compositionality

In Figure 6.2 objects B and C are inside A and do not overlap. Then A is an XOR representation of B and C , so an event in A is in state B or C but not both. Any arrows that leave A apply both to B and C . Thus we are reducing the number of arcs in the

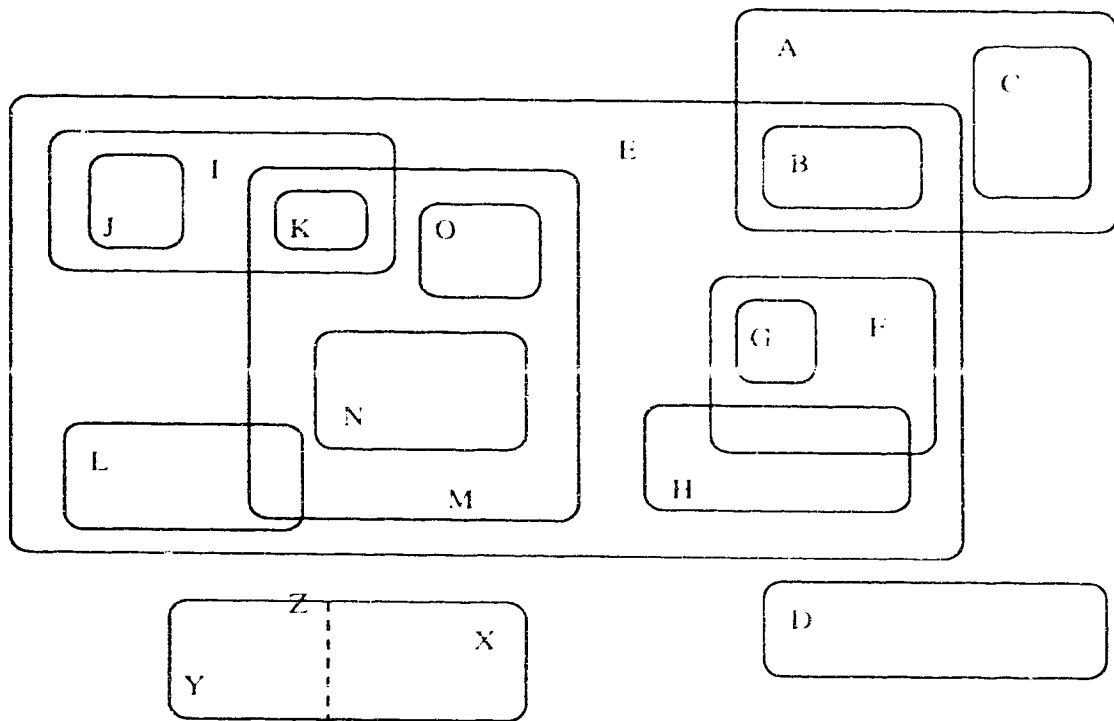


Figure 6.2: A simple Statechart.

representation.

Orthogonality is the dual of XOR decomposition. In Figure 6.2 X and Y are AND decompositions of Z, and are said to be orthogonal. To be in state Z is to be in both X and Y states. The XOR decomposition is also known as *clustering*. Arrows are labelled with names which denote actions. Variables in parentheses are predicates which denote conditions. The depth feature allows one to *zoom* into and out of an object description.

History Mechanisms

A group of states are clustered, and all events in that group are linked in the order of execution. This *history* of states allows one to revisit a group of events. The choice of the event can be specified by an arrow to default to an event in case the connective is appearing for the first time. Figure 6.3 shows the use of such a connective. In Figure 6.3 (a), the connective is the letter H with the circle around it. The default is specified by the arrow pointed to the state C. This mechanism can be used to visit events at any level of detail. Figure 6.3 (b) shows an example. Here F is the default state.

Conditional Transitions

Two other connectives allow conditional invocation of events within a group, and selection of events (see Figures 6.4 and 6.5). In Figure 6.4 (a) a conditional visit to B, C or D is made depending on the truth value of the corresponding predicates $p(P)$, $p(Q)$ and $p(R)$. Figure 6.4 (b) implements the same by using a conditional connective. The value evaluated at the connective will initiate the corresponding event.

Figure 6.5 (a) shows the conditional selection of one of the three events. The history mechanism can also influence the selection of a state. An example is shown in Figure 6.5 (b). Events B, C or D are activated depending on which event was active previously. Other features include the ability to specify delays on events, and allow for unclustering a statechart so that events and states are explicitly displayed.

Activity lists are also a part of each state. These lists specify what action is to be taken when a certain event occurs. [Harel, 1987b] presents a detailed discussion on the

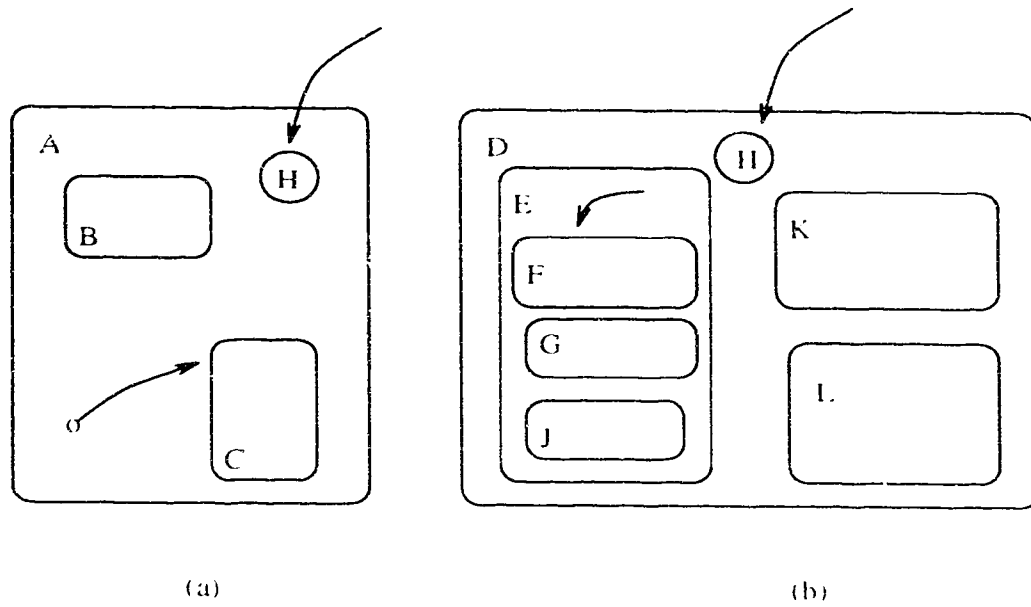


Figure 6.3: (a) History arrow, if entering for the first time, it enters C. (b) History connective applicable to the lowest level; in this case E.

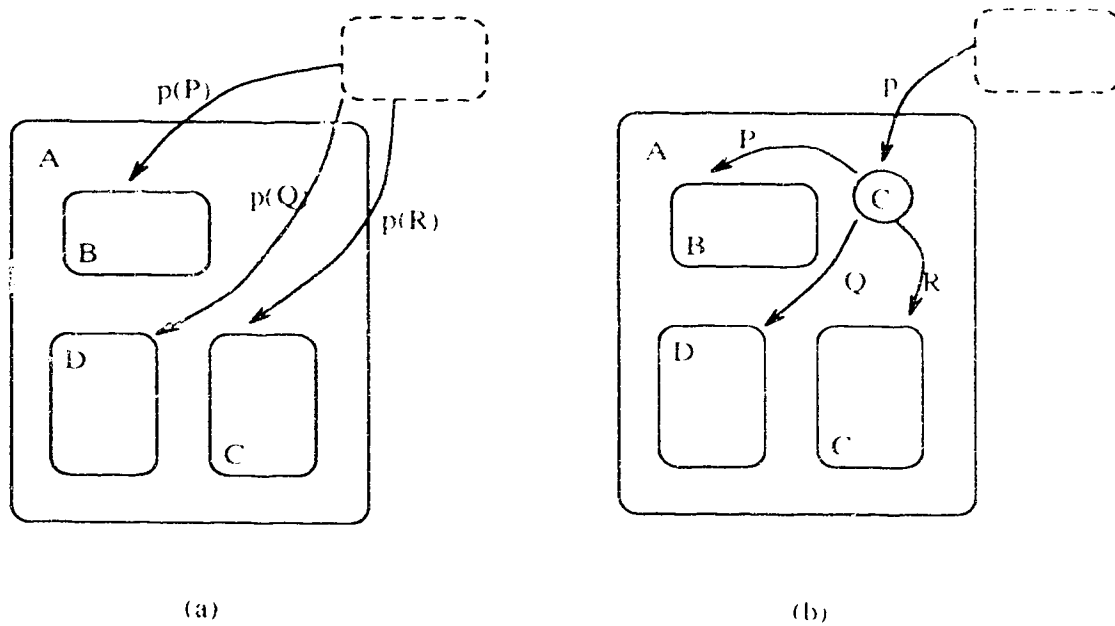


Figure 6.4: (a) Implementing a condition represented by the predicate p . (b) Implementing a condition using a conditional connective.

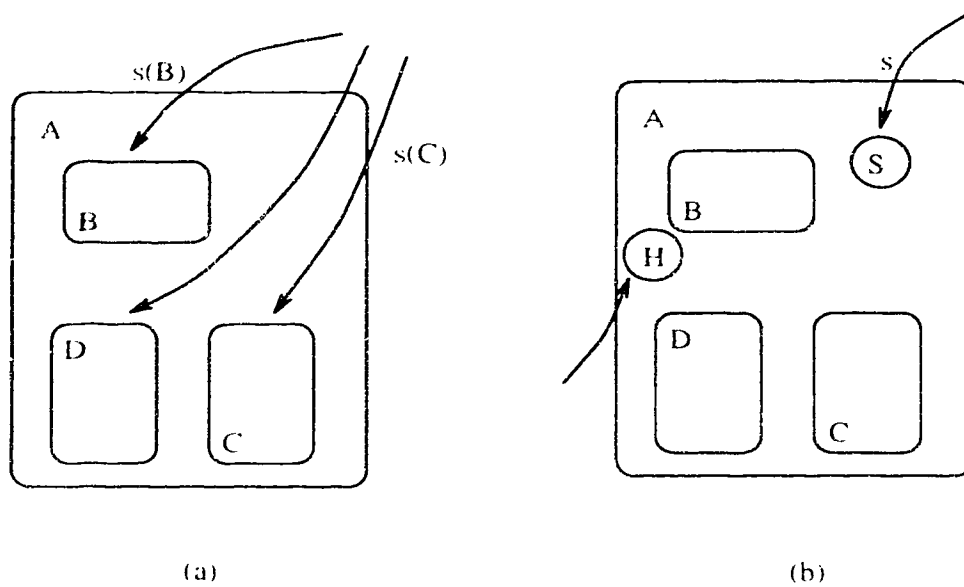


Figure 6.5: (a) Implementing a selection among three events. (b) Implementing a selection among three events using a selection connective and a history input.

semantics and possible extensions to the formalism.

6.1.3 STATEMATE: An Environment for Developing Distributed Systems

The specification method developed by Harel and his coworkers allows three different views of a concurrent, reactive system specification [Harel et al., 1990]: *functionality*, *structure*, and *behavior*.

System Specification by Multilevel Hierarchical Decomposition

Structural information deals with the hierarchical decomposition of the system into components. Information flow between each component is specified. The *functional* component provides hierarchical decomposition of the system based on the functions the system performs. Two types of information flows are specified: flow of data items and flow of control information between functions. These two views do not conflict with each other but complement each other. These two views do not specify information

about when the states of the system change and about temporal relationships between various interacting activities. Essentially these two views do not specify information about the behavior of the system.

The *behavior* is modeled by control flow and temporal relations. Each level in the function hierarchy can have a description of behavior. A statechart describing control flow specifies when, how and why events occur. Such a statechart defines the constraints on execution and generation of events.

The general framework for specification is based on the statecharts. Physical decomposition of a system is specified by *module-charts*, functional decomposition by *activity-charts* and behavioral decomposition by *statecharts*. Module charts and activity charts are based on the statechart formalism but are differentiated by having rectangular shapes (different syntax) and slightly different semantics. Storage modules are rectangular shapes with dashed sides. Solid arcs define information flows, and are labelled *hyperarrows*¹.

In activity-charts, the objects are described by rectangular shapes, as with module-charts, but differ in the semantics of the arcs between objects. In activity charts solid arcs define data flows and dashed arcs define control flows. Activity charts represent two data objects: storage structures and control structures. Control structures are represented within an activity structure by means of statecharts.

Behavioral specification of activities and events is done by statecharts. Some common conditions and events modeled by statecharts are listed in Table 6.1.

Other Tools

Though these graphic structures define decompositions of a software system, they have no way of specifying the structure of a data object. *Forms* language provides the necessary mechanisms to specify the data objects in the form of a slot filler structure. STATEMATE provides editors to specify and edit descriptions interactively. The editors check for syntax correctness before committing specifications.

STATEMATE provides a tool for querying the object specification database. The

¹Hyperarrows can have more than one end point.

Objects	Events	Conditions	Actions
State S	entered(S) exited(S)	in(S)	
Activity A	started(A) stopped(A)	active(A) hanging(A)	start(A) stop(A) suspend(A) resume(A)
Data items D, F Condition C	read(D) written(D) true(C) false(C)	D = F D < F D > F ⋮	D := F make_true(C) make_false(C)
Event E Action A n time units	timeout(E, n)		schedule(A, n)

Table 6.1: Some common events and conditions, and corresponding action sequences for statecharts.

tool keeps a list of incomplete specification components that gets updated as the specifications are completed.

A report generation tool generates reports on data objects and the specified state changes. Reports include various data dictionaries, the behavioral specification of states and associated activities, interface diagrams of modules among many others.

STATEMATE also provides a *document generation language* which users can tailor to meet their specifications. Specifications using this language can be associated with related structures in a specification. The ability to interlace the document specifications with instructions that collate relevant information from the specified model helps generate reports that are very detailed.

Executions and Dynamic Analysis The STATEMATE environment provides a simulator that allows execution of system specifications. This allows for dynamic analysis of the system behavior. External events are generated to simulate the reactive nature of the system. The various components of the specification that are currently being “executed”, are shown in the screen and potential parallel executions are depicted. The simulation can be run at various levels of detail and describes a series of states through which the system passes.

This facility of stepping through a system behavior interactively allows one to debug specifications *visually*. STATEMATE also provides a *simulation control language* that allows a user to program his executions. This language provides all the facilities of a conventional high-level language and the associated tool allows the user to set *break-points* in the simulation to observe and interactively change the control structures. Such simulation runs can be recorded as trace structures and analyzed later.

6.2 Example: Riding the Elevators

In this section we represent the elevator system using statecharts. The states of the system's components are represented by the rounded rectilinear diagrams. The arrows denote events and processes. The elevator system consists of n elevators and m floors. Each floor has two buttons indicating **Up** and **Down** requests for service. An outline representation of the elevator system is given in Figure 6.6.

The elevator object state changes and associated events are modeled in Figure 6.7. An elevator gets as input a request for service from some floor. A request for service will get channeled depending on which state the elevator was in (a history mechanism is in effect here). If the elevator was in the **Waiting** state, the request is processed and an appropriate state is reached. The conditional connective checks the type of request before requesting that either **Up** or **Down** states be reached. **Slist** is the list of requests to be processed.

