CONCISE ENCYCLOPEDIA OF

# SOFTWARE ENGINEERING

EDITORS
DERRICK MORRIS
BORIS TAMM

PERGAMON PRESS

CONCISE ENCYCLOPEDIA OF

# SOFTWARE ENGINEERING

# ADVANCES IN SYSTEMS, CONTROL AND INFORMATION ENGINEERING

This is a new series of Pergamon scientific reference works, each volume providing comprehensive, self-contained and up-to-date coverage of a selected area in the field of systems, control and information engineering. The series is being developed primarily from the highly acclaimed *Systems & Control Encyclopedia* published in 1987. Other titles in the series are listed below.

ATHERTON & BORNE (eds.)
*Concise Encyclopedia of Modelloing & Simluation*

FINKELSTEIN & GRATTAN (eds.)
*Concise Encyclopedia of Measurement & Instrumentation*

PAPAGEORGIOU (ed.)
*Concise Encyclopedia of Traffic & Transportation Systems*

PAYNE (ed.)
*Concise Encyclopedia of Biological & Biomedical Measurement Systems*

PELEGRIN & HOLLISTER (eds.)
*Concise Encyclopedia of Aeronautics & Space Systems*

SAGE (ed.)
*Concise Encyclopedia of Information Processing in Systems & Organizations*

YOUNG (ed.)
*Concise Encyclopedia of Environmental Systems*

**NOTICE TO READERS**

Dear Reader
If your library is not already a standing order/continuation order customer to the series **Advances in Systems, Control and Information Engineering,** may we recommend that you place a standing order/continuation order to receive immediately upon publication all new volumes. Should you find that these volumes no longer serve your needs, your order can be cancelled at any time without notice.

# CONCISE ENCYCLOPEDIA OF
# SOFTWARE ENGINEERING

Editors
DERRICK MORRIS
*UMIST, Manchester, UK*

BORIS TAMM
*Estonian Acadamy of Sciences, Tallinn, Estonia*

Series Editor-in-Chief
MADAN G SINGH
*UMIST, Manchester, UK*

PERGAMON PRESS
OXFORD • NEW YORK • SEOUL • TOKYO

| | |
|---|---|
| UK | Pergamon Press Ltd, Headington Hill Hall, Oxford OX3 0BW, England |
| USA | Pergamon Press, Inc, 660 White Plains Road, Tarrytown, New York 10591-5153, USA |
| KOREA | Pergamon Press Korea, KPO Box 315, Seoul 110-603, Korea |
| JAPAN | Pergamon Press Japan, Tsunashima Building Annex, 3-20-12 Yushima, Bunkyo-ku, Tokyo 113, Japan |

---

Copyright © 1993 Pergamon Press Ltd

*All rights reserved. No part of this publication may be reproduced, stored in any retrieval system or transmitted in any form or by any means: electronic, electrostatic, magnetic tape, mechanical, photocopying, recording or otherwise, without permission in writing from the publishers.*

First edition 1993

**Library of Congress Cataloging in Publication Data**
Concise encyclopedia of software engineering / editors, D. Morris, B. Tamm. — 1st ed.
p. cm. — (Advances in systems, control, and information engineering)
Includes index.
1. Software engineering—Encyclopedias. I. Tamm, B. G. II. Series.
QA76.758.C68 1993
005'.03—dc20 92-27942

**British Library Cataloguing in Publication Data**
A catalogue record for this book is available from the British Library.

ISBN 0-08-036214-1

∞™ The paper used in this publication meets the minimum requirements of the American National Standard for Information Sciences—Permanence of Paper for Printed Library Materials, ANSI Z39.48-1984.

Printed and bound in Great Britain by Butler & Tanner Ltd, Frome

# CONTENTS


| | |
|---|---|
| Honorary Editorial Advisory Board | vi |
| Foreword | vii |
| Preface | ix |
| Guide to Use of the Encyclopedia | xi |
| Alphabetical List of Articles | xiii |
| Articles | 1 |
| List of Acronyms | 375 |
| List of Contributors | 377 |
| Subject Index | 385 |

