Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

190

M.W. Alford J.P. Ansart G. Hommel
L. Lamport B. Liskov G.P. Mullery
F.B. Schneider

Distributed Systems

Methods and Tools for Specification
An Advanced Course

Edited by M. Paul and H.J. Siegert

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

190

M.W. Alford J.P. Ansart G. Hommel
L. Lamport B. Liskov G.P. Mullery
F.B. Schneider

Distributed Systems

Methods and Tools for Specification
An Advanced Course

Edited by M. Paul and H.J. Siegert

Springer-Verlag
Berlin Heidelberg New York Tokyo

Editors
M. Paul
H. J. Siegert
Institut für Informatik, Technische Universität München
Arcisstr. 21, D-8000 München 2, FRG

CR Subject Classification (1982): C.1.2, C.2, D.1.3, D.2, D.3, D.4

ISBN 3-540-15216-4 Springer-Verlag Berlin Heidelberg New York Tokyo
ISBN 0-387-15216-4 Springer-Verlag New York Heidelberg Berlin Tokyo

Printed in Germany

Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.
2145/3140-543210

Preface

The papers comprising this volume were prepared for and presented during the Advanced Course on Distributed Systems – Methods and Tools for Specification. The course was held from April 3 to April 12, 1984 at the Technische Universität München. Due to its success it was repeated from April 16 to April 25, 1985. The organization lay in the hands of the Institut für Informatik, and it was jointly financed by the Ministry for Research and Technology of the Federal Republic of Germany, and the Commission of the European Communities.

Research on distributed systems is in progress within universities as well as in industry and governmental organizations. Networks, particularly high speed local area networks, are often the spur to build distributed systems. In the past a certain agreement on some basic models has been achieved, e.g. on the ISO–OSI–Reference Model, on lower level protocols, and on some synchronization problems. However, concepts and programming paradigms pertinent to higher level protocol layers, to overall concepts for distributed systems, to design choices, and to higher level language support are still important research areas. A discussion and presentation concerning these issues can be found in [Lampson 81b].

Another important research area aimed at improving software quality and reducing software production costs is the support of the specification and design phases within the software life cycle. This problem has received more and more attention during the last decade. Looking at the relative cost or manpower for different phases in the life cycle of software one could see a definite shift of importance from the coding and implementation phase to the specification and design phase. A typical figure is, that about 40% of the total development costs are spent for specification and design. Again we have not yet an agreement on the direction or on the methods and tools to be used for specifying even simple systems.

For a successful specification of distributed systems one has to combine

general specification methods and tools as well as architectural knowledge, modularization concepts and programming paradigms for distributed systems. A presentation of these topics was the major aim of the course. As said before, all aspects involved are still in a research stage to a very high degree. Therefore it is impossible to give a complete picture of all ideas, concepts, methods, and tools. Instead we have tried to show and discuss the range of possible solutions by presenting a specification system used by a commercial company, and in contrast, examples and basic principles for formal specification and verification. It is important of course to have an understanding of programming concepts and paradigms for distributed systems when specifying and designing them. Important concepts and paradigms are presented in chapter 4. As an example for a language for programming distributed systems we have selected the Argus language.

Finally we want to express our gratitude and appreciation

to the lecturers, who have spent considerable time discussing the course contents during the preparation period and preparing the excellent lecture notes, and

to all members of our staff, foremost Mrs. U. Weber and Dr. H. Haifer, who have helped with organizing this course and editing the lecture notes.

The authors and editors are confident, that both the course participants and the readers of these lecture notes will find an in-depth study of the material contained herein rewarding for their own work.

M. Paul
H.J. Siegert

Contents

Chapter 1

Introduction

Production of software for distributed systems, as any other production of industrial goods, requires different activities to be performed. Scanning the literature on software engineering we can find an enormous variety of models for the production of software using different notions for the activities in the production process. In spite of this variety of models and notions we try to filter out the essential activities:

- Acquisition and Analysis
  Gathering, structuring, and analysing informations on the feasibility of a project.
- Requirements Specification
  Specification and analysis *what* the software system should do.
- Design of System Architecture
  Specification and analysis *how* the logical structure of the system should be and *what* each module should do.
- Design of Components
  Specification *how* each module should be realized.
- Implementation
  Specification of the whole system in an executable programming language.
- Integration and Installation
  Make the system run.