The state changes in the **Up** state of an elevator are shown in Figure 6.8. This is one of the states in Figure 6.7. There are essentially two objects being modeled: **Slist** and **Curr_Floor**. Different events take place depending on the input received. As a new floor is added, the **Slist** changes its contents. Depending on the state of **Slist**, other events, such as stopping at a floor and reaching the **Waiting** state and changing direction to reach the **Down** state occur. The two events **Stop** and **Change_Direction** reach another state in the elevator subsystem shown in Figure 6.8. The event **Switch Off(Curr Floor)** sends a broadcast message to the communication link between the elevators and the buttons at each floor.

The state representation of buttons at each floor are simpler. The input is a request

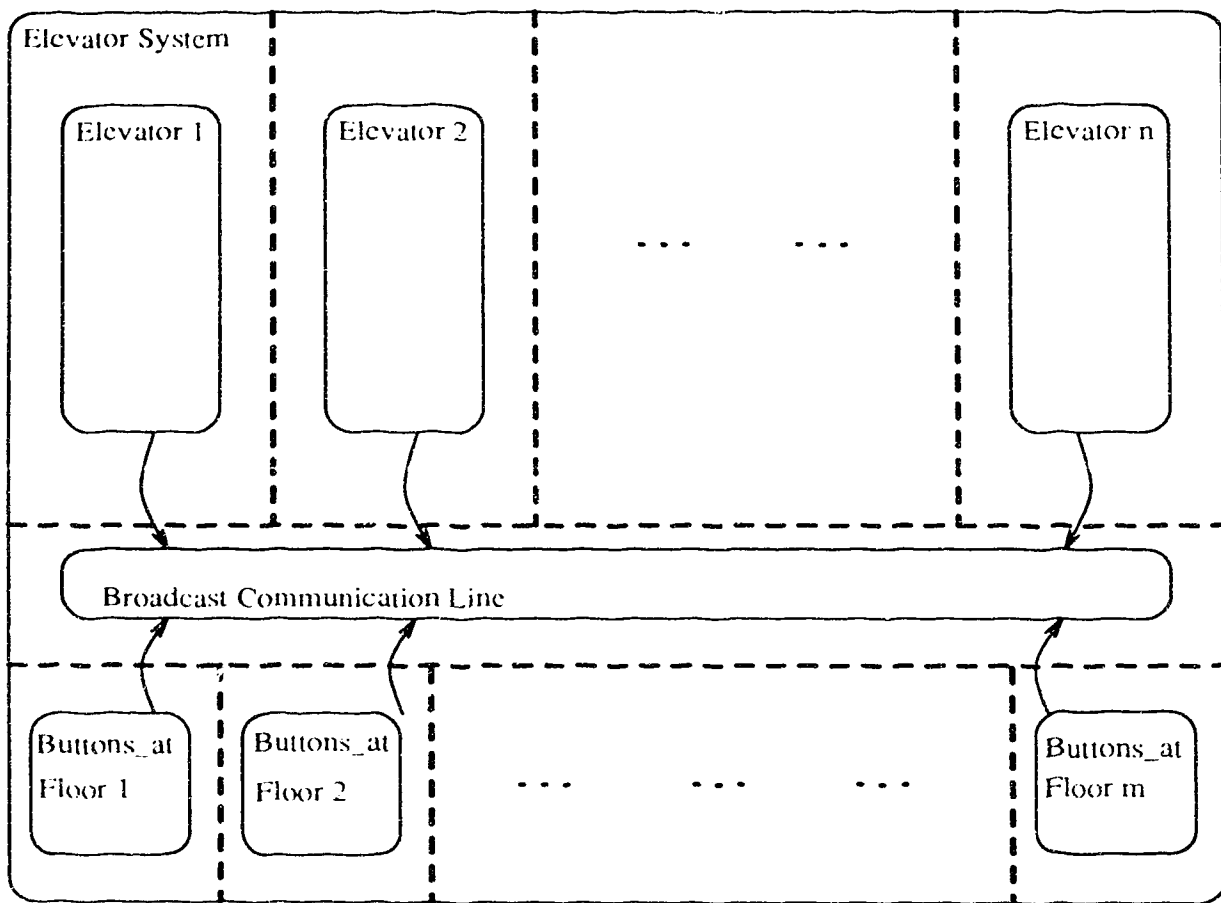


Figure 6.6: A high-level representation of the Elevator System using Statecharts.

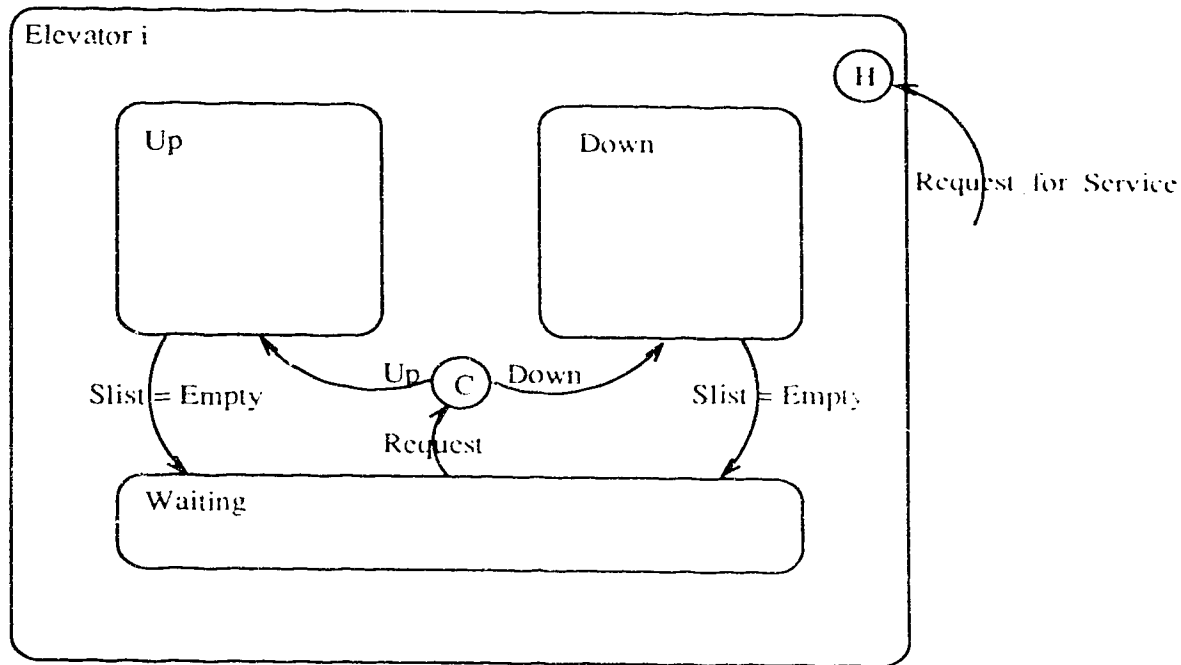


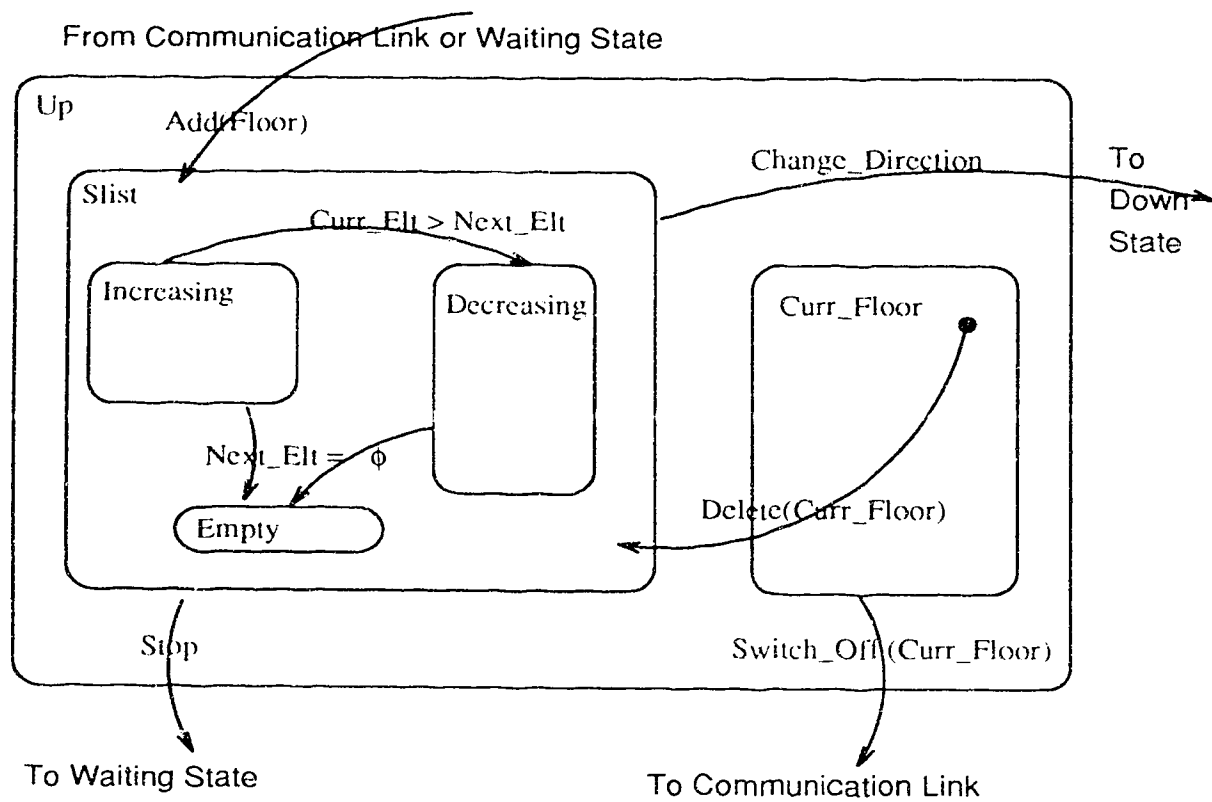
Figure 6.7: The elevator subsystem.

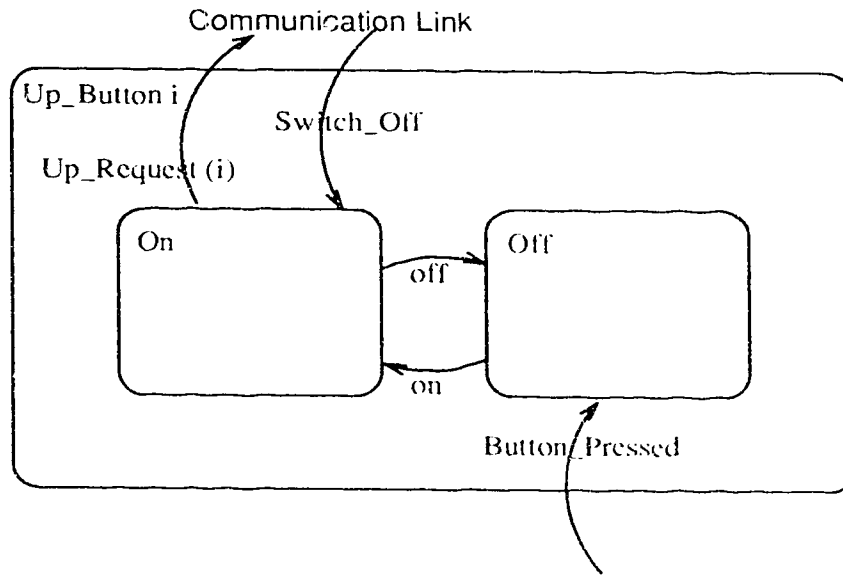
for service, i.e., pressing a button, or a switch off command by one of the elevators. A button can be in two states – On and Off. When a **Switch_Off** request is received, the button switches itself off and goes to the Off state. When a **Button Pressed** event is initiated, it goes to the On state. Then it requests a service by broadcasting a message to all elevators. This behavior is shown in Figure 6.9.

6.3 Analysis of Statecharts

- **Notational Features**

- *Unit of computation:* The basic unit of computation is a state diagram representation. This represents an object or its state.
- *Data abstraction:* Data abstraction is provided as the “insideness” of an object representation.
- *Modularity and Compositionality:* Modularity is again an extension to the data abstraction principles of statecharts. Compositions are formulated by

Figure 6.8: State changes in the **up** state of an elevator i .

Figure 6.9: State changes in the `Up_Button` at floor `i`.

connecting different objects and “modules” by arrows. These arrows are labelled by predicates to denote constraints. Another mechanism of composition is the AND composition of different objects. Overlapping of state representations provides another mechanism of defining composition.

- *Constraints*: Constraints are defined by assertional expressions which specify behavioral constraints on the processes. Processes are represented by arrows and constraints are labels on these arrows.
- *Transitions*: The state transitions are defined as a part of an object description and therefore themselves are a form of constraints on the behavior of the associated objects.
- *Concurrent structures*: Concurrency is represented by AND parallelism constructs and choice constructs. These constructs are graphical and are easy to visualize. The AND decomposition of objects is represented as in Figure 6.2, where objects in state `Z` can be in two possible states `X` and `Y` at the same time.

• Semantic Features

- *Defining assertions*: Assertions are defined as predicate labels on process con-

nectors (arrows). Assertions allow expression of properties of a system based on its behavior. For example, the statement *the system will not stop until all requests have been processed* is represented in Figure 6.8 as the condition when **Slist** gets empty.

- *Reasoning with time:* There do not exist any mechanisms to reason about the temporal relationships between processes. The statecharts are representations of behavioral changes.
- *Defining provable properties:* Properties are defined as logical predicates on an object or one of its states. Such properties are defined as a part of an event or a process that effects the state of an object.

The STATEMATE system provides mechanisms to perform many of the verification tasks for a statechart specification. The semantics of statecharts are based on extended finite-state-automata. Such a formalism does not allow representation of infinitely many states, and therefore specifications in statecharts have a fixed domain to contend with, and are easier to debug.

Statecharts allow the designer to represent relationships between states of objects at different levels. For example, in Figure 6.8, the arrows go into **Slist** from outside the object. This lack of a uniform interface for each object does not support the principle of writing well-defined, complete interfaces for objects.

6.3.1 Statecharts and Graph Grammars

The tools in STATEMATE provide mechanisms for observing and dynamically changing the behavior of a distributed software system. Graph grammars can provide a formal basis for such visualizations. This is because graph grammars provide an excellent framework for defining transformations of representations from one level of detail to another.

In this section, we briefly describe graph grammars and various operations on them, including exploiting implicit parallelism to reduce graphs. Then, we discuss how statecharts can benefit from the graph grammars formalism.

Graph Grammars

Graph grammars consist of

- An initial graph, and
- A set of productions.

A graph grammar is a graph rewriting system², where the set of productions constrain the rewrites. Graph-based computation proceeds by moving values through nodes and arcs of a program graph. Nodes and arcs in a graph represent objects and transitions respectively. Some examples of static graphs systems are: *petri nets* and *data flow diagrams*.

Generalized Graph Grammars

Generalized graph grammars allow the most unconstrained type of graph rewriting. Productions include a left hand side that matches a subgraph of a graph to be rewritten, and a right hand side that tells how to replace the matching subgraph in order to complete the rewrite. Some form of *isomorphism* (or equivalence) is established during the process of *matching*. This poses the problem of how the right hand side is connected (embedded) into the graph that is being rewritten. Different kinds of generalized grammars specify embedding in different ways. Embedding rules for programs that generate $X - Y$ graphs are given in [Goering, 1990]. We present below general techniques of graph composition and decomposition which are useful in distributed problem solving.

Parallel Graph Transformations

We briefly present results of research on graph grammars and concurrency. Decomposed graph representations can be solved over a distributed network of computers. Two methods for composition and decomposition of graphs that preserve consistency of derivations are *split* and *join*. These are defined below.

²A rewriting system consists of a set of production rules and a set of terms. The term is transformed by applying these rules. In a graph rewriting system, the productions define new graphs for a given graph.

