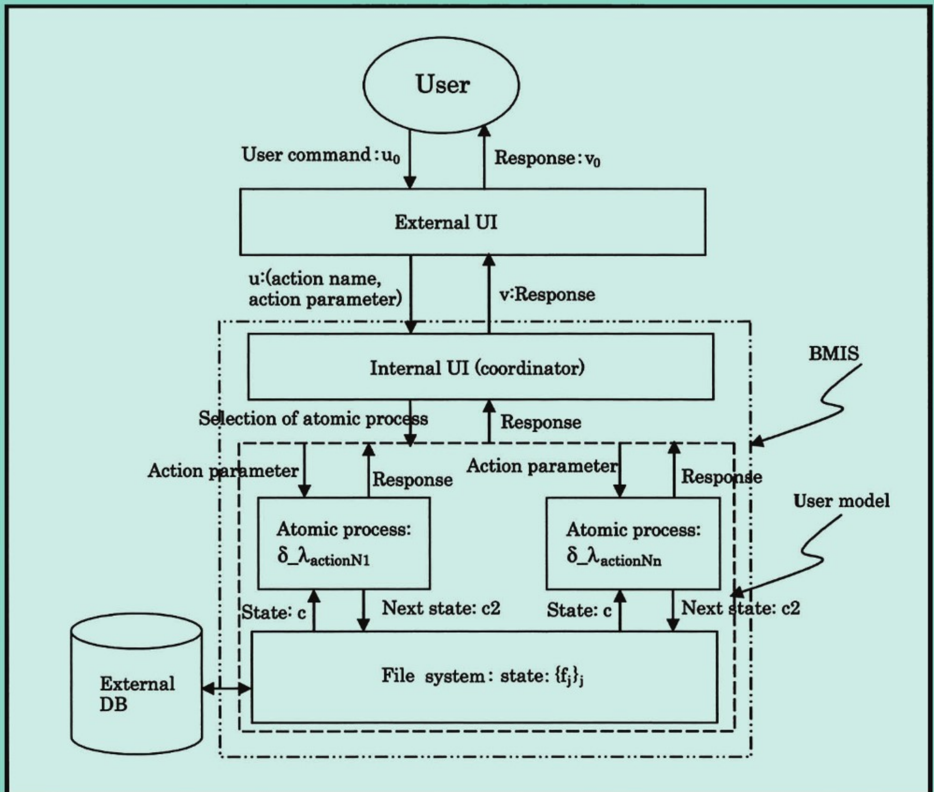
FOUNDATIONS AND APPLICATIONS OF MIS

A Model Theory Approach

YASUHIKO TAKAHARA
YONGMEI LIU



IFSR International Series on Systems Science and Engineering
Volume 24

FOUNDATIONS AND APPLICATIONS OF MIS

A MODEL THEORY APPROACH

International Federation for Systems Research International Series on Systems Science and Engineering

Series Editor: George J. Klir
State University of New York at Binghamton

Editorial Board

Gerrit Broekstra
Erasmus University, Rotterdam, The Netherlands

John L. Casti
Sante Fe Institute, New Mexico

Brain Gaines
University of Calgary, Canada

Ivan M. Havel
Charles University, Prague, Czech Republic

Klaus Kornwachs
Technical University of Cottbus, Germany

Franz Pichler
University of Linz, Austria

Volume 19 *FLEXIBLE ROBOT DYNAMICS AND CONTROLS*
Rush D. Robinett, III, Clark R. Dohrmann, G. Richard Eisler, John T. Feddema, Gordon G. Parker, David G. Wilson, and Dennis Stokes

Volume 20 *FUZZY RELATIONAL SYSTEMS: Foundations and Principles*
Radim Bělohlávek

Volume 21 *ARCHITECTURE OF SYSTEMS PROBLEMS SOLVING, Second Edition*
George J. Klir and Doug Elias

Volume 22 *ORGANIZATION STRUCTURE: Cybernetic Systems Foundation*
Yasuhiko Takahara and Mihajlo Mesarovic

Volume 23 *CONSTRAINT THEORY: Multidimensional Mathematical Model Management*
George J. Friedman

Volume 24 *FOUNDATIONS AND APPLICATIONS OF MIS: A Model Theory Approach*
Yasuhiko Takahara and Yongmei Liu

IFSR was established "to stimulate all activities associated with the scientific study of systems and to coordinate such activities at international level." The aim of this series is to stimulate publication of high-quality monographs and textbooks on various topics of systems science and engineering. This series complements the Federation's other publications.

A Continuation Order Plan is available for this series. A continuation order will bring delivery of each new volume immediately upon publication. Volumes are billed only upon actual shipment. For further information please contact the publisher.

Volumes 1-6 were published by Pergamon Press.

FOUNDATIONS AND APPLICATIONS OF MIS

A MODEL THEORY APPROACH

Yasuhiko Takahara

Tokyo Institute of Technology
Tokyo, Japan

and

Yongmei Liu

Central South University
Changsha, Hunan, China

Yasuhiko Takahara
Professor Emeritus
Tokyo Institute of Technology
Ookayama, Meguro
Tokyo, Japan
yasutk@sd6.so-net.ne.jp

Yongmei Liu
School of Business Administration
Central South University
Changsha, Hunan
China
ymliu@mail.csu.edu.cn

Series Editors:
George J. Klir
Thomas J. Watson School of Engineering
and Applied Sciences
Binghampton University
Binghampton, NY 13901

Library of Congress Control Number: 2006920277

ISBN-10: 0-387-31414-8 e-ISBN: 0-387-35840-4
ISBN-13: 978-0-387-31414-3

Printed on acid-free paper.

©2006 Springer Science+Business Media, LLC
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, U.S.A.), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed in the United States of America. (KeS/IBT)

9 8 7 6 5 4 3 2 1

springer.com

Preface

This book has three objectives. The first is to present a new methodology for management information system (MIS) development as an application of mathematical general systems theory; the second is to establish a theoretical foundation for MIS development; and the third is to demonstrate the utility of set theory combined with extended Prolog (extProlog) for MIS development.

Traditionally, MISs have been developed from a software engineering perspective, leading to the problem that MIS development is limited to the operational framework allowed for by the lifecycle approach or its variations. Thus, while the lifecycle approach sets a starting point for MIS systems engineering, it is unable to provide an operational framework that is more than simply common sense in systems engineering. Historically, the "hard" systems methodology was proposed in engineering before the inception of the lifecycle approach, and although developed independently, the two approaches are almost identical. A criticism of the hard systems methodology is that it is merely engineering common sense and does not provide operational theory. To establish a more sophisticated theory for MIS development, the problem must therefore be investigated from a viewpoint other than a software engineering perspective. This book sets out to demonstrate that a systems theory approach or model theory approach is a more satisfactory alternative that accounts for the systems nature of an MIS.

In general, MIS development is currently performed without a formal disciplinary framework, and as such remains somewhat of an art. This problem has been pointed out repeatedly by those in pursuit of formal approaches to information systems development as a real engineering discipline. However, it should be noted that although formalism plays a fundamental role in the construction of a solid operational theory for MIS development, a simple formalism that does not include systems concepts is not sufficient.

To overcome these problems, a formal (general) systems theory approach to MIS development was developed. The model theory approach presented in this book is one realization of the formal systems theory approach, and can be summarized as follows:

- It is not a software engineering approach but a systems approach. In particular, it views an MIS as a hierarchical system consisting of three components: a transaction processing system, a data transformation system, and a problem-solving/decision support system. A model-theoretic structure is derived for each component based on general systems theory concepts.
- It is a formal approach in the sense that a model is constructed based on the formal structures and described in set theory.
- A system, either a transaction-processing system or problem-solving system, is constructed of two components: a user model component and a standardized component. The former component is dependent on the target problem, and the latter is independent of the problem. System construction is supplemented by the latter component.
- The user model component can be translated into an executable system by automatic system generation. Upon translation, the standardized component is attached to the user model. Thus implementation is an integral part of the approach, facilitating rapid systems development.
- The implementation language is extProlog.

The model theory approach adopts extProlog, an extension of regular Prolog, as the implementation language to treat numerical problems as well as symbolic problems. This language has a narrow semantic and syntactic gap with the set-theoretic description, allowing a model described in set theory to be translated into an extProlog model automatically. The implementation stage can thus be included as an integral part of the model theory approach. This addresses the two criticisms of formal approaches: that they are mainly concerned with the specification of systems development and do not positively address the implementation stage, and that it is perceived that advanced mathematical knowledge is necessary for formal approaches. This book asserts that working knowledge of elementary set theory and logic should be considered easy and indispensable for every professional systems engineer.

The model theory approach claims the following advantages over standard approaches:

- It provides a reliable system specification (a general assertion for formal approaches).
- It facilitates reliable implementation and rapid systems development by providing a user interface (for the transaction processing system) and a goal-seeker (for the problem-solving system) as black box components for MIS development coupled with automatic systems generation. Once a system specification has been given in computer-acceptable set theory, an executable transaction processing or problem-solving system can be produced. The executable system is expected to be a correct realization of the specification.
- It may reduce the cost of systems development by providing a means of accelerating development and by being usable on open-source software (Linux, Apache, postgreSQL, PHP).
- It can realize an intelligent MIS that accounts for both problem-solving and transaction processing functions on an integrated platform.

- It facilitates end-user maintenance by allowing system construction to be performed using elementary set theory rather than a computer language.

An MIS is modeled as a three-layer system in the model theory approach, where the lowest layer is the transaction processing system and the highest layer is the problem-solving/decision support system. These layers are linked by an intermediate layer that acts as a data transformation system, such as a data mining system. The model theory approach is applied to all three layers to derive formal systems-theoretic structures and to specify development procedures based on these structures.

This book consists of six parts. The first part, "New Paradigm of Systems Development," outlines the model theory approach and explains the intention of the model theory approach, which is to support end-user development by exploiting the increasing power of information technology.

The second part, "Model Construction Language and Systems Implementation Language," introduces set theory as a basis for model construction in subsequent chapters and extProlog as an implementation language. A restricted form of set theory called computer-acceptable set theory is introduced to describe the user model. The current approach allows the user model to be translated into an executable system automatically. Familiarity with set theory may be all that is required to develop a system by the model theory approach. Prolog, which is the basis for extProlog, is not a popular programming language, since it is typically understood to be a special language for artificial intelligence applications. However, extProlog is promoted here as a general-purpose language with artificial intelligence as a suitable application. Construction methods for graphical user interfaces in extProlog are also provided as indispensable components for MIS development, although a standard browser-based user interface is used for the transaction processing system.

The third part, "Model Theory Approach to Solver System Development," presents the model theory approach to problem-solving system development, focusing on the main engine of the system: the solver. The structure of the solver is determined by a model-theoretic structure representation and shown to be decomposable into two components: one dependent on the target system (user model), and another independent component (standardized goal-seeker). The former must be prepared by a systems developer, while the latter is produced as a standardized goal-seeker and is provided as a black box component for systems development. The systems development procedure is derived based on this decomposition. The identified structure of the solver provides a means of classification for solver development, and thus assists in making the system development process operational.

The fourth part, "Solver System Applications," demonstrates the development of a solver for each class of problem using a typical example. The last chapter of this part presents a formalization of the entire structure of the problem-solving system as a "skeleton model" and a construction of an intelligent data mining system as an example. The standard data mining system as the intermediate layer is treated as a problem-solving system in this book.

The fifth part, "Model Theory Approach to Transaction Processing Systems Development," discusses transaction processing system development in the model theory

approach. The structure of a transaction processing system is also identified as a model-theoretic structure, and is similarly decomposed into two components: a user model component that is dependent on the target system, and a user interface component that is independent of the target system. The systems development procedure is derived based on this decomposition, in which the independent user interface is provided as a black box component. Systems that include intrinsic problem-solving components but which are usually treated as transaction processing systems are also discussed as a specific class of systems that require application of the model theory approach to deal with both the transaction processing system and the problem-solving system in the same framework. The discussion includes the definition of a browser-based intelligent MIS and an example. An intelligent MIS is the ultimate target of the model theory approach. Database connectivity is also discussed. The model theory approach requires, in particular, object-oriented database connectivity with the transaction processing system.

The sixth part, "Theoretical Basis for extProlog," presents the theoretical foundation of extProlog as background for the model theory approach.

This book is written for those wanting to know how a theoretical foundation can be established for MIS development, how a real working MIS can be developed based on formal systems theory, and conversely, how formal systems theory can be applied to the real world of information practices. It is assumed that the reader has working knowledge of elementary set theory and logic and is familiar with some systems concepts such as automaton models.

A draft of this book has been used at senior and higher levels as a textbook for information-oriented systems engineering courses. The content has been reviewed by a professional systems engineering group and can therefore be used as a textbook for advanced undergraduate and graduate courses as well as a reference for systems engineers and systems scientists. If the reader wishes to quickly understand the idea of the model theory approach, he is recommended to first read Chapter 9 on problem-solving system development and Sections 14.5, 14.6, and 14.7 on transaction processing system development. Chapters and sections marked with $*$ can be skipped at the first reading.

The development of the model theory approach benefited from numerous discussions and contributions from many people since 1985. Initial research began at the Tokyo Institute of Technology (Tokyo, Japan) and now continues at the Chiba Institute of Technology (Chiba, Japan). The authors would like to express their gratitude to all the research partners and former students who have made contributions to the development of the model theory approach and who have applied the approach to various problems. In particular, we must first thank Nomura Research Institute (Tokyo, Japan), which gave financial support and cooperated with the authors' research at T.I.T. Mr. Shimazu, President of Fusion Ltd. (Japan) is gratefully acknowledged for providing the opportunity to work on data mining systems, and without whose help the scope of this approach would be substantially restricted. Prof. X. Chen, Central South University (Changsha, China), also gave the authors many opportunities to present this work in graduate courses at C.S.U. Prof. R. Banerji, Emeritus of St. Joseph's University and Prof. N. Shiba of the Chiba Institute of Technology are humbly acknowledged

for patient discussions on the theoretical aspects of this book. Our appreciation is also extended to the members of the Formal Approach Study Group of the Japan Society of Management Information Systems, which is directed by Prof. T. Saito of Nihon University (Chiba, Japan), for valuable feedback.

Yasuhiko Takahara
Yongmei Liu

2006

Contents

Preface v

Part I New Paradigm of Systems Development

1 **New Systems Development Methodology: The Model Theory Approach** 3
1.1 Necessity for a New Systems Development Methodology for Management Information Systems 3
1.2 Outline of Model Theory Approach 5
1.3 Example of Model Theory Approach to Systems Development 12
Appendix 1.1 User Model in extProlog Generated by Setcompiler: Pattern-Finding Problem 17
Appendix 1.2 User Model in extProlog Generated by Manual: Pattern-Fitting Problem 18

Part II Model Construction Language and Systems Implementation Language

2 **Computer-Acceptable Set Theory for Model Construction** 23
2.1 Implementation Structure of User Model in Computer-Acceptable Set Theory 23
2.2 Well-Formed Formula of First-Order Predicate Calculus 24
2.3 Basic Notation for Computer-Acceptable Set Theory 25
2.4 Sets 27
2.5 Predicate (Relation) 29
2.6 Functions 30
2.7 User Model Description in Set Theory 34
2.8 User Model Compilation 36
2.9 Input–Output Operations in Set Theory 39
Appendix 2.1 System-Defined Predicates and Functions Used in This Book 39

3 Implementation Language: extProlog* ... 43
3.1 Extended Prolog (extProlog) ... 43
3.2 Examples ... 45
3.3 Basic Syntax ... 52
3.4 extProlog as General-Purpose Programming Language ... 53
3.5 Graphical User Interface in extProlog ... 59
3.6 Execution Speed of extProlog ... 59
Appendix 3.1 Graphical User Interface in extProlog ... 60

Part III Model Theory Approach to Solver Systems Development

4 Model Theory Approach to Solver System Development: Outlines ... 67
4.1 Model Theory Approach to extSLV Development ... 67
4.2 Design Procedure of extSLV ... 73
4.3 Classification of Problems for extSLV ... 74
4.4 Implementation Structure of extSLV ... 75

5 User Model and Standardized Goal-Seeker* ... 79
5.1 User Model for the E-C-C Case ... 79
5.2 User Model for I-O-O Case ... 83
5.3 Standardized Goal-Seeker by Modified Dynamic Programming Method ... 86
5.4 Standardized Goal-Seeker by Hill Climbing with Push-Down Stack Method ... 91
Appendix 5.1 Standard DP Solver ... 99
Appendix 5.2 Standard DP Solver ... 102
Appendix 5.3 Proof of Proposition 5.1 ... 106
Appendix 5.4 Proof of Proposition 5.2 ... 106
Appendix 5.5 Proof of Proposition 5.3 ... 106
Appendix 5.6 Proof of Proposition 5.4 ... 106
Appendix 5.7 Proof of Proposition 5.5 ... 107
Appendix 5.8 Proof of Theorem 5.3 ... 107
Appendix 5.9 Proof of Proposition 5.6 ... 107
Appendix 5.10 Proof of Proposition 5.7 ... 108
Appendix 5.11 Proof of Theorem 5.4 ... 108
Appendix 5.12 Proof of Proposition 5.8 ... 108

Part IV Solver System Applications

6 Traveling Salesman Problem: E-C-C Problem ... 113
6.1 Traveling Salesman Problem ... 113

6.2 User Model of Traveling Salesman Problem for DP Goal-Seeker ... 113
6.3 Implementation in extProlog ... 117
6.4 Tuning of Goal-Seeker ... 118
Appendix 6.1 User Model for Traveling Salesman Problem ... 119

7 Regulation Problem: E-O-C Problem ... 127
7.1 Regulation Problem ... 127
7.2 User Model of Regulation Problem for PD Goal-Seeker ... 129
7.3 Implementation in extProlog ... 133
7.4 Validity of PD Method for Regulation Problem ... 135
7.5 Feedback Law Problem and Case-Based Reasoning ... 136
Appendix 7.1 User Model for Regulation Problem ... 137
Appendix 7.2 Chinese Checkers Problem ... 138

8 Linear Quadratic Optimization Problem: E-C-O and E-O-O Problems . 145
8.1 Linear Quadratic Optimization Problem ... 145
8.2 User Model of Linear Quadratic Optimization Problem for PD Goal-Seeker ... 146
8.3 Implementation in extProlog ... 149
Appendix 8.1 User Model for Linear Quadratic Optimization Problem ... 150

9 Cube Root Problem: I-C-C Problem ... 153
9.1 Cube Root Problem ... 153
9.2 User Model of Cube Root Problem for PD Goal-Seeker ... 153
9.3 Implementation in extProlog ... 156
9.4 Tuning of PD Goal-Seeker for Cube Root Problem ... 156
Appendix 9.1 User Model for Cube Root Problem ... 156

10 Knapsack Problem: I-C-O Problem ... 159
10.1 Knapsack Problem ... 159
10.2 User Model of Knapsack Problem for PD Goal-Seeker ... 160
10.3 Implementation in extProlog ... 164
10.4 Tuning of PD Goal-Seeker for Knapsack Problem ... 165
Appendix 10.1 User Model for Knapsack Problem ... 166

11 Class Schedule Problem: I-O-C Problem ... 169
11.1 Class Schedule Problem ... 169
11.2 User Model of Class Schedule Problem for PD Goal-Seeker ... 169
11.3 Implementation in extProlog ... 174
11.4 Tuning of PD Goal-Seeker for Class Schedule Problem ... 176
Appendix 11.1 User Model for Class Schedule Problem ... 176

12 Data Mining Problem: I-O-O Problem 183
12.1 Data Mining Problem 183
12.2 User Model of Data Mining Problem for PD Goal-Seeker 184
12.3 Implementation in extProlog 195
12.4 Tuning of PD Goal-Seeker for Data Mining Problem* 196
Appendix 12.1 User Model for Data Mining Problem 198

13 Task Skeleton Model: Intelligent Data Mining System* 207
13.1 General Concept of Problem-Solving System 207
13.2 Task Skeleton Model 210
13.3 Intelligent Data Mining System 213

Part V Model Theory Approach to Transaction Processing Systems Development

14 Transaction Processing System on Browser-Based Standardized User Interface 225
14.1 Model Theory Approach to Transaction Processing System: Canonical Structure 226
14.2 Realization Structure of BMIS: File System 230
14.3 Atomic Process and Relational Structure of User Model 233
14.4 Dynamic Process of modTPS 235
14.5 Development Procedure for TPS Using the Implementation Structure of the User Model 236
14.6 User Model Construction: Example 238
14.7 Operation of Compiled User Model 243
Appendix 14.1 Derivation of Canonical Representation of BMIS* 245
Appendix 14.2 Derivation of Realization Structure of BMIS on File System* 250
Appendix 14.3 External User Interface* 256

15 Browser-Based Intelligent Management Information System: Temporary Staff Recruitment System 259
15.1 Systems Specification 259
15.2 DFD for Specification and Browser-Based Intelligent MIS 260
15.3 TPS Development for Employment System 262
15.4 Data Transformation System 267
15.5 Solver Development: Jobarrange 269
15.6 Operation of Employment System 273
Appendix 15.1 Transaction Processing Part of Employment System in Computer-Acceptable Set Theory 276
Appendix 15.2 Data Transformation and Solver Parts of Employment System in Computer-Acceptable Set Theory 282

16 Database Connectivity for the Model Theory Approach* 287
16.1 Database Connectivity in extProlog 287
16.2 OODB Language 288
16.3 Implementation of OODB Language 293
16.4 SQL and OODB Languages 297
16.5 Database Connection to Model Theory Approach 310
Appendix 16.1 Proof of Theorem 16.1 315
Appendix 16.2 Database-Handling Model in Computer-Acceptable Set Theory 319

Part VI Theoretical Basis for extProlog

17 extProlog as Logic Programming Language* 325
17.1 Prolog as Logic Programming Language 325
17.2 Predicate Calculus 327
17.3 Special Forms 330
17.4 Theorem Proving in Prolog 332
17.5 Theorem Proving by Resolution Principle 335
Appendix 17.1 Proof of Theorem 17.1 340
Appendix 17.2 Proof of Theorem 17.2 341
Appendix 17.3 Proof of Theorem 17.3 341

18 Implementation of extProlog* 343
18.1 Implementation of extProlog: Generalized PDA (Push-Down Automaton) Model 343

Index 355

Part I

New Paradigm of Systems Development

1

New Systems Development Methodology: The Model Theory Approach

This chapter introduces the new systems development platform presented in this book. This new development methodology for management information systems (MISs) is based on a formal model-theoretic structure derived from the systems concepts of general systems theory (GST). The model is represented in set theory and implemented in a fourth-generation programming language, extProlog, by automated system generation. The extProlog language is an extension of standard Prolog that allows for the implementation of an MIS. As discussed in Section 1.2, the new methodology provides a platform for the development of both transaction processing systems and problem-solving systems as the two principal components of an MIS.

1.1 Necessity for a New Systems Development Methodology for Management Information Systems

An MIS, the target of the model theory approach, can be modeled as a hierarchical system consisting of a transaction (data) processing system (TPS) and a problem-solving/decision support system. Figure 1.1 shows the general scheme of an MIS (see also Fig. 14.5).

A TPS is an input–output system, having a transaction input and user request as inputs, and a transaction output and user response as outputs. The transaction output is typically a modification of a file system within the TPS.

A problem-solving information system is modeled as a goal-seeking system in which the problem (assumed to be associated with a goal) is specified by the user and data from the TPS. The function of the goal-seeking system is to produce a solution satisfying the goal or suggest solution candidates to the user.

The TPS traditionally constitutes the infrastructure of an MIS, and the information system is understood as being almost equivalent to a TPS. However, as discussed in Chapters 4 and 15, the problem-solving/decision support system has become more and more important as requirement levels for the information system have risen. The motivation for the new systems development methodology proposed in this book is founded on the following observations:

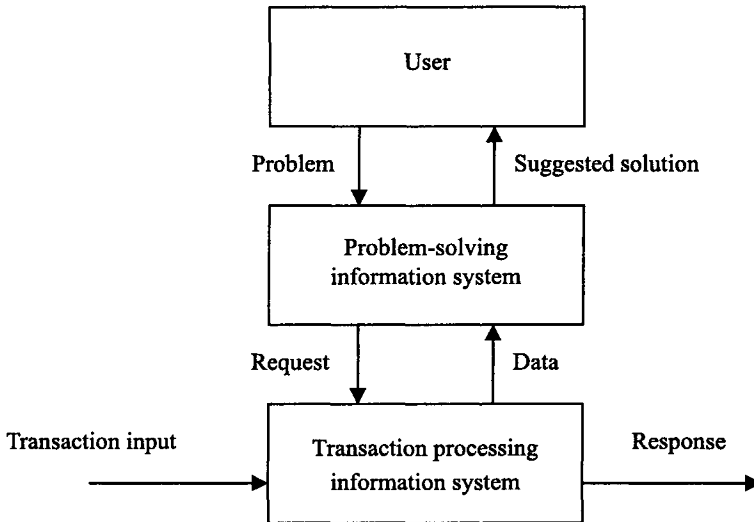


Fig. 1.1. General MIS model.

(1) The rapid increase in computing power, and decreasing cost of workstations and personal computers means that most information systems are now developed on these platforms. Decision makers can also solve problems on these platforms. As such, a new methodology that makes best use of workstations and personal computers must be explored. Popularization of open software also demands a new methodology.

(2) Companies are now facing global competition in a rapidly changing business environment. Decision makers need to make the right decisions at the right times, and require technological support in order to do so. However, the conventional systems development methodologies such as the lifecycle approach are notoriously slow and unwieldy, and as such are not suitable for the problem-solving function. A light, rapid systems development methodology is therefore required.

(3) Traditionally, systems development follows the way in which a user puts forward the request for information, and the system developer then develops an appropriate system. However, since the perceived requirement of the user is semistructured or unstructured and also changeable, it is very difficult for the system developer to understand the request of the user precisely. Furthermore, the users themselves may know only a certain condition of the problem, without understanding the true nature of the problem from the beginning, and quite often a significant amount of time is spent interacting with the solution process before a clearer perspective of the problem is obtained. It is therefore the user who is best equipped to build a system, suggesting that a methodology oriented around end-user development (EUD) is necessary.

(4) Almost every nontrivial system is built assuming that it is to be modified during operation. In general, however, the end user must have detailed knowledge of the

system or prior involvement in its development in order to modify the system effectively. This issue may be resolved by the introduction of a rapid, easy, and not-software-engineering-oriented formalized systems development methodology (See Chapter 14).

1.2 Outline of Model Theory Approach

Although the structure shown in Fig. 1.1 is too general to produce meaningful results, it can be specialized according to the objectives of the proposed model theory approach. Figure 1.2 shows a model of an "intelligent" MIS for the support of intelligent business activities, representing the ultimate target of the model theory approach. (See Chapter 15.)

In Fig. 1.2, the TPS of Fig. 1.1 is represented as an automaton model in which the database (DB) saves the current state while the data processing system (DPS) provides a state transition function [Takahara and Kubota, 1989].

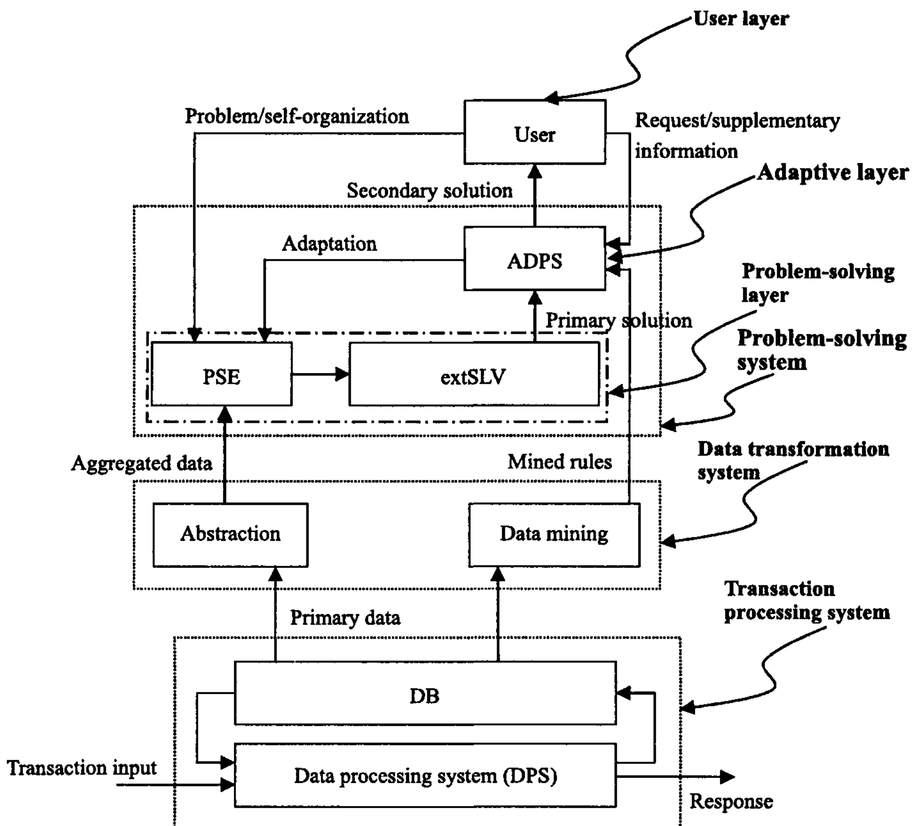


Fig. 1.2. Intelligent MIS model.

The problem-solving information system of Fig. 1.1 is decomposed into two levels in Fig. 1.2: the problem-solving system and the data transformation system. The data transformation system transforms the data of the TPS into information necessary for the problem-solving system by abstraction (aggregation) and/or data mining [Takahara, Asahi and Hu, 2004].

The problem-solving system is broken down into a further two layers: a problem-solving layer and an adaptive layer. The two layers and the user (layer) correspond to the control layer, the adaptive control layer, and the self-organization layer of the layer model of GST, respectively [Mesarovic and Takahara, 1989]. The three-layer model is called a skeleton model, and is discussed in Chapter 13.

The problem-solving layer, the lowest layer of the skeleton model, usually consists of two components: the problem specification environment (PSE) and the extended solver (extSLV), as shown in Fig. 1.2. The PSE specifies the structure of the problem to be solved, but is in general not a problem representation. The problem is instead formed and solved in the extSLV. Since the problem is usually specified outside of the solver, the solver in Fig. 1.2 is called an extended solver, the output of which is the (primary) solution.

Although the TPS is a principal target of current systems engineering, the problem-solving system is much more complicated and theoretically challenging. The first nine chapters (Chapters 4–12) in this book investigate the problem-solving system in the model theory approach. Chapter 13 deals with the data mining function, and Chapters 14–16 are devoted to TPS development.

The objective of the first nine chapters, dealing with the problem-solving system, is to establish a general basis for the construction of a problem-solving system, specifically the design and implementation of the extSLV using a formal problem-solving structure. It should be noted that because the problem-solving layer is responsible for the primary activity of the problem-solving system, the extSLV is the real engine of the problem-solving system. Chapter 15 shows that the intelligent MIS of the model theory approach is constructed by direct application of the extSLV.

The principal function of the extSLV is to yield an optimized (or satisfactory) solution for an identified problem. Although optimization may appear to be too specialized for the MIS concept, it is a valid part of an intelligent MIS model. Uncertainties associated with a problem are dealt with by the adaptive layer and the user layer. The role of the extSLV in this context is discussed in detail in Chapter 13.

As an outline of the model theory approach, the formalized structures of information systems are introduced here without detailed explanation in order to set the context for the formulation at the outset. Full discussions can be found in Chapters 5 and 14.

The model theory approach begins from a general formulation of the two types of information systems, that is, the TPS and the problem-solving system. Thus, the formulation explicitly divides the structures of each information system into two parts: a standard part and a target-specific part. In the TPS, the standard part is formalized as a standardized user interface, and the target-specific part is formalized as a user model structure of a TPS. For the problem-solving system, the standard part is formalized as a standardized goal-seeker, and the target-specific part is formalized as a user model structure of a solver system. The standardized components are then implemented as

standard subsystems in the model theory approach and provided as black box components to systems development.

The formalized user model structure of the TPS is given as follows:

User Model Structure of TPS =

$$\langle \text{ActionName, AttrName, ResName, } \{f_j\}_j, \text{para}, \{\text{Paralist}_i, \text{delta_lambda}(\text{actionN}_i), \text{actionN}_i\}_i \rangle,$$

where

ActionName: set of action (command) names,
ResName: set of response names,
AttrName: set of attribute names used in the file system,
$\{f_j\}_j$: set of file structures (f_j: AttrName→{true, false} or $f_j \subset$AttrName),
para: ActionName→℘(AttrName): action parameter specification function,
where ℘(AttrName) is the power set of AttrName,
Paralist_i = realization(para(actionN_i)), where $\text{actionN}_i \in$ ActionName,
delta_lambda: ActionName→∪{realization(resName)| resName ∈ ResName}:
interface component of an atomic process,
actionN_i: Paralist_i→∪{realization(resName)| resName ∈ ResName}:
implementation component of an atomic process.