An ordering of those activities in time with additional revision cycles is often called a ***software life-cycle model*** or a ***phase model***.

Rapid prototyping means to produce a quick implementation of essential parts of the system in order to show important properties of the system to the user as early as possible. It is especially useful to agree upon requirements on the man-machine interface of a system and is therefore regarded to be a part of requirements engineering.

During all those activities a lot of specifications are produced. Our goal is to produce better quality software and to rationalize the software production process. This can be achieved if we try to find errors in those specifications as soon as possible. The cost for correcting an error made in some activity grows exponentially in time of error detection as can be seen from Figure 1.1.

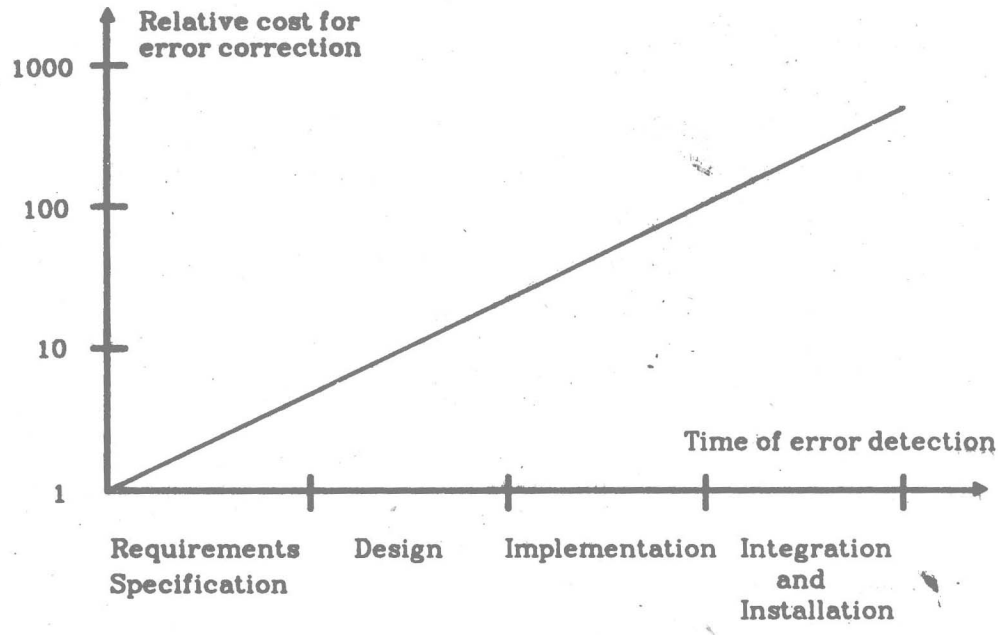

Figure 1.1: Cost for error detection

The extent of how many errors can be detected by analytical tools depends on the degree of formality of a specification. As Figure 1.2 shows, the production of software would ideally start with a complete formal specification of the requirements. By formal specification we understand a specification formalized in syntax and semantics. In this case we could come

to an implementation by using semantics-preserving transformations.