# HONORARY EDITORIAL ADVISORY BOARD


Chairman
John F Coales CBE, FRS
*Cambridge, UK*

Editor-in-Chief
Madan G Singh
*UMIST, Manchester, UK*

D Aspinall
*UMIST, Manchester, UK*

K J Åström
*Lund Institute of Technology*
*Sweden*

A Bensoussan
*INRIA, Le Chesnay, France*

P Borne
*Institut Industriel du Nord*
*Villeneuve D'Ascq, France*

A W Goldsworthy OBE
*Jennings Industries Ltd*
*Victoria, Australia*

J Lesourne
*Conservatoire National*
*des Arts et Métiers,*
*Paris, France*

P A Payne
*UMIST, Manchester, UK*

A P Sage
*George Mason University*
*Fairfax, VA, USA*

Y Sawaragi
*Japan Institute for Systems*
*Research, Kyoto, Japan*

G Schmidt
*Technische Universität München*
*Germany*

B Tamm
*Tallinn Technical University*
*Estonia*

M Thoma
*Universität Hannover*
*Germany*

R Vichnevetsky
*Rutgers University*
*New Brunswick, NJ, USA*

# FOREWORD

With the publication of the eight-volume *Systems & Control Encyclopedia* in September 1987, Pergamon Press was very keen to ensure that the scholarship embodied in the Encyclopedia was both kept up to date and was disseminated to as wide an audience as possible. For these purposes, an Honorary Editorial Advisory Board was set up under the chairmanship of Professor John F. Coales FRS, and I was invited to continue as Editor-in-Chief. The new work embarked upon comprised a series of Supplementary Volumes to the Main Encyclopedia and a series of Concise Encyclopedias under the title of the Advances in Systems, Control and Information Engineering series. This task involved me personally editing the series of Supplementary Volumes with the aim of updating and expanding the original Encyclopedia and arranging for the editing of a series of subject-based Concise Encyclopedias being developed from the Main Encyclopedia. The Honorary Editorial Advisory Board helped to select subject areas which were perceived to be appropriate for the publication of Concise Encyclopedias and to choose the most distinguished experts in those areas to edit them. The Concise Encyclopedias were intended to contain the best of the articles from the Main Encyclopedia, updated or revised as appropriate to reflect the latest developments in their fields, and many totally new articles covering recent advances in the subject and expanding on the scope of the original Encyclopedia.

Software engineering is an area that has developed very quickly in recent years, in step with the development and wider availability of computer systems of greater power and complexity. Professor Derrick Morris of UMIST in Manchester, UK, and Professor Boris Tamm of the Estonian Academy of Sciences in Tallinn, Estonia, a former President of IFAC, have brought together a wide-ranging coverage of the subject of software engineering, emphasizing the aspects of particular relevance to control engineering, such as real-time programming.

This volume will be of particular interest to all those with an interest in the development or use of software systems, both from the software engineer's side and the user's side—for example, people working in information processing areas—and to anyone needing information on the theories and techniques on which the discipline of software engineering is based.

Madan G Singh
*Series Editor-in-Chief*

This page intentionally left blank

# PREFACE

The *Concise Encyclopedia of Software Engineering* is intended to provide compact coverage of the knowledge relevant to the practicing software engineer. It is hoped that it will be of value to new practitioners who need a concise overview and to established practitioners who need information about the "penumbra" surrounding their own specialities. Another anticipated group of readers are professionals from other disciplines who need to gain some understanding of the various aspects of software engineering which underpin complex information and control systems and the thinking behind them.

Thus, the content has been chosen to provide an introduction to the theory and techniques relevant to the software of a broad class of computer applications. This is supported by examples of particular applications and their enabling technologies. The references in the articles provide an entry into the literature for more extended study. A broad overview of the total content can be obtained from the alphabetical list of articles on p. xiii. For those who wish to locate specific information there is a subject index on p. 385.

In planning a work of this kind a major difficulty has to be faced. This is making the decisions on what to exclude. Clearly the subject area is too large for a concise work to provide comprehensive coverage at a detailed level. We therefore chose to limit the coverage of several substantial topics which we believe are well covered in a concentrated way by the text books which underpin computer science—programming, structured programming, programming languages and formal methods—in order to increase the breadth of topics covered. The plan neither attempted to provide a substitute for standard text books, nor did it go to the other extreme and attempt to produce a dictionary of software engineering.

This is not to say that topics such as those mentioned above are ignored, we hope that the anticipated readership will gain insight into them from what is included, and in many cases be motivated to carry out further study.

Another difficulty which arises in carrying out a plan aiming for broad coverage is achieving this coverage, whilst minimizing overlap. We have taken more time over this than originally planned and the need to go to print is now pressing. Many authors we would have wished to obtain contributions from are simply too busy, but we are enormously indebted to those who have contributed. No work of this kind could succeed without the participation of so many experts. They have been very patient about our delays and many of them have helped us out of last minute difficulties by making additional contributions at very short notice. Their names appear at the foot of each article and in alphabetical order in the List of Contributors on p. 377.

The series of Concise Encyclopedias was proposed by Madan Singh. We are grateful to him for inviting us to participate. It has been comforting to find him very supportive of our efforts and always available to advise on problems.

Thanks are also due to the staff at Pergamon Press for the expert guidance we have received, and to Colin Drayton, Helen McPherson and Peter Frank in particular, for their efficiency in dealing with the administrative workload.

Finally, we must thank our own secretarial staff, Pat Cook at UMIST and Ilda Timmerman at Tallinn, for the work that both have put into the project. Pat Cook, in addition to producing many invitations and reminders and collating the responses, has been our day-to-day interface with Pergamon Press. Her role in assembling the material for publication has been crucial.