The user model is constructed as a family of atomic processes.

The formalized user model structure of the problem-solving system is given as follows:

User Model Structure of Solver System (with Dynamic Programming Goal-Seeker) =

$$\langle A, C, \delta, \lambda, \text{ genA, constraint, goalElement, invst}, c_0, C_f \rangle,$$

where

A: set of actions,
C: set of states,
$\delta : C \times A \to C$: state transition function,
$\lambda : C \times A \to A$: output function,
genA: $C \to$ ℘(A): allowable action specification function,
constraint: $C \to$ {true, false}: constraint predicate for a state,
goalElement: $C \to$ Re: evaluation of a state, where Re is the set of real numbers,
invst: $C \to$ {true, fase}: stopping condition for backward solving activity,
$c_0 \in C$: initial state,
$C_f \subset C$: set of final states.

These structures are discussed in detail in Chapters 14 and 5, respectively. These results lead to the following systems development procedure for the model theory approach:

(i) A system developer constructs a model for the target system using the user model structure as a template. The constructed model is called a user model, and is described in set theory.
(ii) The developer rewrites the user model in set theory into a user model in computer-acceptable set theory, which is then translated into a user model in extProlog by a compiler (setcompiler).
(iii) The extProlog interpreter attaches the standardized components, the standardized user interface, or the standardized goal-seeker to the user model in extProlog depending on the type of the target system.
(iv) Finally, the extProlog interpreter executes the combined model to realize the intended system.

The final form of the user model produced by a system developer is called an implementation structure. The implementation structure for a TPS is presented as follows (see Chapter 14):

Implementation Structure of User Model for TPS in (Computer-Acceptable) Set Theory

```
func(<list of function names>);       /*declaration of function names used
                                        in the user model*/
ActionName.g=<list of action names>;  /*specification of ActionNames*/
                                      /*definition of atomic process of
                                        actionN_i*/
<actionN_i>.g=para(actionN_i);        /*specification of parameters for
                                        actionN_i*/
delta_lambda([actionN_i],paralist)=   /*interface component of actionN_i*/
  res<->res:= actionN_i(paralist);
actionN_i(paralist)=res<->            /*implementation component of
  (res,c2):=φ(c,paralist);              actionN_i*/
```

Here, ϕ is a function defining the next state (new contents of the file system) and the user response.

Similarly, the implementation structure for a solver system is given as follows (see Chapter 4):

Implementation Structure of User Model for Solver System in (Computer-Acceptable) Set Theory

```
func(<list of function names>); /*declaration of function names used in
                                  the user model*/
delta(state,action)=nextstate <->
   nextstate:= δ(state,action); //state transition
genA(state)=actionset <->
   actionset:= φ(state);        //allowable actions for a given state
constraint(state) <->
   μ(state);                    //permissible state
initialstate()=c0 <->
   c0:=θ(state);                //initial state
finalstate(state) <->
   ρ(state);                    //final state
```

```
goalElement(state)=r <->
   r:=ξ(state);                //evaluation of a state (output)
invst(state) <->
   ζ(state);                   //stopping condition of backward solving
                                 process
preprocess() <->
   pre();                      //preprocessing for goal-seeker
postprocess() <->
   post();                     //postprocessing for goal-seeker
```

Here, δ, φ, θ, and ξ are functions, and μ, ρ, ζ, pre and post are predicates. The predicates preprocess() and postprocess() can be omitted.

A user model in standard set theory is not suitable for processing on a computer. For example, the symbol $\cup$ cannot be directly read by a computer. The notion of "computer-acceptable" set theory here refers to modification of standard set theory such that the model in set theory can be accepted by the setcompiler. This translation is discussed in Chapter 2. The setcompiler is supported by the model theory approach.

The above is a general outline of systems development by the model theory approach. The details of the development of the TPS and the problem-solving system differ slightly, as shown in the next two figures. Figure 1.3 shows the detailed development procedure for a TPS. The figure also includes operation of the developed TPS.

When a target problem requires TPS development, the system developers are assumed to verify their perception of the problem by creating a data flow diagram (DFD). A DFD is regarded as a generalized block diagram in the model theory approach. A user model in set theory is generated from this image of the problem with the assistance of the formalized user model structure for the TPS. The user model in set theory is then transformed into a representation in computer-acceptable set theory. The representation is finally translated into a user model in extProlog automatically. Upon translation, a standardized user interface, called an internal user interface (stdDPS_UI.p), is attached to the user model in extProlog.

The user model is executed under the control of another user interface, called an external user interface (stdUI.php), to realize the intended function. The external user interface is an interface between the end user and the system, and the internal user interface is an interface between the external interface and the user model.

The systems development procedure for a problem-solving system is shown in detail in Fig. 1.4.

When a target problem requires development of a problem-solving system, the system developers are also assumed to verify their system concept through the use of a block diagram with a structure called the problem specification environment (PSE) as the input (see Chapter 5). Its output is a solution. A user model in set theory is constructed from the block diagram using the formalized user model structure of the problem-solving system given above. The user model is then transformed into a representation in computer-acceptable set theory. The representation is translated into a user model in extProlog in the same manner as for the TPS. On translation a standardized goal-seeker is attached to the user model in extProlog. Finally, a solving system

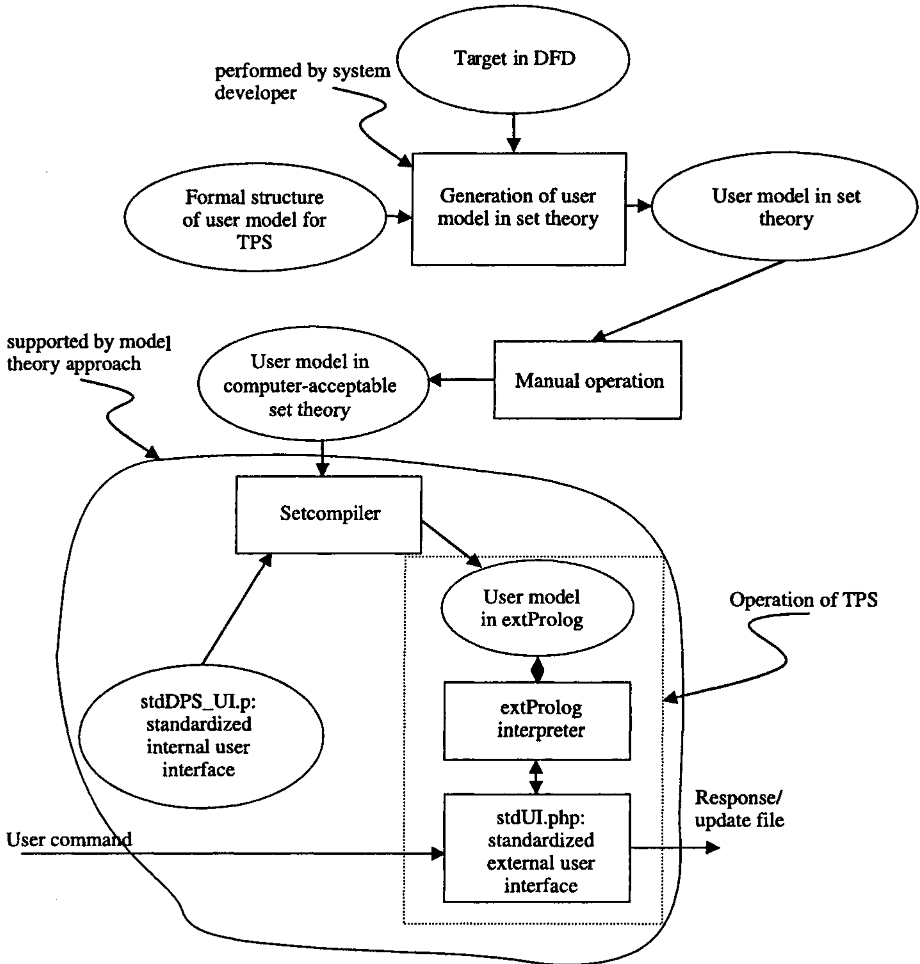


Fig. 1.3. Development procedure for TPS.

is created by the extProlog interpreter. As shown in Fig. 1.4, the solving system is given as an output of the extProlog interpreter. More specifically, the solver system is not a newly created system but a combination of the interpreter and the user model in extProlog.

Appendix 1.1 presents a user model in extProlog generated by the setcompiler, and Appendix 1.2 shows the same extProlog user model generated manually.

The model theory approach can be summarized as follows:

- It is not a software engineering approach but a systems theory approach, based on a systems model.
- It has the model theory structures of information systems, derived from general systems theory concepts.

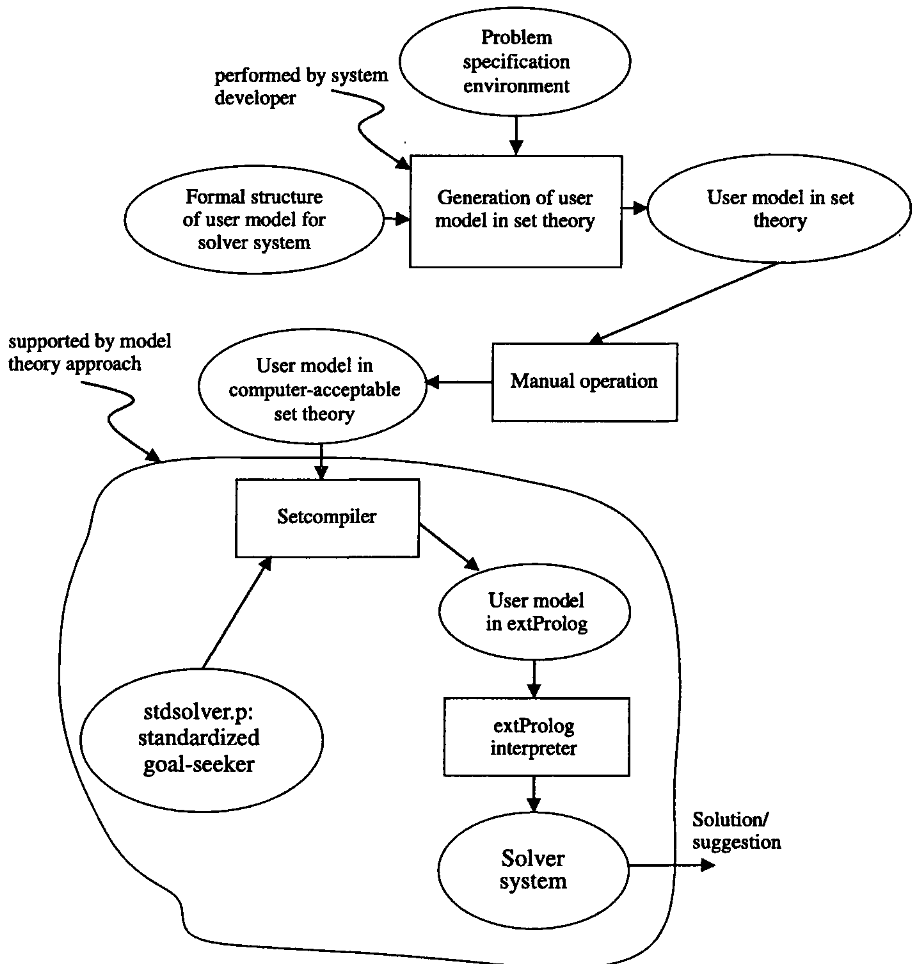


Fig. 1.4. Development procedure for problem-solving system.

- It is a formal approach in the sense that a model is developed using the model theory structure and described in set theory.
- A model in set theory can be translated into an executable system automatically as an integral part of the approach.
- System construction is supplemented by standardized components specified by the formal structure, facilitating rapid systems development.
- The implementation language is extProlog.

Figure 1.5 shows a comparison of the model theory approach with other well-known approaches. It should be noted that the target of the other approaches is TPS development, whereas the model theory approach addresses both TPS and solver development.

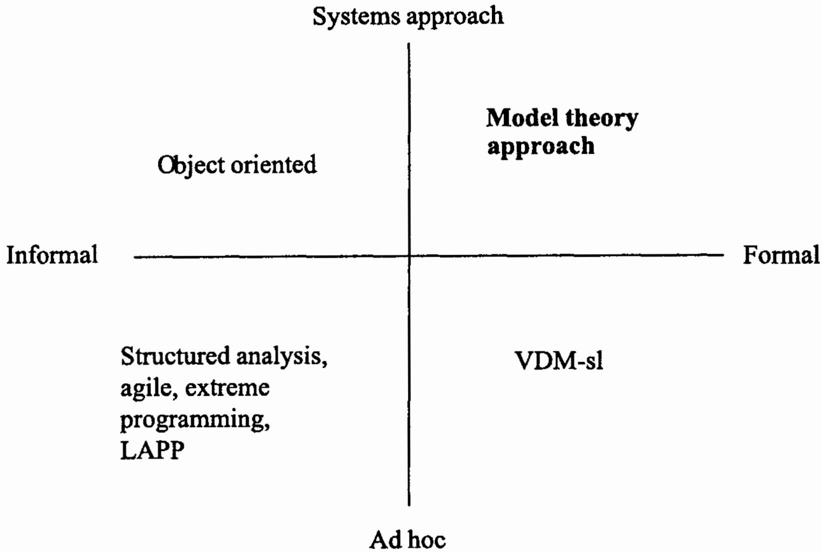


Fig. 1.5. Comparison of model theory approach with other approaches.

It is clear that the structured analysis, agile, extreme programming, LAPP (Linux, Apache, postgreSQL, PHP) and object-oriented approaches are not formal approaches. Although VDM-sl (Vienna Development Method specification language) is a formal approach, it does not include a system model for an information system such as the user model structure mentioned above. The three main advantages of the model theory approach are the provision of a theoretical basis for information systems development, automatic system generation using standardized components, and facilitation of rapid systems development. Consequently, it can provide a methodology that facilitates a more rapid, less expensive, more reliable, and easier system development.

1.3 Example of Model Theory Approach to Systems Development

A pattern-finding problem in data mining is examined here in order to demonstrate the model theory approach [Berndt and Clifford, 1996]. Consider a given set of time-series data and template data. Let the time series $s()$ and template $t()$ be functions given by

$$s : N \to \text{Re},$$
$$t : M \to \text{Re},$$

where $N = \{0, \ldots, n-1\}, M = \{0, \ldots, m-1\}$, and Re is the set of real numbers. The pattern-finding decision problem is then given as follows. Let

$$Y \subset (N \times M)^*$$

be a set of outputs, where $(N \times M)^*$ is the free monoid of $(N \times M)$. If $y = a_0 a_1 \cdots a_l \in Y$, where $a_k = (i_k, j_k) \in N \times M$, is a selected action, then $s(i_k)$ is supposed to be matched to $t(j_k)$ for $k = 0, \ldots, l$.

The desirability of an output $y = y_0 y_1 \cdots y_l$ is then measured by

$$G(y) = \sum_{k=0}^{l} |s(i_k) - t(j_k)|,$$

where $a_k = (i_k, j_k)$.

The decision problem is then to find y such that the following condition is satisfied:

$$G(y) \to \min \text{ with respect to } y \in Y.$$

Constraint conditions can be imposed for $y \in Y$. In the present case, $a_0 = (0, 0)$ and $a_l = (n-1, m-1)$ must hold for $y = a_0 a_1 \cdots a_{l_i}$. Let

$$\text{constraint: } (N \times M)^* \to \{\text{true, false}\}$$

be a unary predicate to specify the set Y, or $Y = \{y | \text{constraint}(y) = \text{true} \ \& \ y \in (N \times M)^*\}$. Then, the decision problem for a problem-solving system or a solver (SLV) can be represented as follows:

$$\langle (N \times M)^*, \text{constraint}, G \rangle.$$

This is a set-theoretic formulation of the problem, and represents the starting point of the model theory approach, that is, it presents a problem specification environment. The final user model structure of the model theory approach is then given by

$$\langle A, C, \delta, \lambda, \text{genA}, \text{constraint}, \text{goalElement}, \text{invst}, c_0, C_f \rangle,$$

which is quoted above. This section shows the explicit construction of the components of the model structure.

The SLV solves the problem, affording the optimal solution. In the model theory approach, the problem-solving process is first formalized as an automaton (the user model structure is composed of an automaton and a goal). The sets and functions necessary for the automaton representation are as follows:

Let

$$A = N \times M; \text{ the set of actions,}$$

$$C = N \times M; \text{ the set of states,}$$

and

$$\text{genA} : C \to \wp(A),$$

where

$$\text{genA}((x, y)) = \{(x+1, y), (x+1, y+1), (x, y+1)\} \text{ for any } (x, y) \in C.$$

Here, genA(c) is a set of allowable actions for the state $c \in C$.

Then let

$$\delta : C \times A \rightarrow C,$$

where

$$\delta((x, y), (u, v)) = c2 \leftrightarrow (u, v) \in \text{genA}(x, y) \,\&\, c2 = (u, v) \,\&\, \text{constraint}(c2).$$

Here, δ is a state transition function.

Finally, let

$$\lambda : C \times A \rightarrow A,$$

where

$$\lambda(c, a) = a.$$

Here, λ is an output function.

This formulation specifies the automaton:

$$\text{solving process automaton} = \langle A, C, \delta, \lambda \rangle.$$

The next step involves formulation of the dynamic optimization problem by introducing a goal: evaluation of the output $y \in Y$. The goal is used to determine the optimal action sequences on A. In the present case, the goal $G : Y \rightarrow \text{Re}$ is decomposed into elements goalElement in order to use a standardized dynamic programming (DP) approach, which is called a DP method in the model theory approach.

Let

$$\text{goalElement} : C \rightarrow \text{Re},$$

where

$$\text{goalElement}((u, v)) = |s(u) - t(v)|.$$

Then, the following relation holds:

$$G(y) = \sum \text{goalElement}(a_i)$$

where

$$y = a_1 a_2 \cdots a_l.$$

In other words, $G(y)$ is an additive function of goalElement(a_i). This leads to the following dynamic optimization problem:

$$\begin{aligned}&\text{dynamic optimization formulation}\\&\quad = \langle A, C, \delta, \lambda, \ \text{genA, constraint, goalElement, invst}, c_0, C_f \rangle,\end{aligned}$$

where $c_0 = (0, 0)$ and $C_f = \{(n - 1, m - 1)\}$. This is a formulation of a user model for the DP method.

If the above formulation is expressed in a form that can be accepted by a computer and is subsequently processed by the setcompiler (Fig. 1.4), an executable

```
/*warp2.set*/
.func([delta,invdelta,genA,
      initialstate,goalelement,
      dis,warpoutput2]);

delta(C,A)=C2 <->
    C2:=A,
    constraint(C2);

invdelta([U,V])=Ys <->
    (U<1 and V<1) ->
           (
           Ys:=[]
           )
    .otherwise
           (
           (U<1 and V>=1) ->
                  (
                  Ys:=[[U,V-1]]
                  )
           .otherwise
                  (
                  (U>=1 and V<1) ->
                         (
                         Ys:=[[U-1,V]]
                         )
                  .otherwise
                         (
                         Ys:=[[U-1,V],
                              [U-1,V-1],
                              [U,V-1]]
                         )
                  )
           );

genA([X,Y])=As <->
    As:=[[X+1,Y],[X+1,Y+1],[X,Y+1]];

constraint(C) <->
    C>=0;

initialstate()=C <->
    C:=[0,0];

finalstate(C) <->
    C=[6,6];

goalElement(C,A)=R <->
    (C=[6,6]) ->
         (
             R:=0
             )
        .otherwise
             (
             R:=dis(C)
             );
dis([X,Y])=D <->
    Ft:=[4,5,6,7,6,5,4],
    Fs:=[4,6,8,10,8,6,4],
    D:=abs(getvalue(Fs,X)-getvalue
       (Ft,Y));

invst(Cs) <->
     C0:=initialstate(),
     member(C0,Cs);
postprocess() <->
     solutionSS(SS),
     C0:=initialstate(),
     Cs0:=warpoutput2(C0,SS),
     Cs:=append([C0],Cs0),
     Rs:=goalValue(Cs),
     xwriteln(0,"Rs=",Rs),
     TCs:=transpose(Cs),
     show1(TCs,"trajectory");

goalValue(Cs)=Rs <->
     (Cs=[]) ->
             (
             Rs:=[]
             )
     .otherwise
             (
             [C|Cs2]:=Cs,
             R:=goalElement(C,A),
             Rs2:=goalValue(Cs2),
             Rs:=append([R],Rs2)
             );

warpoutput2(C,SS)=Cs <->
     (SS=[]) ->
             (
             Cs:=[]
             )
     .otherwise
             (
             [A|As]:=SS,
             C2:=delta(C,A),
             Cs2:=warpoutput2(C2,As),
             Cs:=append([C2],Cs2)
             );
```

Fig. 1.6. User model in computer-acceptable set theory.

system can be obtained. Figure 1.6 shows a user model warp2.set for the pattern-finding problem, which is acceptable by a computer. Appendix 1.1 shows the corresponding executable system warp2.p generated from warp2.set by the setcompiler. Chapter 2 discusses the computer-acceptable description of set theory for a user model.

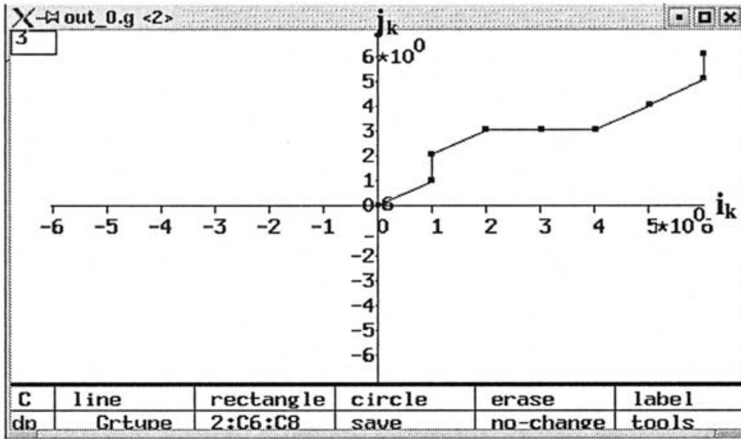


Fig. 1.7. Output of pattern-finding system: optimal warp.

The model description above includes a function invδ (invdelta) that is not an element of the user model formulation; it is the inverse of δ, that is, invδ : $C \to \wp(C)$ such that inv$\delta(c) = \{c'|(\exists a)(\delta(c', a) = c)\}$. Theoretically speaking, invδ can be generated from the user model formulation. However, as such a naive inverse operation is not efficient for implementation of invδ; the function is assumed to be supplied by the user model when the DP method is used. Specifically, a user is assumed to provide an efficient algorithm using a specific property of the target problem. A typical example is the inverse operation of a matrix.

Appendix 1.1 presents the compiled output, warp2.p, which is an executable system. In the warp2.p compile, the standardized dynamic programming (DP) strategy program stdDPsolver62.p is attached by the statement #include "stdDPsolver62.p". The entire code of stdDPsolver62.p is given in Appendix 5.1. Figure 1.7 shows the trajectory $\{(i_k, j_k)_k\}$ of a solution y, where the horizontal coordinate is i_k and the vertical coordinate is j_k. An extra predicate postprocess() is included to generate the graph in warp2.set.

The internal mechanism of the DP method, which can be ignored by an ordinary system developer, is based on a function, idGoal:C $\to$ Re, called an ideal evaluation of a state (see Chapter 5). Let idGoal be recursively defined as

$$\text{idGoal}(c_f) = 0 \text{ where } c_f \in C_f,$$
$$\text{idGoal}(c) = \text{goalElement}(c) + \min\{\text{idGoal}(\delta(c, a))|a \in \text{genA}(c)\}.$$

The evaluation idGoal is a partial function, generated by a generator automaton called genGoal. Suppose $c_0 \in$ dom(idGoal), where dom(idGoal) is the domain of idGoal. Then, a DP strategy

$$\sigma : C \to A$$

is defined by idGoal as follows:

$$\sigma(c) = a^* \leftrightarrow \min\{\text{idGoal}(\delta(c, a))|a \in \text{genA}(c)\} = \text{idGoal}(\delta(c, a^*)).$$

Here, σ is a goal-seeker model in set-theoretic terms and is supplied as a standardized solver (black box component) by the model theory approach (refer to Fig. 5.4).

Let $(c_0, c_1, \ldots, c_k)$ be a solution or an output of the goal-seeker, where $c_{i+1} = \delta(c_i, \sigma(c_i)) \equiv \pi(c_i)$ and $c_k \in C_f$. The generator representation of SLV is then given by

$$\mathrm{SLV} = \langle C, \pi, c_0, C_f \rangle,$$

where $\pi : C \to C$ such that

$$\pi(c) = \delta(c, \sigma(c)).$$

Appendix 1.1 User Model in extProlog Generated by Setcompiler: Pattern-Finding Problem

The following is a model in extProlog generated by the set compiler from the user model shown in Fig. 1.6:

```
//warp2.p

func("warp2.set",[delta,invdelta,initialstate,goalelement,dis,goalValue,
    warpoutput2]) ;
delta(C,A,CC):-
    C2:=A,constraint(C2),
    CC:=C2 ;
invdelta([U,V],Ys):-
    if and(ltsds(U,1),ltsds(V,1))  then
        Ys:=[]
    else
        if and(ltsds(U,1),gesds(V,1))  then
            Ys:=[[U,minussds(V,1)]]
        else
            if and(gesds(U,1),ltsds(V,1))  then
                Ys:=[[minussds(U,1),V]]
            else
                Ys:=[[minussds(U,1),V],[minussds(U,1),
                      minussds(V,1)],[U,minussds(V,1)]]
            end
        end
    end ;
genA([X,Y],As):-
    As:=[[plussds(X,1),Y],[plussds(X,1),plussds(Y,1)],
         [X,plussds(Y,1)]] ;
constraint(C):-
    gesds(C,0) ;
initialstate(C):-
    C:=[0,0] ;
finalstate(C):-
    eqsds(C,[6,6]) ;
goalElement(C,A,R):-
    if eqsds(C,[6,6])  then
        R:=0
    else
        R:=dis(C)
```

```
    end ;
dis([X,Y],D):-
    Ft:=[4,5,6,7,6,5,4],
    Fs:=[4,6,8,10,8,6,4],
    D:=abs(minussds(getvalue(Fs,X),
           getvalue(Ft,Y))) ;
invst(Cs):-
    C0:=initialstate(),
    member(C0,Cs) ;
postprocess():-
    solutionSS(SS),
    C0:=initialstate(),
    Cs0:=warpoutput2(C0,SS),
    Cs:=append([C0],Cs0),
    Rs:=goalValue(Cs),
    xwriteln(0,"Rs=",Rs),
    TCs:=transpose(Cs),
    show1(TCs,"trajectory") ;
goalValue(Cs,Rs):-
    if eqsds(Cs,[]) then
        Rs:=[]
    else
        [C|Cs2]:=Cs ,
        R:=goalElement(C,A),
        Rs2:=goalValue(Cs2),
        Rs:=append([R],Rs2)
    end ;
warpoutput2(C,SS,Cs):-
    if eqsds(SS,[]) then
        Cs:=[]
    else
        [A|As]:=SS ,
        C2:=delta(C,A),
        Cs2:=warpoutput2(C2,As),
        Cs:=append([C2],Cs2)
    end ;
?-funcCall("warp2.set"),stdDPsolver(),postprocess();
#include "stdDPsolver62.p";
```

Appendix 1.2 User Model in extProlog Generated by Manual: Pattern-Fitting Problem

The following is a model in extProlog manually generated from the user model of Fig. 1.6:

```
/*warpDP2.p*/

#include "stdDPsolver62.p";
delta(C,A,CC):-
    C2:=A,
    constraint(C2),
    CC:=C2;
invdelta([U,V],Ys):-
    if U<1 and V<1 then
```

```
            Ys:=[]
        else
            if U<1 and V>=1 then
                Ys:=[[U,V-1]]
            else
                if U>=1 and V<1 then
                    Ys:=[[U-1,V]]
                else
                    Ys:=[[U-1,V],[U-1,V-1],[U,V-1]]
                end
            end
        end;

genA([X,Y],As):-
    As:=[[X+1,Y],[X+1,Y+1],[X,Y+1]];
initialstate([0,0]);
finalstate([6,6]);
constraint(C):-
    C>=0;
goalElement([6,6],A,0):-!;
goalElement(C,A,R):-
    R:=dis(C);
dis([X,Y],D):-
    Ft:=[4,5,6,7,6,5,4],
    Fs:=[4,6,8,10,8,6,4],
    D:=abs(getvalue(Fs,X)-getvalue(Ft,Y));
invst(Cs):-
    initialstate(C0),
    member(C0,Cs);
?-function([dis]),stdDPsolver(),
    warpoutput();
/*draw Figure 1.1*/
warpoutput():-
    solutionSS(SS),
    initialstate(C0),
    warpoutput2(C0,SS,Cs0),
    append([C0],Cs0,Cs),
    goalValue(Cs,Rs),
    xwriteln(0,"Rs=",Rs),
    TCs:=transpose(Cs),
    show1(TCs,trajectory);
goalValue([],[]):-!;
goalValue([[X,Y]|Cs2],Rs):-
    goalElement([X,Y],A,R),
    goalValue(Cs2,Rs2),
    append([R],Rs2,Rs);

warpoutput2(C,[],[]):-!;
warpoutput2(C,[A|As],Cs):-
    delta(C,A,C2),
    warpoutput2(C2,As,Cs2),
    append([C2],Cs2,Cs);
```

References

Berndt, D. J. and Clifford, G. (1996) "Finding patterns in time series—a dynamic programming approach" in *Advances in Knowledge Discovery and Data Mining*, Edited by U.M. Fayard, MIT Press.

Mesarovic, M. D. and Takahara, Y. (1989) "Abstract Systems Theory," *Lecture Notes in Control and Information Sciences*, Springer.

Takahara, Y. and Kubota, H. (1989) "Framework for Conceptual Design of Data Processing System," *Office Automation* **10**(1) (in Japanese), 81–88.

Takahara, Y., Iijima, J., and Shiba, N. (1993) "A model management system and its implementation," *System Science*, **19**(4).

Takahara, Y., Asahi, T., and Hu, J. (2004) "Application of MGST Design Approach to Data Mining Systems: Case of I-O-O Problem," *J. of JAS MIN* **12**(4), 35–50.

Part II

Model Construction Language and Systems Implementation Language

2

Computer-Acceptable Set Theory for Model Construction

To realize an information system by the model theory approach, a user model in set theory as introduced in Chapter 1 must be described in a form that can be accepted by a computer. For example, the symbol "$\notin$" cannot be accepted by a computer, and must be replaced by an appropriate term, in this case "notmember." The system-defined predicates and functions are used to describe the predicates and functions necessary for the user model with greatest efficiency. Set theory created in this way is described here as computer-acceptable set theory, and is the focus of this chapter.

2.1 Implementation Structure of User Model in Computer-Acceptable Set Theory

A user model in computer-acceptable set theory has the following simple structure (see Fig. 1.6):

Function declaration
Set of statements specifying the user model structure
Preprocess predicate
Postprocess predicate

The function declaration consists of the following statement:

$$.\text{func}([\text{function-name}_1, \ldots, \text{function-name}_n]);$$

which should include every user-defined function name. Although standard names need not be declared, redundant declarations are allowed. A statement is a well-formed formula of computer-acceptable set theory, as defined below. The predicates preprocess() and postprocess() are used for initialization and input–output operations and may be omitted.

2.2 Well-Formed Formula of First-Order Predicate Calculus

The basic syntax of the set-theoretic description in the model theory approach is based on first-order predicate calculus. The definition starts with the introduction of a well-formed formula (wff) of standard first-order predicate calculus.

The formulation of a wff is based on the definition of a relational structure [Bridge, 1977], as given by

$$\mathrm{ST} = \langle A, \{r_i | i \in I\}, \{f_j | j \in J\}, \{c_k | k \in K\}\rangle,$$

where I, J and K are index sets and

(1) A, the domain of ST, is a nonempty set.
(2) r_i is a relation on A, i.e., $r_i \subset A \times \cdots \times A$.
(3) f_j is a function on A, i.e., $f_j : A \times \cdots \times A \to A$.
(4) c_k is a constant element of A, i.e., $c_k \in A$.