Split A graph G can be split into G_1 , G_2 and I , where I is an interface graph. Consistency is preserved if I is the intersection of G_1 and G_2 . Such a split is defined by the functions $I \rightarrow G_1$ and $G_2 \rightarrow G$. Such a split is represented by

$$G_1 \uparrow_I G_2 \rightarrow G.$$

The addition is defined over the interface graph I .

Join Two graphs G_1 and G_2 with a common interface graph I , can be joined into a single graph G . Such a join is represented by

$$G \leftarrow G_1 \uparrow_I G_2.$$

Such a join is defined by the functions $I \rightarrow G_1$ and $G_2 \rightarrow G$ between the interface graphs and the constituent graphs, and the constituent graphs and the composite graph.

Subsequent transformations of the graph components must preserve the interface, otherwise the composition is destroyed. If D_1 is a transformation for G_1 , then it is *I -preserving* if there is a unique graph morphism $I \rightarrow D_1$ such that $I \rightarrow D_1 \rightarrow G_1 = I \rightarrow G_1$.

Distributed Graph Grammars

Distributed graph grammars provide a mechanism for defining noninteracting productions on local (component) and global graphs. Formally such a graph grammar is defined as

$$DG = (L_1, L_2, G, I)$$

where L_1 and L_2 are the local graphs and G is the global graph, and I defines the interface (intersection) graph.

Producers generated by such grammars allow for parallel processing of graphs. Generation of causal independent sequences allows viewing them as temporally disjoint sets of actions. Proofs of correctness of such transformations of global graphs to decompositions (i.e., derivations of components to their global graphs are given in [Frig et al., 1987].

CHAPTER 6. GRAPH BASED METHODS

An algebraic approach to graph grammars is based on graphs, graph homomorphisms and gluing constructions of graphs [Thrig, 1978]. Such an approach allows definition of concurrently executable productions. These productions are said to preserve consistency of graph decompositions if the homomorphisms defined on them are proved true later. These homomorphisms are treated as proof obligations.

Graph Grammars and Implications for Statecharts' Metamorphosis

The transformations of statecharts are easily defined as graph productions. One basic problem associated with such an approach is that the "arrows" in statecharts can transcend different levels of abstractions. Such "arrows" do not support modularity and compositionality in a software system as they violate the rules of well defined interfaces.

One way of solving this problem is to define interface graphs between the objects at different levels of detail. In the framework of graph grammars, it is feasible to formally define such interfaces. Such a representation in graph grammars corresponds to finding the intersection graphs for collections of interacting statechart objects. Given such an interface graph, transforming it across a graph is done by graph productions. For a number of classes of graphs there exist polynomial time algorithms for recognizing such intersections [Golumbic, 1980].

6.4 Chapter Summary

STATEMATE is a powerful environment for specification of requirements, design and rapid prototyping of concurrent, reactive systems. The ability to execute specifications allows systems to be designed with specific goals of performance in mind. Systems generated thus are closer to the specifications.

STATEMATE employs visual formalisms for all the stages of software development. This allows increase in productivity of development of software systems. The statecharts formalism is simple and is based on graphs. The algorithms for traversal are not hard and can be implemented efficiently in polynomial time. The ability to specify graph abstractions also helps in localizing the processing of statechart.

STATEMATE also provides generic tools that help in translation of statecharts and associated descriptions of a system into a high-level programming language. Rapid prototyping becomes easier with this kind of facility. Work is being done to generate VHDL code so STATEMATE can generate code for silicon compilation and subsequent fabrication of chips.

The aspect of clean interfaces between modules of statecharts is capably addressed using graph grammars. We have mentioned how “deep” connections between different objects in statecharts are abstracted by graph productions. This entails preliminary transformation of statechart graphs by recognizing various intersection subgraphs.

Chapter 7

Object-based Methods

In this chapter we discuss object-based methods for design of distributed systems. We present the discussion using *Actors*, an object-based model for distributed computing. In Section 7.1 we give a brief historic background of object-based approaches to computing. In Section 7.2 we provide the motivation for using object-based methods for design of software and define the terms *objects* and *classes*. We then provide a working definition for *object-based design*. We also then provide some problems in using object-based methods in designing distributed applications.

In Section 7.3 we present the Actors model as an example to illustrate the methods and issues in using object-based methods for distributed application design. We use a compact language for Actors to highlight the features. We then present a discussion of actors as a design formalism for distributed applications.

7.1 Introduction

SIMULA [Dahl et al., 1970] was the first language to introduce the object-based approach to computing. The language introduced the concept of a *class* to represent data abstraction. Now object-based systems have become one of the promising approaches to development of software at all stages.

The object-based methods evolve around the concept of state-based approach rather than a data-centered approach to software design. Object-based methods allow decomposition of software system as a hierarchical classification of object classes. Most such

systems also provide inheritance as a means of specializing an existing classification, polymorphism and dynamic binding as a means of generalization. [Wegner, 1987] gives an excellent introduction to programming with objects.

7.2 Object-based Design Methods

Structure charts represent software designs as functional units. One drawback with this strategy is that any change in the system function will necessitate a major redesign. This problem is minimized in object-based designs. The basic reason is that information is grouped with associated operations. Only the relevant operation definitions need be change !. The motivation for an object-based approach which emphasizes abstraction of data, is best expressed by the statement ([Gutttag, 1977]).

Subprograms, while well suited for the description of abstract events, are not particularly well suited to the description of abstract objects.

7.2.1 Object and Classes

Object based design was popularized by Booch [Booch, 1986]. Before we define *object-based* design, we first define the term *object* (see [Booch, 1986]). An object can be defined as an entity that

- has a state and can be referred to by a name.
- is characterized by the operations on it by other objects and by the operations by it on other objects.
- is an instance of some class of objects.
- has controlled visibility of and by objects.
- can be viewed either by its specification or by its implementation.

An object can be classified as an actor, agent, or server depending on the relationship with other objects. An *actor* object does not suffer operations but only operates on other objects. A *server* only suffers operations and cannot operate on other objects. An *agent* has both the characteristics of an actor and a server.

An *object class* is a definition from which we can create one or more *object instances*, each of which has its own, independent internal state, but otherwise identical properties. *Object instances* interact by sending *messages* to each other, which cause the receiver to do any or none of the following:

- change its internal state,
- send a message or messages to another object,
- return a resulting object instance to the sender.

New *object classes* can be created by combining different *object classes* and/or by specifying alternate actions to be performed upon the receipt of messages. *Inheritance* is the property of an object system wherein, a child object inherits data structures and associated methods from its parent. *Delegation* is a technique of routing methods of a class to a particular object. This is more flexible than inheritance, as it is easier to change the delegatee object, but not so easy to change the superclass. Terms like *delegation* and *inheritance* refer to various forms of combination and modification of *object classes*. See [Viegas, 1987] for formal definitions of these terms.

7.2.2 Object-based Design

Object-based design is the process of creating object classes (and appropriate object instances thereof) from some base set of builtin object classes to solve a given problem. Good OOD is the creation of a minimal set of efficient object classes which need minimal changes in case of modifications in design.

An object definition provides an abstraction of behavior of a related set of entities associated with an application. To this purpose, an object encapsulates both a specification of its behavior and its representation. Borrowing the idea of information hiding from Parnas [Parnas, 1972], only the pertinent behavior of the object is visible to other objects, and its representation is hidden. This helps address the problems of data abstraction and information hiding. The following concepts are supported by the object-based paradigm.

- **Generalization:** Generalization mechanisms allow the use of a set of objects in

a variety of situations. Modeling polymorphism is one example of generalization.

- **Aggregation:** Encapsulating related concepts into an object definition is aggregation.
- **Association:** Associations are defined by relationships between objects. Examples of association in the object-based framework are the techniques of inheritance and delegation¹.

7.2.3 Object-based development and Concurrency

For sequential computations, an object-based framework offers a number of benefits. But in a distributed or distributed application the notions of inheritance causes problems of synchronization. We define some of the problems associated with synchronizing concurrent events in an object-based environment below.

- Synchronization code embedded in parent classes is difficult to reuse by subclasses that must cooperate with the inherited code.
- Parent class code has no way of knowing the synchronization policy requirements of the child classes.
- Providing different synchronization mechanisms at all class levels negates the tenet of abstraction and information sharing.

7.3 Actors: A Model for Distributed Computation

Actors formalism provides a general framework for programming various concurrent computers. This framework provides a model to represent, at an abstract level, the concepts of communication and concurrency. In addition, it exhibits a number of other features necessary for development of distributed programs.

Actors are active objects in an actor system which send and receive messages. Computations in actor systems are defined as a series of interactions between a number of actors. The actor model defines a behavior system for the set of communicating actors.

¹These terms are well defined in [Wegner, 1987].

In the following sections we define the components of an actor system. We then define the actor model of computation. Next we describe the properties of an actor program. We use the framework built so far and present a simple actor language with the objective of using this language to specify the elevators problem.

7.3.1 Components of an Actor System

What is an Actor?

An actor is a pair $\langle M, \mathcal{B} \rangle$, where M is a mail address (referring to a queue of incoming messages), and \mathcal{B} describes the behavior associated with the actor, as a consequence of processing the mail message. This behavior can consist of the following actions.

1. An actor can *send* messages to other known actors (including itself). The messages are communicated by *tasks*.
2. An actor can *create* new actors. The mail address of this new actor is known initially to the creator, (possibly to the newly created actor also) and may be communicated later to other actors.
3. An actor *must* specify a (replacement) behavior to process subsequent messages. After an actor receives a message, it may perform other actions too. A replacement behavior processes subsequent messages. This processing can occur concurrently to the other actions an actor performs.

Message passing is the basis for communication and concurrency in actor systems. Actor behavior is supported by providing mail buffer for the messages for each actor. Tasks perform communication. We give a compositional definition of tasks and their function next.

Tasks

We have defined tasks as the processes which perform the actual communication. Each task consists of the following.

- A *tag*. The tag is used to identify various tasks as tasks may contain identical addresses and messages to actors.
- A *target address* of an actor to which the communication is to be delivered.
- A *communication* which contains information to be passed to the receiving actor.

Formally, a task is a triple $\{n, m, k\}$, and each triple is an element of \mathcal{T} , where \mathcal{T} is the set of all tasks in an actor system. We define \mathcal{T} as follows.

$$\mathcal{T} = \mathcal{N} \times \mathcal{M} \times \mathcal{K}.$$

There are three ways an actor, while accepting a communication, knows the address of another actor to send communication.

- The address was known before it accepted the communication.
- The address became known after it accepted a communication, which contained the address.
- The address is that of a new actor created, as a result of processing the communication.

Types of Actors

There are two classifications for actors, one based on the type of replacement behavior specified and the other based on the type of actor behavior. The replacement behavior of an actor involves the creation of a new actor that will process the communications in the message buffer. Depending on the type of replacement behavior specified there are two kinds of actors: *serialized* and *unserialized*.

An *unserialized actor* is one whose behavior never changes over its lifetime because the replacement behavior is always caused by an identically behaving actor. Unserialized actors are similar to non-recursive mathematical functions whose behavior does not change with different inputs, i.e., the same set of instructions are executed no matter what the inputs are.

Serialized actors are those whose behavior is characterized by actions that happened earlier. These actors are said to be history sensitive, i.e., the behavior² changes as a result of processing a communication. Dynamic systems (such as adaptive control systems) exhibit such behavior.

7.3.2 The Actor Model

Distributed computation can be modeled in different ways (Section 1.4). The actors model is based on a message passing model and provides message buffers with asynchronous communication between actors. Each actor is viewed as an object having its own behavior. The internal structure is private to each actor, and the only way of effecting the behavior of an actor is by sending it a communication. The actor model does not provide a *class* mechanism, but provides a mechanism for hierarchical³ composition of actors. The newly created actors can be given a behavior to act as processing units for the creator actors. The problems associated with inheritance and concurrency are avoided as a result. This means more work at defining actor interfaces, but the representation is a lot cleaner than when dealing with code sharing problems associated with inheritance.

The message-passing behavior of actors is characterized by a closure on the actor addresses. Identifiers in an actor hold the mail addresses of those actors it knows. Actors at these addresses are termed *acquaintances*. The messages an actor receives generate bindings for the identifiers in the body of the actor.

Computation in an actor system is initiated by sending a message to a *main* actor. This actor sends messages to those actors that start up the rest of the application. The actors which receive their first messages from the external environment are called *receptionists*. The receptionists are similar to a user-interface module. In the actors formalism, it is possible to have more than one such “modules”, one for each collection of actors with similar definitions.

²Formally, unserialized actors can be defined as follows. Suppose an actor α has a behavior β , and it receives a communication k . Suppose also the new behavior is β' . Then α is an unserialized actor if for all $k \in \mathcal{K}$ $\beta = \beta'$.

Otherwise, α is a serialized actor, and β' is the replacement behavior for β as a result of processing the communication k . \mathcal{K} is the universe of possible communication messages.

³The hierarchy is defined by the creation of new actors and the information each actor has on other actors.

Every time an actor receives a message a new behavior is created. The old instance of this actor is destroyed if it has finished all computation necessitated by the message. Hence, an actor can initiate multiple behaviors at the same time. Each behavior is associated with an actor. A collection of such actors (each having a different behavior, but all generated by an actor) is called a *configuration*.

This mechanism is useful in defining transitions in an actor system, as it provides a history of actions. A new configuration evolves from an older configuration when there is a change in the composition of the configuration, either in its set of actors or in the set of tasks to be performed. Thus there is scope for a tool to verify correctness of behaviors of actor programs based on the configurations. We define a configuration and its properties below.

Configurations

A *configuration* is a set of actors and tasks. This is defined in terms of the unprocessed tasks and a *local states function*. A local state function is a (partial) mapping from the set of mail addresses \mathcal{M} to the set of possible behaviors \mathcal{B} .

$$l : \mathcal{M} \dashrightarrow \mathcal{B}.$$

By extending the range to $\mathcal{B} \cup \{ \perp \}$, we make l a total function. All undefined elements are mapped to \perp .

Now, a configuration is defined as a pair (l, T) , where T is the finite set of tasks in the actor system. The definition is constrained by the need for uniqueness, i.e., two configurations will have different tags and mail addresses.

7.3.3 Properties of Actor Programs

The essential components of an actor program are

- *behavior definitions* which define a behavior and tag it to an identifier without creating a new actor,
- *expressions* to create actors.

- *send* commands to create tasks.
- *receptionist declaration* which lists all actors that may receive information from the external environment.
- *external declaration* which lists all actors which are not a part of the current computational configuration that may receive communication from the actors in the computational configuration.

We present a simple actor language and

A Simple Actor Language and its Semantics

We describe a Simple Actor Language (SAL) [Agha, 1986], which illustrates the essential features of actor languages. The language emphasizes the ability to distribute a computation, and generate maximal concurrency for a computation.

In the following discussion, the acquaintance list is enclosed in (\dots) and the communication list is enclosed in $[\dots]$. The behavior definitions in SAL do not create new actors, but bind an identifier to a behavior template. The behavior definitions are written as:

```

< behavior_definition > ::=
  def < behavior_name > ( < acquaintance_list > ) [ < communication_list > ]
    < command > ::=
  end def

```

where *behavior_name* is an identifier, to which the described behavior is bound.