D Morris and B Tamm
*Editors*

This page intentionally left blank

# GUIDE TO USE OF THE ENCYCLOPEDIA

This Concise Encyclopedia is a comprehensive reference work covering all aspects of software engineering. Information is presented in a series of alphabetically arranged articles which deal concisely with individual topics in a self-contained manner. This guide outlines the main features and organization of the Encyclopedia, and is intended to help the reader to locate the maximum amount of information on a given topic.

Accessibility of material is of vital importance in a reference work of this kind and article titles have therefore been selected not only on the basis of article content, but also with the most probable needs of the reader in mind. An alphabetical list of all the articles contained in this Encyclopedia is to be found on p. xiii.

Articles are linked by an extensive cross-referencing system. Cross-references to other articles in the Encyclopedia are of two types: in-text and end-of-text. Those in the body of the text are designed to refer the reader to articles that present in greater detail material on the specific topic under discussion. They generally take one of the following forms:

> ...as is described in the article *Communication Networks for Control Systems.*

> ...the applications of these techniques (see *Artificial Intelligence: Applications*).

The cross-references listed at the end of an article serve to identify broad background reading and to direct the reader to articles that cover different aspects of the same topic.

The nature of an encyclopedia demands a higher degree of uniformity in terminology and notation than many other scientific works. The widespread use of the International System of Units has determined that such units be used in this Encyclopedia. It has been recognized, however, that in some fields Imperial units are more generally used. Where this is the case, Imperial units are given with their SI equivalent quantity and unit following in parentheses. Where possible, the symbols defined in *Quantities, Units, and Symbols* published by the Royal Society of London have been used.

Most of the articles in the Encyclopedia include a bibliography giving sources of further information. Each bibliography consists of general items for further reading and/or references which cover specific aspects of the text. Where appropriate, authors are cited in the text using a name/date system as follows:

> ...as was recently reported (Smith 1990).

> Jones (1988) describes...

The contributors' names and the organizations to which they are affiliated appear at the ends of all the articles. All contributors can be found in the alphabetical List of Contributors, along with their full postal address and the titles of the articles of which they are authors or coauthors.

The most important information source for locating a particular topic in the Encyclopedia is the multilevel Subject Index, which has been made as complete and fully self-consistent as possible.

This page intentionally left blank

# ALPHABETICAL LIST OF ARTICLES


Abstract Data Types
Algorithms for Graphs
Algorithms: Theoretical Basis
Artificial Intelligence
Artificial Intelligence: Applications
Artificial Intelligence: Methods
Artificial Intelligence Software
Availability and Reliability of Software
Code Generation
Communication: Industrial Network Issues
Communication Networks for Control Systems
Communication Protocols in Control Systems
Compatibility and Standards for Software
Compiler Compilers
Compilers
Computer Architectures
Computers for Control
Concurrency
Concurrent Programming
Control System Architecture
Cryptology
Data Management in Plant Control
Data Structure and Algorithms
Databases
Databases, Intelligent
Design Methodologies
Diagnostic Software
Distributed Computing
Distributed Problem Solving
Embedded Systems
Examination and Measurement of Software
Fault Tolerance of Software
File Access Methods
Flexible Manufacturing Systems Software
Formal Methods
Functional Programming
Graphics
Hardware: Description Languages
Hardware: Logic Design Software
Hardware: Standards and Guidelines
Image Processing in Control
Impact Analysis and Hierarchical Inference
Information Management
Information Structuring by Hypertext
Knowledge Engineering
Knowledge Representation
Language Theory
Lifecycles
Logic Programming
Logic Programming, Parallel
Maintenance of Software
Modularization of Software
Natural Language Processing
Neural Networks
Numerical Control Software
Object-Oriented Programming
Operating Systems
Parallel Algorithms
Parallel Algorithms: Performance
Parallel Processing Structures
Petri Nets: An Introduction
Petri Nets: Application Issues
Portability of Software
Problem Domain Analysis
Procedural Programming Languages
Programming Tutors
Project Management: Network Models
Protection of Software
Protocol Engineering
Prototyping
Real-Time Control Software
Real-Time Software: Interprocess Communication
Real-Time Software: Validation and Verification
Requirements Capture
Robots: Software
Safety and Security of Software
Simulation
Simulation Languages: Taxonomy
Software Engineering Environments
Software Engineering Paradigms
Software for Personal Computers
Software Metrics
Software Models
Software Project Management
Specification and Verification of Software
Specification Languages
Systolic Algorithms for VLSI
Task Scheduling
Testing of Software
Testing of Systems Using Software
Time Concepts in Software
Translation, Verification and Synthesis: Comparison
Validation and Verification of Software
Virtual Machines

This page intentionally left blank

# A

## Abstract Data Types

An important factor in the design of software information systems is the modelling of real-world entities or objects. A model of an object will embody the essential characteristics which make it an object of that kind. For instance, if we were creating a model of a vehicle, we would give it characteristics such as wheels, a body, an engine, a gearbox, brakes and so on. Furthermore, individual *instances* of this model may have three or four wheels, a conventional or hatchback body, and a four- or five-speed gearbox. Attributes are not confined to those of the static (unchanging) variety. Other attributes of a vehicle are its speed, acceleration, gear employed and engine rotation, which are dynamic in nature. In software terminology, these models are known as *types*.