The first-order language L(ST) for the structure ST $= \langle A, \{r_i | i \in I\}, \{f_j | j \in J\}, \{c_k | k \in K\}\rangle$, then consists of

(1) individual variables $v_0, v_1, \ldots$.
(2) individual constant symbol c_k for each $k \in K$.
(3) a $\lambda(i)$-ary predicate symbol r_i for each $i \in I$.
(4) a $\mu(j)$-ary function symbol f_j for each $j \in J$.
(5) logical connectives $\neg$ (not) and & (and).
(6) universal quantifier $\forall$.
(7) parentheses (,).

The set of terms of the first-order language L, Term(L), is the smallest set X such that

(1) all individual variables $v_0, v_1, \ldots$ and constant symbols c_k are members of X.
(2) if $t_1, \ldots, t_{\mu(j)} \in X$, then $f_j(t_1, \ldots, t_{\mu(j)}) \in X$ for each $j \in J$.

The set of atomic formulas of L, Atom(L), consists of all elements of the form $r_i(t_1, \ldots, t_{\lambda(i)})$, where $t_1, \ldots, t_{\lambda(i)} \in$ Term(L).

Finally, the set of well-formed formulas (or simply formulas) of L, Form(L), is defined as the smallest set Y such that

(1) Atom(L) $\subset Y$.
(2) if $\phi, \psi \in Y$, then $\neg\phi$, $\phi \,\&\, \psi$, $\forall v_i \phi \in Y$.

Additional logical connectives $\vee$, $\to$, and $\leftrightarrow$ are defined in terms of the primitives as follows:

$$A \vee B \equiv \neg(\neg A \& \neg B),$$
$$A \to B \equiv \neg A \vee B,$$
$$A \leftrightarrow B \equiv A \to B \& B \to A,$$
$$\exists v_i \phi \equiv \neg \forall v_i \neg \phi.$$

The above is a standard definition of a wff. The connectives $\to$ and $\leftrightarrow$ are the main ingredients for a wff in the model theory approach.

2.3 Basic Notation for Computer-Acceptable Set Theory

The following are extensions of the first-order language for computer-acceptable set theory. A symbol is an alphanumeric string starting with an alphabetical character.

(1) The universal quantifiers are not explicitly used in this approach. Square brackets [and] are used to represent a list structure. For example, [1, a, "A"] is a list.
(2) The conventional expression of numbers is used for numeric constant symbols. 12.3 is a numerical constant symbol. Arithmetic and relational operators are used in the usual way. The absolute operator is given by a function abs(). For example, $|-5| = \text{abs}(-5)$.
(3) A constant symbol is given by the form .⟨symbol⟩. For example, .Ab is a constant symbol.
(4) "⟨string⟩" is used as a text-type constant symbol. For example, "order ID cannot be changed" is a text-type constant symbol.
(5) A symbol that is not a constant is a variable. For example, X1 is a variable.
(6) A variable with the suffix ".g" is treated as a special variable called a global variable. A global variable is a variable that is valid over the entire model, whereas a regular variable is valid only within the statement in which it is used. A statement is defined in (13) below. For example, the statement "ActionName.g=["quit", "register"];" defines a global variable ActionName.g.
(7) A file is denoted by the form ⟨filename⟩.lib and is treated as a global variable. Since a file is a list of records (see Chapter 14), it is treated as a set in the model theory approach (see Section 15.3). The term "stdUI_para.lib" in Section 1.2 is a global variable that represents a file.
(8) The equality symbol = is used as a binary predicate symbol.
(9) The symbol := is used as a binary predicate symbol to indicate the assignment of a value v to a variable x, i.e., $x := v$.
(10) Due to restrictions on keyboard input, logical operator symbols are replaced by computer-acceptable symbols (see the list below).

Connective	Computer-acceptable symbol	Example	Replaced example
$\neg$	not()	$\neg p$	not(p)
$\wedge$(&)	and	p and q	p and q
$\vee$	or	p or q	p or q
$\leftrightarrow$	<->	$p \leftrightarrow q$	p < - > q
$\rightarrow$	->	$p \rightarrow q$	p -> q
$\neq$	<>	$p \neq q$	p <> q

Here, "and" is replaced by "," by convention.

(11) A set is represented as a list term. For example, $[1, [2, 3], x]$ is a set expression, where "[...]" is treated as a function. More precisely, [...] is a composition of a binary function symbol dot(). For example, $[1, 2, 3] = \text{dot}(1, \text{dot}(2, \text{dot}(3, \text{nil})))$ where nil denotes the empty list.

(12) Let A, B, and C be formulas. Then, an expression (left-hand side) may be replaced as follows:

$$(A \rightarrow B)\&(\neg A \rightarrow C) \equiv (A)\text{->}(B) \text{ otherwise } (C).$$

(".otherwise" can be also used.) For example,

$$(x >= a)\text{->}(y = u) \text{ otherwise } (y = v).$$

This is a fundamental construction in the model theory approach.

(13) The following four types of formulas terminated by ";" are called statements:

(i) ⟨global variable⟩ = ⟨term⟩. Used to define global variables, e.g.,

ActionName.g = ["quit", "register"];

(ii) ⟨predicate symbol⟩(⟨argument list⟩) <-> ⟨formula⟩. Used to define predicates, e.g.,

$$p(y, x, [])\text{<->}y = x * x;$$

(iii) ⟨function symbol⟩(⟨argument list⟩) = ⟨variable⟩ <-> ⟨formula⟩. Used to define functions, e.g.,

$$\text{createY}([\text{"}a\text{"}, 1, \text{"}b\text{"}, 2]) = Y\text{<->}$$
$$Y := \text{defSet}(p(y, x, []), [x, [\text{"}a\text{"}, 1, \text{"}b\text{"}, 2]]);$$

(iv) atomic formula. Used to define a record of data, e.g.,

connection("tokyo", "oosaka", 5);

(14) A user model in set theory is a set of statements. For example, "warp2.set" in Chapter 1 is a user model.

(15) Quantifiers are not used explicitly in the model theory approach. However, every statement is assumed to be quantified by the universal quantifier.

(16) A metastatement "func([....]);" is used to declare function symbols. (".func([...])" can be also used.) If a predicate is listed in func([....]);, it can be used as a function. For example,

$$\text{func}([p, \ldots]); \qquad p(x, y, z)\text{<->}z = x * y;$$

then, $z := p(2, 3)$ yields $z = 6$. System-defined functions need not be declared. The basic system-defined functions are presented in Appendix 2.1.

(17) Appendix 2.1 lists the system-defined functions and predicates that are used in this book to construct and manipulate sets, predicates, and functions.

The following is an example of a statement defining the function update_order3:

```
update_order3([oid,...,est_payment])=res <->
        (project(order.lib,opos,[oid,...,sdate])) ->
            (
            Date:=getDate2(),
            order.lib:=replaceList(order.lib,opos,[oid,...,Date]),
            res:=["UPDATE_ORDER...",["est_del=",est_del,...,est_payment]]
            )
```

```
.otherwise
          (
          res:="..."
          );
```

Here, oid,...,est_payment are variables and hence terms. Then [oid,...,est_payment] is a term because [..] is a function. Since update_order3 is a function, update_order3 ([oid,...,est_payment]) is a term. Then, since res is also a term, update_order3 ([oid,...,est_payment]) = res is an atomic formula, since = is a predicate. Similarly,

```
project(order.lib,opos,[oid,...,sdate]),
Date:=getDate2(),
order.lib:=replaceList(order.lib,opos,[oid,...,Date]),
res:=["UPDATE_ORDER...",["est_del=",est_del,...,est_payment]]
res:="..."
```

are atomic formulas. The sequence of formulas

```
Date:=getDate2(),
order.lib:=replaceList(order.lib,opos,[oid,...,Date]),
res:=["UPDATE_ORDER...",["est_del=",est_del,...,est_payment]]
```

is a formula because "," is a connective. Then,

```
(project(order.lib,opos,[oid,...,sdate])) -> (..).otherwise(...)
```

is a formula. Finally,

```
"update_order3([oid,...,est_payment])=res<->....otherwise (res:="...");"
```

is a statement defining the function update_order3(). Although universal quantifiers are omitted, every variable in the statement is assumed to be quantified by $\forall$.

It should be noted that not all statements can be meaningful as an element of a model description, as discussed below.

2.4 Sets

a. Set Notation

A set is represented as a list in the model theory approach, where $X = \{x_1, \ldots, x_n\}$ is represented by $X = [x_1, \ldots, x_n]$. An empty set is represented by []. Although this convention is convenient, it should be noted that since in general $[a, b] \neq [b, a]$, checking equality between sets requires sorting a list in the model theory approach. For example, an equality check between sets A and B is performed by sort(A) = sort(B) instead of $A = B$.

b. Set Construction

A set is constructed by identifying elements of a list directly or by an extension of a predicate. "ActionName.g=["quit", "register"]" is an example of set construction by

the identification of elements. An extension is given by the function defSet(). Suppose a set Ys is given by

$$Ys = \{y | p(y, x, \langle\text{parameter list}\rangle), x \in Xs\}.$$

This set is then constructed as

```
Ys:=defSet(p(y,x,[parameter list]),[x,Xs]);
p(y,x,[parameter list]) <->
       definition of p();
```

For example, $\{n * n | n \in \{1, 2, 3, 4, 5\}\}$ is given by

```
Ys:=defSet(p(y,x,[]),[x,[1,2,3,4,5]]);
p(y,x,[])<->y=x*x;
```

c. Set Algebra Notation

Due to restrictions on keyboard input, set-theoretic operator symbols are replaced by the following computer-acceptable terms and atomic formulas:

Operator symbol	Computer-acceptable term	Example	Replaced example
$\cap$	intersection()	$A \cap B$	intersection(A, B)
$\cup$	union()	$A \cup B$	union(A, B)
$\times$	product	$A \times B$	product(A, B)
$-$	minus()	$A - B$	minus(A, B)
$\vert\,\vert$	cardinality()	$\vert A\vert$	cardinality(A)
$\in$	.in (member)	$x \in Xs$	x .in Xs (member(x, Xs))
$\notin$	.notin (notmember)	$x \notin Xs$	x .notin Xs (notmember(x, Xs))
$\subset$	subset()	$A \subset B$	subset(B, A)
$\not\subset$	notsubset()	$A \not\subset B$	notsubset(B, A)

d. Set Modification

A new element x is added to a set (list) Xs as follows.

```
Xs2:=union(Xs,[x]).
```

Since a set is represented as a list, a new element x is appended to a set (list) Xs by

```
Xs2:=append(Xs,[x]).
```

It should be noted that union([1, 2], [2]) = [1, 2] and append([1, 2], [2]) = [1, 2, 2].

An element x is deleted from a set (list) Xs by

```
Xs2:=minus(Xs, [x]).
```

In the above expressions, the element $[x]$ is used. However, a set (list) can replace $[x]$.

An element x of a set Xs is replaced by another element y by

```
pos:=invproject(Xs,x), Xs2:=replaceList(Xs,pos,y);
```

The function invproject(Xs, x) gives the position pos of x in Xs.

In general, $Xs2$ in the above expressions must be a variable that differs from Xs. "Xs := append(Xs, $[x]$)" is not allowed. However, if Xs is a global variable (for example, $Xs = Ys$.g), the following expressions are allowed:

```
Ys.g:=union(Ys.g, [x]);
Ys.g:=append(Ys.g, [x]);
Ys.g:=minus(Ys.g, [x]);
pos:=invproject(Ys.g,x), Ys.g:=replaceList(Ys.g,pos,y);
```

2.5 Predicate (Relation)

a. Relation Construction

A relation is simply represented as a predicate, which is defined in the following form:

$p(v_1, \ldots, v_n)$<->
    conditions on $\{v_1, \ldots, v_n\}$;

For example,

```
p(x,y) <->
    x<=y;
```

An atomic formula representing a record is created by the predicate "assign," as discussed in Section 2.7.

b. Relation Modification

(i) If a relation is given by a set (its extension), the method of modification for the set is applicable to the relation.

(ii) If a relation is given by a predicate p(x), it can be modified by

```
P2(x) <->
(condition(x)) ->
          (
          /*modified statement*/
          )
.otherwise
          (
          p(x)
          );
```

c. Quantifiers

Quantifiers are not used explicitly in the model theory approach. However, if the target set Z is finite, the functions of the quantifiers are handled by defSet() in the following way:

(i) universal quantifier: $(\forall x \in Z)(p(x))$:

$$(\forall x \in Z)(p(x)) \text{ is true if and only if (iff) } Z = \text{defSet}(p2(x, x, []), [x, Z]),$$

where

```
p2(x, x, []) <->
        p(x);
```

(ii) existential quantifier: $(\exists x \in Z)(p(x))$:

$$(\exists x \in Z)(p(x)) = \text{true iff } [] <> \text{defSet}(p2(x, x, []), [x, Z]),$$

where

```
p2(x, x, []) <->
        p(x);
```

d. Data (Record) Handling

Data may be defined as a class of atomic formulas that have a common predicate name. In this case, the data correspond to a table in a database, where each atomic formula represents a record of it. This is discussed in more detail in Section 2.7.

2.6 Functions

a. Function Construction

A function is defined in several ways.

(i) A function may be represented as a set or a relation. Suppose $f : \{1, \ldots, 10\} \to N$ such that $f(n) = n * n$. Then,

$$f = \{(n, n * n) | n \in \{1, \ldots, 10\}\}.$$

In the model theory approach, f is defined by defSet() as

```
f:=defSet(pf(y,x, []), [x, [1,2,3,4,5,6,7,8,9,10]]);
pf(y,x, []) <->
        y = [x, x * x];
```

It should be noted that $[n, n + 1, \ldots, n + k - 1]$ can be generated by the system-defined function genIndex(n, k).

Suppose $f : X \to Y$ is specified as

$$f(x) = \begin{cases} z & \text{if condition}(x) = \text{true}, \\ z' & \text{if condition}'(x) = \text{true}, \\ \bullet & \\ \bullet & \\ z'' & \text{if condition}''(x) = \text{true}. \end{cases}$$

Then, $f : X \to Y$ is described in the model theory approach as

```
f(x)=y <->
          (condition(x)) ->
                     (y:=z),
          (condition'(x)) ->
                     (y:=z'),
                     .
                     .
          (condition"(x)) ->
                     (y:=z");
```

In particular, if f is specified as

$$f(x) = \begin{cases} y_1 & \text{if condition } A \text{ is true}, \\ y_2 & \text{otherwise}, \end{cases}$$

f is described by

```
f(x)=y <->

          (conditionA)->(y:=y1)
          .otherwise (y:=y2);
```

(iii) A function can be defined by a predicate using the metastatement func([...]) introduced in Section 2.3.

(iv) According to recursive function theory, a complicated function can be defined in three ways: composition, primitive recursion, and minimization.

- **Composition**

Suppose $f : X \to Y$ and $g : Y \to Z$ are given. Then, a function $h : X \to Z$ is defined by the composition operation as $h(x) = g(f(x))$. This operation is accepted if f, g, and h are declared in the metacommand func().

- **Primitive Recursion**

Let N be the set of integers. If $f : X \to Y$ and $g : N \times Y \times X \to Y$ are given, the primitive recursion operation defines a function $h : N \times X \to Y$ as

$$h(0, x) = f(x)$$

and

$$h(n+1, x) = g(n, h(n, x), x).$$

In the model theory approach, h is constructed by

$$h(n, x) = y \leftrightarrow (n = 0) \text{ -> } (y = f(x))$$

$$.\text{otherwise}$$

$$(y = g(n, h(n, x), x));$$

The essence of primitive recursion is the recursive operation. To avoid the stack memory problem that may occur in recursive operations, this definition is implemented by the defSet() function as

```
h(n,x)=y <->
        (n=0) ->
                (
                y:=f(x)
                )
        .otherwise
                (
                y.g:=f(x),
                ns:=genIndex(1,n),
                ys:=defSet(precursion(y,z,[x]),[z,ns]),
                y:=project(ys,0)
                );
precursion(y,z,[x]) <->
        y:=g(z,y.g,x),
        y.g:=y;
```

• **Minimization**

Suppose $f : N \times X \to \text{Re}$ is given, where Re is the set of real numbers. Then, a new function $h : X \to N$ is generated by the minimization operation as follows:

$$h(x) = \text{mini}\{n | f(n, x) = 0, n \in N\}.$$

In general, h may be a partial function, whereas f is assumed to be a total function. If N is finite, that is, $N = \{0, 1, \ldots, L\}$, the minimization operation is implemented by defSet() as

```
h(x,N)=y <->
          ys:=defSet(pmin(y,n,[x]),[n,N]),
          y:=project(ys,0);
pmin(y,n,[x]) <->
          (f(n,x)=0) ->
                     (
                     y:=n,
                     .break
                     )
```

Since only finite sets are used in the model theory approach, the composition operation and the defSet() function are sufficient for generating computable functions.

b. Function Evaluation

(i) Suppose a function $f : X \rightarrow Y$ is defined as a set, that is, $f = \{(x, f(x)) | x \in X\}$. In this case, y is obtained by the predicate member(). Execution of member($[x, y], f$) yields y for a given x. Let $f = [[1, 2], [2, 4], [3, 9]]$. Then, member($[2, y], f$) yields $y = 4$.

(ii) Suppose a function $f : X \rightarrow Y$ is defined by

```
f(x)=v <->
      (specification of v);
```

In this case, y is simply obtained by $y := f(x)$ if f is declared in func(). For example, f is defined as

```
f(x)=y <-> y=x*x;
```

Then, $y := f(2)$ yields $y = 4$.

c. Function Modification

(i) Suppose a function $f : X \rightarrow Y$ is defined as a set, that is, $f = \{(x, f(x)) | x \in X\}$. In this case, a new function $f1 : X \rightarrow Y$ where

$$f1(x) = \begin{cases} f(x) & \text{if } x \neq x1, \\ y1 & \text{otherwise,} \end{cases}$$

is created by

```
Ix:=invproject(f,[x1,y0]),
f1:=replaceList(f,Ix,[x1,y1]);
```

The function replaceList(f, Ix, $[x1, y1]$) replaces the Ixth element by $[x1, y1]$ in f.

(ii) Consider a function $f : X \rightarrow Y$ defined as

```
f(x)=y <->
          (condition(x)) ->
                    (y:=z),
          (condition'(x)) ->
                    (y:=z'),
                    .
                    .
          (condition"(x)) ->
                    (y:=z");
```

Then,

```
f1(x)=y <->
          (x=x1) ->
                  (
                  y:=y1
                  )
          .otherwise
                  (
                  y:=f(x)
                  );
```

2.7 User Model Description in Set Theory

The previous sections introduced basic notation for the description of a user model in set theory. This section illustrates how this notation is used in practice for user model construction.

a. State Transition

For a TPS the state set C is usually represented by

$$C = F_1 \times \cdots \times F_n,$$

where

$$F_j \subset (\mathrm{Attr}_{j1} \times \cdots \times \mathrm{Attr}_{jk})^*.$$

Here F_j is a set of files and Attr_{jt} is an attribute set of F_j. State modification can be achieved by two methods: change of the record number in a file, or change of the content of a record.

(i) Change of the record number in a file is easily performed by the functions append(), union(), and minus().

(ii) Change of record content usually requires three steps. Suppose a record r_0 in the file F_i is to be replaced by another record r_1. The procedure is then as follows.

Step 1: Identification of the position of $r_0 = (\mathrm{attr}_{00}, \ldots, \mathrm{attr}_{0k})$ in F_i. This is performed by

$$\mathrm{pos} := \mathrm{invproject}(F_i, r_0);$$

It is important to note that $\mathrm{attr}_{00}, \ldots, \mathrm{attr}_{0k}$ of r_0 can be variables unless a "key" element (e.g., the "ID" element) is a variable. Here, invproject() assigns proper values to the variable elements if the operation is successful.

Step 2: Generation of r_1. Suppose this operation requires replacement of attr_{0p} by a new value v_p. Then, r_1 is given by

$$r_1 := \mathrm{replaceList}(r_0, p, v_p);$$

If the position p is unknown, step 1 can be reapplied to r_0.

Step 3: Replacement of r_0 by r_1: This operation is performed by

$$F_i := \mathrm{replaceList}(F_i, pos, r_1);$$

b. Response Generation

Response generation usually requires evaluation of the current state $c \in C$. If the evaluation can be completed by dealing with one record in a file, it can be performed by project() or member(). However, there are many cases that require every record in a file to be checked. Such operations are carried out by defSet(). Suppose for $F_j \subset (\mathrm{Attr}_{j1} \times \cdots \times \mathrm{Attr}_{jk})^*$ that the last elements of the records satisfying a predicate

p2() are to be collected. This requirement is realized through the use of defSet() as follows:

```
res:=defSet(p1(y,r,[parameter_list]),[r,Fj]);
p1(y,r,[parameter_list]) <->
            attr0:=project(r,0),
            (p2(attr0)) ->
                        (
                        y:=attr0
                        );
```

Here, project(r, 0) provides the last element attr0 of the record r.

c. Data Handling

Since the system-defined function listPred() provides the class of records of data (a file) as a list (a set), data handling is relatively easy in the model theory approach. For example, suppose the following data are provided, where the first element of a list is assumed to be a key:

```
connection(["tokyo","oosaka"],5);
connection(["tokyo","nagoya"],3);
connection(["nagoya","oosaka"],2);
```

Here, ["tokyo", "oosaka"] is a key of the list (["tokyo", "oosaka"], 5). Then,

```
L:=listPred("connection");
```

yields

```
L=[[["tokyo","oosaka"],5],[["tokyo","nagoya"],3],[["nagoya","oosaka"],2]].
```

An atomic formula can be created by the predicate "assign." For example, the statement

```
assign(["connection",["oosaka","fukuoka"]],8);
```

creates the atomic formula

```
connection(["oosaka","fukuoka"],8);
```

Since "assign" can overwrite a formula by applying a key, an atomic formula can be modified. For example, the statement

```
assign(["connection",["oosaka","fukuoka"]],8.5);
```

can replace

```
"connection(["oosaka","fukuoka"],8)"
```

by

```
"connection(["oosaka","fukuoka"],8.5)".
```

An atomic formula can be erased by a predicate "retract_byhead." For example, "connection(["oosaka", "fukuoka"], 8.5)" is erased by

```
retract_byhead(connection(["oosaka","fukuoka"],8.5));
```

Here, since ["oosaka", "fukuoka"] is a key, the predicate retract_byhead(connection (["oosaka", "fukuoka"], x)) can perform the same function with x as a dummy variable. It should be noted that the standard predicate "retract" of Prolog requires that the target object be fully specified. A text-type constant symbol of the above example can be replaced by a regular constant symbol.

2.8 User Model Compilation

A user model in computer-acceptable set theory is translated into a model in extProlog by the setcompiler. Figure 2.1 outlines the compilation scheme.

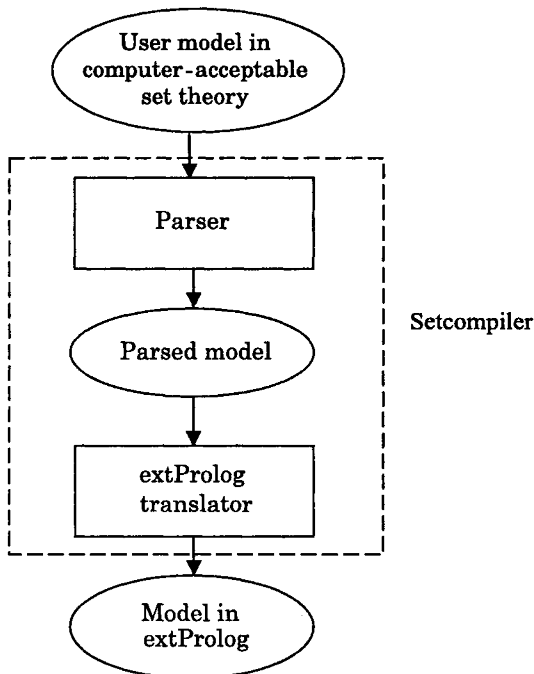


Fig. 2.1. Setcompiler.

The parser translates an arithmetic/logic formula, a predicate, a function, or an if–then formula into a corresponding internal object of extProlog. Thus, each statement of the user model is translated into a list of objects, and the entire user model is translated into a list of lists.

Consider the user model of Fig. 2.2.

```
/*register53.set*/
func([delta_lambda,register]);
ActionName.g=["quit","register"];
/*atomic process of register*/
register.g=["name","institute"];
delta_lambda([.register])=res <->
          paralist:=stdUI_para.lib,
          res:=register(paralist),
          stdUI_res.lib2:=res;
register([name,institute])=res <->
           (notmember'[[name,institute],y],participant.lib)) ->
                     (
                     participant.lib2:=union(participant.lib,
                                         [[[name,institute],[0,0,0]]]),
                     res:=[name,institute,.new_registration,.fee,.track]
                     )
          .otherwise
                     (
                     [fee,track,date]:=participant.lib([name,institute]),
                     (fee <> 0 and track <> 0) ->
                                (
                                res:=[name,institute,.pre_registration,.receipt,
                                      .proceeding]
                                )
                     .otherwise
                                (
                                res:=[name,institute,.incomplete_registration,
                                      .fee,.track]
                                )
                     );
```

Fig. 2.2. Example of a user model in computer-acceptable set theory.

Figure 2.3 shows the parsed result of the function register() in Fig. 2.2.

```
[_eqsds(register([name,institute]),res)
iff
[notmember([[name,institute],y],participant.lib),_imply,
(participant.lib2,_assign,union(participant.lib,[[[name,institute],
[0,0,0]]]),,,res,_assign,[name,institute,_new_registration,_fee,_track])
,_otherwise,([fee,track,date],_assign,participant.lib([name,institute]),,,
[(_and(_nesds(fee,0),_nesds(track,0))),_imply,(res,_assign,[name,institute
,_pre_registration,_receipt,_proceeding]),_otherwise,(res,_assign,
[name,institute,_incomplete_registration,_fee,_track])])]]
```

Fig. 2.3. Parsed form of the function register().

The following are typical correspondences between the original formula and the parsed result:

register([name,institute])=res	↔	_eqsds(register([name,institute]),res),
<->	↔	iff,
(notmember([[name,institute],y], participant.lib))	↔	(notmember([[name,institute],y], participant.lib)),
->	↔	_imply,
fee <> 0 and track <> 0	↔	_and(_nesds(fee,0),_nesds(track,0)).

The translation is performed in the following way:

- Operator symbols are replaced by internal symbols:

$$= \rightarrow _\text{eqsds},$$
$$:= \rightarrow _\text{assign};$$

- Constant symbols have "." replaced by "_":

$$.\text{new_registration} \rightarrow _\text{new_registration};$$

- Variable and numeric symbols are not changed:

$$\text{fee} \rightarrow \text{fee},$$
$$0 \rightarrow 0;$$

- Arithmetic and logic formulas are replaced with the prefixed normal form:

$$\text{fee} <> 0 \rightarrow _\text{nesds}(\text{fee}, 0);$$

  where "_nesds" is the internal representation of "<>" (not equal).
- Predicates and functions are not changed:

$$\text{register}([\text{name},\text{institute}]) \rightarrow \text{register}([\text{name},\text{institute}]).$$

The parsed result is again translated into standard extProlog code by the translator. extProlog is discussed in more detail in Chapter 3. Figure 2.4 shows the extProlog TPS user model generated from Fig. 2.2.

```
//register53.p
func("register53.set", [getDate,fileFunction,invprojectList,invproject,
            getIndex,getSubtable,check_participant,plist,total_fee,trackplist,
            delta_lambda,register,setupparticipant,updateparticipant,
               reg_update]);
actionName.g(["quit","register"]);
register.g(["name","institute"]);
delta_lambda([_register],Res):-
            xread("stdUI_para.lib",StdUI_para.lib,"text"),
            Paralist:=StdUI_para.lib,
            Res:=register(Paralist),
            StdUI_res.lib2:=Res,
            appendf("stdUI_res.lib","w",StdUI_res.lib2 );
register([Name,Institute],Res):-
            xread("participant.lib",Participant.lib,"text"),
            if notmember([[Name,Institute],Y],Participant.lib)  then
                     Participant.lib2:=union(Participant.lib,[[[Name,Institute],
                       [0,0,0]]]),
                     appendf("participant.lib","w",Participant.lib2 ),
                     Res:=[Name,Institute,_new_registration,_fee,_sym]
            else
                     [Fee,Sym,Date]:=fileFunction("participant.lib",
                                      [Name,Institute]),
                     if and(nesds(Fee,0),nesds(Sym,0))  then
                              Res:=[Name,Institute,_pre_registration,_receipt
                                    ,_proceeding]
                     else
```

```
                                        Res:=[Name,Institute,_incomplete_registration
                                             ,_fee,_sym]
                            end
                  end ;
?-funcCall("register53.set"),stdDPS_UI();
#include "stdDPS_UI.p";
```

Fig. 2.4. User model in extProlog "register53.p" generated from "register53.set".

The function names "delta_lambda," "getDate," "fileFunction," "invprojectList," and "invproject" are appended to func by the setcompiler as standard functions. The last statement, #include "stdDPS_UI.p," sets the standardized internal UI to be included upon execution.

2.9 Input–Output Operations in Set Theory

Input and output operations for a user model are usually defined in the preprocess() and postprocess() predicates of a user model. The actual operations are provided by extProlog predicates. As mentioned in Section 2.8, a predicate form is not modified on translation, and hence in principle, the input–output predicates of extProlog can be used directly for the input–output operations of a user model (see Fig. 1.6 and Appendix 1.1). The input–output predicates of extProlog are discussed in Chapter 3.

Appendix 2.1 System-Defined Predicates and Functions Used in This Book

`append()` : binary function;

See Section 2.4.

`assert()` : unary predicate;

See Section 3.4.

`assign()` : binary predicate;

See Section 3.4.

`clearsheet()` : unary predicate;

`clearsheet(Wp)` implies that a spreadsheet of extProlog whose ID is WP is cleared.

`concat()` : ternary predicate;

`concat("ABCD","abcd",x)` implies `x="ABCDabcd"`.

`defSet()` : ternary function;

See Section 2.4.

`deleteList()` : binary function;

`lis:=deleteList([1,[3,4],3],2)` implies `lis=[1,3]`.

`genIndex()` : binary function;

`x:=getIndex(n,k)` implies `x=[n,n+1,...,n+k-1]`.

`getDate()` : nullary function;

`x:=getDate()` implies `x=current time` in the form `[year,month,day,hour,minute,second]`.

`getDate2()` : nullary function;

`x:=getDate2()` implies `x=current time` in the form `[year,month,day]`.

`getvalue()` : ternary predicate;

See Section 3.4.

`inverse()` : unary function;

`x=inverse([a,b,c,d])` implies `x=[d,c,b,a]`.

`invproject()` : binary function;

`x:=invproject([a,b,c],c)` implies `x=3 or 'c'` is the third element of `[a,b,c]`.

`listPred()` : unary function;

See Section 2.7.

`load_go()` : unary predicate;

See Section 15.3.

`makewindowSS()` : 4-ary predicate;

`makewindowSS(Wp,"mytree",W,H)` is to open a spreadsheet `Wp` whose title is `"mytee"` and size is `W x H`.

`project()` : binary function;

`x:=project([1,2,3],2)` implies `x=2`.

`replaceList()` : binary function;

`lis:=replaceList([1,2,3],2,[3,4])` implies `lis=[1,[3,4],3]`.

`retract()` : unary predicate;

See Section 2.7.

`retract_byhead()` : unary predicate;

See Section 2.7.

`setvalue()` : ternary predicate;

See Section 3.4.

`showl()` : binary predicate;

See Appendix 3.1.

`sort()` : unary function;

`x:=sort([b,2,a,1,"e",4,"cd"])` implies `x:=["cd","e",a,b,1,2,4]`.

`sortmax()` : unary function;

`x:=sort([b,2,a,1,"e",4,"cd"])` implies `x:=[4,2,1,b,a,"e","cd"]`.

`sum()` : unary function;

`x=sum([1,2,3])` implies `x=6`.

`transpose()` : unary function;

`x:=transpose([[a,b,c],[1,2,3]])` implies `x=[[a,1],[b,2],[c,3]]`.

`xread()` : binary predicate;

`xread(0,Ans)` is to get data from the dialog window, whose window pointer number is 0 and insert it into the variable `Ans`.

`xwriteln(0,-)` : predicate;

`xwriteln(0,"Ans=",X)` is to display {`"Ans=",X`} in the dialog window.

The following are special predicates for data mining [Takahara, 2003]:

```
procC("entropy",-,-).

procC("getcategory",-,-),

procC("getelement",-,-),
```

References

Bridge, J. (1977) *Beginning of Model Theory*, Oxford University Press.

Takahara, et al. (2003) *Study on 4-th Generation Systems Development Methodology for MIS*, Internal Report, Dept. of Management Information Science, Chiba Institute of Technology, Chiba, Japan.