The replacement behavior in SAL is specified by a **become** command whose syntax is

```
become < expression >
```

where $\langle expression \rangle$ evaluates to a behavior. Commands to perform operations, including specifying replacement behaviors have the syntax

```

⟨ command ⟩ ::= if ⟨ logical_expression ⟩ then ⟨ command ⟩
               { else ⟨ command ⟩ } fi |
become ⟨ expression ⟩ |
⟨ send_command ⟩ |
⟨ let_bindings ⟩ { ⟨ command ⟩ } ⟨ behavior_definition ⟩ |
⟨ command ⟩*

```

The predicate **new** creates a new actor.

```

⟨ new_expression ⟩ ::= new ⟨ behavior_name ⟩ ( { expr { expr }* } )

```

This returns the mail address of the new actor created, and denoted by *new expression*. This address is used for communication after being bound to an identifier. This binding can be accomplished by a **let** binding. This is the only form of assignment supported.

```

⟨ let_binding ⟩ ::= let id = ⟨ expression ⟩ { and it = ⟨ expression ⟩ }*

```

This binding gives the actor an address of another actor, but no details of the actor's data and process structures. Creating tasks is done by sending a communication to other actors. This is accomplished by the **send** command.

```

⟨ send_command ⟩ ::= send ⟨ communication ⟩ to ⟨ target ⟩

```

The communication is a list of values, and the target is an identifier bound to the mail address of another actor.

The acquaintance lists and communication lists are instances of *parameter lists* which have the form:

```

⟨ parameter list ⟩ ::= { id | ⟨ var_list ⟩ } | { id | . ⟨ var_list ⟩ }* | ε
⟨ var_list ⟩ ::= case ⟨ tag-field ⟩ of { variant }i end case
⟨ variant ⟩ ::= ⟨ case_label ⟩ : ( ⟨ parameter list ⟩ )

```

Two more constructs: **call** and **reply** can be added to model communication between actor configurations. The **call** construct is used to request some input from an

actor, and is written as

$$\text{call } g \text{ with } k$$

where g is an identifier bound to a process, and k is a communication to the actor at g . The actor at g can *reply* by using $\text{reply } k$ to construct

$$\text{reply } k \text{ with } [value].$$

7.4 Example: Riding the Elevators

There are two basic objects: the elevators and the floors. The behavior of this system is represented as a set of states of the elevators and the buttons on the floors. The actor representing an elevator will describe the behavior of that elevator. There will be n such actors, one for each elevator. Figure 7.1 gives an example of such an actor. We have omitted a lot of detail, but this does not affect the design of the actor. Each actor gets a message from a message network. This network too can be modeled as an actor. Messages are read off this network in a by elevator actors. The exact concurrency control protocol to be used will be determined later. The elevator, thus is defined by a tuple $\langle \text{state}, \text{request.list}, \text{floor} \rangle$. The *create-elevator* actor will create all the elevator actors. Elevator actors create new actors to service the requests as they appear. Here the processing is sequential.

This type of definition allows the implementation strategy to be enforced later. A number of invariants can be defined along with the conditional statements. These invariants tighten the specification by restricting the scope of function execution. Some of the invariants are:

- If elevator is in *waiting* state, then the buffers are empty.
- If elevator is in *up* state, then *requestID.request* is a list that is in increasing order.
- If elevator is in *down* state, then *requestID.request* is in decreasing order.

Clearly these invariants can be added as conditionals to the above actor definitions. Every change in state brings in a new configuration. In this case the other actor, the

```

def elevator-i
  ...
  if service-requests = empty then
    elevator-i.status = waiting
    call button[status]
  fi
  if service-requests NOT = empty then
    go.requestID = new (go) [ requestID ]
  fi
  ...
end def

def go (requestID)
  [ case requestID.request of
    up :      service-request = become ( serviceup ) [request] ;
    down :    service-request = become ( servicedown ) [request] ;
    floorup : service-floor = become ( servicefloorup ) [request.flstatus] ;
    floordown : service-floor = become ( servicefloordown ) [request.flstatus] ;
  end case ]
  ...
end def

def serviceup ( request )
  ...
  requestID.request = serviced ;
  send de-illuminate to button.request ;
end def

def servicefloorup (request, floorstatus)
  ...
  if floorstatus = serviced then
    exit
  else
    requestID.request = serviced ;
    send de-illuminate to floor.request ;
  fi
end def

```

Figure 7.1. A template of an actor definition for an elevator behavior.

button is interacting with some external actor via a receptionist. We shall not give the details of these two actors as they do not make a major impact on the design.

A change in the state of an actor is denoted by the change in an actor's behavior pattern. The actor specifies a new replacement behavior which now processes the incoming messages. The elevator actor has other messages coming to it. These messages are requests for servicing some jobs. These messages also cause the change in state (represented by a replacement behavior). In Figure 7.1, (though it is not shown) the new replacement behavior for *go* will take care of actually servicing the requests. The new actor too will process similar messages. There are a number of replacement behaviors for this actor configuration.

- Changes in state will cause different behaviors.
- The *go* behavior will cause a change in the status of the buttons.
- A change in the buffer sizes will cause a new behavior, in that the floor value in the state of the elevator changes.

Such a definition of participating actors is activated by a main actor called *create-elevators*. Such an actor will create all the *elevator-i* actors with appropriate properties on the receipt of a start signal from the environment. A definition of such an actor is given in Figure 7.2.

The other two object classes of importance are the buttons in the elevators and on each floor, and the floor requests. A button actor defines the initial behavior (a manifestation of state). The change in behavior is a result of an external event (a communication from the environment). The button actor is defined in Figure 7.3. A button has a state of being *on* or *off*. It also has a *floor* value associated with it. We differentiate between the button in the elevator and the one on each floor by using *button* and *floor* actors respectively. Each elevator will have a button actor. In Figure 7.4, we list the floor button actor.

The replacement behavior for a button will be a change in processing of the *button-i*. There are three behaviors possible: do nothing, turn button off, or turn button on. The behavior of the floor actor will depend on the status of that floor serviced. This actor also sends out messages to signify a new event.

```

def create-elevators (N)
  ...
  if START_SIGNAL then
    if N < n then
      address [N] = new ( elevator-N );
      let N = N + 1 ;
      become create-elevators ( N )
    fi
  fi
  ...
end def

```

Figure 7.2: An actor definition for the main actor: *create-elevators*.

A note on the fairness concerns expressed in the problem definition (Section 2.1). The fairness issue is that of the delivery of communication, generated as a result of processing messages at each actor. The actors formalism supports a non strict fairness mechanism (Section 2.4, [Agha, 1986]). This form of fairness satisfies the two requirements properties of the problem. The reasoning is as follows:

1. All messages arriving at a button actor are always sent out.
2. All messages arriving at an elevator actor are always processed.

Therefore, as no priorities have been assigned, all requests for elevators from floors will be serviced eventually, and all requests for floors from within elevators will be serviced eventually.

7.4.1 A Discussion of Actors

A computation in the actor model is made up of a collection of configurations. Each configuration may have a *receptionist* (an interface actor), and an *external actor*. A problem is solved by this collection of actors by moving from one configuration to another, in the process interacting with other configurations.

In this section we briefly present a few salient points about actor languages. We shall use the criteria we developed in Section 3.2.

```
def button-i (request)
  ...
  [ case button.status of
    on : if pressed then
      do.nothing fi
      if served.floor then
        button.status = off fi
    off : if pressed then
      button.status = on fi
    end case ]
  ...
  if request-is-send-status then
    reply [status]
  fi
  ...
end def

def button (request)
  ...
  if request-is-send-status then
    repeat
      new (button-M) [request]
    until all-buttons-created
  fi
  ...
end def

def floor (request, status)
  ...
  if changed-status then
    floor.status = new-status :
    repeat
      reply elevator-M [request]
    until all-messages-sent
  fi
  if request-status then
    reply [status]
  fi
  ...
end def
```

Figure 7.3: A template of an actor definition for a button.

- **Notational Features**

- *Unit of computation:* An actor is the basic unit of computation. An actor is a dynamic unit of data and process representation.
- *Data abstraction:* Data abstraction is provided by total encapsulation of behavior methods inside an object definition. The only visible components of an object are defined by the object itself by means of new actors. For example, all elevator actors have similar actor definitions.
- *Modularity and Compositionality:* The system of communication between actors is very flexible. Modularity is introduced by the concept of *receptionists* which are actors that act as interfaces to a particular configuration. All communication is then routed through this actor which then decides the best mechanism of handling messages. A host of receptionists define module interfaces. Compositionality in actor systems is defined by communication primitives. Since an actor system is a collection of (possibly interacting) agents, it is easy to define a configuration as a composition of two configurations. Rules for such composition are defined to admit many configurations (see Chapter 7 [Agha, 1986]).
- *Constraints:* Actor languages provide necessary constructs for defining constraints on actor behavior. The fairness constraints are naturally modeled in actors, as the fairness of actors formalism applies to the example's fairness requirements.
- *Transitions:* An execution is a collection of configurations, each of which has evolved from a previous configuration. A change in the state of either an elevator actor or a button actor triggers a new configuration.
- *Concurrent structures:* Message-passing is the basic feature of actors. Communication between configurations is managed by *receptionists*, communication between individual actors is defined by the target addresses.

- **Semantic Features**

- *Defining assertions:* Assertions in actor languages can be defined as constraints on the behavior of collections of configurations. This allows reasoning about behavioral equivalences of configurations.

- *Reasoning with time*: Series of configurations maintain a trace of the execution. There is a clear description of the occurrence of events in terms of evolution of actor configurations. The transition semantics defines the evolution of configurations and provides a framework for reasoning with time.
- *Defining provable properties*: The formalism does not admit any specific assertional formalism, and therefore a property defined in an actor language can be verified only by a reasoning about the execution sequences generated. As yet a theorem prover has not been defined for the actors model.

7.4.2 Drawbacks

The actors formalism represents a paradigmatic shift in software development methods. The approach is based on a collection of functional objects that communicate using message-passing. There is a restriction on the type of communication and the type of architectures upon which this model can be implemented. For our definition of a distributed system, this model is adequate for the development of software systems. The lack of an insight on the type of support tools necessary to make this model of computation a widely usable one makes visualization of the ability to develop correct programs a little difficult.

Another issue is the lack of direction on the syntax of actor languages. Such actor languages should have more structure than is provided by the SAL language. The structure of the software can be expressed better with provision of sophisticated data types.

7.5 Chapter Summary

We have summarized the actors formalism for distributed computation. This model provides a formal framework for development of specification systems. The formal model is based on message-passing between communicating agents. Actor specifications allow behavior definitions to be represented explicitly. Dynamic behavior is modeled as a change in the behavior of an actor caused by the a message received by the actor. Transactions are modeled by *configurations* in actor specifications. The actors formalism

is abstract enough to be applied to the design and implementation stages of software development.

The main issue of concurrency and inheritance (see Section 7.3.2) can be addressed in a number of ways. One way is to keep the synchronization specifications separate from specifications of other actions.

The issue of reuse in software is capably addressed using object-based method. A number of research papers [Farumi et al., 1988; Gossain and Anderson, 1990; Johnson and Russo, 1991] discuss issues of reuse and how object-based methods allow reuse of specification, designs and code.

A number of programming languages have been designed to write concurrent programs. [Yonezawa and Tokoro, 1987] presents several languages as examples: POOL, ABCL, ConcurrentSmalltalk, etc. There are other object-based languages like COOL [Schwan et al., 1986], PO [Corradi and Leonardi, 1990], Flame [De Paoli and Jazayeri, 1990], Trellis [Moss and Kohler, 1987].

Chapter 8

Capabilities of Specification Methods: An Evaluation

We have analyzed four different methods of specifying concurrent systems. In this chapter we present an evaluation of these methods. We also present some results at the end of the evaluation. This forms the basis for the development of a framework for developing specification languages for distributed systems.

8.1 The Evaluation

We present a comparison of the methods we have surveyed so far. The comparison will be done based on the criteria identified in Chapter 3 and repeated below for convenience.

- Expressiveness:
 - Data and process structures,
 - Structures defining constraints on processes and data,
 - Structures defining transitions of processes, and
 - Concurrency and communication structures.
- Modularity and Compositionality.
- Verifiability:
 - Reasoning with time.

- Specification and verification of properties, and
- Semantics.

We present the comparison in a tabular manner with the methods appearing in the order surveyed. In the ensuing sections we keep the discussion as general as possible, but sometimes refer to a specific system when our comments are appropriate for that system and not the method in general.

8.1.1 Expressiveness: Data and Process Structures

All the methods surveyed use some form of object representation to encapsulate information; however object-based and algebraic methods provide better frameworks for encapsulation of information. Data abstraction in most object-based methods is provided by procedural abstraction, whereas, algebraic methods provide data abstraction based on a mathematical foundation of abstract data types [Cook, 1990]. Process structures are best presented in algebraic methods, in which labelled transition systems define the semantics of interactions between processes. Other methods have process representation notions which are new and non-conventional. For example, in the actors formalism, processes are active actor objects. All events take place based on the message passing. Graph-based methods provide a visual picture of the interaction between different processes. Such representations are dependent on unconventional notation. Logic-based methods are the least expressive and have to depend on some representation for the abstract data types for effective data representation. Hence algebraic methods like Lotos have the best mechanisms for data and process representation. A common feature of all four methods analyzed is their support of a declarative approach to representation. Table 8.1 presents the data and process abstraction features of the four different approaches to specification.

8.1.2 Expressiveness: Specifying Constraints

Constraints specification is an important component of good design methods. Different kinds of constraints can be specified. For a specification language which has objects, methods, transitions, and modules as building blocks, constraints can be specified on

Spec Methods	Data and Process Structures for Abstraction
Algebraic Methods	Algebraic methods combine representations for data types with a process definition. They provide means to define data and process abstractions within a unified framework of abstract data types and process algebras.
Logic-based Methods	Logic specification structures are dependent on some adhoc extension to the declarative representation to incorporate data definitions. For example, UNITY provides pascal-like data declaration facilities. UNITY processes are program fragments which have associated declarations and properties.
Graph-based Methods	Data and processes are represented as graph objects. Process interactions are modeled as connections between process and data objects. For example, statecharts use rounded-square figures to represent data objects and encapsulation of data. Abstractions are realized by imploding and exploding each unit in a graph.
Object-based Methods	Object structures provide encapsulation of data and associated processes. Objects interact with each other using message passing.

Table 8.1: Specification methods and associated representation structures: Data and process structures.

each of these entities. Table 8.2 gives an overview of the facilities provided by each specification formalism. The language used to specify constraints is predicate logic or a subset of predicate logic. All the methods surveyed have a rich set of logical operators to specify constraints. The best mechanism is provided by logic formalisms for specification. Logic specifications can represent quantification succinctly whereas this is difficult or impossible to do so in other formalisms.

Spec Methods	Specifying Constraints on Data and Processes
Algebraic Methods	Algebraic methods provides an extended set-theoretic logic for specification of constraints. In most specification methods in this group no explicit constructs are provided, and constraints are a part of the specification.
Logic-based Methods	First order logic operators provide the necessary functions to specify constraints. This is augmented with modal (temporal) operators to specify and prove time-variant properties of specifications.
Graph-based Methods	Constraints are defined by labeled arcs between objects labelled with predicates. The nature of assertions depends on the type of implementation, but is mostly a subset of first order logic.
Object-based Methods	Constraints between objects are specified by the interface of each object. The assertions use relational and logical operators that are usually a part of most modern programming languages. Specification formalisms like actors augment the operators with quantification.

Table 8.2: Specification methods and associated representation structures: Specifying constraints.