In the modelling of objects, many different levels of detail can be perceived. A software system to simulate the road handling of a motor car will need detailed code and data structures to imitate the grip of the car's tires on the road. At a much higher conceptual level, the designer of software to simulate a driver controlling the car along a particular stretch of road may want to program the car to change gear at a particular point. This designer should not need to have any knowledge of the detailed modelling of the gear system, and the effect that changing gear has on factors such as tire grip. Rather, the motor car should appear as an abstract object to which certain well-defined operations, such as change_gear, can be applied.

Abstract data types are based around this "need-to-know" principle. The concept of abstract data types is concerned with

(a) hiding details of the modelling of an entity that are irrelevant to the clients interacting with it, a technique known as information hiding; and

(b) only allowing communication with instances of the type through operations with well-defined interfaces.

### *1. Abstract Data Type Definition*

An abstract data type is a mathematical model of an entity together with a set of operations defined on that model. It has two components: a specification and an implementation.

#### *1.1 Specification*

An abstract data type specification states how an instance of that type appears to any external clients that use it, and is essentially composed of five parts.

(a) A name used to denote the type. This name usually refers to the entity being modelled.

(b) The sets used by the operations.

(c) The syntax of the operations (i.e., how they are denoted grammatically).

(d) The semantics of the operations (i.e., what the operations mean).

(e) Constraints on values of the type (if any).

The specification of the operations should be unrelated to the representation of the abstract data type in any particular programming language. For example, they are commonly defined as mathematical functions, a technique explored in Sect. 2.1.

Operations can be placed into three main categories:

(a) constructor operations, which create an instance of the type;

(b) operations that take a type instance as an argument, and return a modified instance to the client; and

(c) inquiry operations, which return information about the state of an object.

#### *1.2 Implementation*

The implementation of an abstract data type is an emulation of the specification that allows the type to be represented on a real machine. It defines the representational details of the data structures used to implement the type, together with the code for each of the operations in the type specification. The implementation of the operations should comply with their semantic definitions in the specification. However, proving that an implementation is correct with respect to its semantic specification is a nontrivial task, requiring a mathematical description of the implementation language.

As long as the interface the operations present to clients remains fixed, the implementation can be changed at will, without having repercussions on the clients. This is an important benefit of using abstract data types, as it allows the implementer to explore the speed/space/time trade-offs of different algorithms, as well as removing any knock-on effects of code maintenance.

The distinction between specification and implementation is most apparent with imperative languages. Other programming paradigms (see Sect. 3.3) allow an abstract data type to be almost entirely defined by its specification, with a minimum of representational information.

### *2. Abstract Data Type Specification Techniques*

There are two prominent methods of specifying abstract data types: algebraic specification and operational specification. These are described below.

### *2.1 Algebraic Specification*

Algebraic specification (otherwise called axiomatic specification) is probably the most common way of specifying abstract data types. Some notational languages exist that use this method to define abstract data types, notably OBJ (Goguen and Tardo 1986), Clear (Burstall and Goguen 1981), and Larch (Guttag *et al.* 1985).

Using the algebraic specification technique, the operations on an abstract data type are specified as mathematical functions. The prominent feature of this method is that the operations are defined in terms of each other. No operations external to the type being defined are employed.

The classic object to illustrate this technique is a stack. A stack is a list to which items can be added or removed from one end only. Notice that the essential properties of a stack say nothing about the type of objects that will reside in the stack. A stack which held integers could still be specified by an abstract data type, but this would introduce factors into the specification that were unrelated to the intrinsic properties of a stack. An abstract data type that has operations that are not constrained to work with particular types is known as a generic type.

Clients using the stack abstract data type may wish to:

(a) create a new stack;

(b) place ("push") items onto the stack;

(c) remove ("pop") items from the stack;

(d) access the top value on the stack;

(e) test whether the stack is empty.

These operations can be denoted by the functions Init, Push, Pop, Top, and Empty respectively. An algebraic specification of an abstract data type has the constituents described in Sect. 1, with the operations being defined as mathematical functions. The section of the specification that distinguishes the algebraic method is the definition of the operation semantics. This section consists of axioms, which relate the functions to each other.

Figure 1 shows the algebraic specification for the Stack type. Notice that the stack being defined here is unbounded. Any constraints on the size of the stack would be placed in a fifth section. Also, the special value error is used to indicate that an invalid argument has been given to a particular operation. Error handling poses awkward problems in abstract data types. One solution is to add the special value error to the domain of the Stack set. In this case, to be sufficiently rigorous, the action of operations should also be defined when they receive error as an argument—for example: Pop (error) = error.

An important consideration is whether the set of axioms that specify the stack operations is complete. In other words, is there a possible function call that could be made for which the result cannot be determined from the axioms? Also, is there a possible state that a stack can be in that cannot be inferred from the axioms? Answering no to either of these questions means that the axioms are incomplete. This example is so simple that it can be recognized by inspection that the axioms cover all eventualities; however, most abstract data types are much more advanced in terms of complexity, requiring formal proof of completeness.