3

Implementation Language: extProlog*

This chapter introduces extended Prolog (extProlog). Although primarily used as an implementation language for the model theory approach in a hidden manner similar to a machine language, knowledge of this language is also helpful for understanding the model theory approach, for the following reasons:

(1) Input–output operations are provided by predicates of extProlog as well as HTML.
(2) Advanced control schemes, such as those used in Chapter 13, are implemented with direct support of extProlog.
(3) The standardized components of extSLV and TPS are implemented in extProlog.
(4) An efficient user model can be directly constructed in extProlog if a system developer is familiar with extProlog.

The extProlog language is adopted for implementation of the model theory approach due to its semantic closeness to set-theoretic modeling and other advantages as discussed below. This semantic closeness allows a model described in set theory to be translated into an extProlog model automatically by the setcompiler. Consequently, extProlog is discussed in this chapter at considerable depth. It is assumed that readers are familiar with the C programming language, although familiarity is not absolutely necessary.

3.1 Extended Prolog (extProlog)

Practical experiences with the model theory approach projects have led to the following language requirements (R1–R9) for the model theory approach:

R1: It should be readily capable of implementing a model described in set-theoretic terms, and should also be a general-purpose language.
R2: It should support the model integration approach [Takahara, Iijima, and Shiba, 1993] and allow for the representation of a composite system consisting of subsystems. It is common in practice for a complicated system to be built as a system of subsystems (discussed in Chapter 13).

R3: It should have good numeric processing capabilities, since the management problems targeted by the new methodology are basically dependent on numerical processing.
R4: It should have good symbolic processing capabilities, since many real problems require heuristic solution algorithms. This case is illustrated in Chapter 15.
R5: It should have database connectivity.
R6: It should provide a standard graphical user interface (GUI). Although every MIS should have an interface between the system and the user, it is not usually necessary to build a beautiful GUI: rapid development of the GUI is more important. This chapter, and Chapters 14 and 15, discuss standardized GUIs for extSLV and TPS, respectively.
R7: It should provide reasonable program execution speed.

Conventional Prolog already satisfies some of these requirements. However, basic Prolog also presents some difficulties as the implementation language for the proposed development approach:

- It is not a fully general-purpose language and is weak in a number of areas including execution control.
- It does not support the model integration approach.
- It is not particularly good for numerical processing.
- It does not provide database connectivity.
- It does not provide a GUI.
- Programs in standard Prolog are unreadable in the sense that functional expression is not allowed.

To overcome the weaknesses of standard Prolog, an extended form of Prolog called extProlog [Takahara et al. 2003] has been developed. The specific extensions are as follows:

E1: Array (vector and matrix) processing. This is realized in two ways. Firstly, standard Prolog itself is already strong in terms of list processing, but is unable to perform arithmetic operations on lists directly. The arithmetic operations are extended for lists as termwise operations in extProlog. For example, if $X = [X1, X2]$ and $Y = [Y1, Y2]$, $X + Y$ yields $[X1 + Y1, X2 + Y2]$. The same holds for Boolean expressions.
E2: Introduction of a model description language (MDL) as a provision for the model integration approach. This is discussed in Chapter 13.
E3: Numerical execution tools. Numerical processing is improved in extProlog by introducing predicates and functions on lists and numerical computations.
E4: Introduction of object-oriented database connectivity, and provision for standard SQL queries. This is discussed in Chapter 16.
E5: Standardized GUIs. Figure 1.7 shows an example of a standardized GUI. This is discussed in Appendix 3.1.
E6: Program structure control functions such as *for*, *if*, and *repeat* clauses.
E7: Introduction of a mechanism to call a predicate as a function. In general, if Y of a predicate $p(X_1, \ldots, X_n, Y)$ is defined as an output of $p()$, $p()$ can be called as a

function using the form $Y := p(X_1, \ldots, X_n)$ if it is declared in function(). (See Section 3.4.2.) Of course, $p()$ can also be used in the form $p(X_1, \ldots, X_n, Y)$.

3.2 Examples

This section presents two typical examples of extProlog programming. The first is a numerical programming example, and the second is a symbolic programming example.

3.2.1 Numerical Example

Prolog is usually considered a language most suitable for symbol processing. However, extProlog is extended to handle numeric processing for management in a reasonable way. This section will discuss numerical programming in extProlog using a linear simultaneous equation as an example. The problem is solved by the Gauss–Jordan method in both C and extProlog programs.

The following is a C program for solving a linear simultaneous equation:

```
/*simuleq3.c*/
/*solution of simultaneous equations*/
/*A'x=b;A=[A',b]*/
#define N 100
#include <stdio.h>
main()
{
    float a[N][N+1]={{1,2,5},{3,1,5}};
    float x[N];
    int i,n=2;

    /*solve by Gauss-Jordan method*/
    for (i=0;i<n;i++)
    {
        mypivot(a,n,i);
    }
    for(i=0;i<n;i++)
        x[i]=a[i][n];
    for (i=0;i<n;i++)
        printf("x[%d]=%e\n",i,x[i]);
}

mypivot(a,n,i)
    float  a[N][N+1];
    int n,i;
{
    float  aii,aki;
    int k,j;

        aii=a[i][i];
        for (j=i;j<=n;j++)
            a[i][j]=a[i][j]/aii;
        for (k=0;k<n;k++)
        {
            if(k!=i)
```

```
            {
            aki=a[k][i];
            for (j=i;j<=n;j++)
                a[k][j]=a[k][j]-a[i][j]*aki;
            }
        }
}
```

Programs in C can be translated into extProlog in a straightforward manner, as discussed below. However, the converse is not true.

In principle, a Prolog program consists of a set of rules (clauses), and each rule is of the following form:

$$q():-p_1(),\ldots,p_n();$$

where $q()$, $p_1()$, ..., $p_n()$ are predicates. The comma, "," should be understood as the "and" connective. The $q()$ component is the head of the rule, and :- $p_1()$, ..., $p_n()$; is the tail. Although the rule is usually understood to indicate the following implication relation,

if the predicates $p_1()$, ..., $p_n()$ are true, then the predicate $q()$is true,

this interpretation is not used in this chapter. The theory of Prolog as a logic programming language is discussed in detail in Chapter 17. Here, a rule is interpreted as a subroutine. In a numerical program of extProlog, $p_i()$ is a predicate representation of a numerical equation and $q()$ represents a procedure head. As shown below, there is no structural difference between a numerical program in C and its counterpart in extProlog.

The primary modifications involved in translation from C to extProlog are as follows:

1. The declaration part is omitted.
2. All variable names should start with an uppercase letter.
3. The terminal symbol ";" is replaced with "," except for the very end terminal ";".
4. Array expressions are replaced with special predicates.
5. A subroutine name is used as the head of the corresponding rule.
6. A special rule "?-main();" is appended to the Prolog program to initialize execution of the main() rule.
7. System-defined subroutines in C are replaced with system-defined predicates in extProlog.

The first modification follows from the fact that variables in Prolog are typeless. This feature makes Prolog the most convenient platform for complicated system development. The development methodology presented in this book relies heavily on this feature.

The above C program consists of two routines, main() and mypivot(), and therefore the corresponding Prolog program consists of two rules, with heads main and mypivot.

After the declaration part has been omitted and the variable names have been modified, the main routine appears as follows:

```
main()
{
    for (I=0;I<N;I++)
    {
                mypivot(A,N,I);
    }
    for(I=0;I<N;I++)
                X[I]=A[I][N];
    for (I=0;I<N;I++)
                printf("X[%d]=%f\n",I,X[I]);
}
```

Furthermore, inserting the data assignments $A := [[1,2,5],[3,1,5]]$, $N :=$ strlen(A), and $X :=$ constantlist(0, N) into the above code in place of the corresponding C codes, $a[N][N+1] = \{\{1,2,5\},\{3,1,5\}\}$, $n = 2$, and float $x[N]$, results in the following extProlog program, which corresponds to main() in the C program:

```
/*simuleq10.p*/
/*solution of simultaneous equations*/
/*A'x=b;A=[A',b]*/
main():-
        /*specify [A',b]*/
        A:=[[1,2,5],[3,1,5]],
        /*get dimension of A*/
        N:=strlen(A),
        X:=constantlist(0,N),
        /*solve by Gauss-Jordan method*/
        for(I:=0;I<N;I++)
        begin
                mypivot(A,N,I)
        end,
        for(I2:=0;I2<N;I2++)
        begin
                getvalue(A,[I2,N],Ain),
                setvalue(X,I2,Ain)
        end,

        /*Ans=solution*/
        xwriteln(0,"Ans=",X);
?-main();
```

$X[I] = A[I][N]$ is replaced by two special predicates: getvalue() and setvalue(). They are discussed in Section 3.4.2.

In the same way, the extProlog rule representing the subroutine mypivot() can be obtained as follows:

```
mypivot(A,N,I):-
   getvalue(A,[I,I],Aii),
   for(J:=I;J<=N;J++)
   begin
            getvalue(A,[I,J],Aij),
            setvalue(A,[I,J],Aij/Aii)
   end,
   for(K:=0;K<N;K++)
```

```
begin
            if K<>I then
                  getvalue(A,[K,I],Aki),
                  for(J2:=I;J2<=N;J2++)
                  begin
                    getvalue(A,[K,J2],Akj2),
                    getvalue(A,[I,J2],Aij2),
                    setvalue(A,[K,J2],Akj2-Aij2*Aki)
                  end
            end
end;
```

Combining the two rules main() and mypivot() gives a complete extProlog program for the solution of a set of simultaneous linear equations. It should be noted that since the size of the problem is evaluated by the line $N := \text{strlen}(A)$, the program can solve a system of simultaneous linear equations of any dimension; that is, the program need not be modified at all even if the size of the problem changes. This is, of course, more than a simple translation of the C program.

The above discussion demonstrated how a Prolog program can be directly translated from a C program. Although the generated program is workable, it may not be optimally efficient. Using the system-defined functions and predicates provided by the extension, the program can be made more efficient [Takahara et al. 2003].

A numerical program in C basically consists of assignment statements, conditional statements (if clauses etc.), and repeat statements (for clauses etc.). Since these statements are allowed in extProlog (refer to Section 3.4), any routine (main routine or subroutine) of a numerical C program can be translated almost directly as a single extProlog rule, as demonstrated above.

3.2.2 Symbolic Example

Prolog and extProlog are strong in terms of symbolic programming. It is important to observe that although a C program can be translated into an extProlog program easily, the converse is not true for a symbolic program. This is a significant advantage over C for systems development.

The following is a trivial example of a symbolic processing program in Prolog:

```
/*socrates.p*/
die(X):-animal(X);
die(X):-man(X);
man(plato);
man(socrates);
?-die(X),xwriteln(0,X,"will die");
```

The program represents the following logical relations:

If X is an animal, X will die.
If X is a man, X will die.
plato is a man.
socrates is a man.

The program infers who will die from the above relations.

In extProlog, predicates are categorized as belonging to either the self-executable predicate class or the rule head class. All of the system-defined predicates and predicates for numerical computation belong to the former class. For example, the system-defined predicate xwriteln() in the above program is a self-executable predicate that displays a text specified by its argument in a window when executed. The predicates setvalue() and getvalue() in the numerical example in Section 3.2.1 are also self-executable predicates.

On the other hand, all of the user-defined predicates belong to the rule head class. The predicate die() in the above program is a member of the rule head class. A rule is considered as a subroutine definition in this chapter. Execution of a predicate in the rule head class implies execution of the body (tail) of the rule corresponding to the predicate. The biggest difference between a subroutine in C and that in Prolog is that more than one subroutine can be defined for one rule head predicate.

Consider die(X) in the above example. This is a rule head predicate for which two subroutines, die(X):-animal(X) and die(X):-man(X), are defined. Although the tail specifies a logical condition and the head represents a logical conclusion in the conventional interpretation of Prolog, this chapter interprets them in a different way; the head is considered to represent a procedure head and the tail represents its body, as mentioned above.

If there is more than one subroutine definition for a rule head predicate, the first subroutine is usually executed first, and the remaining subroutines are stored in a special stack to be executed later if execution of the first fails. This mechanism is called backtracking, and is an important feature of Prolog. If a rule is selected by a rule head predicate, the selection is called matched, and unification succeeds.

The body of a subroutine in Prolog, that is, the tail of a rule, can be considered to be composed of a series of subroutine calls. For example, the subroutine die(X):-animal(X) consists of one subroutine call, animal(X). Execution of die(X) is called successful if the subroutine call of animal(X) is successfully completed.

The Prolog interpreter, an engine to execute a Prolog program, saves the necessary subroutine calls as a list in its push-down stack and executes subroutine calls from the list. It is usual that execution of one subroutine call triggers other subroutine calls, which are appended to the subroutine call list to yield a new list, and each call is executed in a last-in, first-out manner.

The initial subroutine call list is given by a special rule of the following form:

$$?\text{-}p_1(), \ldots, p_n();$$

where $(p_1(), \ldots, p_n())$ is the initial list in the push-down stack. The above rule is called a query clause. Execution is carried out in this sequence. For example, the initial list of the above example of socrates.p is (die(X), xwriteln(0, X, "will die")). As such, matching of die(X) is attempted first. Since matching with the first rule, die(X):-animal(X), triggers another subroutine call, animal(X), the subroutine call list is updated as (animal(X), xwriteln(0, X, "will die")). That is, die(X) is replaced with animal(X).

The next call is the predicate animal(X). Since no subroutine is prepared for animal(X) or no subroutine head cannot be matched to animal(X), the subroutine call fails and due to the backtracking operation, the second rule, die(X):-man(X), is attempted. Consequently, the subroutine call man(X) is induced, and the list is updated as (man(X), xwriteln(0, X, "will die")).

The subroutine call man(X) is matched to man(plato), yielding X = plato. Since the selected subroutine does not have a tail and the matching does not require the execution of any more tasks for the subroutine, the call list becomes (xwriteln(0, X, "will die")). When a rule does not have a tail, it is called a fact. The rule man(plato) is a fact.

Since xwriteln() is a self-executable predicate, and since X = plato, execution of the predicate displays the text "plato will die" in a window (window ID = 0). Consequently, the list becomes empty and the program execution ends.

It is important to note that a Prolog variable is local, with meaning valid only in the rule in which it is defined. In other words, the same variable symbol can be used in different rules with a different meaning. This is discussed in detail in Chapter 18.

The real advantage provided by extProlog originates from its capacity for symbol processing. To illustrate this fact, let us consider a maze search program, which is again a trivial program but more meaningful than the "socrates.p" program presented above. Figure 3.1 illustrates the maze problem.

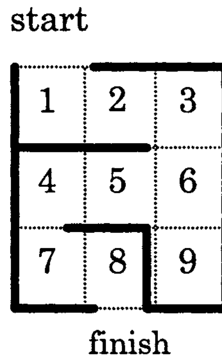


Fig. 3.1. Maze problem.

A program for solving the problem is given below [Convington, Nute and Vellino, 1997].

```
/*mazeProlog4.p*/
maze():-
    mypath([start],Solution),
    xwriteln(0,"solution=",Solution);
mypath([finish|RestOfPath],[finish|RestOfPath]):-!;
mypath([CurrentLocation|RestOfPath],Solution):-
    connected_to(CurrentLocation,NextLocation),
```

```
        not(member(NextLocation,RestOfPath)),
        mypath([NextLocation,CurrentLocation|RestOfPath],Solution);
    connected_to(Location1,Location2):-
        connect(Location1,Location2);
    connected_to(Location1,Location2):-
        connect(Location2,Location1);
    ?-maze();
    connect(start,1);
    connect(1,2);
    connect(2,3);
    connect(3,6);
    connect(4,5);
    connect(6,9);
    connect(5,6);
    connect(4,7);
    connect(7,8);
    connect(8,finish);
```

The program is constructed using a typical strategy in Prolog, that is, it makes extensive use of the backtrack function. The principal subroutine is mypath(). The list [CurrentLocation|RestOfPath] of mypath() represents a path (list of location IDs) from the start location to CurrentLocation, where CurrentLocation is the first element of the list and RestOfPath is the remaining sublist. The subroutine mypath() extends the path to one that connects the start location to the finish location. The extension is made by a recursive call. The subroutine first finds NextLocation, which is connected to CurrentLocation, using the data set connect(). The predicate not(member(NextLocation, RestOfPath)) requires that NextLocation not be in RestOfPath or a path must not form a loop. This requirement can be satisfied by the backtracking function. If a NextLocation is found, the necessary path is obtained by extension of the extended path [NextLocation, CurrentLocation|ResrOfPath]. The extension operation is performed by calling mypath() recursively. The operation ends when the extension reaches the finish location.

The subroutine call mypath([start], Solution) states that a solution is an extension of the initial path "[start]."

Such a maze-solving program would be much more involved in C. The simplicity of the Prolog (extProlog) program as illustrated above originates from the following features:

(i) A Prolog variable is typeless, making it unnecessary to specify data types before writing a program, as is required in C.
(ii) Prolog has a backtracking function, which is very useful for problem-solving. In C, such a backtracking function needs to be coded, significantly increasing the complexity and length of the program in C.
(iii) In general, complicated data structures (for example, tree structures with substructures on nodes) can be conveniently represented as a simple list structure in Prolog, and the structure can be dynamically constructed and modified on execution. This convention cannot be used in C.

Consequently, as a problem becomes more complicated, programs written in C become much longer than the corresponding solution written in extProlog.

3.3 Basic Syntax

This section will introduce the basic syntax of extProlog. Since the complete syntax of extProlog is more complicated than that of standard Prolog due to the extension, the basic syntax, which is sufficient for understanding essential parts of subsequent discussions in this book, is introduced.

The syntax structure of Prolog is given below. Its description employs the following notation:

```
program body=progbody
rule=clause
predicate=literal
query clause=qclause
```

It should be clear that predsym, conssym, and varsym indicate predicate symbol, constant symbol, and variable symbol, respectively.

Basic Syntax of extProlog [Maier and Warren, 1988]:

```
<program>::=<progbody><qclause>|#include"<conssym>.p";
            <progbody><qclause>
<progbody>::= ε|<clause><progbody>
<clause>::=<literal><tail>;
<qclause>::=?-<litlist>;
<literal>::=<predsym>(<arglist>)
<arglist>::= ε|<term><argtail>
<argtail>::= ε|,<arglist>
<term>::=<conssym>|<varsym>
<tail>::= ε|:-<litlist>
<litlist>::=<literal><littail>
<littail>::= ε|,<litlist>
```

where

```
predsym = lc(lc+uc+digit)*
<conssym> = lc(lc+uc+digit)* |digit*
<varsym> = uc(lc+uc+digit)*
lc = any lowercase letter
uc = any uppercase letter
digit = any digit
```

The first line of the syntax states that a Prolog program consists of a progbody and a qclause, or #include "⟨conssym⟩.p," a progbody and a qclause. The symbol "|" denotes "or." The other lines can be understood in the same way. For example, the second line states that a progbody can be empty (ε) or consist of a clause and a progbody.

Figure 3.2 shows how the program body of "socrates.p" can be derived from the syntax specification:

```
<progbody>
→<clause><progbody>
→<literal><tail>;<progbody>
→<predsym>(<arglist>)<tail>;<progbody>
→die(<arglist>)<tail>;<progbody>
→die(<term><argtail>)<tail>;<progbody>
```

```
→die(<varsym><argtail>)<tail>;<progbody>
→die(X<argtail>)<tail>;<progbody>
→die(X)<tail>;<progbody>
→die(X) :-<litlist>;<progbody>
→die(X) :-<literal><littail>;<progbody>
→die(X) :-<predsym>(<arglist>)<littail>;<progbody>
→die(X) :-man (<arglist>)<littail>;<progbody>
→die(X) :-man (<term> <argtail>)<littail>;<progbody>
→die(X) :-man (<varsym><argtail>)<littail>;<progbody>
→die(X) :-man (X<argtail>)<littail>;<progbody>
→die(X) :-man (X)<littail>;<progbody>
→die(X) :-man (X);<progbody>
→......
→die(X) :-animal (X);die(X):-man(X);<progbody>
→die(X) :-animal (X);die(X):-man(X);<clause><progbody>
→die(X) :-animal (X);die(X):-man(X);<literal><tail>;<progbody>
→die(X) :-animal (X);die(X):-man(X);<predsym>(<arglist>)<tail>;<progbody>
→die(X) :-animal (X);die(X):-man(X);man (<arglist>)<tail>;<progbody>
→die(X) :-animal (X);die(X):-man(X);man (<term><argtail>)<tail>;<progbody>
→die(X) :-animal (X);die(X):-man(X);man (<conssym><argtail>)<tail>;
       <progbody>
→die(X) :-animal (X);die(X):-man(X);man (plato<argtail>)<tail>;<progbody>
→die(X) :-animal (X);die(X):-man(X);man (plato)<tail>;<progbody>
→die(X) :-animal (X);die(X):-man(X);man (plato);<progbody>
→......
→die(X) :-animal (X);die(X):-man(X);man (plato);man(socrates);<progbody>
→die(X) :-animal (X);die(X):-man(X);man (plato);man(socrates);
```

Fig. 3.2. Derivation of socrates.p.

3.4 extProlog as General-Purpose Programming Language

Prolog is fundamentally a logic programming language, and is usually not considered to be a general-purpose language. However, Prolog is extended here as extProlog to function as a general-purpose language suitable for the model theory approach. Section 3.2 demonstrated this general-purpose functionality of extProlog.

3.4.1 Execution Control in extProlog

Program structure control is facilitated by the following clauses (formulas) in extProlog:

(1) *if* clause

An *if* clause in extProlog is written as follows:

```
if b(X) then
        P1
else
        P2
end;
```

Here b(X) is a formula, and P1 and P2 are litlists. An *if* clause is used in "mypivot(A, N, L)" of Section 3.2.1.

(2) *switch* clause

Suppose C is a variable that represents a predicate a(Y) or b(Y). A *switch* clause in extProlog is then written as follows:

```
switch(case(C), X), /*X=case.C(Y)*/
   X;
  case.a(Y):-
           P1;
  case.b(Y):-
           P2;
```

If $C = a(Y)$, then $X =$ case.a(Y). Consequently, X induces execution of the rule case.a(Y):- P1.

(3) *for* clause

A *for* clause in extProlog is written as follows:

```
for (b(X)) begin
        P1
end;
```

A *for* clause is used in "simuleq10.p" of Section 3.2.1.

(4) *do-while* clause

This case is examined using a specific example: computation of the sum $X = 0 + 1 + \cdots + 99$.

The corresponding clause in extProlog is as written below:

```
sum(X):-
 assign(regI,0),
 assign(regSum,0),
 repeat,
        assign(regSum,getvalue(regSum)+getvalue(regI)),
        inc(regI,1),
        getvalue(regI)>=100,!,
 regSum(X);
```

where inc() is a system-defined predicate introduced to enhance the execution speed of extProlog.

3.4.2 Operations on Sets and Lists

This section will discuss extensions related to set manipulation [Takahara et al. 2003]. A set is manipulated in extProlog as it is in computer-acceptable set theory.

a. Representation of Sets, Relations, and Functions

(i) Sets

A set is represented as a list:

$$\text{set} = [a_1, \ldots, a_n],$$

where elements a_i represent atomic elements of Prolog: an integer, real, constant, string, or a more complicated element. Hence, naturally, any set can be represented

in extProlog. In particular, if a_i ($i = 1, \ldots, n$) is a list, the expression of a set is a representation of a matrix. For example, [[1, 2, 3], [4, 5, 6]] is an expression of a matrix, where [1, 2, 3] and [4, 5, 6] are row vectors. An empty set is represented by [] or nil.

(ii) Relations

A relation is simply represented as a predicate.

(iii) Functions

A function is represented as a special relation. For sets X and Y, a function is a binary relation $f \subset X \times Y$ that must satisfy the condition

$$(\forall (x, y), (x, y'))((x, y), (x, y') \in f \rightarrow y = y').$$

While standard Prolog cannot handle functions of the form $Y := f(X)$, extProlog allows such expressions, as discussed in Section 3.1. In order to use the function form of an expression for a predicate, a function declaration must be made for it. Consider the following example:

```
q():-
        function([nextV, prevV]),
        X:=6,
        XX:=10,
        Z2:=2+prevV(2+nextV(X,XX),XX),
        xwriteln(0,"Z2=",Z2);
nextV(X,XX,Y):-
        Y:=X+2+XX;
prevV(X,XX,Y):-
        Y:=X-2+XX;
?-q();
```

Here, function([nextV, prevV]) declares that the predicates nextV() and prevV() will be called as functions with value given by the last variable, while the rest of the parameters specify the domain. In fact, func() in computer-acceptable set theory is translated into function().

b. Construction of Sets

A set is constructed by specifying elements of a list directly or by extension of a predicate of Prolog. In extProlog, the latter construction is supported by the function defSet(). Consider the following program, which constructs the set $Ys = \{y | y = x * 5, y < 16, x \in \{1, 2, 3, 4, 3, 2, 1\}\} = \{5, 10, 15\}$. (If $\{1, 2, 3, 4, 3, 2, 1\}$ is a set, it should be just $\{1, 2, 3, 4\}$ because it does not have duplicated elements.)

```
/*testdefSet.p*/
q():-
            Xs:=[1,2,3,4,3,2,1],
            Ys:=defSet(consY(Y,X,[0]),[X,Xs]),
            xwriteln(0,"Ys=",Ys);
consY(Y,X,P):-
            Y:=X*5,
            Y<16;
?-q();
```

It should be noted that defSet() is used in the same way as it is in computer-acceptable set theory.

c. Operation on Sets

The following standard operations on sets are supported by the predicates member, subset, union, intersection, minus, and &, in extProlog:

$$X \in Xs \Leftrightarrow \text{member}(X, Xs) = \text{true};$$
$$X \subset Xs \Leftrightarrow \text{subset}(X, Xs) = \text{true};$$
$$Zs = Xs \cup Ys \Leftrightarrow Zs := \text{union}(Xs, Ys);$$
$$Zs = Xs \cap Ys \Leftrightarrow Zs := \text{intersection}(Xs, Ys);$$
$$Zs = Xs - Ys \Leftrightarrow Zs := \text{minus}(Xs, Ys);$$
$$Zs := Xs \times Ys \Leftrightarrow Zs := \&(Xs, Ys).$$

Since a set is represented as a list, it can be more flexibly manipulated in extProlog using the order property of a list than in standard set theory.

d. Operations on Lists (Functions)

The following are system-defined predicates provided to facilitate operations on lists:

(i) Unary and Binary Operations on Lists

Most unary and binary arithmetic operations are extended as a termwise operation applicable to lists. For example,

$$\log([a1, \ldots, an]) = [\log(a1), \ldots, \log(an)],$$
$$[a1, \ldots, an] > b \Leftrightarrow a1 > b, \ldots, an > b.$$

The operations $+$, $-$, $*$, $/$, $<$, $=$, $>$, $>=$, $<=$ are extended in the same manner. A scalar b can be dealt with as the list $[b, \ldots, b]$ in a list operation. The following demonstrates the extension: Let $X := [1, 2, 3]$ and $Y := [4, 5, 6]$. Then,

$$Z := X/Y \Leftrightarrow Z = [1/4, 2/5, 3/6];$$
$$Z2 := X/7 \Leftrightarrow Z2 = [1/7, 2/7, 3/7];$$
$$Z3 := 8/Y \Leftrightarrow Z3 = [8/4, 8/5, 8/6];$$
$$Z4 := X * Y \Leftrightarrow Z4 = [1 * 4, 2 * 5, 3 * 6].$$

The "*" operation has some exceptions when used for matrices and vectors, as explained below.

(ii) Special Functions and Predicates for Lists

Many special functions and predicates are defined for lists. Suppose $Xs = [a1, \ldots, an]$ and $Ys = [b1, \ldots, bn]$. Then, the following are typical functions:

$$\text{ln}(Xs) = \text{strlen}(Xs) = n(\text{number of elements}),$$
$$\text{sum}(Xs) = a1 + \cdots + an,$$

$$\min(Xs) = \text{the smallest element of } Xs,$$

$$\max(Xs) = \text{the largest element of } Xs,$$

$Z := \text{distinct}(Xs) \Leftrightarrow Z$ is the sublist of Xs where duplicated elements are deleted.
Let $Xs = [1, 2, 3, 4, 3, 2, 1]$. Then,

$$Z := \text{distinct}(Xs) \Leftrightarrow Z = [1, 2, 3, 4].$$

It is an important feature of extProlog that functions are usually defined recursively with respect to the list structure. For example,

$$\text{sum}([[a1, \ldots, an], [b1, \ldots, bm]]) = [\text{sum}([a1, \ldots, an]), \text{sum}([b1, \ldots, bm])].$$

Basic manipulations of a list are performed by the predicates project, append, select, replace and a function, replaceList. [Takahara et al. 2003]. The following are typical examples:

$$\text{project}([a1, a2, a3], 2, X) \leftrightarrow X = a2,$$

$$\text{project}([a1, a2, a3], X, a2) \leftrightarrow X = 2.$$

$$\text{append}([a1, a2, a3], [b1, b2], X) \leftrightarrow X = [a1, a2, a3, b1, b2],$$

$$\text{select}(a2, [a1, a2, a3], X) \leftrightarrow X = [a1, a3].$$

An element of a list is replaced by "replace" and by "replaceList":

$$\text{replace}([a1, a2, a3], 2, b2, X) \leftrightarrow X = [a1, b2, a3],$$

$$X := \text{replaceList}([a1, a2, a3], 2, b2) \leftrightarrow X = [a1, b2, a3].$$

(iii) Matrix Manipulation
The product operation "*" on matrices and vectors follows the conventional rules, as illustrated by the following examples. Let

$$A = [[1, 2], [3, 4]],$$

$$B = [[5, 6], [7, 8]],$$

$$X = [2, 3].$$

Then,

$$A * B = [[19, 22], [43, 50]],$$

$$A * X = [8, 18],$$

$$X * A = [11, 16],$$

$$\text{transpose}(A) = [[1, 3], [2, 4]].$$

The inner product given as follows:

$$\text{sum}([1, 2] * [3, 4]) = \text{sum}([1 * 3, 2 * 4]) = 1 * 3 + 2 * 4.$$

An element of a matrix is manipulated by two predicates, getvalue() and setvalue(). Suppose A is a matrix. Then, the (I,J)th element of A is obtained by

```
getvalue(A, [I,J],X).
```

The Ith row vector is obtained by

```
getvalue(A,I,X).
```

The (I,J)th element is replaced with

```
setvalue(A, [I,J],V).
```

Similarly, the Ith row vector is replaced with

```
setvalue(A,I,V).
```

If A is a vector, the Ith element is manipulated by

```
getvalue(A,I,X),
setvalue(A,I,V).
```

For example, let $A = [[1, 2, 3], [a, b, c]]$. Then, the second row vector of the matrix can be obtained by getvalue(A, 1, V) where $V = [a, b, c]$.
(iv) Construction of Lists
There are cases in which we want to have a list rather than a set. There is a function defList() for list operations, which corresponds to defSet() for set operations. Suppose a datum Xs is given that shows evaluations of the items "a" and "b":

$$Xs = [[a, 2], [b, 4], [a, 2], [b, 5], [a, 3], [b, 5]].$$

The evaluation of "a" is given by the list [2, 2, 3]. If defSet is applied to get the list, [2, 3] is generated rather than [2, 2, 3]. The function defList is used to get [2, 2, 3]. The following is a program to generate [2, 2, 3]:

```
/*testdefList.p*/
q():-
        Xs:=[[a,2],[b,4],[a,2],[b,5],[a,3],[b,5]],
        Ys:=defList(consY(Y,X,[a]),[X,Xs]),
        xwriteln(0,"Ys=",Ys);
consY(Y,[a,E],[a]):-
        Y:=E;
?-q();
```

The program generates Ys=[2, 2, 3].

3.4.3 Global Variables

A Prolog variable is in principle a local variable in a program. However, there is an occasion where a global variable must be used. A global variable is supported by the system-defined predicate assert() in standard Prolog. In extProlog, its modified version assign() is also used, with the following syntax:

$$\text{assign}([\langle\text{predname}\rangle, \text{conssym}_1, \ldots, \text{conssym}_n], \langle\text{parameter list}\rangle)$$

If the above predicate is executed, a new fact $\langle\text{predname}\rangle(\text{conssym}_1, \ldots, \text{conssym}_n, \langle\text{parameter list}\rangle)$ is created if a fact $\langle\text{predname}\rangle(\text{conssym}_1, \ldots, \text{conssym}_n, X)$ does not exist, where X is any parameter list. If such a fact exists, X is replaced with $\langle$parameter list$\rangle$. The difference between assert() and assign() is that assert() always creates a new rule or a new fact whenever it is executed, whereas assign()

does not necessarily create a fact but modifies an existing one. For example, execution of assign([goal, 2], V) creates a fact goal(2, V) if goal(2, X) does not exist, and if goal(2, X) exists, it is modified to goal(2, V) when assign([goal, 2], V) is executed.