8.1.3 Expressiveness: Specifying and Recording Transitions

Table 8.3 compares methods based on their ability to specify state transitions. Transitions in a concurrent system are important and are the main mechanism to reason about the dynamic behavior of concurrent software. Some formalisms provide specification mechanisms for transitions indirectly, for example, object-based methods. Actors specify transitions via the configurations feature, and hence provide the powerful mechanisms to reason about such transitions. Graph-based methods provide a visual representation of transitions via the state transition representation, and typically have adequate facilities to reason about the dynamics of the software specification. Logic-based methods view transitions as linear (or non linear, depending on the underlying temporal logic) sequences of events. But UNITY does not provide any facility for specifying transitions. The availability of facilities in logic-based methods is dependent on the type of reasoning mechanism implemented. Algebraic methods do not provide any explicit constructs, but allow reasoning based on the concept of bisimulation (see Section 4.1.4).

Spec. Methods	Specifying State Transitions
Algebraic Methods	No specific constructs are available, but tools exist. For example, Lotos has tools to recognize ordering of events.
Logic-based Methods	Logic-based methods use lists to record transitions. The organization of these lists is based on the underlying temporal logic. But UNITY does not have any facility to reason about transitions, as, transition representations are implicit and are very constrained due to the nature of the programming logic (5.4.2).
Graph-based Methods	Transitions are represented visually as arcs in a graph representation. Most graph-based methods use some form of assertions to label these transitions, providing a mechanisms to constrain transitions.
Object-based Methods	State transitions are difficult to model in object-based methods. But actors provides a good mechanism to represent transitions as relationships between <i>configurations</i> .

Table 8.3: Specification methods and associated representation structures: Representing state transitions.

8.1.4 Expressiveness: Concurrency and Communication Structures

Explicit communication between interacting processes must be modeled in concurrent software specifications. Concurrent processes must be represented explicitly. Table 8.4 discusses the communication representation facilities in the methods we have discussed.

Algebraic and logic-based methods provide explicit constructs to represent concurrent structures and communication events. These representations are therefore the most versatile of the four methods. Object-based methods do not explicitly state concurrency in a software system. But it is possible to deduce implicit concurrency by an analysis of the static structure of the system. For example, Actors has a built-in message passing mechanism but not structures to specify concurrency. The situation is similar in graph-based methods. Concurrency is implicit and communication is modeled as events between objects/processes. An exception to this is the statecharts which represent concurrency explicitly, but do not have mechanisms to represent communication, except through the specification of dependent variables or connecting arcs.

Spec. Methods	Facilities to represent Concurrent Structures
Algebraic Methods	Process algebra provides a very rich set of constructs to model concurrency. Communication is modeled by dependent variables.
Logic-based Methods	Logic-based methods provide constructs similar to those provided by algebraic methods. Communication is modeled by shared variables, and guards provide synchronization points.
Graph-based Methods	Communication between objects is visually represented using arcs and concurrency is not always explicitly represented.
Object-based methods	Message passing primitives exist to describe communication between objects. Concurrency is not explicit.

Table 8.1: Specification methods and associated representation structures: Representing concurrent structures.

8.1.5 Modularity and Compositionality

Table 8.5 compares methods based on their ability to develop software specifications as interacting modules. Modules provide a mechanism of programming large information systems. Algebraic methods allow compositional and modular development as they are based on a formal theory of modules. For example, Lotos **specification** structure provides the ability to specify distributed systems as compositions of modules. Lotos also provides a compositional algebra based on labelled transition systems to deduce properties of compositions of modules from properties of individual modules.

All object-based methods provide mechanisms for modular and compositional development of concurrent systems. Actors provide a flexible method for modular development of software. Modules in Actors consist of loosely connected actor configurations. Interfaces between such groups are defined by the receptionist actors. Actors

also provides a set of rules to deduce properties of compositions of configurations from properties of individual configurations. Therefore verification of compositions is easy in Actor systems.

Logic-based methods lack modular software development facilities. Compositional development of concurrent process is supported at a small-scale. For example, UNITY provides a compositional view of interacting processes by way of *programs*. Collections of programs form compositional elements. Composing UNITY specifications from smaller units of specification is done using a set of rules.

Graph-based methods do not have any explicit constructs to allow for modular specification of software systems, but modularity can be implemented as an abstraction on related graph objects. One basic problem with graphs is that objects at different levels can communicate. In general, this may not support the development of self-contained units of software. Statecharts, however, provide a graph-based abstraction mechanism for modular specification via the explosion of graph objects.

Of the four methods, compositional specifications are visualized most easily in statecharts. This is because the correctness of compositions is easily verified using the rules for transformations of statecharts. Algebraic methods have a formal basis to both modularity and compositional specification development, and hence provide the most rigorous method of software development.

8.1.6 Verifiability: Reasoning with Time

Logic-based methods normally do not combine temporal information with specifications of data and process structures. The elements for reasoning about time are separate, and apply to the specifications of data and processes. For example, UNITY does not provide specification structures for representing temporal information, but has an elaborate logic for verifying time-variant properties of UNITY specifications. Logic-based methods provide the best mechanism for reasoning about time. Algebraic methods provide an event ordering mechanism which is supported by the process algebras. Not all object-based methods have notions of time; however Actors provides this by the history mechanism. Each actor transition has a time of generate and associated information available within itself. This provides a powerful mechanism to reason about transitions as configura-

Spec. Methods	Constructs for Modular Specifications
Algebraic Methods	A formal theory of modules serves as a basis for modular and compositional software development. For example, in Lotos, the specification and process constructs allow a software system to be decomposed as a set of interacting processes.
Logic-based Methods	Do not support modular and compositional development of software. However, UNITY provides such mechanisms on a small scale. The view of a UNITY specification is similar to the view of a Pascal program. But composing specifications in UNITY is easy because of the ability to design interacting program fragments.
Graph-based Methods	Modular development is not supported by most graph-based methods. Statecharts provide an abstraction by using different statecharts for different modules. Composing these modules into a system is done by another statechart.
Object-based Methods	Most object-based methods support modular specifications, but tend to do so in a small scale. This is due to the increase in complexity of processing with increase in the hierarchical density. But actor systems achieve modular development by way of collections of actors in a configuration.

Table 8.5: Specification methods and associated representation structures: Representing modules.

tions of actors define a hierarchy of transitions over a period of time. Each actor records the time of creation and events that happen subsequently. Table 8.6 summarizes the facilities provided by the four specification methods.

Spec. Methods	Reasoning with Time
Algebraic Methods	Algebraic methods provide time ordering mechanisms that allow reasoning about the ordering of processes.
Logic-based Methods	Logic-based methods provide separate logics for representing and reasoning about time-variant properties of specifications.
Graph-based Methods	Graph-based methods do not have temporal reasoning mechanisms.
Object-based Methods	Object-based methods do not provide any temporal reasoning mechanisms. But actors provides this mechanism in an indirect fashion, via the configurations mechanism.

Table 8.6: Specification methods and associated representation structures: Reasoning with time.

8.1.7 Verifiability: Specifying and Verifying Properties

Table 8.7 compares the methods based on the facilities to specify and deduce properties of concurrent specifications. Of the methods surveyed, logic-based methods provide excellent facilities for specifying and verifying properties of designs. Logic-based meth-

ods provide verification mechanisms separate of specifications. Algebraic methods also provide good facilities. Object-based systems do allow assertions but do not a logic for specification and verification of properties. Graph-based systems do not have explicit mechanisms for such reasoning. But, statecharts provides tools for verification of properties. Properties are defined as assertions that should be true on incoming and outgoing arcs on objects in a statechart. Graph grammars and petri nets do allow specification and verification of properties of software systems, but lack mechanisms to define structure of a system. For example, the lack of variables complicates the process of specifying any property of a system for verification.

Spec. Methods	Specifying and Verifying Properties
Algebraic Methods	Algebraic methods provide mechanisms for deducing properties of concurrent systems. These mechanisms are based on process algebras.
Logic-based Methods	Logic-based methods provide temporal logic based mechanisms to reason about the behavior of specifications.
Graph-based Methods	Verifying that a particular property holds is not possible directly without augmenting the graph representation with some form of marking.
Object-based Methods	Assertions can represent properties of an object based system, but there are no facilities yet to deduce them.

Table 8.7: Specification methods and associated verification support facilities. Specifying and verifying properties.

8.1.8 Semantics of Specification Formalisms

Table 8.8 compares the different methods based on the underlying semantics. The type of semantics is important in understanding the expressivity and power of the logic provided. For example, transition systems are the simplest of representations for concurrent systems [Hennessey and Milner, 1985]. UNITY is the most limited form of transition systems. Hence, though UNITY has a comprehensive proof system, and in most cases it is able to design any type of concurrent system, it comes up short for dynamic and reactive systems. In the elevator example, the parameters N and M are not changeable in UNITY. Lotos is a good example of a system based on transition semantics. The transition semantics are a good model for concurrency, but like UNITY are inadequate when trying to verify dynamic systems. Lotos has the ability to specify dynamics systems basically because of its ability to provide abstraction at many levels.

Graph-specification methods use different semantics models such as graph grammars

and petri nets. Statecharts are based on extended state-transition automata. These automata are extended with nondeterministic features adding a power of expression beyond that of simple finite-state automata. The statecharts have a semantics based on graph structures. The statecharts also are modified versions of hypergraphs.

Most object-based methods are imperative and there are a number of efforts to define semantics for object-based languages. For example POOL admits layered semantics [America and Rutten, 1990], whereas Actors' semantics are based on message-passing schemes. Actors' semantics [Agha, 1990] appear to have been developed independently of other object-based approaches. The semantics are more powerful than the transition systems as they are able to specify and explain dynamic configurations of concurrent systems. In addition, all actor objects are first-class actors, thus providing a uniform treatment of actor structures.

Spec. Methods	Semantics of Specification
Algebraic Methods	Semantics for process specifications in algebraic methods are given by labelled transition systems. Data representation semantics are defined by initial algebras.
Logic-based Methods	Verification semantics are based on temporal logic. UNITY is still limited and its semantics are based on extensions of Floyd/Hoare axioms to parallel designs. They essentially are limited transition systems.
Graph-based Methods	The semantics of graph-based methods are based on either graph grammars or petri nets or graph event structures (extended finite-state automata). Statecharts semantics are based on graph event structures and extend transition system semantics in the area of dynamic software specifications.
Object-based Methods	One way of defining semantics of object-based systems is layered transition systems. But actor systems are based on object algebras and are not conventional.

Table 8.8: Specification methods and associated verification support facilities: Semantics of specifications.

8.1.9 A Discussion of the Evaluation

We have concentrated on the power of expression of the specification formalisms presented in the previous chapters. We also have discuss these methods with respect to their ability provide verification frameworks. In this section we elaborate on the descriptive powers of these formalisms. There are two other criteria which have not been

presented so far¹: *availability* and *architecture independence* of the methods. After a discussion on the descriptiveness of these methods we present a brief discussion of the four methods with respect to these criteria.

Descriptive Simplicity

Graph models of computation are based on *graph reduction*. A computation in such a model is purely declarative and hence is totally independent of the implementation architecture. Communication and concurrent aspects of an application are modeled implicitly and hence recognized dynamically. A major problem is in the cost of implementing such a model. Statecharts overcome this difficulty, to a great extent, by having a well-defined set of transformations from one level of description to another.

Lotos is based on a transition system. Researchers in Lotos have developed a set of tools that provide a well-defined set of transformations from one level of specification to another. The expressive power of Lotos is less than graph-based models such as statecharts or graph grammars, because of the inability to specify efficiently dynamically changing systems.

UNITY provides a Pascal-like syntax to a highly declarative mechanism for representing specifications. The specifications essentially consist of guards for statements. UNITY scheduling is fair, therefore, all statements are chosen for execution eventually. UNITY is expressive enough to model most concurrent and distributed application. Like Lotos, it is based on a limited form of transition systems and hence is not expressive enough to model dynamic systems.

Since Actors is based on message-passing, the concurrency model is more restricted than corresponding models of other methods. But Actors is more expressive than either Lotos or UNITY as it can specify dynamically evolving systems. This is due to its ability to represent configurations of evolving actors. The formalism has a formal framework for the compositional development of specifications. Actors also provides guidelines of implementing (and increasing) concurrency by the use of *futures*.

¹These do not present any new information on the expressiveness of the formalisms but provide insights into the features a specification formalism should have.

Architecture Independence

None of the four formalisms we have studied assume a base architecture, like a SIMD model or a distributed environment. For a specification, Lotos and Statecharts use tools to transform it to target architectures. UNITY does not have such tools, but provides rules for transformations. Since the Actors formalism is based on a message-passing model, it is difficult to implement, because at present there is a lack of efficient methods to provide object management on a distributed or parallel architecture. Efforts are underway to implement actor systems on distributed architectures [Agha, 1991].

Availability of the Formalism

Of the formalisms we have studied, only two (Lotos [Diaz and Vissers, 1989] and Statecharts [Harel et al., 1990]) have been implemented as a part of software development environments. Distributed implementations for actors are being built [Agha, 1991] and a couple of prototype UNITY implementations have been completed. Proof systems have not been developed for Actors and UNITY; whereas, Lotos and Statecharts have tools for verification of specifications.

8.2 Chapter Summary

In this chapter we have evaluated four of the most popular methods of specifying concurrent systems. As a result of this evaluation we have a number of formalism independent observations, as listed below.

- There are two main components of a specification formalisms: notational aspects and verification aspects.
- These components must be provided within a semantic framework.
- The notational and verificational aspects are related, but need not be closely coupled. Tools for notational development and for verification of the specifications may be separate.
- Specification formalisms should be declarative, i.e., architecture independent, and preferably supported by tools for empirical evaluation.

- Specification formalisms must support an evolutionary model of software development.

As a result of the evaluation, we found two methods, Lotos and Statecharts have tools that allow extensive experimentation for an empirical evaluation. We also found Lotos to have the most extensive support for specification and verification of complex concurrent systems. UNITY and Actors do not yet have tools and systems built around themselves.

Our survey and evaluation reveals that, ideally, a framework for specification formalisms for distributed systems should extract the best concepts from all the four groups of methods. Such a framework should address the following features.

- Object Structures for representation,
- algebraic or graph-based semantics,
- proof rules and inference procedures,
- graph-based analysis methods wherever feasible,
- temporal logic-based analysis methods to verify safety and liveness properties, and
- knowledge-based support for storage and retrieval of design representations.

Ideally, such a framework should support an incremental transformation model of concurrent software development and also address the issue of reuse of designs and code. We shall discuss such a framework in the next chapter.

Chapter 9

A Proposed Framework for Design Representation Languages

9.1 Introduction

In this thesis we have assumed the **IncTr** model of software development and surveyed software specification methods that support this model. Based on the discussion in Chapter 2 and the evaluation in Chapter 8 we propose, in this chapter, a framework for design specification languages for distributed systems. First we present the different phases of development addressed by the framework and then discuss the functionality of tools needed at each phase. These tools support transformational development of distributed systems. We then describe some problems in the area of distributed software systems development before summarizing the work presented in this thesis.

9.2 Specification Formalisms

Based primarily on our survey, we conclude that the following components are needed to support distributed software design assuming the *IncTr* model for systems development.

- Tools relating requirements specifications with design specifications.
- Design specification language constructs for describing

- data and process abstractions,
 - modular compositions,
 - constraints, assertions and exceptions,
 - transactions, and transitions that take place as a result of the transactions,
 - constructs to specify concurrency and communication.
- Tools for verifying specifications.
 - Tools relating design specifications to implementation constructs.

In the following subsections we elaborate on these four points. We develop a notational framework for design specification languages based on the semantics associated with transformations between different representation systems. For this purpose we will have to consider two other representation systems: the requirements specification language and the implementation language. In most cases, tools provide the semantics of transformations. We present a discussion of how such tools may be realized.