**Type** Stack

**Sets** Stack, Item, Boolean

**Operations**
Init () → Stack
Push (Item, Stack) → Stack
Pop (Stack) → Stack
Top (Stack) → Item
Empty (Stack) → Boolean

**Axioms**
Empty (Init ()) = True
Empty (Push (v, s)) = False
Top (Push (v, s)) = v
Top (Init ()) = error
Pop (Push (v, s)) = s
Pop (Init ()) = error

***Figure 1***
Algebraic specification of a stack

More than a cursory glance at the axioms is required to realize that the object being modelled in Fig. 1 is indeed a stack. A problem with specifications of this kind is that they are very far removed from most people's intuitive ideas of the object they represent. A consequence of this is that it requires substantial skill to create specifications of more complex objects.

### *2.2 Operational Specification*

Operational specification differs from algebraic specification in that the semantics of the operation of a type are specified in terms of other operations outside its domain. Essentially, a mathematical model of the type is constructed (hence this technique is also called constructive specification), and the operations are defined in terms of that model. The external operations used may be those of another abstract data type, or they may be well-defined mathematical operations, such as those of Boolean algebra.

A characteristic of this method is that it specifies the semantics of operations in terms of preconditions (conditions that must be true before applying the operation) and postconditions (those that will be true after the operation), together with invariants (conditions that are always true). Some notational techniques based on this method are VDM (Jones 1986) and Z (Spivey 1989).

Figure 2 shows the specification of the semantics of a stack in VDM. Mathematically, a stack can be represented by a sequence. The sequence operators of

```
Stack = seq of Element

INIT () r : Stack
post r = []

PUSH (e : Element, st : Stack) r : Stack
post r = [e]⌢st

READ (st : Stack) r : Element
pre st ≠ []
post r = head st

POP (st : Stack) r : Stack
pre st ≠ []
post ∃ e ∈ Element . [e]⌢r = st

EMPTY (st : Stack) r : Bool
post r ⇔ st = []
```

***Figure 2***
VDM specification of a stack

concatenation and head extraction that the stack operations use are of a fundamental nature, and are presumed to be well understood.

## *3. Programming Language Support for Abstract Data Types*

### *3.1 Imperative Languages*

The first language to incorporate a means of supporting abstract data types was CLU (Liskov *et al.* 1977). More modern languages such as Ada (Sommerville and Morrison 1987) and Modula-2 (King 1988) have given more consideration to abstract data types at the language design stage.

As earlier stated (Sect. 1.2), the model of execution of imperative languages has to emulate the operation functions. Owing to the inherent nature of imperative languages, they do not always implement operations as functions. Often, operations that change the state of a type instance will not create and return a new type instance, but will use side-effects to modify the original instance belonging to the client.

Ada will be used to exemplify the handling of abstract data types in imperative languages. An abstract data type in Ada is composed of two parts.

(a) A specification part declares the name of the type and the parametric interface of the operations that are used to manipulate type instances. This is the only part that is visible to clients of the type.

(b) An implementation part is the algorithmic definition of the operations, and is not accessible to clients. All the representational details of the type should be encapsulated in this section.

The specification part of an Ada abstract data type lacks a means of stipulating what the type operations should do (i.e., their semantics). The Anna system (Luckham and von Henke 1985) goes some way towards this end by allowing Ada programs to be augmented with full specifications.

```
generic
type Stack_Elt is private;
package Stack is

type S is private;

procedure Init (A_Stk : in out S);
procedure Push (value : in Stack_Elt; A_Stk : in out S);
procedure Pop (A_Stk : in out S);
function Top (A_Stk : in S) return Stack_Elt;
function Empty (A_stk : in S) return BOOLEAN;

private
type Stk;
type S is access Stk;

end Stack;
```

***Figure 3***
Ada specification of a stack

To illustrate abstract data type definitions in Ada, the stack will again be used. The set of stack operations discussed in Sect. 2.1 leads to the Ada specification shown in Fig. 3.

The two classical implementations of a stack are as an array and as a linked list. The latter is used for the implementation shown in Fig. 4.

```
package body Stack is
type Stk is record
  Val : Stack_Elt;
  Next : S;
end record;

procedure Init (A_Stk : in out S) is
begin
  S := null;
end Init;

procedure Push (Value : in Stack_Elt; A_Stk : in out S) is
begin
  A_Stk := new Stk'(Value, A_Stk);
end Push;

procedure Pop (A_Stk : in out S) is
begin
  A_Stk := A_Stk.Next;
end Pop;

function Top (A_Stk : in S) return Stack_Elt is
begin
  return A_Stk.Value;
end Top;

function Empty (A_Stk : in S) return BOOLEAN is
begin
  return A_Stk = null;
end Empty;
```