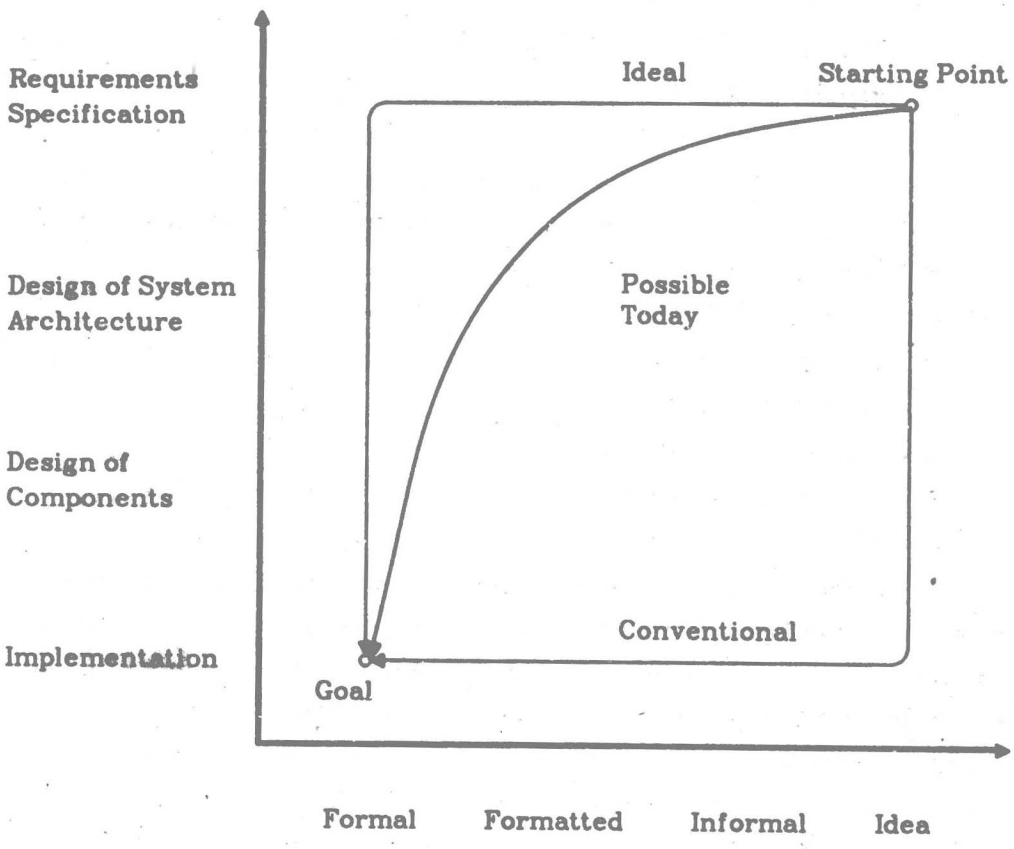

Figure 1.2: Process of software production

Conventionally the specification of requirements, of the system architecture, and of components exists only as a vague idea in the head of the programmer who is starting with coding immediately.

Tools available today allow to go the third realistic way using also informal and formatted specifications. Informal specifications consist of natural language and arbitrary graphs. In formatted specifications there is a well-defined syntactical frame with some informal semantics in it.

Tools can be classified using the following criteria:

- Activities which are supported by a tool. Mostly a tool is applicable only for one or few activities.
- Underlying theoretical models. Those are typical the entity-relationship model, Petri net theory, the finite state machine, etc.
- Form of representation, either graphical or in a linear notation.
- Guidelines for the way to succeed. Some tools even claim not to restrict the user at all and support any way the user wants to take without giving any recommendation.
- Degree of formalization.
- Degree of computer support. Some tools even do without any computer support.
- Availability and cost of tools.
- Scope of intended application.

If we do not have the necessary methods and experience to design a system we cannot blame our tools for that. The most important methods used in software production are the reduction of complexity by decomposition and abstraction. In decomposing systems we try to identify well-known patterns, often called paradigms. Such paradigms may be algorithms (as for example sorting and searching algorithms in sequential programming) or high level language constructs. Successful application of methods is a mental, intuition-guided activity that can not be automized and needs a lot of exercise and experience.

After discussing methods and tools for specification we will take a look at the aspect of distribution. There are different reasons for using distributed systems:

- *Load sharing* to better exploit available processing capacity.
- *Resource sharing* to use expensive resources or scarcely used special equipment.
- *Data sharing* to access distributed databases.
- The *geographical structure* may be inherently distributed. The bandwidth of the communication lines or the weakness of analogue signals may force their processing in loco.

- The ***logical structure*** may be simpler e.g. if each parallel process is located in a separate processor.
- The ***reliability*** of a system can be enhanced by tailoring an appropriate structure.
- The ***flexibility*** of a system is increased having the possibility to add and delete single processors.

Let us have a closer look at the aspect of reliability. Reliability can be defined as the degree of suitability to perform well under specific operating conditions during a specific time. A probabilistic measure for reliability is the ***availability*** of a system. The mean value of the availability A of a system is usually defined as A = MTBF / (MTBF + MTTR), with MTBF meaning the meantime between failures and MTTR meaning the meantime to repair.