9.2.1 Tools Relating Requirements to Designs

We can relate requirements to design only if there is a common semantic basis between the two specification notations. If the same language is used to specify requirements and design such as Lotos or UNITY, then the language provides facilities for refinement of specifications. If two different languages are used (as in Telos and TDL [Mylopoulos et al., 1990; Borgida et al., 1991]), there is need for tools to transform requirements to designs. In either case, a transformation involves

- translating data and process structures in the requirements modeling language to corresponding constructs in the design language,
- verifying correctness and completeness of translations,
- translating/reformulating behavioral constraints and assertions, and
- verifying completeness of translation of constraints specifications.

Most data and process representations can be translated to similar structures in the design notation, but any change in the representation of object between requirements

and design specifications will involve not only a shuffling of the scope of any constraint expressions, but also result in generating new object descriptions and constraint expressions to specify design. This is a hard task as it is difficult to prove the equivalence of the structures before and after transformation. The difficulty would arise from the lack of a common model for computation for the two formalisms. Transformations normally involve self-contained modules in a complex system.

One important function of tools that map requirements to design is to identify implicit concurrency in a system description. This assists in defining implementation strategies at a later point in the development.

9.2.2 Design Specification Language Notation Considerations

The basic representation facilities must include the ability to specify data objects and their relationships. Based on the survey and evaluation, we believe that specifications should be able to model at least the following entities.

- *data objects*, which have the ability to represent abstract data types,
- *transaction definitions*,
- *legal transitions*,
- *function definitions*,
- *concurrency* (both communication and parallelism),
- *module definitions*, and
- *interfaces* between modules.

The specification should be declarative to allow architecture independent representation. One technique of providing a declarative scheme for objects and operations on objects is to define operations to be (active) objects. This allows processes to be specified declaratively. For example, operations can be defined as constraints for a particular object (as in Telos [Mylopoulos et al., 1990]).

The representation should also be able to define *constraints*, *exceptions* and *assertions* on these objects. Assertions and exceptions differ from constraints only in the

scope of application. The following types of constraints and assertions must be specifiable.

- Data Integrity constraints: which preserve the consistency of data representations.
- Behavioral constraints on transactions: invocation, sequencing and synchronization constraints.
- Constraints on transitions.
- Constraints on communication.
- Constraints on modules and their invocation.
- Assertions about objects listed above. These assertions will specify *properties* of the objects and object compositions, and may include the following.
 - Precondition, post condition and invariant assertions.
 - Exceptions. These are normally a part of assertions and constraints on transactions, transitions, communication specifications and modules.

The important issue is that these facilities must be provided within a formal framework. We use the statecharts formalism to model such a framework. Specifications in this framework are made up of a number of module specifications (Figure 9.1). This framework views structures for data and process representation and verification as two different but closely coupled entities. The module interface definition defines the objects that are public to other modules. It also defines a set of transactions that can be used by other modules to access objects in the current module. Figure 9.2 defines the structure of a module interface.

The basic object supporting compositional development in such a framework is a module. As depicted in Figure 9.3, a module consists of a number of object definitions along with function and transaction definitions. Concurrency structures can be specified as a part of object definitions, transaction definitions and function definitions.

Object state changes are modeled by transitions, and are therefore defined as a part of object definitions. A module definition can define local variables for the purposes of intra-module synchronization, or just for transmitting results to the external environment or other modules. Functions may operate on these variables and are therefore

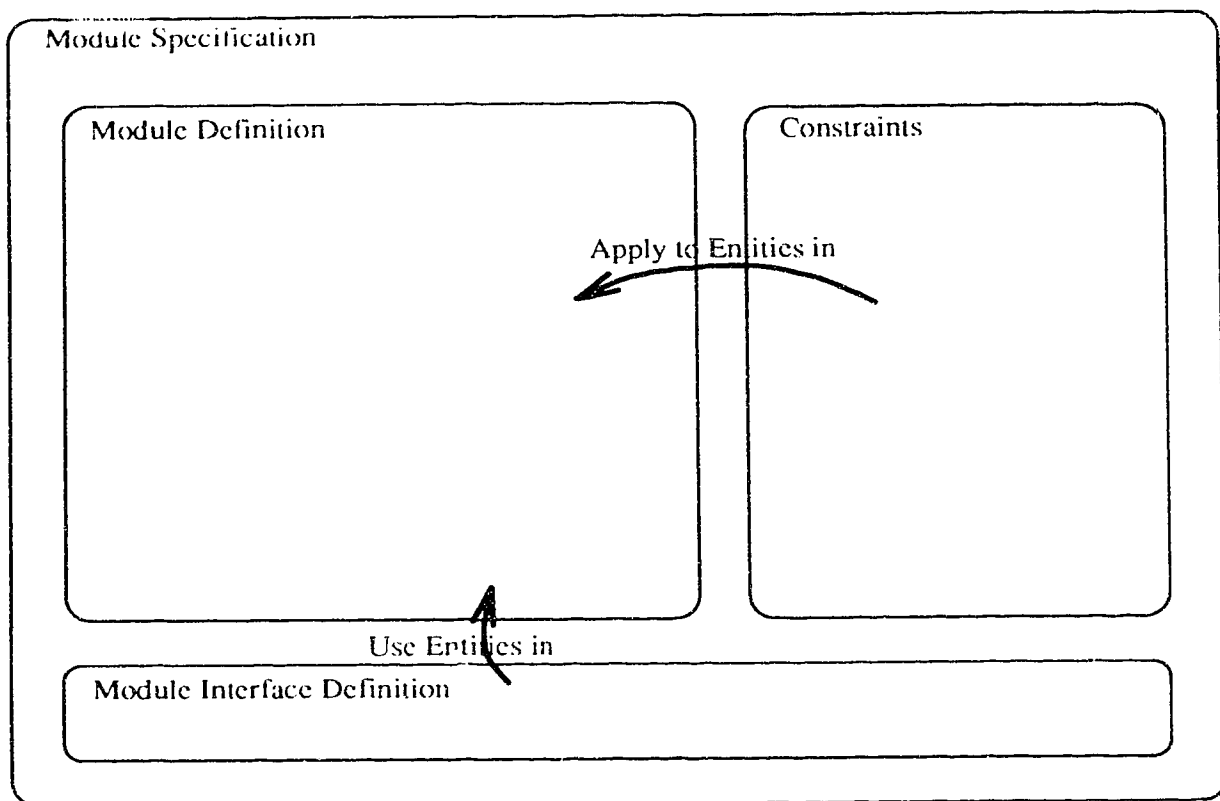


Figure 9.1: Essential features of a formal design specification language and their relationships.

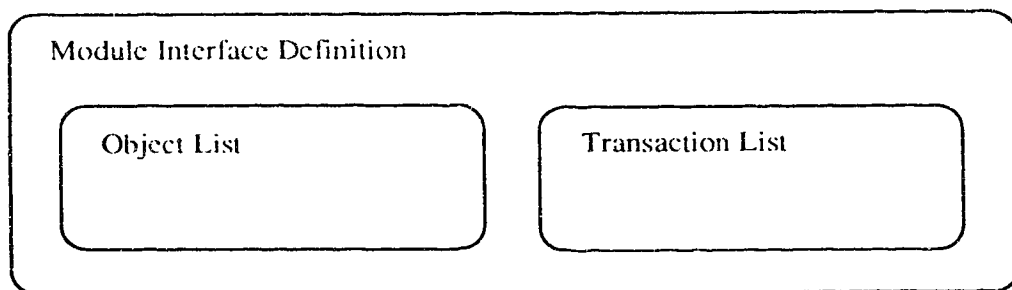


Figure 9.2: Module interface definitions.

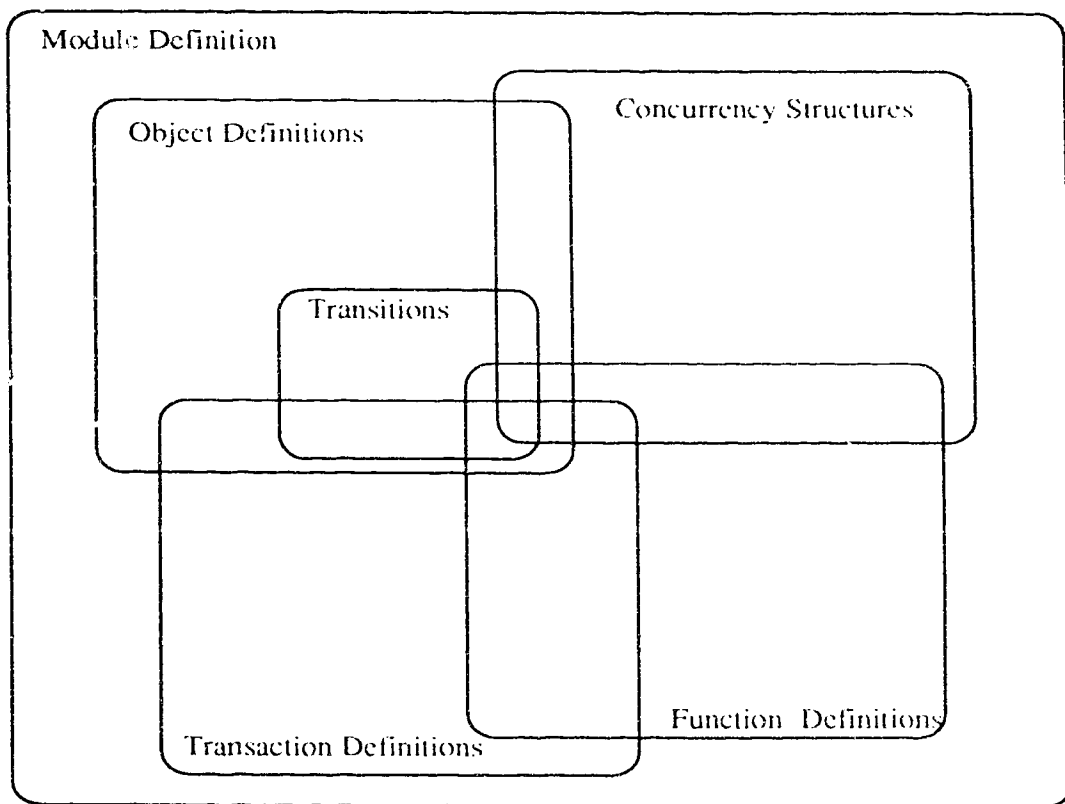


Figure 9.3: Module definitions.

not entirely a part of object descriptions. Functions may also be encapsulated in object descriptions, limiting their scope to the object. Transactions cause state transitions and such transactions are defined as a part of object descriptions. Transactions can also change the state of the module; such transactions are defined as a part of the module definition. Concurrency structures allow description of concurrency and communication between objects within a module and between modules. CSP-like communication primitives are adequate for describing communication. Possible concurrent constructs that should be supported are: *parallel composition*, *nondeterministic OR* and *general choice* (see Section 4.1.3).

The constraint component of the framework assists in the refinement of specifications. The assertional language is normally a subset of first-order logic, augmented with temporal operators to allow specification and verification of formal properties of designs. Exceptions are a part of the assertional language and apply to all the data and process structures. Constraints can be imposed on functions by preconditions and postconditions. Transactions, communication and concurrency constructs may allow exceptions and other forms of constraints. Though Figure 9.1 shows them to be two separate entities, constraints are normally specified as part of the definitions of objects, functions, transitions, transactions, concurrency constructs and modules. Figure 9.4 defines the relationships between constraints, assertions and exceptions.

9.2.3 Tools for Verifying Specifications

One way of developing quality software is to analyze and test the specifications before implementing. Specifications can be checked for notational consistency ensuring that an implementation will be based on a consistent document. Below we present some symbolic checks that need to be performed on specifications.

- Data type consistency: References to data types and their definition must be consistent.
- Consistency between defined object states and object transitions.
- Objects and modules that have been defined and not used can be flagged.
- Compositional consistency: Modular compositions do not have conflicting syntactic and semantic definitions.

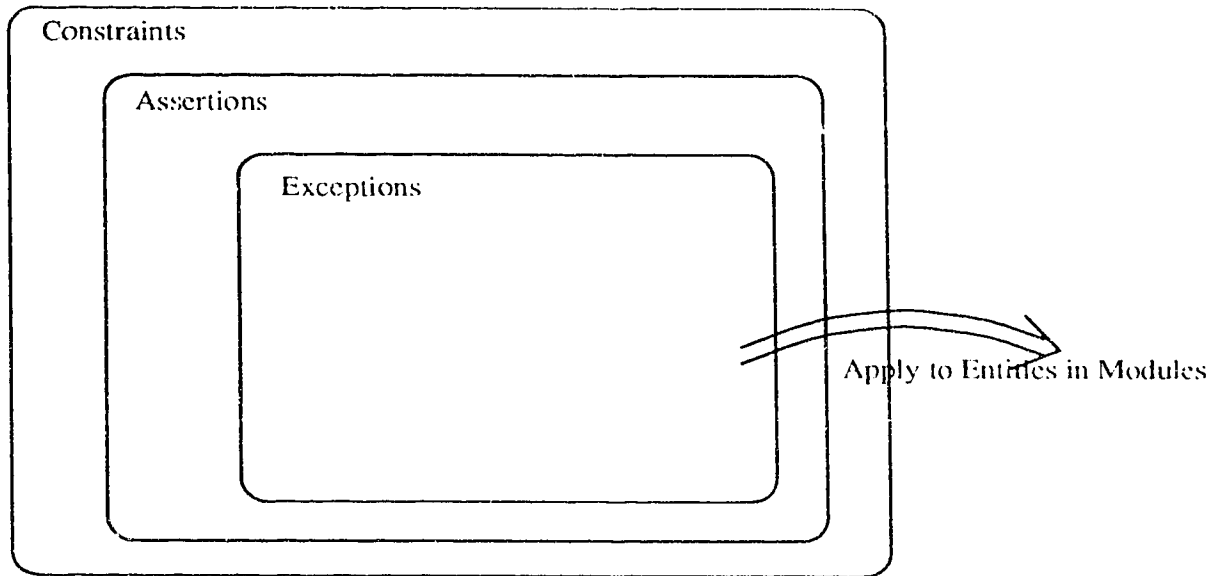


Figure 9.4: Constraints, assertions and exceptions.

These checks on the specification only ensure a notationally correct specification. What is more important is that the specification be functionally correct. To achieve this, we need tools to simulate an execution of the specification. Such a tool must have consistency checking mechanisms built into it. Section A.1 formally defines the notions of *consistency*, *soundness* and *completeness* for a given formal system and used in the process of reification. Some mechanisms for verifying constraints are:

- Verifying general assertions: All assertions on data objects, transitions and modules must be validated. Examples are
 - assertions that are true when an object is created,
 - assertions that are true when any transaction is being activated,
 - assertions that are true when a transition from one state to another is taking place, and
 - assertions that are true before and after a message is communicated.
- Verifying legal transitions.
- Verifying desirable properties of specification components.

To elaborate on the aspect of specifying and verifying properties of concurrent programs, we observe that there is no automatic method for identifying and generating these properties — they have to be defined by the system designer. These properties can be mechanically verified using the proof rules associated with the logical reasoning component. The cost may be prohibitive, however. The refinement of specifications enables the process of verification to be more manageable. This is because the verification is done as the refinement proceeds in small steps. It is easier to define correctness criteria for small refinements.

This type of verification is necessitated by the lack of Hoare-style [Hoare, 1972] axiomatic proof rules for programming. Such rules would enable deductive software development. Until axiomatic proof rules are developed to facilitate deductive synthesis of software, stepwise refinement appears to be the best method of verifying software.

9.2.4 Tools Relating Designs to Implementations

In transformations at the design to implementation stage, the information added is dependent on the problem solving environment; for example, converting the design data representations to data structures for the implementation language. At the stage when we want to convert designs to more implementation dependent code, we are confronted with two transformational issues: *data and behavior specifications*, and *transaction specifications*.