***Figure 4***
Ada implementation of a stack

```
abstype 'a Stack = NEW_STACK|PUSH of 'a Stack
with
  fun Init () = NEW_STACK
  fun Push (v, s) = PUSH (v, s)
  fun Pop (PUSH (v, s)) = s
  fun Top (PUSH (v, s)) = v
  fun Empty (NEW_STACK) = true
  |     Empty (PUSH (v, s)) = false
end;
```

*Figure 5*
SML specification of a stack

### 3.2 Object-Oriented Languages

Object-oriented languages are based on the concept of classes. Classes are analogous to abstract data types, and consist of definitions of the data structures needed to represent an instance (or an object) of the class, along with the operations that can be performed on an instance. Smalltalk (Goldberg and Robson 1983) is perhaps the supreme example of a pure object-oriented language. More recent languages such as C++ (Spivey 1989) and Eiffel (Moyer 1988) are based on imperative languages, but accommodate object-oriented concepts such as classes. They include facilities to restrict the interaction with a class instance to a strict set of operations.

### 3.3 Functional Languages

Modern functional languages such as SML (Wikström 1988) and Haskell (Hudak *et al.* 1990) have extensive abstract data type facilities, which have close associations with algebraic specification. Functional languages allow abstract data types to be declared in terms of constructor operations, which are used to construct instances of the type, and functions, which are used to inspect and manipulate the instances. Representational details of abstract data types are minimized or excluded altogether. Genericity of abstract data types is also a feature of modern functional languages. The SML language is used in Fig. 5 to specify a generic stack abstract data type. The Stack type is declared to have two constructor operations: NEW_STACK and PUSH. NEW_STACK creates an empty stack, while PUSH takes a stack as an argument and produces a new stack. These constructor operations are not directly accessible to clients of the type—all interaction with a type instance must be made by using the visible functions Init, Push, Pop, Top and Empty, which are defined in terms of the constructors. Note that no attempt at handling errors (e.g., trying to evaluate POP (NEW_STACK)) has been made. To accommodate error-handling, an extra constructor called ERROR could be added to the Stack type, and the operations defined over it, as described in Sect. 2.1.

*See also:* Data Structure and Algorithms; Knowledge Representation; Procedural Programming Languages

## Bibliography


Azmoodeh M 1988 *Abstract Data Types and Algorithms*. Macmillan, London

Burstall R M, Goguen J A 1981 An informal introduction to specifications using CLEAR. In: Boyer R S, Stothers-Moore J (eds.) *The Correctness Problem in Computer Science*. Academic Press, London

Goguen J, Tardo J 1986 An introduction to OBJ: a language for writing and testing algebraic program specifications. In: Gehani N, McGettrick A D (eds.) *Software Specification Techniques*. Addison-Wesley, Reading, MA

Goldberg A, Robson D 1983 *Smalltalk 80: The Language and its Implementation*. Addison-Wesley, Reading, MA

Guttag J 1977 Abstract data types and the development of data structures. *CACM* **20**, 6

Guttag J 1980 Notes on type abstraction. *IEEE Trans. Soft. Eng.* **6**, 1 (January)

Guttag J, Horning J, Wing J 1985 The LARCH family of specification languages. *IEEE Software* **2**, 5

Guttag J, Horowitz E, Musser D 1977 Some extensions to algebraic specifications. *SIGPLAN Not.* **12**, 3

Guttag J, Horowitz E, Musser D 1978 The design of data type specifications. In: Yeh R (ed.) *Current Trends in Programming Methodology*, Vol. IV. Prentice-Hall, Englewood Cliffs, NJ

Guttag J, Horowitz E, Musser D 1978 Abstract data types and software validation. *CACM* **21**, 12

Hudak P, Peyton-Jones S, Wadler P, Boutel B, Fairbairn J, Fasel J, Guzmán M M, Hammond K, Hughes J, Johnsson T, Kieburtz R, Nikhil R, Partain W, Peterson J 1992 Report on the programming language Haskell: a non-strict, purely functional language, version 1.2. *SIGPLAN Not.* **27**, 5

Jones C B 1980 *Software Development: A Rigorous Approach*. Prentice-Hall, Englewood Cliffs, NJ

Jones C B 1986 *Systematic Software Development Using VDM*. Prentice-Hall, Englewood Cliffs, NJ

King K 1988 *Modula-2: A Complete Guide*. Heath, Lexington, MA

Liskov B H 1988 Data abstraction and hierarchy. *SIGPLAN Not.* **23**, 5

Liskov B H, Snyder A, Atkinson R, Schaffert C 1977 Abstraction mechanisms in CLU. *CACM* **20**, 8

Liskov B H, Zilles S N 1974 Programming with abstract data types. *Proc. ACM SIGPLAN Conf. Very High-Level Languages. SIGPLAN Not.* **9**, 4

Liskov B H, Zilles S N 1975 Specification techniques for data abstractions. *IEEE Trans. Soft. Eng.* **1**, 1

Luckham D C, von Henke F W 1985 An overview of Anna, a specification language for Ada. *IEEE Software* **3**