Chapter 2 introduced two global variables, *.lib and *.g, for computer-acceptable set theory. The global variable *.g is implemented by the predicate assign().

3.5 Graphical User Interface in extProlog

A graphical user interface (GUI) is an indispensable component of an information system. Every application discussed in Chapters 6 to 16 has a GUI. A GUI not only presents results in an understandable way to an end user, but also assists in understanding the target problem and helps one to improve the solving system. A GUI is also crucial for data input in transaction processing.

The model theory approach supports GUI construction in two ways: by providing extProlog predicates to build basic GUIs and by allowing for the use of a popular browser with PHP. The first way is mainly used for problem-solving, whereas the second is for transaction processing in the current book. Since a predicate of extProlog can be directly used in computer-acceptable set theory, the first way provides a systems developer with every kind of basic GUI function in the simplest way: static or dynamic, character-oriented or graphical. Appendix 3.1 presents GUIs in extProlog [Takahara, Y. et al. 2003].

3.6 Execution Speed of extProlog

Since extProlog is an interpreter language, program execution speed is much slower for extProlog than for compiler languages. This may be the only major problem with the use of extProlog. To face this problem, basic computations are implemented as C subroutines in extProlog, which can be called as Prolog predicates. The practicability of extProlog with respect to execution speed can be evaluated, for example, by examining a simplex tableau algorithm of linear programming (LP):

Consider an LP problem given by

$$c^T x \to \max$$
$$Ax \leqq b, x \geqq 0.$$

The program for solving this problem is about 275 lines in length, and the response time is measured with respect to n, the dimension of the matrix A.

Tolerable response times can be defined in many ways. One definition is that execution time of less than 1 s is desirable, 1–3 s is acceptable, and longer than 3 s is undesirable. According to this definition, suppose a tolerable response time is 3 s. The result shows that the current implementation of extProlog can solve an LP problem of size 30×30, which requires a data size of about 1000, in much less than 3 s. Although $n = 30$ would not encompass the entire domain of problems, it represents reasonably good coverage of small-scale problems. Thus, execution speed is not considered to be a serious issue, considering the rapid progress in computer power.

Appendix 3.1 Graphical User Interface in extProlog

a. Tree Display in a Spreadsheet

One of the most important applications of a spreadsheet as a GUI for a problem-solving system is a tree display. A tree display is used in Chapter 12. The tree display is implemented by the higher-level predicate [Takahara, Y. et al. 2003]

drawtree(Title, D, Wp),

where Title is a name assigned to the spreadsheet window, and Wp is the window pointer. Every window is given an identifying number, called a window pointer. Finally, D is a data element that takes the following data structure:

$$D = [[\text{node}_1,\ \text{branch}_1,\ \text{nnode}_1], \ldots, [\text{node}_n,\ \text{branch}_n,\ \text{nnode}_n]],$$

where a tree is represented as in Fig. 3.3.

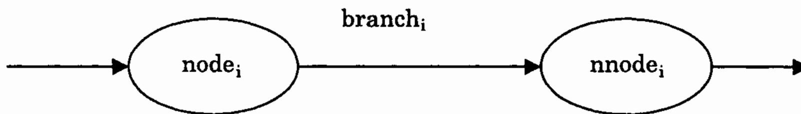


Fig. 3.3. Tree structure.

In Fig. 3.3, nnode$_i$ is a child node of node$_i$ and the nodes are linked by a branch denoted by branch$_i$. Suppose D is given by

```
D:=[
[node0,reduced,nil],
[node0,normal,node1],
[node1,no,node2],
[node1,yes,node3],
[node2,young,nil],
[node2,pre_presbyopic,nil],
[node2,presbyopic,node4],
[node3,hypermotrope,node5],
[node3,myope,nil],
[node4,myope,nil],
[node4,hypermetrope,nil],
[node5,young,nil],
[node5,pre_presbyopic,nil],
[node5,presbyopic,nil]].
```

Figure 3.4 shows the representation of D on a spreadsheet graph with window pointer of 3.

It should be noted that the ith branch datum branch$_i$ is displayed with the child node nnode$_i$ on the spreadsheet. For example, [node1, no, node2] in D creates node1 and node2 at the cells $[B, 2]$ and $[C, 3]$, respectively, and the branch datum "no" is displayed in the cell $[C, 3]$ with "node2".

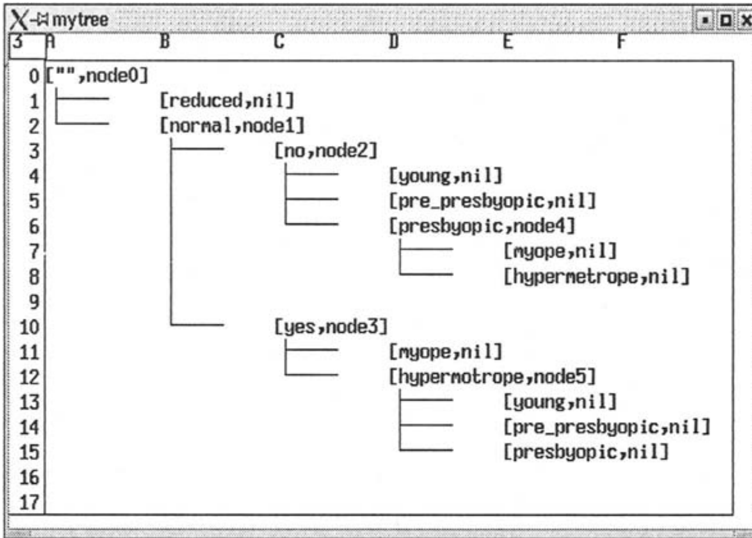


Fig. 3.4. GUI of a tree.

b. GUI on Graph Window

A graph—plot graph, bar graph, pie graph, radar graph, xyplot graph, or trajectory graph—is opened and displayed by the higher-level predicate

$$\text{show1}(D, \langle\text{graph type}\rangle),$$

where show1() is a subroutine (Prolog rule) [Takahara, Y. et al. 2003]. The following program uses show1() to display two graphs:

```
/*testgraph.p*/

q():-

    Data:=[[1,2,3],[5,3,1]],

    Data2:=[[1,2,3],[1,3,5]],

    show1(Data,plot),

    show1([Data,Data2],trajectory,"multiple");

?-q();
```

The data are given by Data and Data2.

Figure 3.5 shows a plot graph with data given by the data structure $D = [\langle\text{list of data}\rangle_1, \ldots, \langle\text{list of data}\rangle_n]$ where each map is specified by $\langle\text{list of data}\rangle_i$.

A multiple-trajectory graph is generated by the following predicate:

show1(Data, trajectory, "multiple").

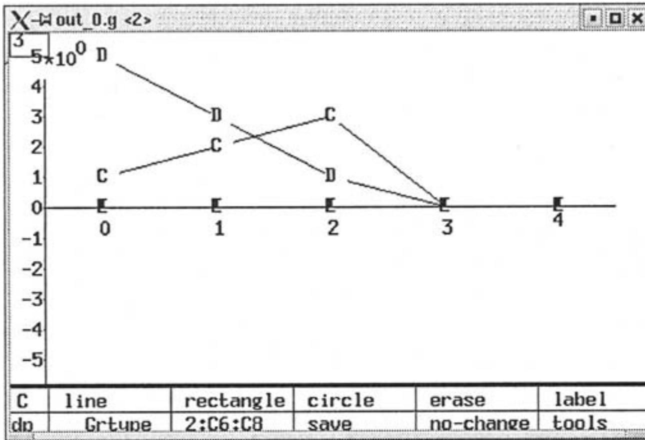


Fig. 3.5. Plot graph.

The data structure is given by

$$\text{Data} = [\langle\text{trajectory data}\rangle_1, \ldots, \langle\text{trajectory data}\rangle_m].$$

Using these, multiple-trajectories are displayed in a window, where each trajectory is specified by ⟨trajectory data⟩. Figure 3.6 shows an example of a multiple-trajectory graph. Chapters 7 and 8 illustrate more meaningful multiple-trajectory graphs.

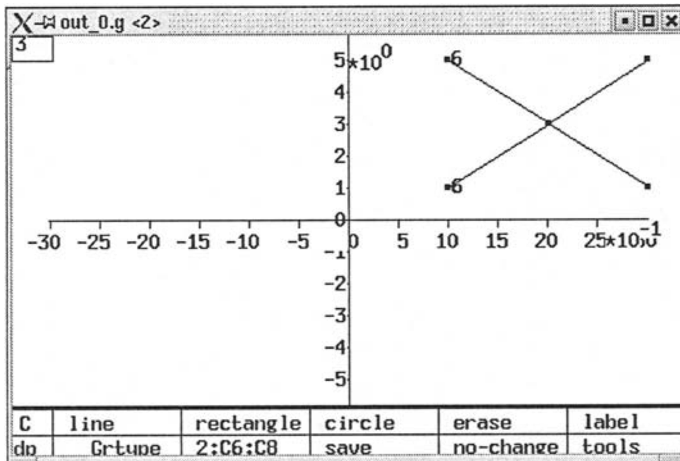


Fig. 3.6. Multiple-trajectory graph.

References

Maier, D. and Warren, D. S. (1988) *Computing with Logic*, Benjamin.

Convington, M., Nute, D. and Vellino, A. (1997) *Prolog Programming in Depth*, Prentice Hall.

Takahara, et al. (2003) *Study on 4-th Generation Systems Development Methodology for MIS*, Internal Report, Dept. of Management Information Science, Chiba Institute of Technology, Chiba, Japan.

Takahara, Y. et al. (2003) *User's Manual for the 4-th Generation Systems Development Methodology*, internal report, Dept. of Management Information Science, Chiba Institute of Technology, Chiba, Japan.

Part III

Model Theory Approach to Solver Systems Development

4

Model Theory Approach to Solver System Development: Outlines

This chapter outlines the process of problem-solving system development by the model theory approach as a preliminary for Chapter 5. Specifically, development of extSLV is the target. A problem classification system is introduced for development. Throughout this book, extSLV development is carried out based on this classification scheme.

If the reader wishes to get a sketchy idea of the model theory approach to problem-solving system development, he is recommended to first read Chapter 9.

4.1 Model Theory Approach to extSLV Development

Before presenting the model theory approach to extended solver (extSLV) development, let us first examine how general systems theory (GST) contributes to information system development. Mesarovic and Takahara [1974] proposed a general scheme in which GST can be used for systems practices, as outlined in Fig. 4.1.

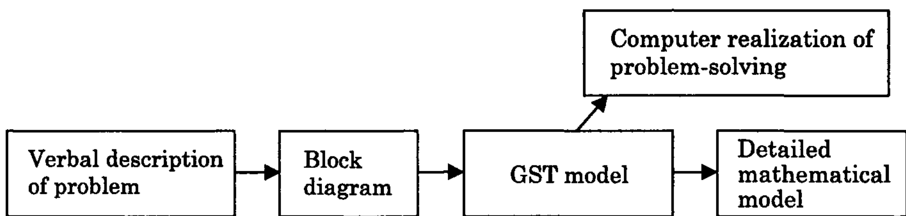


Fig. 4.1. Basic strategy of general systems theory approach.

It is usual that system analysis and synthesis starts from a verbal description of a system and its block diagram representation. However, there is a large gap between a block diagram description and a detailed mathematical model. In the model above, a GST model can be used to fill this gap, since a GST model is expected to explicitly describe the structural properties of the target system. Since it is unwise to seek

a solution for a system problem before understanding its structure, the GST description can be helpful for generating a meaningful mathematical model and assist with manipulation.

The same is expected to hold for the relationship between a block diagram description of a problem and information system development. It should be noted that a computer program is a type of detailed formal model. A GST model is required as a necessary mediator. In this book, the model theory approach provides the required GST model. The approach, in fact, plays a bigger role in information system development than GST for the case of detailed mathematical modeling, because the implementation phase (realization of an executable system) is also addressed as an integral part of the model theory approach. Its model is more than a mediator.

Figure 4.2 shows the scheme of the model theory approach presented in this book for construction of extSLV.

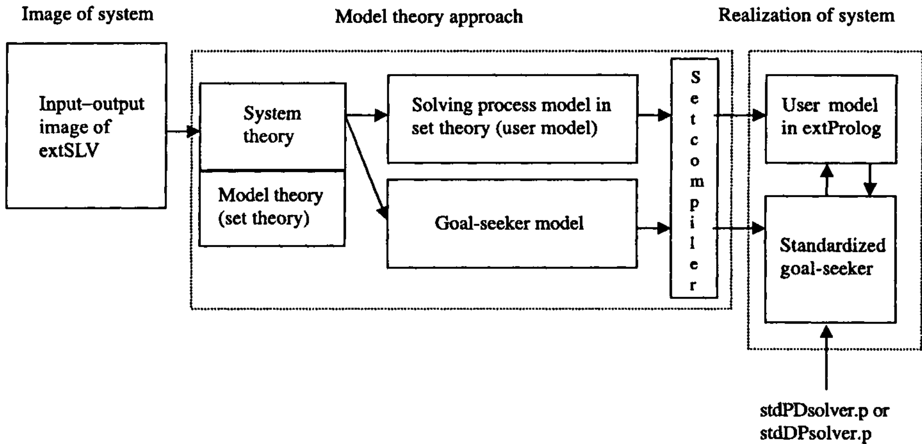


Fig. 4.2. Model theory approach to extSLV.

The input–output image of extSLV corresponds to the block diagram shown in Fig. 4.1, in which the input is a problem specification environment (PSE) and the output is a solution. (See Fig. 1.2.) The GST model shown in Fig. 4.1 is given by a combination of a solving process model in set theory and the goal-seeker model shown in Fig. 4.2. The solving process model is a user model in Section 1.2. It is produced by formalizing the basic understanding of GST that a problem-solving activity is essentially an incremental dynamic process.

The problem-solving process is outlined in Fig. 4.3, where c_0 and C_f denote the initial state and final state set of the problem-solving activity, respectively.

The problem is defined as the gap between c_0 and C_f, and the problem-solving activity is expressed as a dynamic process that takes the initial state to the final state set. The dynamic process is represented by an automaton. The state transition occurs

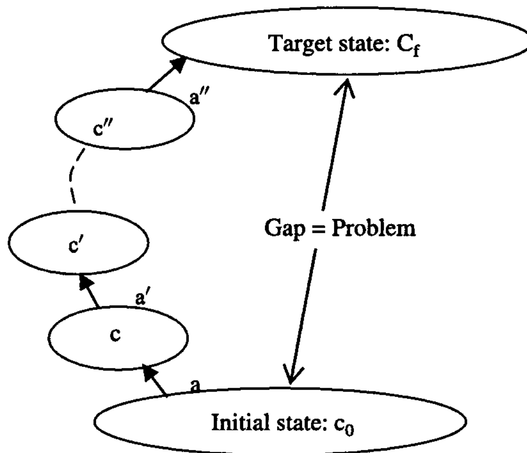


Fig. 4.3. Dynamic problem-solving process.

due to an action "a," and a successful sequence of actions, aa′ ... a″, takes the initial state to the final state.

The problem-solving process shown in Fig. 4.3 can be realized by the goal-seeking model in GST [Mesarovic and Takahara, 1974]. Figure 4.4 shows the goal-seeking model specifically for problem-solving.

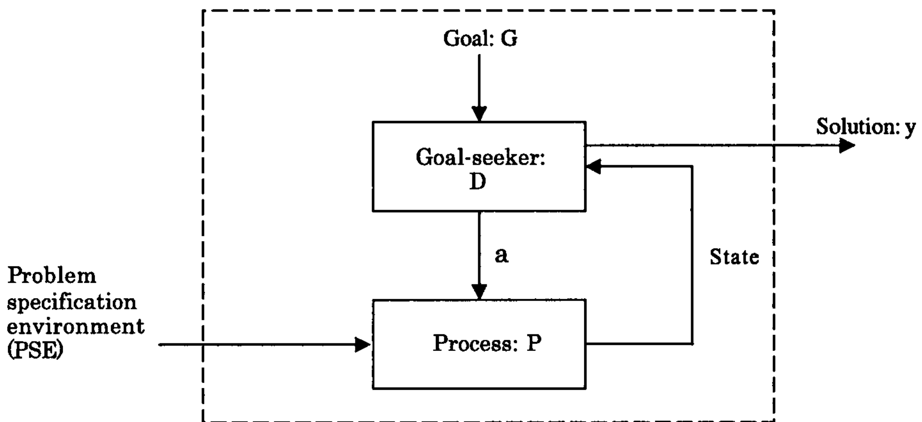


Fig. 4.4. Goal-seeking model for problem-solving.

The goal-seeking model consists of two basic parts: process P and goal-seeker D. The process P is an object that is a target of the goal-seeking activity of D. The process is an input–output system with inputs consisting of two components: the external input

PSE and the input parameter called the decision or manipulating variable "a" specified by D.

The PSE provides conditions to determine the target problem. These conditions are usually represented as a relational structure, which is formally given by

$$\mathbf{PSE} = \langle \text{class of base sets, class of relations, class of functions, class of constants} \rangle.$$

Since the PSE is fixed during the problem-solving activity, P is described as having only one internal input "a." The goal-seeking system is then a closed system that behaves as a generator, that is, as an automaton without an external input. In this respect, the system differs considerably from that for the transaction processing system (see Chapter 14), which is an open system.

The goal-seeker D, using feedback information on the state of the process, selects a suitable value for "a" in reference to the goal G, which represents an evaluation of the desirability of the behavior of P controlled by "a." Although the goal may or may not be given explicitly by the environment, the goal-seeker always requires the existence of a goal in the model theory approach, and the goal-seeker is designed to yield an optimum or satisfactory solution for that goal.

Embedding the problem-solving process (Fig. 4.3) into P (Fig. 4.4) yields the problem-solving system. For the problem-solving process, let

$$A = \text{set of values of the action variable}$$

and

$$C = \text{set of values of the state variable.}$$

The problem-solving process is then given by the following automaton representation:

$$\mathbf{problem_solving_process} = \langle A, C, \delta, c_0, C_f \rangle,$$

where $\delta : C \times A \to C$.

Replacing P (Fig. 4.4) with the problem-solving process gives the problem-solving system shown in Fig. 4.5. This system is called an extended solver (extSLV) in the model theory approach. Although the PSE specifies the structure of the problem to be solved, it is in general not a problem representation. The target problem is therefore formed and solved inside the extSLV, whereas for a regular solver, the problem is specified externally.

The target of the goal-seeker in Fig. 4.5 consists of a process (state transition), a goal, and auxiliary information relevant to the PSE. The combined structure is called a decision problem in GST. In the model theory approach, the decision problem is called a user model for the extSLV.

The goal-seeker specifies a value for the control parameter "a" of the process (automaton) to achieve the specified goal using information from the user model. The output of the goal-seeker is the solution "y."

In the model theory approach, the goal-seeker is designed for the relational structure model of a user model as presented in Section 1.2 such that the goal-seeker is independent of the individual properties of the target problem. The individual properties

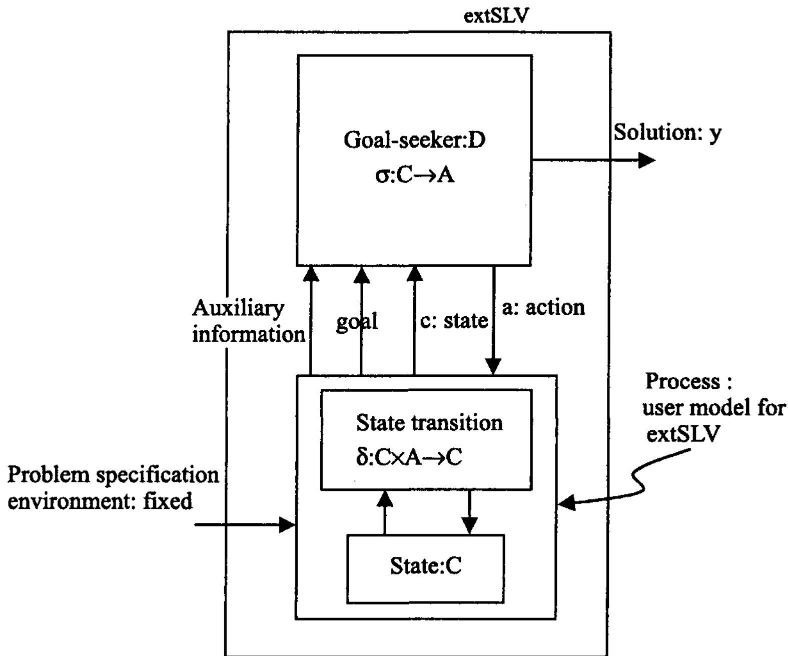


Fig. 4.5. Problem-solving system: extSLV.

are embedded in the instantiation of the user model. This independence was illustrated in the example in Section 1.3, where stdDPsolver62.p was the independent goal-seeker. The goal-seeker is then standardized and provided as a black box component.

The goal-seeker is characterized by a strategy to derive "a." The basic solving strategy is given in two ways: by the hill-climbing method with a push-down (PD) stack, which is called a PD method, or by a modified dynamic programming (DP) method, which is called a DP method. Both are discussed in Chapter 5.

According to Fig. 4.5, the extSLV is described using the structure

$$\textbf{extSLV} = \langle \text{goal-seeker, user-model-structure-of-extSLV} \rangle.$$

The goal-seeker is then represented by

$$\textbf{goal-seeker} = \langle A, C, \sigma \rangle,$$

where $\sigma : C \to A$.

Since a user model is a decision problem in GST [Mesarovic and Takahara, 1989], the relational structure is given as

$$\textbf{user-model-structure-of-extSLV} = \langle A, C, \delta, \lambda, \text{ genA, constraint, goal, st}, c_0, C_f \rangle,$$

where

A: set of actions,
C: set of states,

$\delta : C \times A \rightarrow C$: state transition function,
$\lambda : C \times A \rightarrow A$: output function,
genA: $C \rightarrow \wp(A)$: allowable action specification function,
constraint: $C \rightarrow$ {true, false}: constraint predicate for a state,
goal: $C \rightarrow$ Re: evaluation of a state, where Re is the set of real numbers,
st: $C \rightarrow$ {true, false}: stopping condition,
$c_0 \in C$: initial state,
$C_f \subset C$: set of final states.

The state transition function δ is generally a partial function, and genA() and constraint() are used here to ensure that δ behaves properly. The function genA(), an allowable action specification function, specifies the allowable or preferable actions for a given state (see Chapter 9). The predicate constraint() determines whether a state is permissible. In the knapsack problem, for example, although any jewel can be put into the knapsack, the resultant state may be prohibited due to problem restrictions. The predicate constraint is also an effective means of representing heuristics for a target problem (see Chapter 8). The state transition function δ is naturally assumed to satisfy

$$\delta(c, a) = c' \rightarrow a \in \text{ genA(c) and constraint}(c') = \text{true}.$$

The stopping condition st : $C \rightarrow$ {true, false} is, in many cases, given by

$$\text{st}(c) = \text{true} \leftrightarrow c \in C_f.$$

If an instance of the user model structure is given in set theory for the target problem, a solver system for the problem is automatically created by the setcompiler.

Some comment should be made regarding other approaches that have dealt with optimization problems using set theory and logic [Domenico et al., 2001; Hooker, 2000], and logic programming languages for implementation. Although these methods appear quite similar to the model theory approach, there are a number of important differences. First, the most fundamental difference is that these previous approaches are concerned with the derivation of a solving algorithm for a given specific problem, whereas the model theory approach is first concerned with the structure of a problem-solving system in general, which is formalized as a combination of the standardized goal-seeker and a user model, and then development of a solving structure of a specific problem based on this structural analysis. In short, a problem-solving system is constructed following the metasystem model or the relational structure in the model theory approach.

The second difference is therefore in the classification of problems. In algorithm-oriented theories, problems are usually assumed to be defined on numerical spaces (e.g., Euclidean spaces), and are classified using metric space concepts such as linear and nonlinear problems, and convex problems. The classification of the model theory approach, as introduced in Section 4.3, is concerned with the structure of a problem and not with its expression.

Finally, it is natural that the resultant solving algorithm should differ between the two approaches. Since the present scheme uses an analysis result of the solving system

to develop an algorithm, the resultant algorithm is typically procedural. This clearly distinguishes it from constraint programming, in which the declarative style of programming is emphasized.

4.2 Design Procedure of extSLV

The design procedure derived from the model theory approach in Fig. 4.2 is shown in Fig. 4.6.

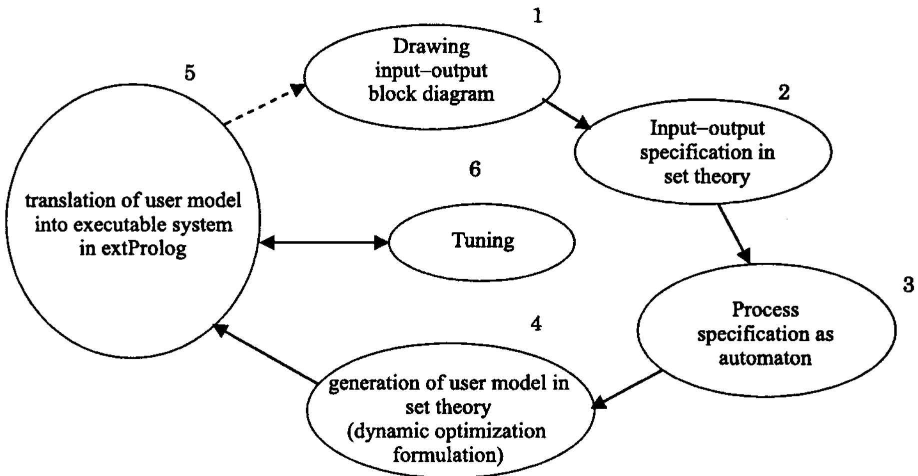


Fig. 4.6. Design procedure of extSLV.

The procedure consists of six stages: drawing input–output block diagram, input–output specification in set theory, process specification as an automaton, generation of user model in set theory (dynamic optimization formulation), translation of the user model into an executable system in extProlog, and tuning.

extSLV is divided into the two independent components of process and goal-seeker in Fig. 4.5. Since the process represents the dynamism of a problem-solving activity as appropriate for the target problem, it must be specified as a user model by a system developer following the user model structure. The minimum tasks required for the developer are then (i) formalizing the problem-solving activity in the user model form, and (ii) translating it into an executable system using the setcompiler. The first four stages performs the former task. The goal-seeker is attached to the user model on translation at the fifth stage.

Although the standardized goal-seekers can perform their function satisfactorily for many problems, they may not be quite efficient for certain problems. In such cases, tuning must be performed for the goal-seeker after a working solver has been constructed.

Chapter 5 will discuss each stage in detail. The most difficult part of user model formulation may be identification of the state of the automaton. It is shown in Chapter 5 that the present methodology provides a basic and general procedure for identifying the states.

4.3 Classification of Problems for extSLV

It is clear that implementation of the design procedure depends on the character of the problem under consideration. In order to make the procedure operational, classification of problems is required.

In general, problems can be categorized into those suitable for a conventional solver algorithm and those that are not. In some fortunate situations, the PSE can be transformed into a problem to which a conventional solver (or solving algorithm) is directly applicable. Figure 4.7 outlines this case, in which the extSLV is decomposed into a problem formulator (PRF) and a conventional solver.

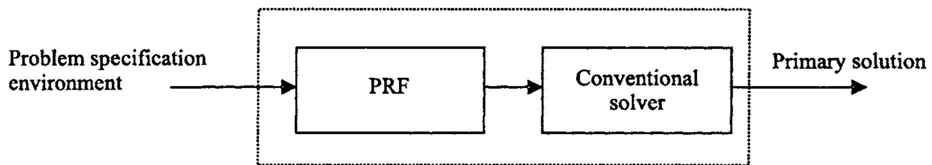


Fig. 4.7. Conventional solver.

A typical example of a conventional solver is the linear programming (LP) algorithm, which requires a problem to be represented in a standardized form. The PRF in Fig. 4.7 transforms the parameters of the PSE into a standard form as required by the solver. In this case, problem identification may be difficult because the problem must be formulated to satisfy the required form. The model theory approach is concerned with the latter case, which is more common in practice. The latter case requires further classification of the problem.

The model theory approach uses the following three dimensions for categorization of problems when a conventional solver is not applicable:

1. Explicit solving action — implicit solving action (E/I)
2. Closed goal — open goal (C/O)
3. Closed target — open target (C/O)

An explicit solving action indicates that an output (solution) of the extSLV is a sequence of actions of the solution process. The dynamic process δ in Fig. 4.5 is then defined by the PSE, and the output in a natural way, as will be shown in Chapter 5. Typically, the traveling salesman problem, which is discussed in Chapter 6, has an explicit solving action, whereas the knapsack problem, which is discussed in Chapter 10, does not. How a solution is derived is not relevant to the solution of the knapsack problem.

A closed-goal problem implies that evaluation of the output is given by the problem specification. The traveling salesman problem is a closed-goal problem because if a traveling route (output) is given, its evaluation measure, the traveling distance, naturally follows.

A closed-target problem is a problem for which the final state (C_f in Fig. 4.3) of the solving process can be directly determined when process δ is specified. In general, the final state is not uniquely specified. The traveling salesman problem can be described as a closed target, while the knapsack problem cannot. If the problem is an open target, the stopping condition of the solving activity cannot be easily determined. It should be noted that even if the goal-seeker reaches a final state C_f, it does not necessarily mean that a solution is obtained. A path to a final state may be a solution only if it satisfies a specific criterion, for instance, to be the shortest.

There are eight possible cases for classification. Table 4.1 lists the classification and examples.

Table 4.1. Problem classification.

Notation	Example
E-C-C	Traveling salesman problem
E-C-O	Linear quadratic dynamic optimization problem
E-O-C	Regulation problem
E-O-O	General control problem
I-C-C	Cube root problem
I-C-O	Knapsack problem
I-O-C	Class scheduling problem
I-O-O	Data mining problem

The traveling salesman problem and the knapsack problem are well known as typical operations research problems. The regulation problem is a classical control engineering problem in which the state of a target system is required to be brought to a set point by a control action as soon as possible. Although the control engineering problem is very different from conventional MIS problems, it will be shown that the problem is handled in the same way as MIS problems. This demonstrates the generality of the model theory approach. The class schedule problem is a problem to develop a class schedule of a school. These problem types are the subjects of Chapters 6–12.

4.4 Implementation Structure of extSLV

An implementation structure represents a complete user model in set theory, which is directly derived from the relational structure user-model-structure-of-extSLV. Once it is constructed and compiled, a workable system is generated. Figure 4.8 shows the

implementation structure of an extSLV, in which the PD method is used as the standardized goal-seeker.

```
func();                            //declaration of function names
delta:CXA→C;                       //state transition
genA:C→℘(A);                       //allowable actions for a given state
constraint:C→{true,false};         //permissible state
initialstate:()→C
    (nullary function);            //initial state
finalstate:C→{true,false};         //final state
goal:C→Re;                         //goal
st:C→{true,false};                 //stopping condition of the solving process
preprocess()→{true,false};         //preprocessing for goal-seeker
postprocess()→{true,false};        //post-processing of the goal-seeker
```

Fig. 4.8. Implementation structure in extProlog using stdPDsolver.

The functions in Fig. 4.8, delta(), genA(), constraint(), initialstate(), and finalstate(), are determined by an automaton representation of the solving process for a given problem specification environment. The function goal() directs the solving process activity, while the predicate st() determines when the solving activity must be stopped. The predicate preprocess() prepares a solving process environment including modification of the PSE. The predicate postprocess() prepares an output form of a solution. They can be omitted.

The implementation structure for an extSLV in which the DP method is used is shown in Fig. 4.9.

```
func();                            // declaration of function names
delta:CXA→C;                       //state transition
invdelta:C→℘(C);                   //inverse state transition
genA:C→℘(A);                       //allowable actions for a given state
constraint:C→{true,false};         //permissible state
initialstate:()→C
    (nullary function);            //initial state
finalstate:C→{true,false};         //final state
goalElement:CXA→Re;                //goal
invst:℘(C)→{true,false};           //stopping condition of the solving process
preprocess()→{true,false};         //preprocessing for goal-seeker
postprocess()→{true,false};        //pos-tprocessing of the goal-seeker
```

Fig. 4.9. Implementation structure in extProlog using stdDPsolver.

The implementation structure also consists of func(), a user model for the DP method, and two predicates, preprocess() and postprocess(). It should be noted that in Fig. 4.9, a user model of the DP method includes an extra function

$$\text{invdelta:C} \rightarrow \wp(\text{C}),$$

and goal() and st() in Fig. 4.8 are replaced by

$$\text{goalElement:C} \times A \rightarrow \text{Re}$$

and

$$\text{invst:}\wp(\text{C}) \rightarrow \{\text{true, false}\},$$

respectively. The example presented in Section 1.3 illustrates the use of these functions.