Data and process specifications and associated behavior are not easy to transform to the implementation language structures. This difficulty mostly arises from the difference in the computational models. At this stage what is needed is a tool that will provide a mapping between the computational models of design specifications and the implementation language. Verification of such transformations can be avoided if the function that is modeled by the tool is proved correct.

The transaction specifications tool is more complex as it must model dynamic situations. Since the transactions affect the state of data objects, tools are needed which show simultaneously the state changes in data objects and their behavior, and the corresponding transaction which caused the effect. Such a tool will generate the implementation specific code depending on the type of simulation representations generated.

At the point between the design and implementation stages, we need tools to perform the following functions.

- A data and behavior representation translator, which will add necessary data type information to model data types in the programming language used in implementation.
- A translation system which will take in transaction specifications and the associated assertions and generate representations in implementation code.
- A temporal ordering tool which will produce an ordering of events and module actions of the entire system, so corresponding synchronization code can be generated. This translation can depend on the method to be used for implementing concurrency.

The refinement method of the **IncTr** model helps define the semantics associated with such tools. At the implementation stage verifying the correctness of the transformation tools is sufficient. We typically do not need formal verification tools for implementations, as at that level, testing is the most common method of verifying the behavior of an implementation. Tools to generate test data would of course be helpful from the user perspective.

9.2.5 Tool Support for Formal Reasoning

Until now there have been very few tools that could help a programmer reason about properties of programs. Some such tools are the Rewrite Rule Laboratory [Kapur et al., 1986], LCF [Paulson, 1987], the **B** tool [Abrial, 1988] for proof checking **Z** specifications and the Boyer-Moore theorem prover [Boyer and Moore, 1979]. But these tools need constant input from the user and generally require expertise with the logics of the proof system. What is needed is a set of automated tools which assist in verification. The interaction with such a tool set will simulate the usage of a structure editor. An ideal tool set will have the following mathematical capabilities.

- Ability to extend the assertional language with useful user defined primitives like predicates, functions and variables.

- Ability to postulate and form the *axiom* set for making inferences.
- Ability to abstract and define short hand notations for existing definitions.
- Ability to generate new inference rules from the axioms and already generated rules.
- Ability to refine “informal” proofs (involving graphical or other representations) to formal proofs.
- Ability to develop and apply strategies for verification.
- Ability to collate results into theories about specifications and use them to reason about other specifications.

As yet there are no tools that satisfy the above requirements. Note that the requirements do not specify the type of inference system to be used. Deduction is a powerful means to infer, but induction will also help prove properties of abstract data types. One way of using both is to use Gentzen’s Natural Deduction system (see [Paulson, 1987], pages 46-47) for reasoning and supplement it with a set of rules that define inductive properties of abstract data types that are being used in the specification. The onus of verification of these inductive rules is on the designer who supplied them.

In addition, facilities to store collections of constant declarations, definitions and abbreviations, axioms, derived rules and conjectures, complete and incomplete proofs, and proof strategies are needed to make verification computations cost effective. Such facilities also help in the reuse of *theories*. Certain types of reasoning occur frequently, and it is more efficient to have compiled/coded versions of such patterns. It may be possible to construct a single rule that represents a number of related patterns of reasoning. The type of logics supported will also influence the flexibility and complexity of the reasoning mechanism. Some simple logics include: first order predicate logic, Hoare’s axioms, and temporal logic. It is not reasonable to include non-monotonic logics, as this would mean we must also change the deduction system.

One important area of research for transforming designs to implementations is rewriting. Development of a theory and subsequent development of tools for rewriting will provide a powerful mechanism to realize correct software. Concurrent rewriting [Meseguer, 1990] extends known logical theories of rewriting to distributed software

development. Graph rewriting is a technique based on proving equivalences of graph representations. Models for distributed systems have been proposed based on this technique [Degano and Montanari, 1987].

9.3 Research Directions

Our survey, analysis and the proposed framework reveal many issues that need attention. In the area of tool support, a number of other tools are needed for the process of specification. Some are tools for database support for specifications that involve persistent stores for design specifications and supporting multi-user cooperation based software development. Some of these types of tools are provided by Integrated Project Support Environments (IPSEs) [Brereton, 1987]. The major issue in the development and use of such tools is the integration of tool support. Difficulties arise in integration because of the need to support multiple methods of software development in a group development environment. Integration problems in IPSEs are discussed in [Brown and McDermid, 1992].

One of the major issues in distributed software development is the inability to provide a global view of the application. As a result, it becomes difficult to visualize how the many components in a software system interact and behave. The non sequentiality of execution introduces an element of nondeterminism in the behavior. Transformation theory¹ has not yet been able to provide a consistent mechanism to translate high level representations of software to implementations when there is no global control over the process of distributed software design.

A widely accepted theory of concurrency allows the transformation process to maintain control over the development of software. Four main approaches have been attempted at providing a theory of concurrency that is necessary to support the transformation process.

- Combining process specification formalisms like CCS or CSP with abstract data

¹Some transformation systems address software development issues, but do not address the issue of lack of global control. [Partsch and Steinbrüggen, 1983] gives a survey of different program transformation systems. [Goguen, 1990] presents an algebraic framework for rewriting as a refinement technique. PLEASE, an executable specification language, supports incremental transformations [Terwilliger and Campbell, 1989].

type specifications [ISO, 1987].

- Combining petri net formalisms with abstract data types [Krämer, 1991].
- Combining abstract data types with denotational semantics based formalisms like the stream formalisms [Broy, 1988].
- Treating processes as abstract data types [Astesiano et al., 1985; Astesiano et al., 1990].

Transforming the entire specification set to a level of detail where implementation becomes feasible, is not pragmatic. The main reason is that there is no notion of global control. An advantageous side effect of lack of global control of the software transformation process is that parts of specifications can be refined in parallel. This view to transformation of specifications has not received enough attention.

It is our view that combining the formal software development methods with partial evaluation and transformation of high-level representations will result in robust software. Specifications are broken down into smaller subsets and distributed for refinement. These transformations are partial and a union of these partial transformations should provide a complete² specification. Distributed graph grammars provide a formal basis for this type of transformations (see Section 6.3.1). There are a number of issues that have to be addressed in such an approach to transformation, the most important being management of the distributed transformation processes. Factors involved in decomposition of the initial specification, and later re-composition are also important.

Another area which has not received much attention is the *incompleteness* of software specifications. Incompleteness of specification can mean two things:

- The specification has omitted some of the key aspects of a system.
- The specification models only the required aspects of a system, ignoring the unnecessary aspects.

Ideally, a specification must be proved complete with respect to the requirements (see Section A.1 for a formal definition of completeness of specifications). But conceptual

²Proof obligations for such compositions are very important.

models, which define requirements, are not detailed enough to define which aspects of the system are necessary and which aspects are not. The incompleteness issue is of real importance in the partial evaluation of specifications. If the decomposition does not generate disjoint units of the specification, then it is not possible to show completeness of the entire specification without specifying the dependencies between these specifications.

9.4 Summary

In this thesis we have characterized the features of design specification languages for distributed software systems development. We choose the **IncTr** model of incremental transformation of specifications for software development. We have provided a broad taxonomy of design specification methods for distributed systems, classified according to the underlying formal framework. We have examined methods that typify each category. This taxonomy was helpful in separating the issues in distributed software system design. Based on the design criteria for classification and the subsequent evaluation of the methods, we have proposed a framework for developing design specification languages for distributed systems. Such a framework describes

- support for stepwise refinement of specifications;
- tools that provide semantics of transforming requirements specification to design specifications;
- a framework for design specification languages;
- tools that provide semantics for verifying specifications;
- tools that provide semantics of transforming design specifications to implementation constructs.

Bibliography

- Abrial, J. R. [1988] "The B Tool (Abstract)." In Bloomfield, R., L. Marshall, and R. Jones, editors, *VDM'88: VDM - The Way Ahead*, volume 328 of *LNC'S*, pages 86-87. Springer-Verlag.
- Ackroyd, M. and D. Daum [January, 1991] "Graphical Notation for Object-Oriented Design and Programming." *Journal of Object-Oriented Programming*, 4[1]:18-28.
- Agha, G. [1990] "The Structure and Semantics of Actor Languages." In [de Bakker et al., 1990], pages 1-59.
- Agha, G. [1991] "Personal Communication."
- Agha, G. A. [1986] *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press.
- Alpern, B. and F. B. Schneider [October, 1985] "Defining Liveness." *Information Processing Letters*, 21:181-185.
- America, P. and J. Rutten [1990] "A Layered Semantics for a Parallel Object-Oriented Language." In [de Bakker et al., 1990], pages 91-123.
- Astesiano, E., A. Giovini, and G. Reggio [1990] "Processes as Data Types: Observational Semantics and Logic." In Guessarian, I., editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *LNC'S*, pages 216-234. Springer-Verlag.
- Astesiano, E., G. F. Mascari, G. Reggio, and M. Wirsing [March, 1985] "On the Parameterized Algebraic Specification of Concurrent Systems." In Ehrig, H., C. Floyd, M. Nivat, and J. Thatcher, editors, *Mathematical Foundations of Software Development: Proceedings of TAPSOFT 85*, volume 185 of *LNC'S*. Springer-Verlag.
- Bal, H. E., J. G. Steiner, and A. S. Tanenbaum [September, 1989] "Programming Languages for Distributed Computing Systems." *ACM Computing Surveys*, 21[3]:261-322.
- Baldassari, M., V. Berti, and G. Bruno [1988] "Object Oriented Conceptual Programming based on PROT Nets." In *International Conference on Computer Languages*, pages 226-233.
- Bartussek, W. and D. L. Parnas [1978] "Using Assertions about Traces to write Abstract Specifications for Software Modules." In *Proceedings of Second Conference of European Cooperation on Informatics*, volume 65 of *LNC'S*, pages 211-236. Springer-Verlag.

- Bear, S., P. Allen, D. Coleman, and F. Hayes [1990] "Graphical Specification of Object Oriented Systems." In *ECCOOP/OOPSLA '90*, pages 28-37.
- Bergstra, J. A. and J. W. Klop [1984] "Process Algebra for Synchronous Communication." *Information and Control*, 60:109-137.
- Best, E. [1984] "Concurrent Behavior: Sequences, Processes and Axioms." In *Seminar on Concurrency*, pages 221-245.
- Björner, D. and C. B. Jones, editors [1982] *Formal Specification and Software Development*. Prentice-Hall.
- Booch, G. [February, 1986] "Object-Oriented Development." *IEEE Transactions on Software Engineering*, 12[2]:211-221.
- Borgida, A., J. Mylopoulos, and J. W. Schmidt [1991] "The TaxisDL Software Description Language." TDL Working Report.
- Boudol, G. [1990] "Flow Event Structures and Flow Nets." In *Semantics of Systems of Concurrent Processes*, pages 62-95.
- Boudol, G. and K. G. Larsen [1990] "Graphical versus Logical Specifications." In Arnold, A., editor, *Colloquium on Trees in Algebra and Programming*, volume 431 of *LNCS*, pages 57-71. Springer-Verlag.
- Boyer, R. and J. S. Moore [1979] *A Computational Logic*. Academic Press.
- Brereton, P., editor [1987] *Software Engineering Environments*. Ellis Horwood.
- Brown, A. W. and J. A. McDermid [1992] "Learning from HPSE's Mistakes." *IEEE Software*, 9[2]:23-28.
- Broy, M. [October, 1988] "Requirements and Design Specification for Distributed Systems." In Vogt, F.H., editor, *CONCURRENCY 88: Proceedings of International Conference on Concurrency*, volume 335 of *LNCS*, pages 33-62. Springer-Verlag.
- Chandy, K. M. and J. Misra [1988] *Parallel Program Design: A Foundation*. Addison-Wesley.
- Cohen, B., W. T. Harwood, and M. I. Jackson [1986] *The Specification of Complex Systems*. Addison-Wesley.
- Cook, W. R. [1990] "Object-Oriented Programming versus Abstract Data Types." In [de Bakker et al., 1990], pages 151-178.
- Corradi, A. and L. Leonardi [1990] "Parallelism in Object-Oriented Programming Languages." In *Proceedings of the 1990 International Conference on Computer Languages*, pages 271-280.
- Dahl, O.-J., B. Myrhaug, and K. Nygaard [1970] "Simula 6 Common Base Language." Technical Report S-22, Norwegian Computing Center.
- de Bakker, J. W., W. P. de Roever, and G. Rozenberg, editors [1990] *Foundations of Object-Oriented Languages*, volume 489 of *LNCS*. Springer-Verlag.

- De Paoli, F. and M. Jazayeri [1990] "FLAME: A Language for Distributed Programming." In *Proceedings of the 1990 International Conference on Computer Languages*, pages 69-78.
- Degano, P. and U. Montanari [1987] "A Model for Distributed Systems based on Graph Rewriting." *Journal of the ACM*, 34[2]:411-449.
- Diaz, M. and C. Vissers [November, 1989] "SEDOS: Designing Open Distributed Systems." *IEEE Software*, 6[6]:24-33.
- Diaz-Gonzalez, J. P. and J. E. Urban [1988] "Language Aspects of ENVISAGER: An Object-Oriented Environment for the Specification of Real-Time Systems." In *International Conference on Computer Languages*, pages 214-225.
- Dijkstra, E. W. [1976] *A Discipline of Programming*. Prentice-Hall.
- Ehrig, H. [1978] "An Introduction to Algebraic Theory of Graph Grammars." In Ehrig, H., M. Nagl, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1-69. Springer-Verlag.
- Ehrig, H., P. Boehm, U. Hummert, and M. Löwe [1987] "Distributed Parallelism of Graph Transformations." In Göttler, H. and H.J. Schneider, editors, *Graph-Theoretic Concepts in Computer Science*, volume 314 of *LNCS*, pages 1-19. Springer-Verlag.
- Fernandez, J.-C. and L. Mournier [1991] "A Tool Set for deciding Behavioral Equivalences." In *CONCUR '91*, pages 23-42.
- Francez, N. [1986] *Fairness*. Springer-Verlag.
- Gerth, R. and A. Pnueli [1989] "Rooting UNITY." In *Fifth International Workshop on Software Specification and Design*, pages 11-19.
- Ghezzi, C., M. Jazayeri, and D. Mandrioli [1991] *Fundamentals of Software Engineering*. Prentice-Hall.
- Goering, S. K. [1990] "A Graph Grammar Approach to Concurrent Programming." Ph.D Thesis UIUCDCS-R-90-1715, University of Illinois, Urbana-Champaign.
- Goguen, J. A. [1981] "More Thoughts on Specification and Verification." *ACM SIGSOFT*, 6[3]:38-41.
- Goguen, J. A. [1990] "An Algebraic Approach to Refinement." In Bjørner, D., C. A. R. Hoare, and H. Langmaack, editors, *VDM '90: VDM and Z-Formal Methods in Software Development*, volume 428 of *LNCS*, pages 12-28. Springer-Verlag.
- Goguen, J. A., J. W. Thatcher, and E. G. Wagner [1977] "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types." In Yeh, R. T., editor, *Current Trends in Programming Methodology : Volume IV*, chapter 5, pages 80-149. Prentice-Hall.
- Golumbic, M. C. [1980] *Algorithmic Graph Theory and Perfect Graphs*, Academic Press.
- Gorrieri, R. and U. Montanari [1990] "SCONE: A Simple Calculus of Nets." In *CONCUR '90*, pages 2-30.