Martin J J 1986 *Data Types and Data Structures*. Prentice-Hall, Englewood Cliffs, NJ

Meyer B 1988 *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ

Shaw M 1984 Abstraction techniques in modern programming languages. *IEEE Software* **1**, 4

Sommerville I, Morrison R 1987 *Software Development with Ada*. Addison-Wesley, Reading, MA

Spivey J M 1989 *The Z Notation*. Prentice-Hall, Englewood Cliffs, NJ

Stroustrup B 1986 *The C++ Programming Language*. Addison-Welsey, Reading, MA

Wikström A 1988 *Functional Programming Using Standard ML*. Prentice-Hall, Englewood Cliffs, NJ

R. S. Turner
[University of Hull, Hull, UK]

# Algorithms for Graphs

Graphs have applications in nearly all areas of science, and have been studied by pure and applied researchers for hundreds of years. Owing to their suitability for modelling many aspects of computation, they have also been scrutinized, since the advent of the electronic computer, from an algorithmic point of view; a study of the resources (i.e., time and space) required to solve certain problems concerning graphs has been undertaken. For example, it has been shown that even the most simply stated graph problem may require an inordinate amount of time to solve (longer than the age of the universe), and upon this analysis many banks base their secret codes for the transfer of funds. Not only do graphs model aspects of computing science but the programmer uses them explicitly as data structures within programs, and also in the design phase of programs. Algorithms for manipulating graphs can then be applied—to facilitate the fast retrieval of data, for example, or to establish possible redundancies in some design.

## *1. Basic Definitions*

This article attempts to show how solutions to algorithmic problems in graph theory are of considerable importance to many practical problems arising in software engineering. It begins by giving the basic graph-theoretical definitions relating to this paper and presents a brief introduction concerning the importance of polynomial time algorithms as compared to exponential time algorithms. It then examines three specific problems concerning graphs (and digraphs) before giving applications of algorithms for solving these problems in software engineering. The problems encountered in this paper are sufficiently complex that to present an algorithm for the solution of one of them (and the proof of correctness) would take up too much space. Consequently, the algorithms presented are relatively simple and are given in a colloquial style—that is, as an English description. We emphasize an understanding of the graph-theoretical problem and the applications of any algorithm for the solution of the problem, as opposed to detailing actual algorithms, although references are given as to where appropriate algorithms may be found.

A graph $G=(V,E)$ of size $n$ consists of a set $V$ of $n$ vertices, usually numbered 1 to $n$, and a set $E$ of (undirected) edges, each edge being a pair of vertices. A graph can be represented diagrammatically by drawing points to represent the vertices and lines joining points to represent the edges. A directed graph, or digraph, is similar except that each (directed) edge is an ordered pair of vertices, with the edges of a digraph represented diagrammatically as arrows from one vertex to another. Notice that in a graph an edge between vertex $i$ and vertex $j$ is also regarded as an edge between vertex $j$ and vertex $i$, but in a digraph this is not so. In a digraph, it might be the case that there is an edge between vertex $i$ and vertex $j$ but none between vertex $j$ and vertex $i$. A vertex $i$ of a graph or a digraph is called a neighbor of a vertex $j$ if there is an edge between vertex $j$ and vertex $i$.

A graph (or digraph) lends itself very well to computer representation. The two main representations of a graph $G$ are as an adjacency matrix or as a series of linked lists. An adjacency matrix is a two-dimensional array $M$ of size $n$, with $M[i,j]=1$ if there is an edge between vertex $i$ and vertex $j$, and 0 otherwise (notice that $M$ is symmetrical when $G$ is a graph, but need not be so when $G$ is a digraph). The linked-list representation consists of $n$ linked lists, one for each vertex, with list $i$ detailing the neighbors of vertex $i$. The linked-list representation tends to be used when it is known that there are $O(n)$ edges in the graph; that is, when the graph is "sparse".

Having defined graphs and digraphs, we now consider algorithms concerning graphs. The ability of graphs to model many real-life situations, coupled with the fact that they may be easily manipulated by a computer, has made the study of graph algorithms a flourishing field. However, even with modern-day computers, the best algorithms found so far for completing some of the (seemingly) most simple of operations on graphs are not practical, even for graphs of relatively small size. For example, it should be clear that one can write a computer program that determines whether there is a tour, along adjoining edges, of all the vertices in a graph. It is unknown whether there is such an algorithm that yields the answer within a polynomial (in the size of the graph) number of operations; that is, in "polynomial time." All such algorithms currently available use an exponential number of operations; that is, they run in "exponential time" (an exponential function of $n$ blows up relatively soon, compared to a polynomial one).