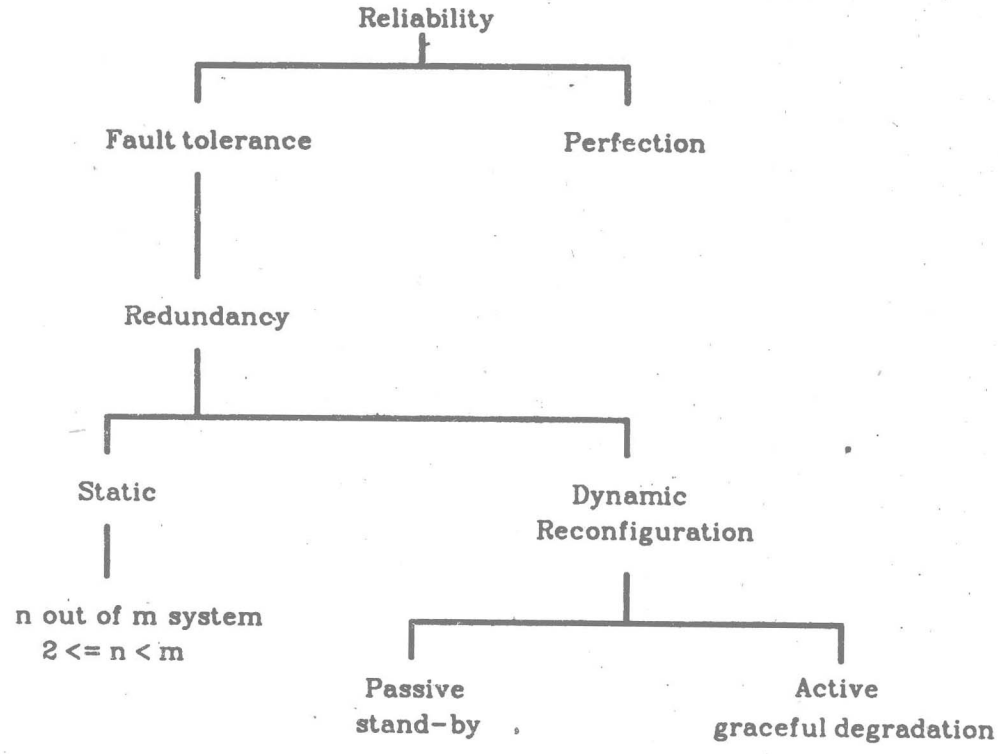

Figure 1.3: Reliability of a system

Figure 1.3 shows that reliability can either be achieved by *perfection*, that means constructing perfect hardware and software, or by *fault tolerance*. Fault tolerance can be achieved by *redundancy* which may either be static (n out of m system) or dynamic, requiring *reconfiguration*. Reconfiguration can be done either using passive *stand-by* processors or using already active processors giving up some of the less important functions of the system (*graceful degradation*).

The course material has been selected such that not only some methods and tools are presented for all activities of the software production process but also fundamentals to understand those methods and tools.

We also tried to present currently known thinking patterns in the field of distributed systems as to alleviate the decomposition process. We are shure that this course does not provide a closed theory or fool-proof recipes how to produce software for distributed systems and that a lot of research and development remains to be done.

Chapter 2
Basic Concepts

2.1. Introduction to Models

This section presents an overview of the models commonly used as the foundation for specifying properties of a distributed systems. This will of necessity only review a few selected models -- an review of all of the models used by different specification techniques is beyond the time and space limitations for this course.

Before examining individual models, it is useful to consider why one should be interested in the subject of models for specifications. The major motivation can be derived from the following observation: a model is used to precisely define desired characteristics of a system -- what is not specified cannot be verified, and what is not verified may be in error. The purpose of a model is to precisely define specific properties or characteristics of a system under consideration to be built or analyzed, and provides the foundation for verifying those properties. Different models are used to specify different properties; alternatively, to express a specific property of a system, one must select from a class of models which represent that property.

The kinds of properties necessary for the development of distributed systems include the following: sets, sequences, and structures of data; transformations of one data set into another, and the implied input/output relationships between the transformations and the data sets; sequences and

concurrency of data sets which arrive or are generated at different points of time; transformations of one time sequence into another; sequences of transformations; data flow between transformations; concurrency of transformations; control of interactions between concurrent transformations; time to perform a transformation; and reliability/ availability of performing a transformation in an environment of faults.

If we compare the properties of a flow chart or pseudo-code to this list of desired characteristics, we see that a flow chart or pseudo-code (structured or otherwise) usually expresses sequence, selection, and iteration of processing steps; the characteristics of data flow, concurrency, and performance are not present. A program structure chart for a serial program usually identifies all subprograms CALLED by a subprogram, flow of control, and flow of data; but no concurrency would be represented.