References

Cantone, D., Omodeo, E., and Policriti, A. (2001) *Set Theory for Computing: From Decision Procedures to Declarative Programming with Sets*, Springer.

Hooker, H. (2000) *Logic-Based Method for Optimization*, John Wiley & Sons.

Mesarovic, M. D. and Takahara, Y. (1974) *General Systems Theory: Mathematical Foundation*, Academic Press.

Mesarovic, M. D. and Takahara, Y. (1989) "Abstract Systems Theory," *Lecture Notes in Control and Information Sciences*, Springer.

Takahara, Y. and Kubota, H. (1989) "Framework for Conceptual Design of Data Processing Systems," *Office Automation*, **10**(1) (in Japanese).

Takahara, Y., Asahi, T., and Hu, J. (2003) "Application of MGST Design Approach to Data Mining Systems: Case of I-O-O Problem," *J. of JASMIN*.

5

User Model and Standardized Goal-Seeker*

This chapter presents detailed explanations of the design stages of Fig. 4.6 and theoretical results from the user models and standardized goal-seekers featured in Fig. 4.5. The theoretical results can be omitted if readers are interested in practical aspects of the model theory approach. There are eight cases of user models according to the classification of Section 4.3. This chapter investigates two extreme cases: the E-C-C case and the I-O-O case. The other cases can be considered as combinations of these two extreme cases.

5.1 User Model for the E-C-C Case

This section investigates the case of E-C-C, explicit solving action, closed-goal, and closed-target states.

5.1.1 Drawing Input–Output Block Diagram

At this initial stage, the problem specification environment (PSE) and its solution (output) or solution candidates of a target extended solver (extSLV) are conceptually described. Figure 5.1 illustrates an input–output block diagram of the traveling salesman problem.

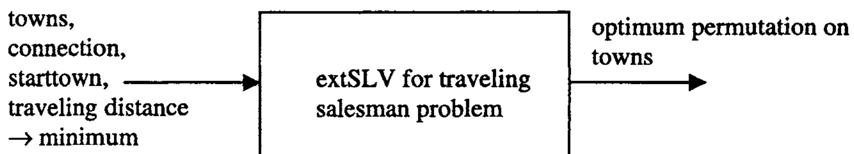


Fig. 5.1. Input–output block diagram of traveling salesman problem.

Figure 5.1 indicates that towns, connecting information among them, and the starting town constitute the input structure. A solution is a traveling sequence of the towns with a minimum traveling distance.

5.1.2 Input–Output Specification in Set Theory

At this stage, the input PSE and the output of the block diagram are represented in set-theoretic terms. It is usual that the PSE is specified by a relational structure [Bridge, 1977] rather than a simple family of sets. It must be noted that the output set of the input–output specification is not a set of solutions; rather it is a set of solution candidates. A real solution of extSLV is selected from the candidates. Then,

Input structure:

$$\langle \text{class of base sets, class of relations, class of functions, class of constants} \rangle.$$

Output set Y:

$$Y \subset A^*,$$

where A is the set of solving actions and A^* is the free monoid of A. The representation $Y \subset A^*$ comes from the assumption of the explicit solving action. An element $a \in A$ usually has a complicated structure, which is determined by the PSE. In the example of Section 1.3, the input structure is $\langle N, M, s, t, G \rangle$ and A is specified by $N \times M$ or a set of pairs of integers. In this case, A does not have a complicated structure.

5.1.3 Process Specification as Automaton

At this stage, the solving activity process δ of Fig. 4.5 is represented as an automaton. Because $Y \subset A^*$ in the current case, a general form of δ can be specified as

$$\delta : Y \times A \to Y$$

such that

$$\delta(y, a) = ya,$$

where ya is the concatenation of y and a, and Y and A are then the state set and the action (input) set of the state transition function δ, respectively.

The output function $\lambda : Y \times A \to A$ takes a special form for the current case:

$$\lambda(y, a) = a.$$

In many cases, Y is too large to be used as a state set. However, we can often find a map $\omega : Y \to C'$ such that ω satisfies the relation

$$(\forall y, y', a)(\omega(y) = \omega(y') \to \omega(ya) = \omega(y'a));$$

ω is called a state reduction map. Then we have the following fact.

$$\begin{array}{ccc} Y \times A & \xrightarrow{\delta} & Y \\ \omega\downarrow \quad \downarrow I & & \downarrow \omega \\ C' \times A & \xrightarrow{\delta'} & C' \end{array} \qquad \begin{array}{ccc} Y \times A & \xrightarrow{\lambda} & A \\ \omega\downarrow \quad \downarrow I & & \downarrow I \\ C' \times A & \xrightarrow{\lambda'} & A \end{array}$$

Fig. 5.2. Automaton homomorphism by ω.

Proposition 5.1. ker$\omega \subset Y \times Y$ *is a Nerode equivalence, or* ω *is an automaton homomorphism, where* $(y, y') \in \ker\omega \leftrightarrow \omega(y) = \omega(y')$ *[Mesarovic and Takahara, 1989]. That is, there is an automaton* $\langle A, C', \delta', \lambda' \rangle$ *such that the diagrams in Fig. 5.2 are commutative.*

Proof. Refer to Appendix 5.3.

If ω is found, $(Y/\ker\omega)$ will be used as a state set. Moreover, $(Y/\ker\omega)$ is expected to be of a manageable size. If we allow the trivial case that $\omega(y) = y$, it follows that ω always exists. As such, the state set will be denoted by C and the state transition function and the output function will be denoted by $\delta : C \times A \to C$ and $\lambda : C \times A \to A$, respectively.

The example of Section 1.3 has $\omega : Y \to C$, where $C = N \times M$ and $\omega(a_1, \ldots, a_k) = a_k$. Then the relation $\omega(y) = \omega(y') \to \omega(ya) = \omega(y'a)$ trivially holds for $a \in A = N \times M$. Other examples of ω are given in the subsequent chapters.

There may be a misunderstanding that an automaton formulation can be introduced because the target problem has the explicit solving action property or is a dynamic problem. This is not true. The automaton formulation comes from the dynamics of the solving process. This issue is discussed in Section 5.2.

As mentioned in Section 4.1, δ is, in general, a partial function. In order to ensure that δ behaves properly, the following two functions are used:

$$\text{genA} : C \to \wp(A)$$

and

$$\text{constraint} : C \to \{\text{true}, \text{false}\},$$

where $\wp(A)$ is the power set of A. The following should hold:

$$\delta(c, a) = c' \to a \in \text{genA}(c) \text{ and constraint}(c') = \text{true}.$$

The output function $\lambda : C \times A \to A$ is

$$\lambda(c, a) = a.$$

Let the initial state be

$$c_0 \in C,$$

where $c_0 = \omega(\Lambda)$ for the null string $\Lambda \in A^*$. Obviously,

$$\text{constraint}(c_0) = \text{true}$$

should hold.

Because a problem is a closed-target problem for the E-C-C case, let the set of target states be

$$C_f \subset C;$$

C_f may be or may not be a singleton set, and

$$(\forall c_f \in C_f)\ (\text{constraint}(c_f) = \text{true})$$

must hold.

An automaton specification of the process is then as follows:

Automaton specification of process $= \langle A, C, Y, \delta, \lambda, \text{genA}, \text{constraint}, c_0, C_f \rangle$.

In general, $C \neq Y$.

5.1.4 Dynamic Optimization Formulation

Because a state transition function is a discrete state space representation, if an evaluation function

$$G : Y \to \text{Re}$$

is introduced for the output, we have a dynamic optimization problem, where Re is the set of real numbers.

Because a problem in the E-C-C case is a closed-goal problem, an evaluation function is given by the PSE, which is expressed by $G : Y \to \text{Re}$. We assume that y is desirable if $G(y)$ is small. A dynamic optimization formulation of the process specification is then as follows:

Dynamic optimization formulation$= \langle A, C, Y, \delta, \lambda, G, \text{genA}, \text{constraint}, \text{st}, c_0, C_f \rangle$,

where $st : C \to \{\text{true}, \text{false}\}$ is a stopping condition. In many cases, st is as follows:

$$st(c) = \text{true} \leftrightarrow c \in C_f.$$

In many practical cases of explicit solving action, the following additivity condition is satisfied for G.

Definition 5.1. Additivity Condition
If $G : Y \to$ Re satisfies the condition

$$G(ya) = G(y) + \text{goalElement(a)}$$

for some function goalElement:$A \to$ Re, where $y \in A^*, a \in A$, goalElement(a) > 0, and $G(\Lambda) = 0$, G is called additive. When G is additive, the dynamic optimization formulation will be called an additive formulation.

The example of Section 1.3 illustrates a case of additive formulation. A simple but important case of an additive goal is given by

$$G(y) = \text{length of } y$$

or goalElement$(a) = 1$. This goal requires the shortest path for a solution.

When the additive condition holds, its dynamic optimization formulation is given by

$$\textbf{Dynamic optimization formulation} = \langle A, C, Y, \delta, \lambda, \text{ goalElement, genA, constraint}, st, c_0, C_f \rangle.$$

5.2 User Model for I-O-O Case

This section discusses another extreme user model of the case of I-O-O, which is the least structured case.

5.2.1 Drawing Input–Output Block Diagram

This stage is exactly the same as that of Section 5.1.1. Figure 5.3 illustrates an input–output block diagram of the data mining system problem, which is discussed in Chapter 12. The data mining system problem belongs to the I-O-O case.

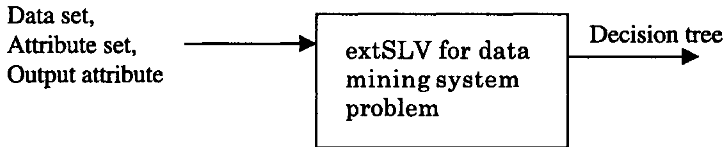


Fig. 5.3. Data mining system problem.

The data mining system of Fig. 5.3 is assumed to generate a decision tree that describes implications of a given data set.

5.2.2 Input–Output Specification in Set Theory

This stage is also essentially the same as that of Section 5.1.2. The PSE is represented by a structure.

Input structure:

$$\langle \text{class of base sets, class of relations, class of functions, class of constants} \rangle.$$

Because the case is of implicit solving activity, the output is not given in a relation with a solving activity. Then formally,

Output set:

$$Y,$$

where $y \in Y$ can have an involved structure reflecting the property of the target problem. Certainly, a decision tree of Fig. 5.3 cannot be represented by a simple structure (see Chapter 12).

5.2.3 Process Specification as Automaton

At this stage, the solving activity process δ is determined as an automaton. Because the problem is of implicit solving activity, the state transition is specified in a way different from that in Section 5.1. In general let

$$A = \{a | a : Y \to Y\}.$$

Then, $\delta : Y \times A \to Y$ is defined as

$$\delta(y, a) = a(y).$$

It is obvious that A is often too large to be manipulated conveniently. In order to overcome this difficulty, heuristics can be used to reduce the size of A [Takahara and Saito, 2005]. This book assumes that A is represented as a parameter set rather than as a set of mappings. This assumption is illustrated in the examples of the subsequent chapters.

We will use the symbol C for the state set Y, i.e., $\delta : C \times A \to C$. In general, δ is also a partial function in the current case. Then, the following two functions are used to ensure that δ behaves properly:

$$\text{genA} : C \to \wp(A)$$

and

$$\text{constraint} : C \to \{\text{true, false}\}.$$

The function genA can be used to effectively reduce the action set depending on the state. δ must satisfy

$$\delta(c, a) = c' \to a \in \text{genA}(c) \text{ and constraint}(c') = \text{true}.$$

The output function $\lambda : C \times A \to Y$ is different from that of Section 5.1. In the current case it is given by

$$\lambda(y, a) = y.$$

Let the initial state be

$$c_0 \in C.$$

Heuristically, a trivial state is used to specify c_0. If an element of C is a set, $c_0 = \varnothing$ (empty set) is a usual candidate. Obviously, constraint(c_0) = true must hold.

In the case of I-O-O, the set C_f of target states is not given a priori. Unless the dynamic programming method discussed in Section 5.3 is used, target states are used only for specifying the stopping condition. In many cases of I-O-O, including the data mining case, C_f can be satisfactorily specified as

$$C_f = \{c | (\forall a \in A)(\delta(c, a) \text{ is undefined})\}.$$

That is, the stopping condition implies the situation from which the solving process activity cannot proceed further. Another case is illustrated by an optimization problem of control engineering of Chapter 8, in which the time span for the solving process is given a priori. For example, if the goal is specified as

$$\int_0^T \text{goal(c, a)}\, dt,$$

then $[0, T]$ is the span for process operation. The stopping time is given by T, which is the stopping condition.

Using the above heuristic consideration, the automaton specification of the process is then given as

Automaton specification of process $= \langle A, C, Y, \delta, \lambda,$ genA, constraint, $c_0, C_f\rangle$.

5.2.4 Dynamic Optimization Formulation

Because the current problem is of the I-O-O type, there is no legitimate goal for the problem. Nevertheless, a goal is needed and is used to evaluate an action even in the case of an open goal. In fact, a goal can be a design parameter in this case. Heuristics are again used for determination of a convenient goal. In many cases, an estimation of the length of the solving path is used, or desirability of an output is described using a standard form (for example, a quadratic form). These heuristics are illustrated in Chapters 7, 11, and 12. Let a goal be formally represented as

$$\text{goal} : C \to \text{Re}.$$

If $C = Y$ has a preference relation $\leq$, it is desirable that the following compatibility relation holds:

$$c \leq c' \to \text{goal}(c) \geq \text{goal}(c').$$

Naturally, this relation cannot always be satisfied in practice.

A dynamic optimization formulation of the process is then given as follows:

Dynamic optimization formulation $= \langle A, C, Y, \delta, \lambda,$ goal, genA, constraint, $st, c_0, C_f\rangle$,

where $st : C \to \{\text{true, false}\}$ is a stopping condition, i.e.,

$$st(c^*) = \text{true} \leftrightarrow c^* \in C_f \,\&\, \min\{\text{goal}(c) | c \in C_f\} = \text{goal}(c^*).$$

5.3 Standardized Goal-Seeker by Modified Dynamic Programming Method

The final step of the design is to attach a standardized goal-seeker to a dynamic optimization formulation. This section discusses a standardized goal-seeker based on dynamic programming (DP) and hence $\langle A, C, Y, \delta, \lambda$, goalElement, genA, constraint, $st, c_0, C_f\rangle$ is used as its target structure.

We assume that C_f is a singleton set, i.e., $C_f = \{c_f\}$. This assumption is not essential. The external behavior of a goal-seeker is formally given by a mapping $\sigma : C \to A$. Then, combining σ and δ, we have a dynamical system $\pi_\sigma : C \to C$, where

$$\pi_\sigma(c) = \delta(c, \sigma(c)).$$

Here π_σ is called a dynamical mapping, which is a generator expression of extSLV.

The function σ is characterized by a strategy that specifies an action "a" depending on "c." This section will use a modified DP method as the strategy, simply called a DP method. Figure 5.4 illustrates the structure of the DP method.

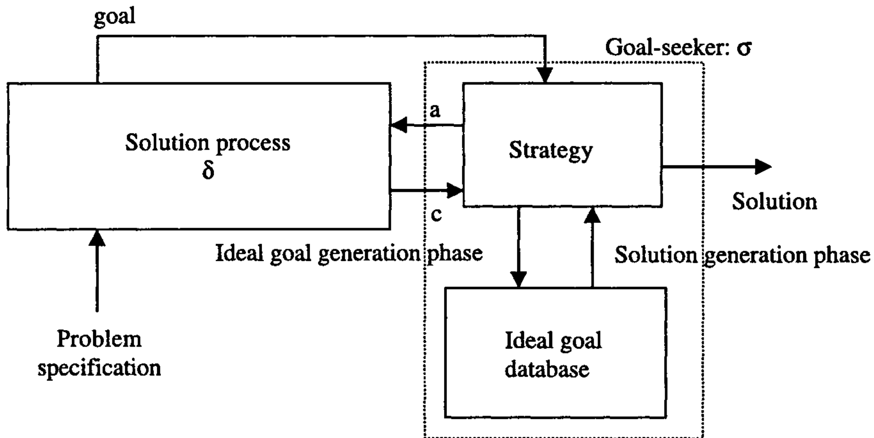


Fig. 5.4. Goal-seeker by dynamic programming (DP) method.

In the method, a solution is obtained in two phases: the ideal goal generation phase and the solution generation phase. The concept of these phases is illustrated by a maze problem in Fig. 5.5 in which a cell is given an identification number. The solution is obviously the path 1–4–5–6–9–8.

An ideal goal is a function idGoal:{cell} $\to$ Re, where {cell} is the set of cells. It shows the length of the shortest path to the exit. In the ideal goal generation phase the system proceeds backward. Because cell 8 is the exit of the maze, the ideal goal idGoal is specified as idGoal(8) $= 0$. Because cell 8 can be accessed by cell 9 in one step, idGoal(9) $= 1$. Similarly, because cell 6 accesses cell 9 in one step or cell 8 in two steps, idGoal(6) $= 2$. In this way, the ideal goal idGoal() is generated as shown in Fig. 5.5.

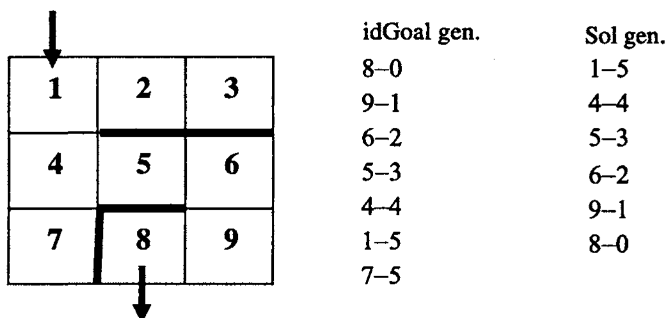


Fig. 5.5. Goal generaion and solution generation.

In the solution generation phase the system proceeds forward and starts from initial cell 1. Suppose the solution generation activity is in cell c_i. Then it proceeds to cell c_j, which satisfies the condition

$$\min\{\text{idGoal}(c')|c' \text{ is directly connected to } c_i\} = \text{idGoal}(c_j),$$

where c_i is directly connected to c_j. The solution path moves from c_i to c_j. We will prove below that c_j can be always found. The total solution path 1–4–5–6–9–8 is specified in this way.

When the DP method is used, two auxiliary functions, inv$\delta : C \to \wp(C \times A)$ and invst:$\wp(C) \to \{\text{true, false}\}$, are used in order to improve computation speed. They are shortenings of "inverse δ" and "inverse st." Let invδ() be

$$\text{inv}\delta(c_2) = \{(c, a)|\delta(c, a) = c_2 \;\&\; \text{constraint}(c) = \text{true} \;\&\; a \in \text{genA}(c)\}.$$

Let invst() be

$$\text{invst}(C') = \text{true} \leftrightarrow c_0 \in C'.$$

The function invδ will be used for the backward operation of the idGoal generation, while the function invst provides a stopping condition for the backward operation.

5.3.1 Reduced-Layer Model

The most serious problem with the standard DP method is that because it has to evaluate almost every state of the state space with respect to a given goal, heavy computation is required for a real problem. In order to avoid heavy computation we must find a way to reduce the number of states to be evaluated. The concept of a reduced-layer model is introduced for that purpose.

Definition 5.2. Backward Layer Structure $\{C_k \subset C\}_k$
Let a class of subsets $C_k \subset C (k = 0, 1, 2, \ldots)$ be defined as follows:

$$C_0 = \{c_f\}.$$
$$C_k = \{c_2|(\exists c \in C_{k-1})(\exists a \in A)(\delta(c_2, a) = c \;\&\; \text{constraint}(c_2) = \text{true} \;\&\; a \in \text{genA}(c_2))\} - \cup\{C_p | p < k\}.$$

Here $\{C_k \subset C\}_k$ will be called the backward layer structure of C.

C_k is the set of states reachable going backward from C_{k-1} in one step. It should be clear that for Fig. 5.5,

$$C_0 = \{8\};\ C_1 = \{9\};\ C_2 = \{6\};\ C_3 = \{5\};\ C_4 = \{4\};\ C_5 = \{1, 7\}.$$

Definition 5.3. Restricted Allowable Activity Function
Because $C_i \cap C_j = \varnothing$ holds for $i \neq j$, a function

$$\text{genA}^* : \cup\{C_k\}_k \to \wp(A)$$

is defined as follows: Let $c \in \cup\{C_i\}_i$ be arbitrary. Let k be the unique integer such that $c \in C_k$. Then, if $k > 0$,

$$\text{genA}^*(c) = \{a | \delta(c, a) \in C_{k-1}\} \cap \text{genA}(c).$$

Otherwise,

$$\text{genA}^*(c_f) = \varnothing.$$

Here genA* is called the restricted allowable activity function.

Of course, $\text{genA}^*(c) \subset \text{genA}(c)$ holds.

Definition 5.4. Reduced-Layer Model
The following dynamic optimization formulation

$$\textbf{Reduced-layer model} = \langle A, C, Y, \delta, \lambda, \text{goalElement},\ \{C_k\}_k,\ \text{genA}^*,\ \text{constraint}, st, c_0, c_f\rangle$$

will be called the reduced-layer model.

The difference between the dynamic optimization formulation and the reduced-layer model is that the latter uses genA* instead of genA.
The following are fundamental facts for a reduced-layer model.

Proposition 5.2. *Suppose $c_0 \in C_k$ for some k. Then, there exists $a_1 a_2 \cdots a_k \in A^*$ such that $\delta(c_0, a_1) = c_1 \in C_{k-1}, \delta(c_0, a_1 a2) = c_2 \in C_{k-2}, \ldots, \delta(c_0, a_1 a_2 \cdots a_k) = c_f$. Furthermore, $a_k \in \text{genA}^*\ (c_{k-1})$.*

Proof. Refer to Appendix 5.4.

Proposition 5.3. *Suppose $\delta(c_0, y) = c_f$ for some y. Then $c_0 \in C_k$ for some k.*

Proof. Refer to Appendix 5.5.

Combining Proposition 5.2 and Proposition 5.3, we have the following theorem.

Theorem 5.1. *If $\delta(c_0, y) = c_f$ for some y, then there exist $k, c_0 \in C_k$, and $a_1 a_2 \cdots a_k \in A^*$ such that $\delta(c_0, a_1) = c_1 \in C_{k-1}, \delta(c_0, a_1 a2) = c_2 \in C_{k-2}, \ldots,$ $\delta(c_0, a_1 a_2 \cdots a_k) = c_f$, and $a_{i+1} \in \text{genA}^*\ (c_i)$.*

Theorem 5.1 asserts that if there is a sequence y that takes the system from c_0 to c_f, another sequence can be found for the reduced-layer model that can do the same thing.

Definition 5.5. Semioptimal Solution
An optimal solution of a reduced-layer model will be called a semioptimal solution.

The following are conceptual results.

Corollary 5.1. *If* genA = genA*, *a semioptimal solution is an optimal solution of a dynamic optimization formulation.*

Corollary 5.2. *A semioptimal solution yields the shortest step path from c_0 to c_f. Furthermore, it yields an optimal path among the shortest paths from c_0 to c_f.*

In the subsequent discussions $\delta(c_0, y) = c_f$ is assumed to hold for some y.

5.3.2 Ideal Goal Generation

We will develop a DP method for the reduced-layer model.

Definition 5.6. idGoal Family
Let $c_0 \in C_k$ for some k. Let us define a class of functions $\text{idGoal}_p : C_p \rightarrow \text{Re}$ for $p = 0, 1, 2, 3, \ldots$ recursively.
(i) Let $\text{idGoal}_0 : C_0 \rightarrow \text{Re}$ be given as

$$\text{idGoal}_0(c_f) = 0.$$

(ii) Suppose idGoal_{p-1} is given. Then, for $c \in C_p$,

$$\text{idGoal}_p(c) = \min\{\text{goalElement}(a) + \text{idGoal}_{p-1}(\delta(c, a)) | a \in \text{genA}^*(c)\}.$$

$\{\text{idGoal}_p\}_p$ will be called the idGoal family.

The idGoal family of the maze problem of Fig. 5.5 is given in Fig. 5.5.

Let us reformalize the above recursive relation as an automaton (generator), genGoal, using invδ. For the sake of notational convenience, $\delta^* : C \times A \rightarrow C \times \text{Re}$ is introduced as

$$\delta^*(c, a) = (\delta(c, a),\ \text{goalElement}(a)).$$

If invδ is used, $\{\text{idGoal}_p\}_p$ is given by the following relation.

Proposition 5.4. *Suppose $c \in C_p$. Then*

$$\text{idGoal}_p(c) = \min\{\text{goalElement}(a) + \text{idGoal}_{p-1}(c') | (c, a) \in \text{inv}\delta(c') \& c' \in C_{p-1}\}$$

holds.

Proof. Refer to Appendix 5.6.

Let $\delta\text{Goal} : \{C_p\}_p \times \{\text{idGoal}_p\}_p \times \wp(C) \to \{C_p\}_p \times \{\text{idGoal}_p\}_p \times \wp(C)$ be

$$\delta\text{Goal}(C_p, \text{idGoal}_p, C_p^*) = (C_{p+1},\ \text{idGoal}_{p+1}, C_{p+1}^*),$$

where $(C_{p+1}, \text{idGoal}_{p+1}, C_{p+1}^*)$ is specified by the following relations:

$$C_{p+1} = \cup\{\{c|(\exists a)((c,a) \in \text{inv}\delta(c_2))\}|c_2 \in C_p\} - C_p^*,$$

$$C_{p+1}^* = C_p^* \cup C_{p+1},$$

and for $c \in C_{p+1}$,

$$\text{idGoal}_{p+1}(c) = \min\{\text{goalElement}(a) + \text{idGoal}_p(c_2)|(c,a) \in \text{inv}\delta(c_2)\ \&\ c_2 \in C_p\}.$$

The stopping condition, invst:$\wp(C) \to$ {true, false}, for genGoal is given in the user model as

$$\text{invst}(C_p) = \text{true} \leftrightarrow c_0 \in C_p.$$

Using the functions δGoal and invst, we can define a generator, genGoal.

Definition 5.7. genGoal
Let an automaton (generator) be

$$\text{genGoal} = \langle\{C_p\}_p \times \{\text{idGoal}_p\}_p \times \wp(C), \delta\text{Goal},\ \text{invst},\ (\{c_f\}, \text{idGoal}_0, \{c_f\})\rangle,$$

where

$$\text{idGoal}_0 = \{(c_f, 0)\}.$$

Let genGoal be called a generator for the idGoal family.

The process of the ideal goal generation phase is given by genGoal.

5.3.3 Solution Generation

Once the idGoal family, $\{\text{idGoal}_p\}$, is given, a semioptimal solution is obtained in a straightforward way. Suppose $c_0 \in C_k$ holds for some k, i.e., $\text{invst}(C_k) = \text{true}$.

Proposition 5.5. *Let $p \le k$. Then*

$$(\forall c \in C_p)(\exists a \in \text{genA}^*(c))(c' = \delta(c,a) \in C_{p-1}\&\ \text{idGoal}_p(c) - \text{idGoal}_{p-1}(c') = \text{goalElement}(a)).$$

Furthermore, if $\text{idGoal}_p(c) - \text{idGoal}_{p-1}(\delta(c,a^*)) = \text{goalElement}(a^*)$ *for* $a^* \in A$,

$$\min\{\text{goalElement}(a) + \text{idGoal}_{p-1}(\delta(c,a))|a \in \text{genA}^*(c)\} = \text{goalElement}(a^*) + \text{idGoal}_{p-1}(\delta(c,a^*)),$$

i.e., $a^* \in \text{genA}^*(c)$ *and is a desired action.*

Proof. Refer to Appendix 5.7.

The proposition says that for c in C_p we can always find a desirable action a in genA*(c) that takes c to c' in C_{p-1}, which is characterized by the values of idGoal for them.

Theorem 5.2. *A semioptimal solution* $y = a_1 a_2 \cdots a_k, \delta(c_0, y) = c_f$, *is given by the following relation:*

$$c_1 = \delta(c_0, a_1) \in C_{k-1} \&\ \mathrm{idGoal}_k(c_0) - \mathrm{idGoal}_{k-1}(c_1) = \mathrm{goalElement}(a_1).$$

$$c_2 = \delta(c_1, a_2) \in C_{k-2} \&\ \mathrm{idGoal}_{k-1}(c_1) - \mathrm{idGoal}_{k-2}(c_2) = \mathrm{goalElement}(a_2).$$

$$\vdots$$

$$c_f = \delta(c_{k-1}, a_k) \in C_0 \&\ \mathrm{idGoal}_1(c_k) - \mathrm{idGoal}_0(c_f) = \mathrm{goalElement}(a_k).$$

Theorem 5.2 gives a procedure to generate a semioptimal solution.

Let k be the first number that satisfies $c_0 \in C_k$ and let it remain fixed. Let strategy: $C \times A \to C \times A$ be for $c \in Cp$ and $a \in A$

$$\mathrm{strategy}(c, a) = (c_2, a_2) \leftrightarrow \delta(c, a_2) = c_2 \& c_2 \in C_{p-1} \&\ \mathrm{idGoal}_p(c) - \mathrm{idGoal}_{p-1}(c_2) = \mathrm{goalElement}(a_2).$$

Existence of a strategy is asserted by Theorem 5.2. The stopping condition $st : C \times A \to \{\text{true, false}\}$ is given in the user model as

$$st(c, a) = \text{true} \leftrightarrow c \in C_f.$$

Definition 5.8. genSol
Let an automaton (generator) be

$$\mathrm{genSol} = \langle C \times A, \mathrm{strategy}, st, (c_0, a_0) \rangle,$$

where a_0 can be any element in A. Then genSol will be called the generator for a solution generation.

The strategy is a combination of a state transition function and an output function. A solution is given by an automaton (generator) genSol.

5.4 Standardized Goal-Seeker by Hill Climbing with Push-Down Stack Method

The goal-seeker is easily hampered by a deadlocked situation due to ill-structured constraints of a problem-solving system. This section investigates a strategy equipped with a push-down stack that enables backtracking to avoid such a deadlocked situation. If the system meets a deadlocked situation, it must return to a previous state (backtracking) to proceed in another direction. The stack saves information necessary for backtracking. A strategy can be designed using a combination of the hill-climbing method and a push-down stack. This method will be called the PD method.

The target formulation of a user model is assumed to be given by

$$\textbf{Dynamic optimization formulation} = \langle A, C, Y, \delta, \lambda, \text{goal}, \text{genA}, \text{constraint}, st, c_0, C_f\rangle,$$

where goal() is given as a function whose domain is the set of states. If the formulation is given as

$$\textbf{Dynamic optimization formulation} = \langle A, C, Y, \delta, \lambda, \text{goalElement}, \text{genA}, \text{constraint}, st, c_0, C_f\rangle,$$

where goalElement:$A \to$ Re, it can be reformulated in the form with goal: $C \to$ Re. Let $C^e = C \times$ Re,

$$\delta^e : C^e \times A \to C^e \text{ such that } \delta^e((c, r), a) = (\delta(c, a), r + \text{goalElement}(a)),$$
$$\lambda^e : C^e \times A \to Y \text{ such that } \lambda^e((c, r), a) = \lambda(c, a),$$
$$\text{genA}^e : C^e \to \wp(A) \text{ such that } \text{genA}^e((c, r)) = \text{genA}(c),$$
$$\text{constraint}^e : C^e \to \{\text{true, false}\} \text{ such that } \text{constraint}^e((c, r)) = \text{constraint}(c),$$
$$c_0^e = (c_0, 0),$$
$$C_f^e = \{(c_f, r) | c_f \in C_f, r \in \text{Re}\},$$
$$st^e : C^e \to \{\text{true, false}\} \text{ such that } st^e((c, r)) = st(c).$$

Consequently, this section assumes that the target formulation is given as $\langle A, C, Y, \delta, \lambda,$ goal, genA, constraint, $st, c_0, C_f\rangle$.

Figure 5.6 illustrates the structure of the hill-climbing method with a push-down stack.

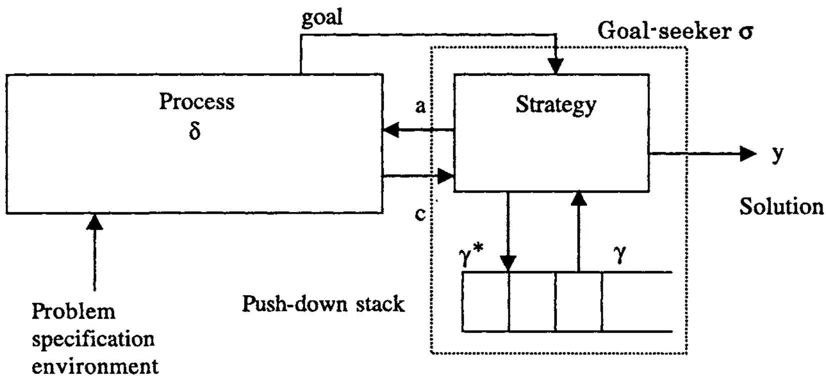


Fig. 5.6. Goal-seeker by hill-climbing with push-down stack.