A decision problem is a problem consisting of a set of instances (e.g., the set of all graphs) with a subset of yes-instances (e.g., the set of graphs that have a tour of all vertices). An algorithm solves some decision problem if it determines whether a given instance is or is not a yes-instance of the problem. We denote the class of problems solvable in polynomial time by P, and the class of problems solvable in exponential time by EXPTIME. There is another class of decision problems, NP, containing P and contained within EXPTIME, that contains many of the exponential-time decision problems not known to be solvable in polynomial time, and also many of the decision problems encountered in graph theory and computer science in

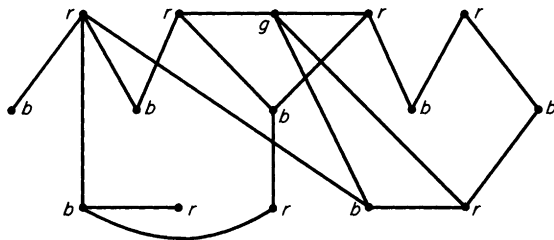


***Figure 1***
An optimally colored graph (where $r$ = red, $b$ = blue and $g$ = green

general. It is not known whether the inclusions $P \subseteq NP \subseteq EXPTIME$ are strict: moreover, it can be shown that there are certain decision problems in NP, termed NP-complete problems, with the property that if it can be shown that any NP-complete problem is in P, then the classes of problems P and NP necessarily coincide. There are other problems, not necessarily decision problems (e.g., the problem of obtaining a tour of all vertices in a graph) nor necessarily belonging to the class NP, with the property that if any of these problems can be solved in polynomial time, than P and NP coincide: these are termed "NP-hard" problems.

## 2. *Examples of Algorithms*

We now consider some problems concerning graphs and digraphs for which a "good" (i.e., polynomial time) algorithm might be required (although it may be the case that we have yet to find one), and we present some applications of the algorithms within certain areas of software engineering. The reader is referred to Aho *et al.*(1974) and Garey and Johnson (1979) for more information on measuring the time complexity of an algorithm (i.e., how fast it runs in terms of the size of the input); and to Harary (1969), Christofides (1975), Minieka (1978) or Golumbic (1980) for more information on graphs and their algorithms.

### *2.1 Coloring Graphs*

A (proper) coloring of a graph $G$ is an assignation of a color to each vertex of $G$ such that if two vertices are joined by an edge then they have different colors. An optimal coloring is a coloring using the least number of colors possible: an optimally colored graph is shown in Fig. 1. It has been shown (cf. Garey and Johnson 1979) that the problem of obtaining an optimal coloring of an arbitrary graph is NP-hard, and so all known algorithms run in (at least) exponential time.

An algorithm for obtaining an optimal coloring of an arbitrary graph (which is better than the obvious "try-all-possibilities" algorithm) can be found in Christofides (1975). However, owing to the NP-hardness of the problem, we shall consider an approximate algorithm running in polynomial time—that is, an algorithm that gives a good, but not necessarily optimal, coloring (i.e., the number of colors used is usually near to optimal). Consider the following algorithm:

> Color the vertices in turn, with colors represented by the positive integers, such that a vertex is colored by the integer of lowest value not used by its colored neighbors.

This algorithm can be shown to run in $O(n^2)$ time, but the number of colors used is highly sensitive to the order in which the vertices are colored. (The reader is referred to Gibbons (1985) and Garey and Johnson (1979) for a discussion concerning approximate algorithms.)

Another method for obtaining polynomial-time optimal coloring algorithms is to restrict the class of graphs to which the algorithm applies. As we shall see, it is often the case that the graphs arising from a real problem are of a specific type. In a following application, the class of interval graphs is considered, where an interval graph $G$ has a set of vertices $V$ corresponding to some intervals of the real line (or of some well-ordered set) and there is an edge between two vertices if and only if the corresponding intervals intersect. Not all graphs can be realized as interval graphs. It has been shown—see, for example, Gupta *et al.* (1982)—that interval graphs can be optimally colored in polynomial time. For examples of other classes of graph for which there exist polynomial-time optimal coloring algorithms, the reader is referred to Appel and Haken (1977a), Appel and Haken (1977b), Garey and Johnson (1979), Tucker (1984), Hsu (1986), and Stewart (1989).

As an example of when a graph-coloring algorithm might be required, consider the following problem. Tasks $\{t_1, \ldots, t_n\}$ are to be executed using some available processors, and we need to assign some processor to each task so that the tasks are completed as quickly as possible. It is known that each available processor is capable of completing any task within one unit of time. However, these tasks are such that each requires continual read-only access to some portion of the memory $M$ shared by the processors, and any memory cell of $M$ may be read by only one processor at once. Consequently, when a processor is involved with some task and has access to the relevant area of memory, no other processor can access that memory portion.

Denote the portion of memory required by the task $t_i$ as $M_i$. In order to allocate processors to tasks, form a graph $G$ on $n$ vertices, with the vertex $i$ corresponding to task $t_i$. Join two vertices $i$ and $j$ with an edge if and only if $M_i$ and $M_j$ have at least one memory cell in common. This implies that the tasks $t_i$ and $t_j$ cannot be executed in parallel by different processors. Now color the graph $G$, and note that if two vertices have the same color then the corresponding tasks may be completed simultaneously. An optimal coloring of $G$ clearly corresponds to completing the tasks in the shortest amount