Whether one thinks these representations to be sufficient for representing serial programs, clearly they are insufficient for addressing the problems of concurrent distributed software. To represent these properties, we will examine the following models: mathematical function; finite state machine; functional hierarchy; Petri Net; and graph model of computation.

2.1.1 Mathematical Function

To define a mathematical function, one must specify three things: an input domain (e.g., a set of input variables); an output domain (e.g., a set of output variables); and a rule for transforming the inputs into the outputs. For the transformation to be a function, it must always produce the same outputs for the same input data set.

There are several relevent aspects of mathematical function which affect its applicabilty as a model for specifying distributed systems. First, a mathematical function is not an algorithm -- a function can be specified by providing an algorithm, but it is a design issue to construct an algorithm which performs a transformation within a specified accuracy.

For example, one can specify a transformation by

$$y = \sin(x)$$

but any one of at least three algorithms can be used to accomplish it:

1) y = x (for small values of x)
2) y = polynomial in x (different polynomials for different desired accuracy); or
3) y = Taylor series (calculated iteratively).

A second relevent aspect of a mathematical function is that it can be "decomposed", i.e. specified by a combination of logic and lower level functions. For example, one can define the absolute value function by

$$\begin{aligned} ABS(g(x)) &= x \text{ if } g(x)>0 \\ &= -x \text{ if } g(x)<0 \end{aligned}$$

This has the effect of specifying a function in terms of a structure of functions, and thus specifying an algorithm approach. The structure can be described in terms decision tables, pre-conditions and post-conditions, or flow-charts. Since one would like to specify a transformation and not an algorithm in order to separate requirements from design, use of a mathematical function appears to be very desirable.

The HOS specification approach [Hamilton 77] provides techniques for decomposing any arbitrary function in terms of logical operators JOIN, INCLUDE, and OR together with lower level functions, and repeating the decomposition until the lowest level arithmetic operators are encountered. This process decomposes both control flow and data flow simultaneously, and provides tools to check that the two are consistent.

However, a this approach is limited because a mathematical function inherently has no memory -- given an input, it produces an output but saves no data. This means that a collection of mathematical functions cannot be used to specify the required data contents of distributed computer systems. Attempts to use recursion (i.e., a function invoking another copy of itself to process a subsequent input) to overcome the lack of memory results drives the complexity of the function description exponentially.

In view of this limitation, it appears that a mathematical function is a necessary ingredient but not a sufficient model for specification of a distributed system. By itself, it can only be used to specify functions which require no memory; the model must be augmented to address the

problems for which distributed systems are most widely used.

2.1.2 Finite State Machine

The concept of a Finite State Machine (FSM) seems to be tailor made for the specification of processing for a data processor. Essentially, an FSM is composed of a set of inputs X, a set of outputs Y, a set of states S, an initial state So, and a pair of functions which are used to specify the outputs and state transitions which occur as a result of an input. The transformation function specifies the outputs which result from an input when the FSM is in any of its possible states; and the state transition function specifies the next state which results from an input for each possible state. In other words,

X is the set [Xi] of inputs
Y is the set [Yi] of outputs
S is the set [Sj] of states
So is initial state
F maps Xi and Sj onto Yi
G maps Xi and Sj onto Sj+1, the next state.

Figure 2.1 provides an illustration of such a model.

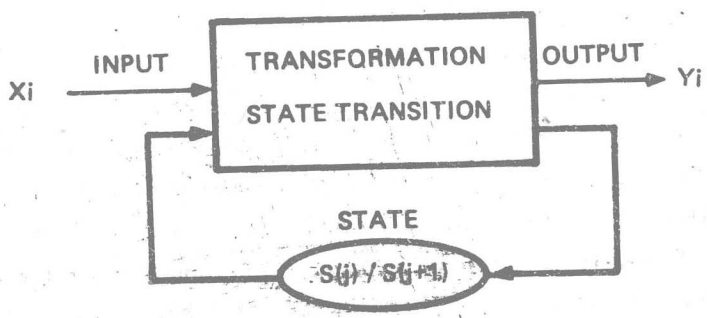

Figure 2.1: Finite State Machine Model