This section investigates general properties of Fig. 5.6. Let us start by introducing some underlying concepts.

Definition 5.9. legalAs: $C \to \wp(A^*)$
Let us define a function legalAs:$C \to \wp(A^*)$: for $c \in C$,

$$
\begin{aligned}
a_1 a_2 \cdots a_p \in \text{legalAs}(c) \text{ iff } & \delta(c, a_1) = c_1 \,\&\, a_1 \in \text{genA}(c) \,\&\, \text{constraint}(c_1) = \text{true} \\
& \& \, \delta(c_1, a_2) = c_2 \,\&\, a_2 \in \text{genA}(c_1) \& \text{constraint}(c_2) = \text{true} \\
& \bullet \\
& \bullet \\
& \& \, \delta(c_{p-1}, a_p) = c_p \,\&\, a_p \in \text{genA}(c_{p-1}) \,\&\, \text{constraint}(c_p) \\
& = \text{true}.
\end{aligned}
$$

Definition 5.10. Solvable State and Unsolvable State
Let $c \in C$ be called a solvable state iff $(\exists \alpha \in \text{legalAs}(c))(\delta(c, \alpha) \in C_f)$. Let $c \in C$ be called an unsolvable state iff c is not solvable, i.e., $(\forall \alpha \in \text{legalAs}(c))(\delta(c, \alpha) \notin C_f)$.

Definition 5.11. Forward Layer Structure $\{C_p\}_p$ on C
A class of subsets $\{C_p\}_p$, $C_p \subset C$ $(p = 0, 1, 2, \ldots)$, will be defined recursively. Let

$$
\begin{aligned}
C_0 &= \{c_0\}; \\
C_{p+1} &= \{c' | (\exists c \in C_p)(\exists a \in A)(a \in \text{genA}(c) \& \delta(c, a) \\
&= c' \,\&\, \text{constraint}(c') = \text{true})\} - \cup \{C_r | r \leq p\}.
\end{aligned}
$$

The class $\{C_p\}_p$ is the forward layer structure on C.

Note the difference between Definition 5.2 and Definition 5.11.

Definition 5.12. Linear Order Relation $\leq$ on $\cup C_k$
Let $c, c' \in \cup C_k$ be arbitrary. Then p and q are unique such that $c \in C_p$ and $c' \in C_q$. Let

$$c \leq c' \leftrightarrow p \leq q.$$

Definition 5.13. Restricted Allowable Activity Function of PD Method: genA*:$\cup C_k \to \wp(A)$
Let a function genA*:$\cup C_k \to \wp(A)$ be defined as follows: let $c \in \cup C_k$. Because $i \neq j \to C_i \cap C_j = \varnothing$, there is a unique p such that $c \in C_p$. Then,

$$\text{genA}^*(c) = \{a | a \in \text{genA}(c) \,\&\, \delta(c, a) \in C_{p+1}\}.$$

Here genA* is the restricted allowable activity function of the PD method.

Definition 5.14. Solution Path and Efficient Solution Path
Let $\alpha \in \text{legalAs}(c_0)$ be called a solution path iff $\delta(c_0, \alpha) \in C_f \subset C$. Let $\alpha = a_1 a_2 \cdots a_p \in \text{legalAs}(c_0)$ be called an efficient solution path iff α is a solution path and satisfies the following relations:

$$\delta(c_0, a_1) = c_1 \in C_1,$$
$$\delta(c_1, a_2) = c_2 \in C_2,$$
$$\bullet$$
$$\bullet$$
$$\delta(c_{p-1}, a_p) = c_p \in C_p = C_f,$$

where for $i < p$, $C_i \neq C_f$.

The efficient solution path will be used as a desirability criterion for the goal-seeker of the PD method.

Definition 5.15. Backtrack: $C \to \wp(A)$
Let a function backtrack: $C \to \wp(A)$ be such that for any solvable $c \in C$,

$$a \in \text{backtrack}(c) \text{ iff } a \in \text{genA}(c) \text{ and } \delta(c, a) \text{ is a solvable state.}$$

Although a goal is used to select an action to yield the next proper state on a C_k, it is not necessarily compatible with the order of C. However, if it is compatible, it will be shown to derive stronger results. We use the following compatibility condition.

Definition 5.16. Compatibility Condition for Goal
A goal is said to satisfy a compatibility condition iff it satisfies the following relation: for $c, c' \in \cup C_p$,

$$\text{goal}(c) \geq \text{goal}(c') \leftrightarrow c \leq c'.$$

Definition 5.17. Strategy of the Hill-Climbing Method with Push-Down Stack: $\sigma_{pd} : \cup C_p \to A$
Let $\sigma_{pd} : \cup C_p \to A$ be

$$\sigma_{pd}(c) = a^*,$$

where $a^* \in$ backtrack(c) and satisfies

$$\min\{\text{goal}(\delta(c, a)) | a \in \text{backtrack}(c)\} = \text{goal}(\delta(c, a^*)).$$

Then σ_{pd} will be called the strategy of the hill-climbing method with a push-down stack.

The goal-seeker of this section is characterized by σ_{pd}. It should be noted that the definition of σ_{pd} is independent of the compatibility of the goal: σ_{pd} is not a simple feedback law because it requires a backtrack operation to ensure that $\delta(c, \sigma_{pd}(c))$ is a solvable state.

Definition 5.18. Loop-Free Strategy
A strategy $\sigma : C \to A$ is called loop free if it satisfies

$$(\forall p, q)(p \neq q \to \pi_\sigma^p(c_0) \neq \pi_\sigma^q(c_0)).$$

where π_σ is the dynamical mapping of σ, i.e., $\pi_\sigma(c) = \delta(c, \sigma(c))$ and $\pi_\sigma^{p+1}(c_0) = \pi_\sigma(\pi_\sigma^p(c_0))$.

The following is an obvious but fundamental fact for problem-solving.

Theorem 5.3. *Suppose C is finite and the initial state is solvable. Then, if a strategy σ is loop free, p exists such that $\pi_\sigma^p(c_0) \in C_f$ or it yields a solution.*

Proof. Refer to Appendix 5.8.

The following is also intuitively clear.

Proposition 5.6. *Let $c \in C_p$ be arbitrary. Then*

$$(\forall\alpha)(\exists q)(\alpha \in \text{legalAs}(c)\ \&\ \delta(c, \alpha) = c' \rightarrow c' \in C_q).$$

Proof. Refer to Appendix 5.9.

The following is also a fundamental fact for the PD method.

Proposition 5.7. *Let $\alpha \in$ legalAs(c_0) be arbitrary. Let $c' = \delta(c_0, \alpha)$. Then, $(\exists\alpha' \in$ legalAs$(c_0))(\alpha' = a_1 \cdots a_k \& \delta(c_0, a_1, \ldots, a_i) \in C_i$ for each $i \& \delta(c_0, \alpha') = c')$.*

Proof. Refer to Appendix 5.10.

Theorem 5.4. *There is a solution path iff an efficient solution path exists.*

Proof. Refer to Appendix 5.11.

We will characterize an efficient solution path. The following should be obvious.

Corollary 5.3. *Suppose $\alpha = a_1a_2 \cdots a_p \in$ legalAs(c_0) is a solution path. Suppose α satisfies the following relations:*

$$\delta(c_0, a_1) = c_1 \notin C_0,$$

$$\delta(c_1, a_2) = c_2 \notin C_0 \cup C_1,$$

$$\bullet$$

$$\bullet$$

$$\delta(c_{p-1}, a_p) = c_p \notin \cup\{C_k | k < p\}.$$

Then, α is an efficient solution path.

An efficient solution path is a solution path that never goes backward (with respect to the order of C) on the way to a target state. The following should be also obvious.

Corollary 5.4. *Suppose $\alpha = a_1a_2 \cdots a_p \in$ legalAs(c_0) is a solution path. Suppose α satisfies the following relations:*

$$a_1 \in \text{genA}^*(c_0),$$

$$a_2 \in \text{genA}^*(c_1), \text{ where } \delta(c_0, a_1) = c_1,$$

$$a_3 \in \text{genA}^*(c_2), \text{ where } \delta(c_1, a_2) = c_2,$$

$$\bullet$$

$$\bullet$$

$$a_p \in \text{genA}^*(c_{p-1}), \text{ where } \delta(c_{p-2}, a_{p-1}) = c_{p-1}.$$

Then, α is an efficient solution path.

When the goal is compatible, we have strong results for σ_{pd}.

Proposition 5.8. *Let $c \in C_p$ be solvable. Suppose the goal is compatible. Let $a^* \in \sigma_{pd}(c)$, where σ_{pd} is a strategy of the PD method. Then,*

$$\delta(c, a^*) \in C_{p+1}.$$

Proof. Refer to Appendix 5.12.

Corollary 5.5. *If the goal is compatible, σ_{pd} is loop free.*

Corollary 5.5 indicates that if the goal is compatible and if the strategy is σ_{pd}, the usual technique of problem-solving that the trajectory of states is saved to avoid repetition of states is not necessary.

Theorem 5.5. *The strategy σ_{pd} of the PD method always yields an efficient solution path if the initial state is a solvable state and the goal is compatible.*

In practice there are two ways to derive an efficient solution path. The first way is to use Corollary 5.3, which says that an efficient solution path is produced if a next state c_i is found outside of $\cup\{C_p | p < i\}$ for every i. However, because it is usual to consume a large amount of memory to memorize $\cup\{C_p | p < i\}$, Corollary 5.3 has difficulty with respect to computation speed and memory limitation. Then, if we use a heuristic that the history H of the visited states may cover an essential part of $\cup\{C_p | p < i\}$, H may be substituted for $\cup\{C_p | p < i\}$. In general, this heuristic is useful because in any case H must be saved in order to make a strategy loop free.

The second way to derive an efficient solution path is to use Theorem 5.5. In that case, the goal is required to be compatible. Although the compatibility condition might be considered very exceptional, that is not necessarily true. If a target problem belongs to the class of implicit solving actions and $a : Y \rightarrow Y$ is selected as an action to improve an output y, and if the goal is defined to evaluate the improvement, the compatibility condition holds. The data mining system problem of Chapter 12 corresponds to this case.

Figure 5.7 shows the total process of a general goal-seeker based on Corollary 5.3 and Theorem 5.5.

All necessary information to yield an action $\sigma_{pd}(c)$ is stored in a push-down stack. The goal-seeker, first, tries to pop up the top information. If the trial fails, in general it indicates failure of the goal-seeker. However, in the case of an open target, it may indicate that the goal-seeker has reached a final state. (Refer to the definition of C_f in Section 5.2.3.) In either case, the operation of the goal-seeker must finish. If the trial is successful, the goal-seeker gets (c, As, H, α), current state, set of available actions $\subset$ genA(c), the history of the visited states, and the history of selected actions as the top information of the stack. It then selects (a^*, RA), where a^* is assumed to satisfy the relation

$$\min\{\text{goal}(\delta(c, a)) | a \in As\} = \text{goal}(\delta(c, a^*))$$

and $RA = As - \{a^*\}$.

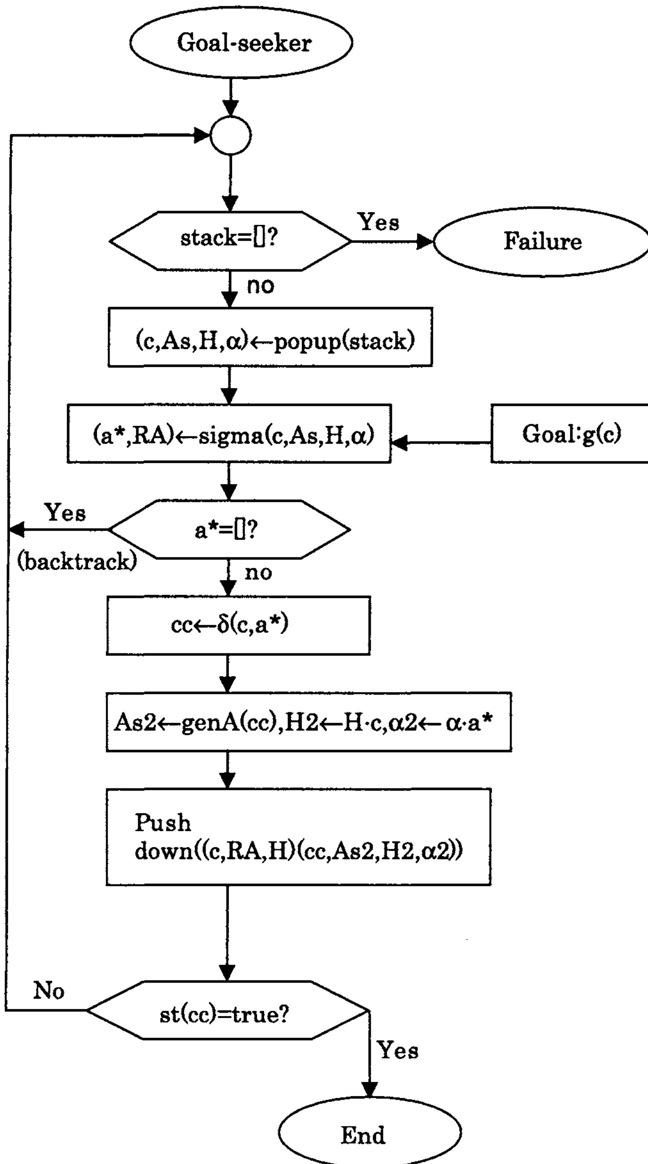


Fig. 5.7. Total process of goal-seeker based on Corollary 5.3 and Theorem 5.5.

Note that $\delta(c, a^*)$ must not be equal to an element of H (refer to Theorem 5.5). If c is a deadlocked state, $a^* = []$ is assumed. In this case the goal-seeker must do the backtrack operation; that is, it pops up the next information from the stack. This backtrack operation ultimately ensures the condition "$\delta(c, a^*)$ is a solvable state" or $a^* \in$ backtrack(c) holds and hence $a^* = \sigma_{pd}(c)$.

If a^* is a regular action ($a^* \neq []$), the goal-seeker applies a^* to the process δ to get the next state $cc = \delta(c, a^*)$. It generates the set of available activities $As2$ for the next state cc. The goal-seeker saves (pushes down) two pieces of information, (c, RA, H, α) and $(cc, As2, H2, \alpha 2)$, where $H2 = H \cdot c$ and $\alpha 2 = \alpha \cdot a^*$, i.e., the current state c and the action a^* are appended to H and α, respectively. The former information (c, RA, H, α) will be used when the goal-seeker backtracks to the state c. The latter information is used at the next state cc. Then, the stopping condition $st()$ is checked. If $st(cc) =$ true, the total process must finish. If $st(cc) \neq$ true, it goes to the pop up stage.

The total extSLV, which consists of the goal-seeker and the process δ, is formalized as a generator:

$$\textbf{extSLV} = \langle \text{Stack, solProc, } st, (c_0, \text{ genA}(c_0), [], []) \rangle,$$

where Stack is the state set of extSLV and $(c_0, \text{ genA}(c_0), [], [])$ is an initial state. Stack, solProc:Stack $\rightarrow$ Stack and $st : C \rightarrow \{\text{true, false}\}$ are defined as follows:

Let

$$\begin{aligned} \Gamma &= C \times \wp(A) \times H \times \text{ Sol; stack element,} \\ H &= C^*; \text{ history of states,} \\ \text{Sol} &= A^*; \text{ history of actions,} \\ \text{Stack} &= \Gamma^*. \end{aligned}$$

Let sigma: $\Gamma \rightarrow A \times \wp(A)$ be defined by

$$\text{sigma } (c, As, \varphi, \alpha) = \begin{cases} (a^*, As - \{a^*\}) & \text{if min } \{\text{goal}(\delta(c, a)) | a \in As \\ & \quad \& \text{ constraint}(\delta(c, a)) = \text{true}\} \\ & \quad = \text{goal}(\delta(c, a^*)), \\ ([], []) & \text{otherwise.} \end{cases}$$

In the definition of sigma, the condition $a^* \in$ backtrack(c) is not required because the condition "$\delta(c, a^*)$ is a solvable state" is supported by the backtracking, as mentioned above. Then,

solProc:Stack $\rightarrow$ Stack is given by

$$\text{solProc}(\gamma^* \gamma_n) = \begin{cases} \gamma^* \gamma_n' \gamma_{n+1} & \text{if sigma } (\gamma_n) = (a^*, RA) \text{ and } a^* \neq [], \\ \gamma^* & \text{otherwise,} \end{cases}$$

where

$$\begin{aligned} \gamma_n &= (c, As, \varphi, \alpha), \\ \gamma_n' &= (c, As - \{a^*\}, \varphi, \alpha), \\ \gamma_{n+1} &= (\delta(c, a^*), \text{genA } (\delta(c, a^*)), \varphi \cdot c, \alpha \cdot a^*). \end{aligned}$$

Finally, $st : C \rightarrow \{\text{true, false}\}$ is given by

$$st(c) = \text{true} \leftrightarrow c \in C_f.$$

The generator is actually a push-down automaton without input.

Appendix 5.1 Standard DP Solver

The following program is listed as a reference, whose detailed explanation is omitted [Takahara et al. 2004]:

```
/*stdDPsolver62.p*/
//enhanced DP solver;State=N times R
/*this is a solver for traveling salesman problem:tsales2.set*/
/*this is a solver for warp2.set*/
/*state=[last_visited_city,remaining_cities]*/
//state=[X,Y]

stdDPsolver():-
          if preprocess() then
          else
                    Dummy1:=1
          end,
          retract([regH,regE]),
          stdDPsolver0(),
          if postprocess() then
          else
                    Dummy2:=1
          end;
/*main part*/
stdDPsolver0():-
          function([goal]),
          depCity.g(DepCity),
          finalstate(Cf),
          assign(selectN,20),
          repeat,
                    retract([goal]),
                    saveGoal(Cf,0),
                    assign(regH,[Cf]),
                    assign(goal,[]),
                    genGoal([Cf],CCs0),
                    if CCs0=[]
                    then
                              selectN(SN),
                              SNN:=SN*2,
                              assign(selectN,SNN),
                              Done:=0
                    else
                              assign(regCCs,CCs0),
                              Done:=1
                    end,
                    Done=1,!,
          regCCs(CCs),
          C0:=initialstate(),
```

```
            getGoal(C0,V0),
            assign(regE,[V0]),
            assign(regH,[C0]),
            process(C0,[],Cf,SS),!,
            xwriteln(0,"SS=",SS),!,
            assign(solutionSS,SS);
            saveGoal([T,R],V):-
            switch(goal(T,R,V),Goal),
            univ(Goal,[N,R,V]),
            assign([N,R],V);
            getGoal([T,R],V):-
            switch(goal(T,R,V),Goal),
            Goal,!;

/*goal generation*/
genGoal(Cs,CCs):-
            assign(regH,[]),
            assign(regVs,[]),
            assign(regI,1),
            invdelta0(Cs,Cs0),
            regVs(Vs),
            selectC(Vs,Cs0,Cs2),
            if invst(Cs2) or Cs2=[] then
                        CCs:=Cs2,
                        if Cs2=[] then
                        end

            else
                        genGoal(Cs2,CCs)
            end;

invdelta0([],[]):-!;
invdelta0(Cs,CCs):-
assign(rregCs,Cs),
            assign(rregCCs,[]),
            repeat,
                        rregCs([X|Xs]),
                        invdelta1(X,Ys),
                        append(rregCCs,Ys,rregCCs),
                        assign(rregCs,Xs),
                        Xs=[],!,
            rregCCs(CCs),
            retract([rregCs,rregCCs]);

invdelta1(C,Cs):-
            getGoal(C,V),
            assign(regCs,[]),
            regH(H),
            invdelta(C,CADs),
            if CADs=[] then
                        Cs=[]
            else
            /*generation of new elements of goal*/
            assign(regCADs,CADs),
            repeat,
                        regCADs([[C2,D]|Xs]),
                        if getGoal(C2,V2) then
```

```
                              if V+D<V2 then
        saveGoal(C2,V+D)
                              end
                        else
                              regI(I),
                              II:=I+1,
                              assign(regI,II),
        append(regVs,[[V+D,I]],regVs),
                              append(regH,[C2],regH),
                              saveGoal(C2,V+D),
                              append(regCs,[C2],regCs)
                        end,
                        assign(regCADs,Xs),
                        Xs=[],!,
        regCs(Cs),
        retract([regCs,regCADs])
        end;

/*solution generation*/
process(C,Sol,CC,SSol):-
        strategy(C,A,C2),
        append(Sol,[A],Sol2),
        if C2=CC
        then
                        SSol:=Sol2
        else
                        process(C2,Sol2,CC,SSol);

strategy(C,A,CC):-
        getGoal(C,V),
        regH(H),
        genA(C,As),
        assign(regAs,As),
        RegA.g:=[],
        RegC.g:=[],
        RegV.g:=V+1,
        repeat,
                        regAs([A2|Xs]),
                        delta(C,A2,C2),
                        if not(member(C2,H)) and
getGoal(C2,V2) and V2<RegV.g then
                              RegV.g:=V2,
                              RegC.g:=C2,
                              RegA.g:=A2
                        end,
                        assign(regAs,Xs),
                        Xs=[],!,
        A:=RegA.g,
        CC:=RegC.g,
        retract([regAs]);

selectC(Vs,Cs,Cs0):-
        N:=strlen(Cs),
        selectN(SelN),
        xwriteln(0,"Num. of backward states=",N),
        if N<=SelN then
                        Cs0:=Cs
```

```
            else
                            sort(Vs,Vs1),
                            transpose(Vs1,[Vlst,Ilst]),
                            project(Ilst,[[1,SelN]],Ilist0),
                            project(Cs,Ilist0,Cs0)
            end;
```

Appendix 5.2 Standard DP Solver

The following program is listed as a reference, whose detailed explanation is omitted [Takahara et al. 2003]:

```
/*stdPDsolver122.p*/
stdPDsolver():-
            if preprocess() then
            else
                            Dummy:=1
            end,
            xwriteln(0,"trace mode?(y/n)"),
            xread(0,Ans),
            if Ans="y" then
                            assign(trace,-3)
            end,
            stdPDsolver0(),
            if postprocess() then
            else
                            Dummy2:=1
            end;
stdPDsolver0():-
            delGamma(0),
            assign(gammaId,0),
            initialstate(C0),
            if genA(C0,As0) then
            else
                            xwriteln(0,"I guess your genA has some problem"),
                            fail
            end,
            goal(C0,V0),
            assign(_regV,[V0]),
            getGammaId(GId),
            saveGamma(GId,[C0,As0,[],[]]),
            solProc(C0,[GId]),
            _regStack(Stack2),
            retract([_regStack,_regC]),
            project(Stack2,1,GId2),
            loadGamma(GId2,[Yf,Asf,Solf,Hf]),
            assign(_Solf,Solf),
            xwriteln(0,"Solf=",Solf),
            assign(_Yf,Yf),
            xwriteln(0,"Yf=",Yf),!;
saveGamma(GId,D):-
            switch(gamma(floor(GId/5),GId,D),GI),
            assert(GI);
loadGamma(GId,D):-
```

```
          switch(gamma(floor(GId/5),GId,D),GI),
          GI,!;
delGamma(I):-
          concat([gamma,".",I],GIT),
          parse(GIT,[GI0]),
          univ(GI,[GI0,Id,D]),
          if GI then
                      retract([GI0]),
                      delGamma(I+1)
          end;

solProc(C0,Stack0):-
          assign(_regStack,Stack0),
          assign(_regC,C0),
          repeat,
                      _regStack(Stack1),
                      _regC(C1),
          goalseeker(C1,Stack1,CC1,A,C2,Stack2),
                      if C2<>cfail then
                               goal(C2,V2),
                               _regV(L),
                               append(L,[V2],L2),
                               Len:=strlen(L2),
                               if Len>200 then

          project(L2,["r",200,Len],L22)
                               else
                                          L22:=L2
                               end,
                               assign(_regV,L22),
                               show1(L22,plot)
                      end,
                      if C2<>cfail and st(C2) then
                               Done:=1
                      else
                               Done:=0
                      end,
                      assign(_regStack,Stack2),
                      if Stack2=[] then
                               xwriteln(0,"NO NEXT JOB!!!")
                      end,
                      assign(_regC,C2),
                      Done=1 or Stack2=[],!;

goalseeker(C,[G|Gs],CC,A,C2,Stack2):-
if status(trace,-3) then
xwriteln(0,"*****************************"),
xwriteln(0,"[state C   ]=",C),stop
end,
          mu(C,G,CC,A,C2,Gs2),
          append(Gs2,Gs,Stack2);

mu(cfail,G,CC,A,C2,Gs):-!,
          loadGamma(G,[CC,AAs,SSol,HH]),
if status(trace,-3) then
xwriteln(0,"[state C   ]=",CC),stop
end,
```

```
            if HH=[] then
                        H:=[]
            else
                        project(HH,-1,H)
            end,
            if SSol=[] then
                        Sol:=[]
            else
                        project(SSol,"head",Sol)
            end,
  if status(trace,-3) then
  xwriteln(0,"[AAs        ]=",AAs),stop
  end,
            sigma([CC,AAs,Sol,H],A,C2,RA),
  if status(trace,-3) then
  xwriteln(0,"[selected A]=",A),
  xwriteln(0,"[next C     ]=",C2),
  xwriteln(0,"[RA         ]=",RA),stop
  end,
            if A<>[] then
                        append(Sol,[A],Sol2),
                        exp_to_text(CC,TCC),
                        append([TCC],H,H2),
                        getGammaId(GId),
                        saveGamma(GId,[CC,RA,
                        Sol2,H2]),
                        Gs:=[GId]
            else
                        Gs:=[]
            end;

  mu(C,G,C,A,C2,Gs):-
            loadGamma(G,[CC,AAs,SSol,HH]),
  if genA(C,As) then
            else
                        xwriteln(0,"I guess yor genA has a problem"),
                        fail
            end,
  if status(trace,-3) then
  xwriteln(0,"[As         ]=",As),stop
  end,
            sigma([C,As,SSol,HH],A,C2,RA),
  if status(trace,-3) then
  xwriteln(0,"[selected A]=",A),
  xwriteln(0,"[next C     ]=",C2),
  xwriteln(0,"[RA         ]=",RA),stop
  end,
            if A<>[] then
                        delta(C,A,CC2),
                        append(SSol,[A],Sol2),
                        exp_to_text(C,TC),
                        append([TC],HH,H2),
                        getGammaId(GId),
                        saveGamma(GId,[C,RA,Sol2,H2]),
                        Gs:=[GId,G]
            else
```

```
                        Gs:=[G]
            end;

sigma([C,[],Sol,H],[],cfail,[]):-!;
sigma([C,As,Sol,H],A,C2,AAs):-
            assign(_regAs,As),
            assign(_regGs,[]),
            assign(_regAAs,[]),
            assign(_regCCs,[]),
            repeat,
                        _regAs([X|Xs]),
                        if delta(C,X,CC0) then
                                    CC:=CC0
                        else
                                    CC:=cfail
                        end,
                        append(_regCCs,[CC],_regCCs),
                        if CC=cfail then
                                    xwriteln(0,"constraint violated"),
            append(_regGs,[1.0e+12],_regGs)
                        else
                                    exp_to_text(CC,TCC),
                        if member(TCC,H) or CC=C then
                                    xwriteln(0,"repeated state"),
                        append(_regGs,[1.0e+12],_regGs)
                        else
                                    goal(CC,V),
                        append(_regAAs,[X],_regAAs),
                        append(_regGs,[V],_regGs)
                        end
            end,
            assign(_regAs,Xs),
            Xs=[],!,
_regGs(Gs),
if status(trace,-3) then
xwriteln(0,"[goal(As)   ]=",Gs),stop
end,
            Min:=min(Gs,I),
            if Min=1.0e+12 then
                        A:=[],
                        C2:=cfail,
                        AAs:=[]
            else
                        project(As,I+1,A),
                        _regCCs(CCs),
                        project(CCs,I+1,C2),
                        _regAAs(AAs0),
                        minus(AAs0,[A],AAs)
            end,
            retract([_regAs,_regGs,_regAAs,_regCCs]);

getGammaId(Id):-
            gammaId(Id),
            assign(gammaId,Id+1);
```

Appendix 5.3 Proof of Proposition 5.1

Let $C' = (Y/\ker\omega)$. Then ω is identified as $\omega(y) = [y]$. Let $\delta' : C' \times A \to C'$, where δ' is defined as

$$\delta'([y], a) = [\delta(y, a)].$$

Note that if $[y] = [y']$ or $\omega(y) = \omega(y')$, then $\omega(\delta(y, a)) = \omega(ya) = \omega(y'a) = \omega(\delta(y', a))$ for any a. That is, $[\delta(y, a)] = [\delta(y', a)]$. Then, the left diagram is commutative. Similarly, let $\lambda' : C' \times A \to A$ be given by

$$\lambda'([y], a) = a.$$

Then the right diagram is also commutative. Then ω is an automaton homomorphism from $\langle\delta, \lambda\rangle$ to $\langle\delta', \lambda'\rangle$. Q.E.D.

Appendix 5.4 Proof of Proposition 5.2

Because $c_0 \in C_k$, there is a_1 such that $\delta(c_0, a_1) = c_1 \in C_{k-1}$ and $c_1 \in \text{genA}^*(c_0)$. Similarly, if $c_p \in C_{k-p}$, there is a_{p+1} such that $\delta(c_p, a_{p+1}) = c_{p+1} \in C_{k-p-1}$ and $a_{p+1} \in \text{genA}^*(c_p)$. Then, $\delta(c_0, a_1 a_2 \cdots a_k) \in C_0 = \{c_f\}$. Q.E.D.

Appendix 5.5 Proof of Proposition 5.3

Let $y = a_p a_{p-1} \cdots a_1$ be such that $\delta(c_0, y) = c_f$. Let $\delta(c_0, a_p a_{p-1} \cdots a_2) = c_1$, $\delta(c_0, a_p\, a_{p-1} \cdots a_3) = c_2, \ldots, \delta(c_0, a_p) = c_{p-1}$.

Because $\delta(c_1, a_1) = c_f, c_1 \in C_1$. If $c_0 \notin C_1$, there must be a first $c_{i1} \in C_2$ such that $c_j \in C_1$ for $j < i1$. Similarly, if $c_0 \notin C_2$, there must be a first $c_{i2} \in C_3$ such that $c_j \in C_1 \cup C_2$ for $i1 \leq j < i2$. In this way, we can generate $i1, i2, \ldots, ik$ such that $c_{ip} \in C_{p+1}$ and $c_{ik} = c_0$. Q.E.D.

Appendix 5.6 Proof of Proposition 5.4

Suppose

$$\begin{aligned}&\min\{\text{goalElement}(a) + \text{idGoal}_{p-1}(\delta(c, a)) | a \in \text{genA}^*(c)\}\\ &\quad = \text{goalElement}(a^*) + \text{idGoal}_{p-1}(\delta(c, a^*))\end{aligned}$$

for some $a^* \in \text{genA}^*(c)$. Let $c' = \delta(c, a^*)$. Then, $c' \in C_{p-1}$ because $a^* \in \text{genA}^*(c)$. Furthermore, $(c, a^*) \in \text{inv}\delta(c')$. Consequently,

$$\begin{aligned}&\text{goalElement}(a^*) + \text{idGoal}_{p-1}(\delta(c, a^*)) \geq \min\{\text{goalElement}(a)\\ &\quad + \text{idGoal}_{p-1}(c') | (c, a) \in \text{inv}\delta(c') \& c' \in C_{p-1}\}.\end{aligned}$$

Conversely, suppose

$$\begin{aligned}&\min\{\text{goalElement}(a) + \text{idGoal}_{p-1}(c') | (c, a) \in \text{inv}\delta(c') \& c' \in C_{p-1}\}\\ &\quad = \text{goalElement}(a^*) + \text{idGoal}_{p-1}(c'^*),\end{aligned}$$

where $(c, a^*) \in \text{inv}\delta(c'^*)$ and $c'^* \in C_{p-1}$. Then, $\delta(c, a^*) = c'^*$. Because $c \in C_p$, $a^* \in \text{genA}^*(c)$. Consequently,

$$\min\{\text{goalElement}(a) + \text{idGoal}_{p-1}(\delta(c, a)) | a \in \text{genA}^*(c)\}$$
$$\leq \text{goalElement}(a^*) + \text{idGoal}_{p-1}(c'^*). \qquad \text{Q.E.D.}$$

Appendix 5.7 Proof of Proposition 5.5

It is obvious that for any $c \in C_p$ there is $a \in \text{genA}^*(c)$ such that $\delta(c, a) = c' \in C_{p-1}$. Furthermore, for any $a \in \text{genA}^*(c)$, $\delta(c, a) \in C_{p-1}$. Because

$$\text{idGoal}_p(c) = \min\{\text{goalElement}(a) + \text{idGoal}_{p-1}(\delta(c, a)) | a \in \text{genA}^*(c)\},$$

the following holds for some $a^* \in \text{genA}^*(c)$:

$$\text{idGoal}_p(c) = \text{goalElement}(a^*) + \text{idGoal}_{p-1}(\delta(c, a^*)).$$

Suppose $\text{idGoal}_p(c) = \text{goalElement}(a^*) + \text{idGoal}_{p-1}(\delta(c, a^*))$ for a^* in A. Because idGoal_{p-1} is defined for $\delta(c, a^*)$, $\delta(c, a^*) \in C_{p-1}$ and hence $a^* \in \text{genA}^*(c)$. If

$$\min\{\text{goalElement}(a) + \text{idGoal}_{p-1}(\delta(c, a)) | a \in \text{genA}^*(c)\} < \text{goalElement}(a^*)$$
$$+ \text{idGoal}_{p-1}(\delta(c, a^*)),$$

then

$$\text{idGoal}_p(c) = \min\{\text{goalElement}(a) + \text{idGoal}_{p-1}(\delta(c, a)) | a \in \text{genA}^*(c)\}$$
$$< \text{goalElement}(a^*) + \text{idGoal}_{p-1}(\delta(c, a^*)) = \text{idGoal}_p(c),$$

which is a contradiction. Q.E.D.