- Gossain, S. and B. Anderson [1990] "An Iterative-Design Model for Reusable Object Oriented Software." In *Proceedings of ECOOP/OOPSLA '90*, pages 12–27.
- Gribomont, E. P. [1990] "A Programming Logic for Formal Concurrent Systems." In Baeten, J. C. M. and J. W. Klop, editors, *CONCUR '90: Theories of Concurrency: Unification and Extension*, volume 458 of *LICS*, pages 298–313. Springer Verlag.
- Guttag, J. V. [June, 1977] "Abstract Data Types and the development of data structures." *Communications of the ACM*, 20[6]:396–405.
- Guttag, J. V. and J. J. Horning [1978] "The Algebraic Specification of Abstract Data Types." *Acta Informatica*, 10:27–52.
- Harel, D. [May, 1987a] "On Visual Formalisms." *Communications of the ACM*, 31[5]:514–530.
- Harel, D. [1987b] "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming*, 8:231–274.
- Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shull, Trauring, and M. Trakhtenbrot [April, 1990] "STATEMATE: A Working Environment for the Development of Complex Reactive Systems." *IEEE Transactions on Software Engineering*, 16[4]:403–414.
- Hennessey, M. and R. Milner [January, 1985] "Algebraic Laws for Nondeterminism and Concurrency." *Journal of the ACM*, 32[1]:137–161.
- Hoare, C. A. R. [1972] "Proofs of Correctness of Data Representations." *Acta Informatica*, 1[1]:271–281.
- Hoare, C. A. R. [1985] *Communicating Sequential Processes*, Prentice-Hall.
- Hogger, C. J. [1984] *Introduction to Logic Programming*, Prentice-Hall.
- ISO [1987] "Information Processing Systems – Open Systems Interconnection – "LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behavior". Technical Report DIS 8807, International Organization for Standardization.
- Jalote, P. [March, 1989] "Functional Refinement and Nested Objects for Object Oriented Design." *IEEE Transactions on Software Engineering*, 15[3]:264–270.
- Johnson, D. S. [1983] "The NP-Completeness Column: An Ongoing Guide." *Journal of Algorithms*, 4[3]:286–300.
- Johnson, D. S. [1984] "The NP-Completeness Column: An Ongoing Guide." *Journal of Algorithms*, 5[4]:595–609.
- Johnson, R. E. and V. F. Russo [1991] "Reusing Object-Oriented Designs." Technical Report UIUCDCS-R-91-1696, University of Illinois, Urbana.
- Kaplan, S. [1989] "Algebraic Specification of Concurrent Systems." *Theoretical Computer Science*, 69:69–115.

- Kapur, D., G. Sivakumar, and H. Zhang [1986] "RRL: A Rewrite Rule Laboratory." In Siekmann, J. H., editor, *Proceedings of 8th International Conference on Automated Deduction*, volume 230 of *LNC'S*, pages 692–693. Springer-Verlag.
- Kelly, J. C. [1987] "A Comparison of four Design Methods for Real-time Systems." In *International Conference on Software Engineering*, pages 238–252.
- Krämer, B. [1991] "Introducing the GRASPIN Specification Language *SEGRAS*." *Journal of Systems Software*, 15:17–31.
- Lehmann, M. M., V. Stenning, and W. M. Turski [April, 1984] "Another Look at Software Design Methodology." *ACM SIGSOFT Software Engineering Notes*, 9[2]:38–53.
- Lucas, P. [1982] "Main Approaches to Formal Specifications." In [Björner and Jones, 1982], chapter 1, pages 3–23.
- Manna, Z. and A. Pnueli [1983] "How to cook a temporal proof system for your pet language." In *Proceedings Tenth Annual Symposium on Principle of Programming Languages*.
- Manna, Z. and A. Pnueli [1987] "Specification and Verification of Concurrent Programs by \forall -Automata." In Banieqbal, E., H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, volume 398 of *LNC'S*. Springer-Verlag.
- Manna, Z. and A. Pnueli [1988] "The Anchored version of the Temporal Framework." In deBakker, J. W., W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNC'S*, pages 201–284. Springer-Verlag.
- Meseguer, J. [1990] "Rewriting as a Unified Model of Concurrency." In Baeten, J. C. M. and J. W. Klop, editors, *CONCUR '90 – Theories of Concurrency: Unification and Extension*, volume 458 of *LNC'S*, pages 384–400. Springer-Verlag.
- Milner, R. [1980] *A Calculus of Communicating Systems*, , volume 92 of *LNC'S* Springer-Verlag.
- Milner, R. [1983] "Calculi for Synchrony and Asynchrony." *Theoretical Computer Science*, 25:267–310.
- Milner, R. [1989] *Communication and Concurrency*, Prentice-Hall.
- Moss, J. E. B. and W. H. Kohler [1987] "Concurrency Features for the Trellis/Owl Language." In *Proceedings of ECOOP '87*, volume 276 of *LNC'S*, pages 171–180. Prentice-Hall.
- Mylopoulos, J., A. Borgida, M. Jarke, and M. Koubarakis [1990] "Telos: A Language for Representing Knowledge about Information Systems." Technical report, Dept. of Computer Science, Univ. of Toronto.
- Olderog, E.-R. [1987] "Operational Petri Net Semantics for CCSP." In *Advances in Petri Nets 1987*, pages 196–223.
- Parnas, D. L. [December, 1972] "On the Criteria to be used in Decomposing Systems into Modules." *Communications of the ACM*, 15[12]:1053–1058.

- Partsch, H. and R. Steinbrüggen [September, 1983] "Program Transformation Systems." *ACM Computing Surveys*, 15[3]:199–236.
- Paulson, L. A. [1987] *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press.
- Sanders, B. A. [November, 1989] "Stepwise Refinement of Mixed Specifications of Concurrent Programs." Technical Report 116, Eidgenössische Technische Hochschule, Zurich.
- Schneider, F. B. and G. R. Andrews [1986] "Concepts for Concurrent Programming." In deBakker, J. W., W.-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, volume 224 of *LNC'S*, pages 669–716. Springer-Verlag.
- Schwan, K., R. Ramnath, S. Sarkar, and S. Vasudevan [1986] "COOL – A Language to Construct and Tune Parallel Programs." In *Proceedings of the 1986 International conference on Computer Languages*, pages 62–71.
- Shields, M. W. [1987] "Algebraic Models of Parallelism and Net Theory." In *Concurrency and Nets*, pages 423–433.
- Stark, E. W. [1989] "Concurrent Transition Systems." *Theoretical Computer Science*, 64:221–269.
- Tarumi, H., K. Agusa, and Y. Ohno [1988] "A Programming Environment Supporting Reuse of Object-Oriented Software." In *Proceedings of Tenth International Conference on Software Engineering*, pages 265–273.
- Terwilliger, R. B. and R. H. Campbell [1989] "PLEASE: Executable Specifications for Incremental Software Development." *Journal of Systems and Software*, 10:97–112.
- Turner, K. J., editor [1988] *First International Conference on Formal Description Techniques*. North-Holland.
- Turski, W. M. and T. S. E. Maibaum [1987] *The Specification of Computer Programs*. Addison-Wesley.
- Usenet [1991] "Usenet Discussion on comp.specification." .
- Vermeir, D. [1986] "OOPS: A Knowledge Representation Language." In *International Conference on Computer Languages*, pages 156–168.
- Wegner, P. [1987] "The Object-Oriented Classification Paradigm." In Shriver, B. and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 479–560. The MIT Press.
- Yau, S. S. and J. J.-P. Tsai [June, 1986] "A Survey of Software Design Techniques." *IEEE Transactions on Software Engineering*, 12[6]:713–721.
- Yonezawa, A. and M. Tokoro, editors [1987] *Object-Oriented Concurrent Programming*. The MIT Press.

Appendix A

A Formal View of Specifications in IncTr

In this appendix, we present a formal view of the **IncTr** model and describe some of its properties. We use the elevator example from section 2.1 to illustrate some of these properties.

A formal system may be used to specify a software system. The specification S is a set of statements in the language L of this formal system \mathcal{F} . The rules defined by the *consequence closure* operator C_n are observed while synthesizing the specification statements [Lehmann et al., 1984; Turski and Maibaum, 1987].

It is obvious that $S \subset L$. L provides a notation for representation, and C_n provides a set of rules for reasoning about the theories that are constructed. Thus, a specification has two interpretations.

1. One way of interpreting specifications is as a set of statements describing various objects of the software system, and operations on these objects. Until recently, most software was developed with this view of specifications. When the development team was satisfied that the specification closely models the intended behavior of the system, they started the implementation process.
2. Another view is to use the consequence closure of the specification as the starting point for reasoning about the behavior of the system. The closure defines all possible behaviors of the software system being modeled by the specification. In this way properties of a specification and its formal system such as consistency, soundness, completeness, etc. can be defined.

Specifications are abstract, and one can formalize the process of refinement of (abstract, non-executable) specifications to executable forms (implementations) using the following theorem.

Theorem A.1 (The Specification Theorem) *Given an input x satisfying an input condition C of a system, find an output z (which is the output of some program $P(x)$)*

which satisfies a given input/output relation $R(x, z)$. The relation R is the transformation function, and the above specification can be characterized as

$$\forall x. C(x) \supset \exists z. R(x, z)$$

This theorem presents the *deductive approach* to program synthesis from formal specifications.

A.1 Properties of Specifications

There are a number of properties a specification should have. These properties also reflect on the characteristics a specification system. The description below is adapted from [Cohen et al., 1986].

A.1.1 Consistency

A specification should not derive contradictory theorems. Thus it must be *consistent*. For a formal system $\mathcal{F} = \langle L, C_n \rangle$, a specification $S \subset L$ is said to be *consistent* if and only if the consequence closure of S is also included in the language L . Formally,

$$C_n(S) \subset L.$$

For example, we cannot deduce that an elevator must service two floors at the same time. Since implementation can be viewed as a specification at a higher level of abstraction, we can define it analogously as below.

A specification S_i in a formal system $\mathcal{F}_i = \langle L_i, C_{n_i} \rangle$ is said to *implement* a specification S in a formal system $\mathcal{F} = \langle L, C_n \rangle$ if and only if

$$L \subseteq L_i \quad \text{and} \quad C_n(S) \subset C_{n_i}(S_i).$$

This states that the implementation (S_i) of S may have a more descriptive language, but all consequences that can be deduced from S must be included in the consequences that can be deduced from S_i . The above statement reflects on the information contents of the specification and its implementation.

A.1.2 Soundness and Completeness

Two important properties of a formal system are *soundness* and *completeness*. A formal system is *complete* if the rules for deduction are powerful enough to derive all theorems derivable. It is said to be *sound* if all derivable theorems are valid, i.e., they are included in the set of consequences for all possible theories. Applying these definitions to specifications and implementation, a formal system for specification \mathcal{F} is *sound* with respect to the formal system for implementation \mathcal{F}_i , if and only if

$$\forall S \subseteq L : C_n(S) \subseteq C_{n_i}(S) \quad \text{where } L \subseteq L_i.$$

Simply put, every behavior (theorem) of S , which is observable (derivable) in the formal system of the specification must also be observable in the formal system of implementation. \mathcal{F} is said to be complete with respect to \mathcal{F}_i if and only if

$$\forall S \subseteq L : C_{n_i}(S) \subseteq C_n(S) \quad \text{where } L \subseteq L_i.$$

This means that every behavior of S observable in the formal system of implementation \mathcal{F}_i must be observable in the formal system of specification \mathcal{F} .

A.1.3 Complete Specifications

A complete specification is one in which the behaviors of every object in the system are so defined that every implementation **must** yield the same behavior for all the objects. Formally,

$$\forall S_i : S_i \text{ **Impl** } S \Rightarrow \{ x \mid x \in C_{n_i}(S_i) \wedge \text{syn}(x) = \text{syn}(S) \}_i = C_n(S).$$

Here, the relation **Impl** defines implementation; i.e., S_i implements S . $\text{syn}(S)$ gives the syntax of S . This definition combines soundness, and completeness.

In the elevator example, completeness would entail considering all possible behaviors of the elevator (some of which could otherwise be eliminated by constraints). A sound specification will help constrain the set of all possibilities to just those situations that are practical. A complete specification will explore possible behaviors like the elevator servicing requests at two different floors at the same time.

A.1.4 Tightness

Another property involving specifications is relations between specifications having the same language but different consequence operators. Given two formal systems \mathcal{F}_1 and \mathcal{F}_2 with the same language L , having the property

$$\forall S \subseteq L : C_{n_1}(S) \subseteq C_{n_2}(S),$$

we say that \mathcal{F}_1 is *looser* than \mathcal{F}_2 or \mathcal{F}_2 is *tighter* than \mathcal{F}_1 . A system is tighter if it has more information, is less abstract, and has more constraints defined. An extremely tight specification is the basis for implementation. We present examples of this property in the later chapters, when we analyze each method of specification.

A.1.5 Extension

Consistency, along with soundness is the most important property a specification must have. A *consistent extension* to a specification $S \subseteq L$ is another specification S' with the same language, such that

$$C_n(S) \subseteq C_n(S') \subseteq L.$$

From this it is obvious that loose specifications are more abstract and therefore can have more consistent extensions. They also can be used to define a greater variety of similarly structured systems, and thereby capture the notion of reusability.

A.1.6 Systems of Specifications

A specification of a large system tends to get complex, and using just one formal system may not only limit the ability to compose specifications for the system, but also generate large, and unwieldy structures. It may be advantageous to have many formal systems and a method for composing specifications in all these formal systems. What is required is a compact manner of integrating the use of two or more formal systems for specification.

In a specification system consisting of two formal systems \mathcal{F}_1 and \mathcal{F}_2 , a composite specification is a set of statements which is *expressible* in both systems. A set of statements is said to be expressible in such a system if

- this set is finite, or
- this set is the consequence closure, under C_{n_1} or C_{n_2} , of an expressible set of statements.

A specification under such a system is defined by the consequence closure under either C_{n_1} or C_{n_2} . A related concept that has a similar property is *abstraction*. Loose (or abstract) specifications have less strict closure operators, whereas tight specifications have very strict closure operators, possibly a combination of many closure operators. In other words, loose specifications may be closed under just one constraint, whereas, tight specifications may have more than one constraint applied on them.

A.1.7 Parameterization

Parameterization is the property of applying some concept to more than one class of objects. For example, we can have sets of natural numbers, and also sets of other objects, like sets of sets, or sets of cities. The concept of a set is applicable to a variety of objects. By providing the class of objects, we are providing a parameter defining the type of object to which this concept will apply. Formally, a parameterized specification is a pair

$$\langle C_n(S), \sigma \rangle$$

where $\sigma \subseteq \text{syn}(S)$ picks that part of the syntax of S which is acting as the parameter. To instantiate the parameterized specification $\langle C_n(S), \sigma \rangle$, we must apply it to an argument which provides a definition of the parameter. This argument is possibly another specification, with its own rules and related mappings from its syntax to the parameter. Formally, such an argument is represented as

$$\langle P, \theta : \sigma \mapsto \rho \rangle \quad \text{where } \theta : \text{syn}(S) \mapsto \text{syn}(P)$$

and $\rho \subseteq \text{syn}(P)$. θ is an injective mapping. By applying the instantiation discussed earlier, $\langle C_n(S), \sigma \rangle$ to $\langle P, \theta : \sigma \multimap \rho \rangle$ we get

$$C_n(\theta(S) \cup P) .$$

The resultant property now constrains both the definition of the parameter, and its specification in that, the instantiation should be consistent. This desirable property only complicates the correctness arguments. An instantiation such as the above is not guaranteed to be consistent, even if $C_n(S)$ and P are consistent. This property must be proved for a particular specification system. Given a common set of symbols, this property holds for notion of equality [Cohen et al., 1986]. Adding inequality operators tends to make the specification language more tight, and therefore create uncertainties, as there is no ordering relation between $C_n(S)$ and P .