Appendix 5.8 Proof of Theorem 5.3

Let $C^p = \{c_0, \pi_\sigma(c_0), \ldots, \pi_\sigma^p(c_0)\}$. If $C^p = C$, $C^p \cap C_f \neq \varnothing$. Suppose $C^p \neq C$. Because σ is loop free, $C^1 \subsetneq C^2 \subsetneq \cdots \subsetneq C^p$. Because C is finite, $C^p \cap C_f \neq \varnothing$ for some p. Q.E.D.

Appendix 5.9 Proof of Proposition 5.6

We use mathematical induction.

(i) Suppose $\alpha = a \in \text{legalAs}(c)$. Then $a \in \text{genA}(c)$ and $\text{constraint}(\delta(c, a)) = \text{true}$. Let $c' = \delta(c, a)$. Due to the definition of C_{p+1},

$$c' \in C_{p+1} \vee c' \in \cup\{C_r | r \leq p\}.$$

If $c' \notin C_{p+1}$, then $(\exists r)(c' \in C_r)$.
(ii) Suppose if $\text{length}(\alpha) \leq l$, then $\delta(c, \alpha) \in C_q$ for some q. Let $\beta = \alpha a$ and $\text{length}(\beta) = l + 1$ where $a \in \text{genA}(\delta(c, \alpha))$ is arbitrary. The induction hypothesis implies $\delta(c, \alpha) \in C_q$ for some q. Then, the argument (i) implies that $\delta(\delta(c, \alpha), a) \in C_{q+1}$ or $\delta(\delta(c, \alpha), a) \in C_r$ for some r. Q.E.D.

Appendix 5.10 Proof of Proposition 5.7

Because of Proposition 5.6, k exists such that $\delta(c_0, \alpha) = c' \in C_k$. Because $c' \in C_k = \{c' | a \in \text{genA}(c)\ \&\ \delta(c, a) = c' \& c \in C_{k-1}\ \&\ \text{constraint}(c') = \text{true}\}\} - \cup\{C_r | r \leq k-1\}$,

$$c' = \delta(c_{k-1}, a_k) \& c_{k-1} \in C_{k-1} \& a_k \in \text{genA}(c_{k-1}) \& \text{constraint}(c') = \text{true}.$$

Because $c_{k-1} \in C_{k-1}$, c_{k-2} and a_{k-1} exist such that

$$c_{k-1} = \delta(c_{k-2}, a_{k-1}) \& c_{k-2} \in C_{k-2} \& a_{k-1} \in \text{genA}(C_{k-2}) \& \text{constraint}(c_{k-1}) = \text{true}.$$

In this way we can find $a_1, a_2, \ldots, a_k, c_1, \ldots, c_{k-1}$ such that

$$\delta(c_0, a_1) = c_1 \in C_1 \& a_1 \in \text{genA}(c_0) \& \text{constraint}(c_1) = \text{true},$$

$$\delta(c_1, a_2) = c_2 \in C_2 \& a_2 \in \text{genA}(c_1) \& \text{constraint}(c_2) = \text{true},$$

$$\bullet$$

$$\bullet$$

$$\delta(c_{k-1}, a_k) = c' \in C_k \& a_k \in \text{genA}(c_{k-1}) \& \text{constraint}(c') = \text{true}.$$

Consequently, $\alpha' = a_1 \cdots a_k \in \text{legalAs}(c_0)$, $\delta(c_0, a_1 \cdots a_i) \in C_i$ and $\delta(c_0, \alpha') = c'$.
Q.E.D.

Appendix 5.11 Proof of Theorem 5.4

The if part is obvious. The only if part comes from Proposition 5.7, where $c' \in C_f$.
Q.E.D.

Appendix 5.12 Proof of Proposition 5.8

Because c is a solvable state, Proposition 5.7 asserts that there is $\alpha = a_1 \cdots a_k \in$ legalAs(c) such that

$$\delta(c, a_1) = c_1 \in C_{p+1},$$

$$\delta(c_1, a_2) = c_2 \in C_{p+2},$$

$$\bullet$$

$$\bullet$$

$$\delta(c_{k-1}, a_k) = c_k \in C_f.$$

Because $a_1 \in \text{genA}(c)$ and $\delta(c, a_1)$ is a solvable state, $c < \delta(c, a_1)$ implies goal(c) > goal($\delta(c, a_1)$) $\geq$ goal($\delta(c, a^*)$), that is, $c < \delta(c, a^*)$. Then, the definition C_{p+1} implies that $\delta(c, a^*) \in C_{p+1}$.
Q.E.D.

References

Mesarovic, M. D. and Takahara, Y. (1989) "Abstract Systems Theory," *Lecture Notes in Control and Information Sciences*, Springer.

Bridge, J. (1977) *Beginning of Model Theory*, Oxford University Press.

Takahara, Y., Liu, Y., and Yano, Y. (2003) "Formal Systems Approach to Solver Design-Hill Climbing Method with Push-Down Stack," *J. of Systems Science and Systems Engineering* **12**(2).

Takahara, Y., Liu, Y., and Yano, Y. (2004) "A Systems Theoretic Approach to the Design and Implementation of a Solver Component for a Management Information System," *Int. J. of General Systems* **33**(1).

Takahara, Y. and Saitou, T. (2005) "An Operational Method for User Model Building of Solver System by Systems Approach," *J. of JASMIN* **13**(4), 1–16 (in Japanese).

Part IV

Solver System Applications

6

Traveling Salesman Problem: E-C-C Problem

This chapter examines the case E-C-C of the problem classification presented in Table 4.1. The famous traveling salesman problem is used as a typical example of the case. We develop extSLV for the problem following the development procedure of Chapter 4.

The stdDPsolver will be used for the present system as the goal-seeker.

6.1 Traveling Salesman Problem

This chapter discusses the traveling salesman problem as a typical example of the E-C-C case. It is the simplest problem according to the problem classification of the model theory approach and is also the most classical topic in the problem-solving literature. Because the problem of the E-C-C or the final state is explicitly identified from the problem-solving environment (PSE), the modified dynamic programming (DP) method will be adopted to attack the problem.

First, the general method of Chapter 4 will be applied to derive a basic extended solver (extSLV) for the problem and then tuning will be done for the goal-seeker.

6.2 User Model of Traveling Salesman Problem for DP Goal-Seeker

A user model for the traveling salesman problem is built based on Fig. 4.6.

6.2.1 Drawing Input–Output Block Diagram

At the first stage, the input and the output of the problem must be specified conceptually. Figure 6.1 shows an input–output block diagram of the traveling salesman problem.

The input is a problem specification environment that provides information for the problem structure. The input to the traveling salesman problem must give which towns

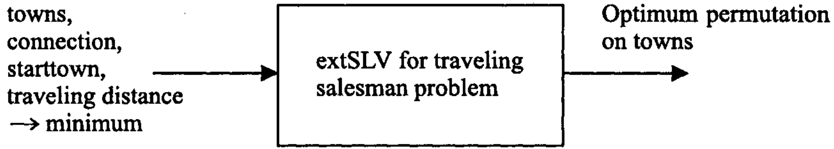


Fig. 6.1. Input–output block diagram of traveling salesman problem.

the salesman must visit, how the towns are connected, the traveling distances between the towns, and which town the salesman must start from and return to. The problem is to find the shortest route that visits every town once and only once. The solution, which is an output of the system, is the sequence of towns traveled.

6.2.2 Input–Output Specification in Set Theory

At this stage the input and the output of the first stage are described in set-theoretic terms. The description forms the basis of the later formulation. Figure 6.2 shows stage 2 of the traveling salesman problem.

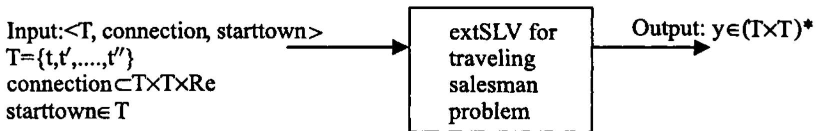


Fig. 6.2. Input–output specification in set-theoretic treatment of the traveling salesman problem.

(i) Input Specification

The input is given by a structure

$$\langle T, \text{ connection, starttown}\rangle,$$

where $T = \{t, t', \ldots, t''\}$ is a set of towns, connection $\subset T \times T \times \text{Re}$ is a relation that specifies the connections among the towns and the distances between them. An element $(t, t', r) \in$ connection implies that the towns t and t' are directly connected and the distance between them is r. Finally, starttown $\in T$ is the town from which the salesman starts and to which he must return.

(ii) Output Specification

The output set Y is a subset of $(T \times T)^*$, the free monoid of $T \times T$, i.e.,

$$Y \subset (T \times T)^*.$$

If $y = (t_1, t_2)(t_2, t_3) \cdots (t_{k-2}, t_{k-1})(t_{k-1}, t_k)$, the salesman is supposed to visit towns in the sequence $(t_1, \ldots, t_k)$. Of course, there should be no repetition in the sequence. A pair (t_{i-1}, t_i) will be called an action. The action (t_{i-1}, t_i) implies that the salesman visits the town t_i after t_{i-1} if it is feasible. A real solution is an optimum element of Y.

6.2.3 Process Specification as Automaton

(i) Action Set

The action set A is given by the problem specification environment (PSE)

$$A = T \times T.$$

(ii) State Transition Function

According to the general procedure of Chapter 5, the state set is given by $(T \times T)^*$. However, for the traveling salesman problem, a desirable state reduction mapping $\omega : Y \to C$ is easily found. Let

$$\omega(y) = (\text{last}(y), T - \text{set}(y)) \in T \times \wp(T)$$

and

$$\omega(\Lambda) = (\text{starttown}, T),$$

where last(y) is the last town of y and set(y) is the set of towns in y. If $y = (t_1, t_2)(t_2, t_3)$, last(y) $= t_3$ and set(y) $= \{t_1, t_2, t_3\}$. Then it is easy to check that $(\forall y, y', a)(\omega(y) = \omega(y') \to \omega(ya) = \omega(y'a))$ holds for $a \in A$. Let

$$y = y_1 \cdots y_m(t_1, t),$$

$$y' = y'_1 \cdots y'_n(t_2, t),$$

$$a = (t, t'),$$

where (t_1, t) and (t_2, t) are the last actions.

Let

$$\omega(y) = \omega(y').$$

Then,

$$(t, T - \text{set}(y)) = (t, T - \text{set}(y')).$$

Let

$$\omega(ya) = (t', T - \text{set}(ya))$$

and

$$\omega(y'a) = (t', T - \text{set}(y'a)).$$

Because $T - \text{set}(y) = T - \text{set}(y')$,

$$T - \text{set}(ya) = T - \text{set}(y'a),$$

or $\omega(ya) = \omega(y'a)$.

Furthermore, for any $c_f \in C_f$, we have $\omega(c_f) = (\text{starttown}, \varnothing)$ or $\omega(C_f) = \{(\text{starttown}, \varnothing)\}$. Then, if $y \in \omega^{-1}(\{(\text{starttown}, \varnothing)\})$, we have $\omega(y) = (\text{starttown}, \varnothing)$, i.e., $y \in C_f$. (last(y), T − set(y)) will be used as a state for the traveling salesman problem.

Let

$$C = T \times \wp(T).$$

The state transition function $\delta : C \times A \to C$ is then

$$\delta((t, T'), (t, t')) = (t', T' - \{t'\}).$$

If an action is (t, t'), the salesman travels from the town t to the town t', which becomes the last town visited and the remaining towns are $T' - \{t'\}$.

(iii) Output Function

The output function $\lambda : C \times A \to A$ is

$$\lambda(c, a) = a.$$

(iv) genA

The allowable activity function is

$$\text{genA}((t, T')) = \{(t, t') | t' \in T' \& (\exists v)((t, t', v) \in \text{connection})\}.$$

An action (t, t') is allowed for a state (t, T') if t is connected to t'.

(v) constraint

The constraint is

$$\text{constraint}((t, T')) = \text{true} \leftrightarrow t \notin T' \text{ or } (t = \text{starttown and } T' = T).$$

Note that $t \notin T'$ is an obvious requirement. If the last town visited is the starttown, we have an exception for the requirement.

(vi) Initial State

An initial state c_0 is

$$c_0 = (\text{starttown}, T).$$

(vii) Final State

A final state c_f is

$$c_f = (\text{starttown}, \varnothing).$$

Here C_f is a singleton set.

(viii) Inverse Stopping Condition

The inverse stopping condition invst: $C \to \{\text{true, false}\}$ is

$$\text{invst}(t, R) = \text{true} \leftrightarrow R \neq \varnothing \text{ and } t = \text{starttown}.$$

(ix) Inverse Transition

The inverse transition $\text{inv}\delta : C \to \wp(C)$ is

$$\text{inv}\delta(c) = \{c' | (\exists a \in A)(\delta(c', a) = c \& a \in \text{genA}(c') \& \text{constraint}(c'))\}.$$

6.2.4 DP Optimization Formulation

The traveling salesman problem is a closed-goal problem. The goal is given as an evaluation of the output. Let goalElement : $A \to \text{Re}$ be such that

$$\text{goalElement}((t, t')) = \text{distance}(t, t'),$$

where goalElement$((t, t'))$ is the distance between t and t' given by the connection relation of the PSE.

Then, if $y = (t_1, t_2)(t_2, t_3) \cdots (t_{k-2}, t_{k-1})(t_{k-1}, t_k)$,

$$\text{goal}(y) = \text{goalElement}((t_1, t_2)) + \text{goalElement}((t_2, t_3)) + \cdots + \text{goalElement}((t_{k-1}, t_k)).$$

In this case, the function goal satisfies the additivity condition, i.e.,

$$\text{goal}(ya) = \text{goal}(y) + \text{goalElement}(a).$$

A dynamic optimization formulation of the traveling salesman problem is given for the modified DP goal-seeker as

$$\textbf{user model} = \langle A, C, Y, \delta, \text{inv}\delta, \lambda, \text{genA}, \text{constraint}, \text{goalElement}, \text{invst}, c_0, c_f \rangle.$$

6.3 Implementation in extProlog

If a system developer implements delta(), invdelta(), genA(), constraint(), initialstate(), finalstate(), and goalElement() in computer-acceptable set theory to derive an implementation structure of Fig. 4.9 for the traveling salesman problem, a basic extSLV can be obtained for it by compilation. The entire user model in set theory is presented in Appendix 6.1.

The input data consists of 26 Japanese cities. This case cannot be solved by a simple backtracking algorithm of Prolog [Covington et al. 1997].

The output or a solution of the solver is presented in Fig. 6.3.

The solution is given by the variable Route in the dialog window, which starts from Tokyo and returns to it from Chiba. Figure 6.4 shows the behavior of the goal.

The horizontal coordinate corresponds to the order of the 26 towns the salesman visits, while the vertical coordinate corresponds to the value of the goal. According to the experiences induced from solver constructions, an optimal solution usually displays a smooth behavior and does not change its value abruptly. Figure 6.4 is compatible with this heuristic judgment.

```
DIALOG
],[20,17],[17,19],[19,21],[21,11]]
"Route="[tokyo,sapporo,aomori,akita,sakata,fukushima,s
endai,nagoya,fukuoka,kumamoto,kagoshima,ooita,takamatu
,okayama,hiroshima,matue,kobe,oosaka,kyoto,kanazawa,ni
igata,toyama,nagano,utsunomiya,mito,chiba,tokyo]
"You can change Function of Graph:""out_0.g"" (source/
sink)"
"Default is [sink]"
"You can change range of X-cord. or timespan(integer)"
"Default is [5]"
"You can change Type of Graph:""out_0.g"
"Type:plot,bar,pie,radar,xyplot,trajectory"
"Default is [plot]"
"MODEL:""out_0.g"" GO"
tsalesDP44.p" ends"
"normal reset state"
"May I help you?"
X>
```

Fig. 6.3. Solution of a traveling salesman problem.

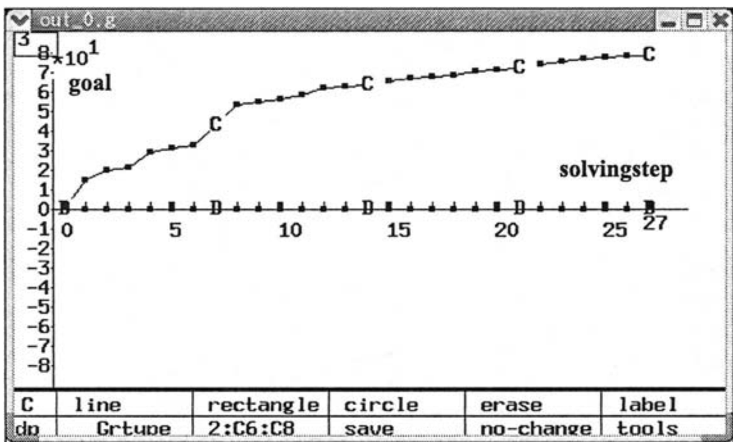


Fig. 6.4. Behavior of goal.

6.4 Tuning of Goal-Seeker

It is natural that as the number of towns increases, the computation time and the amount of memory required increase, until finally, a solution cannot be obtained. This problem comes from the following facts:

1. Because the relation connection is given as connection $\subset T \times T \times \mathrm{Re}$, the processing of genA : $C \to \wp(A)$ becomes drastically slower as the cardinality $|T|$ of T becomes larger. Note that in genA, $(t, t', D) \in$ connection must be decided and can be determined by checking each element of connection.

2. Because $C = T \times \wp(T)$, we have $|C| = n \times 2^n$, where $n = |T|$. Then, the number of the checking operations of the ideal goal elements increases exponentially as n increases.
3. Because the function invdelta is given as invdelta : $C \to \wp(C)$, computation of the backward operation of the DP method increases rapidly as n increases.

We cannot control the size of T or C. However, execution speed can be made faster by the following tuning.

In order to face the first issue, the set of elements of the connection is reorganized. That is, instead of using an element (t, t', r) directly, a new relation connect(t) $\subset (T \times \text{Re})^*$ is defined for each $t \in T$ as

$$\text{connect}(t) = \{(t', r) | (t, t', r) \in \text{connection}\}.$$

Then, in order to check $(t, t', r) \in$ connection we have only to check the small relation connect(t). The predicate connect() is generated by transDB() as the preprocess in Appendix 6.1.

For the second issue, a goal element goal$((t, T'), r) \in$ goal is represented in the form goal.$t(T', r)$. Then, checking of goal$((t', R), r')$ can be done by directly checking the predicate goal.t'(). This modification can provide a remarkable improvement in the execution speed.

For the third issue, it should be noted that

$$\begin{aligned}\text{invdelta}(C') &= \cup\{\{c'|(\exists a)(a \in \text{genA}(c') \,\&\, \delta(c', a) = c)\}, c \in C'\}\\ &= \cup\{\text{first}(\text{connect}(t)) \times \{R \cup \{t\}\} | (t, R) \in C'\},\end{aligned}$$

where first(connect(t)) $= \{t'|(\exists r)((t, t', r) \in \text{connection})\}$. Then we can avoid computing the existential quantifier operation by using connect(t). The direct product operation of sets is available in computer-acceptable set theory. This can avoid a large time-consuming operation associated with the existential quantifier.

Obviously, invdelta recursively produces a large subset of the state set. If the produced subsets are processed in a naive way, the required computation time easily becomes prohibitive. In order to avoid this difficulty, evaluation of states at the goal generation phase should be restricted to relevant ones. The selection criteria of the relevant states are determined by heuristics. In particular, it should be noted that the number of states is controlled by the parameter selectN of stdDPsolver62.p. Its value is tuned to a given problem by the trial and error method.

With the above improvements, a traveling salesman problem of 26 towns was successfully solved by the DP method on a PC in less than 2 s. The 26 towns constitute a realistic route for a 1-month traveling period.

Appendix 6.1 User Model for Traveling Salesman Problem

```
/*tsales2.set*/
/*this is a solver for traveling salesman problem*/
/*state=[last_visited_city,remaining_cities]*/
```

```
.func([getRoute,getDistance,initialstate,goal,delta,invdelta.sort,
       getconnect,invproject,project2,genA]);

delta([T,R],[T,T2])=C2 <->
     R2:=minus(R,[T2]),
     C2:=[T2,R2],
     constraint(C2);

invdelta([C2,R2])=CADs <->
        R22:=append([C2],R2),
        R:=sort(R22),
        getconnect(C2,L),
        LT:=transpose(L),
        Ts:=project(LT,1),
        Ts2:=minus(Ts,R),
        (Ts2<>[]) ->
                  (
                  Is:=invproject(Ts,Ts2,"m"),
                  Ds:=project(LT,2),
                  Ds2:=project2(Ds,Is),
                  Cs:=product(Ts2,[R]),
                  CADs:=transpose([Cs,Ds2])
                  )
         .otherwise
                  (
                  C:=depCity.g,
                  TSize:=totalCSize.g,
                  (cardinality(R)=TSize and
                   project(Ts,I,C)) ->
                  (
                  Ds:=project(LT,2),
                  D:=project(Ds,I),
                  CADs:=[[[C,R],D]]
                  )
         .otherwise
                  (
                  CADs:=[]
                  )
         );

genA([T,R])=As <->
    getconnect(T,L),
    LT:=transpose(L),
    Ds:=project(LT,1),
    Ds2:=intersection(Ds,R),
    As:=product([T],Ds2);

constraint([L,R]) <->
          C:=depCity.g,
          TCs:=totalC.g,
          (notmember(L,R) or (L=C and
          cardinality(R)=cardinality(TCs)));

invst([[C,R]|Xs]) <->
     R<>[],
     C=depCity.g;
```

```
initialstate()=Z <->
            TCs:=totalC.g,
            C=depCity.g,
            Z:=[C,TCs];

finalstate(Cf) <->
          C:=depCity.g,
          Cf=[C,[]];

preprocess() <->
          retract([.connect]),
          totalC.g:=[],
          TId.g:=1,
          transDB(),
          TCs0:=totalC.g,
          totalCSize.g:=cardinality(TCs0),
          delGoal(),
          xwriteln(0,"Which city do you want to
                   start?",TCs),
          xread(0,DepCity0),
          Itokyo:=invproject(TCs0,"tokyo"),
          DepCity:=Itokyo,
          depCity.g:=DepCity;

delGoal() <->
          I:=TId.g,
          procC("createindex",[1,I-1],[Js]),
          Z:=defSet(pdelGoal(y,J,[]),[J,Js]);
pdelGoal(y,J,[]) <->
          concat(["goal",".",J],GoalJ),
          retract([GoalJ]);

postprocess() <->
          SS:=SS.g,
          TCs:=totalC.g,
          Route:=getRoute(TCs,SS),
          xwriteln(0,"Route=",Route),
          Dis0:=getDistance(TCs,SS),
          append([0],Dis0,Dis),

          show1(Dis,"plot"),
               delconnect();
               delconnect() <->
               I:=totalCSize.g,
               procC("createindex",[1,I],[Js]),
               Z:=defSet(pdelconnect(Y,J,[]),[J,Js]);
pdelconnect(Y,J,[]) <->
                   concat([.connect,".",J],Con),
                   retract([Con]);

getDistance(TCs,SS)=Dis <->
         Ds.g:=[],
         Ds:=defSet(pDistance(Y,X,[TCs]),[X,SS]),
//xwriteln(0,"Ds.g=",Ds.g),
         Dis:=Ds.g;
pDistance(Z,[X,Y],[TCs]) <->
         project(TCs,X,XN),
         project(TCs,Y,YN),
```

```
          distance(XN,YN,D),
          Z:=D,
          (Ds.g=[]) ->
                   (
                   Ds.g:=[Z]
                   )
          .otherwise
                   (
          Ds.g:=append(Ds.g, [project(Ds.g,0)+Z])
                   );

distance(X,Y,D) <->
          connection(X,Y,D),!;
distance(X,Y,D) <->
          connection(Y,X,D);

getRoute(TCs,SS)=Route <->
          R0:=defSet(pRoute(Y,X, [TCs]), [X,SS]),
          C0:=depCity.g,
          project(TCs,C0,CON),
          Route:=append(R0, [CON]);
pRoute(Z, [T,T2], [TCs]) <->
          project(TCs,T,TN),
          Z:=TN;

/*connection(tokyo,sapporo,15)->connection.tokyo
  (sapporo,14)*/
transDB() <->
          Cs:=listPred("connection"),
          XX:=defSet(ptransDB(Y,X, []), [X,Cs]);
ptransDB(Y, [A,B,D], []) <->
          TC0:=totalC.g,
          (project(TC0,I0,A)) ->
                  (
                  I:=I0,
                  TC1:=TC0
                  )
          .otherwise
                  (
                  getTId(I),
                  append(TC0, [A],TC1)
                  ),

          (project(TC1,J0,B)) ->
                  (
                  J:=J0,
                  TC2:=TC1
                  )
          .otherwise
                  (
                  getTId(J),
                  append(TC1, [B],TC2)
                  ),
          totalC.g:=TC2,
          L1:=getconnect(I),
          (L1<>[]) ->
                  (
```

```
                append(L1,[[J,D]],L2),
                saveconnect(I,L2)
                )
        .otherwise
                (
                saveconnect(I,[[J,D]])
                ),
        L11:=getconnect(J),
        (L11<>[]) ->
                (
                append(L11,[[I,D]],L22),
                saveconnect(J,L22)
                )
        .otherwise
                (
                saveconnect(J,[[I,D]])
                ),
        Y:=1;

getconnect(I)=L <->
        concat([.connect,".",I],Con),
        L0:=listPred(Con),
        (L0=[]) ->
                (
                L:=L0
                )
        .otherwise
                (
                L:=project(L0,1)
                );

saveconnect(I,L) <->
        concat([.connect,".",I],Con),
        assign(Con,L);

getTId(I) <->
        I:=TId.g,
        TId.g:=I+1;

/*base data*/
connection("aomori","sapporo",5);
connection("aomori","akita",1);
connection("sendai","akita",3);
connection("aomori","sendai",2);
connection("hiroshima","okayama",1);
connection("takamatu","okayama",1);
connection("kobe","okayama",1);
connection("kobe","oosaka",1);
connection("kobe","matue",1.5);
connection("sapporo","oosaka",12);
connection("sendai","oosaka",8);
connection("tokyo","oosaka",4);
connection("fukuoka","oosaka",6);
connection("kyoto","oosaka",0.5);
connection("kanazawa","tokyo",3);
connection("kanazawa","oosaka",3);
connection("nagoya","oosaka",2);
```

```
connection("kanazawa","toyama",1);
connection("kanazawa","kyoto",2);
connection("utsunomiya","tokyo",1);
connection("fukushima","mito",1);
connection("utsunomiya","mito",1);
connection("utsunomiya","nagano",2);
connection("tokyo","mito",1);
connection("chiba","mito",1);
connection("chiba","nagoya",3.5);
connection("tokyo","nagoya",3);
connection("tokyo","fukuoka",13);
connection("tokyo","sapporo",15);
connection("nagoya","fukuoka",10);
connection("nagoya","sapporo",18);
connection("fukuoka","sapporo",28);
connection("niigata","tokyo",4);
connection("niigata","kanazawa",1);
connection("nagano","tokyo",3);
connection("toyama","tokyo",6);
connection("niigata","toyama",1);
connection("nagano","toyama",1.5);
connection("chiba","tokyo",0.5);
connection("yokohama","tokyo",0.5);
connection("chiba","yokohama",1);
connection("kyoto","tokyo",4.5);
connection("kyoto","nagoya",1.5);
connection("toyama","kyoto",4);
connection("takamatu","tokyo",8.5);
connection("takamatu","nagoya",5.5);
connection("takamatu","oosaka",3);
connection("takamatu","fukuoka",6.5);
connection("toyama","matue",2);
connection("kyoto","matue",1.5);
connection("sendai","sapporo",8);
connection("sendai","tokyo",7);
connection("sendai","nagoya",10);
connection("sendai","fukuoka",17);
connection("niigata","sakata",4.5);
connection("sendai","sakata",1.5);
connection("ooita","tokyo",15);
connection("ooita","oosaka",8);
connection("ooita","fukuoka",1.5);
connection("takamatu","ooita",3);
connection("niigata","fukushima",2);
connection("utsunomiya","fukushima",5);
connection("sendai","fukushima",1.5);
connection("sakata","fukushima",2);
connection("aomori","tokyo",10);
connection("sakata","akita",8);
connection("sakata","aomori",8.5);
connection("sakata","sapporo",10);
connection("sakata","fukuoka",20);
connection("kumamoto","tokyo",18);
connection("kumamoto","nagoya",15);
connection("kumamoto","fukuoka",2);
connection("ooita","kumamoto",2);
connection("kagoshima","tokyo",18.5);
```

```
connection("kagoshima","nagoya",15.5);
connection("kagoshima","fukuoka",2.5);
connection("ooita","kagoshima",2.5);
connection("kumamoto","kagoshima",1);
connection("hiroshima","tokyo",9);
connection("okayama","tokyo",8.5);
connection("hiroshima","fukuoka",3);
connection("takamatu","hiroshima",2);
connection("matue","hiroshima",1.5);
```

The following should be noted about the model:

1. The predicate transDB() generates connect() and represents a town name in an integer. For instance, Itokyo is an integer to represent Tokyo.
2. In postprocess() the solution is obtained by _SS(SS), in which town names are represented by integers. The integers are translated into the original names by getRoute(). Here getDistance() calculates the traveling distance, which show1() displays in Fig. 6.4.

References

Covinton, M., Nute, D. and Vellino, A. (1997) *Prolog Programming in Depth*, Prentice Hall.

7

Regulation Problem: E-O-C Problem

Chapters 7 and 8 discuss a control engineering problem. Control engineering problems are interesting for three reasons. First, although control is a type of management, because a target of control engineering is a continuous dynamic physical system, a control engineering problem is naturally different from a management problem. On the other hand, the methodology of this book has been developed and aimed at management problems. However, because it is based on the concepts of general systems theory (GST), it should be a general theory, which implies that our methodology must be applicable to control problems. It is interesting to see how our theory can work as a general theory in a field that is considered outside the scope of the original intention.

Second, as the role of the data processing function of the MIS indicates, numerical processing is a basis of the MIS. If the model theory approach can handle control engineering problems that present the most typical numerical problems, we can expect that it can effectively address numerical aspects of MIS problems.

Third, targets of our extended solver (extSLV) are, in general, ill structured. However, the problems of control engineering are well structured and their models are described by differential equations. Therefore, we can expect that if an extSLV is applied to such a well-structured problem, a deep insight into the solver may be obtained because a detailed analysis is possible for a well-structured problem.

The stdPDsolver will be used for the present system as the goal-seeker.

7.1 Regulation Problem

For the sake of simplicity, a simple but typical dynamic problem shown in Fig. 7.1 will be used as the target of this chapter.

As Fig. 7.1 shows, the dynamic problem consists of a body of mass m and a spring with a spring constant of k. The body is fixed by the spring to a wall. The input of the system is a force $a()$, which is a time function, applied to the body, and the output is the displacement of the body. Suppose that an initial condition (u_0, v_0) is given arbitrarily, where u_0 is an initial displacement and v_0 is an initial velocity. Then the regulation

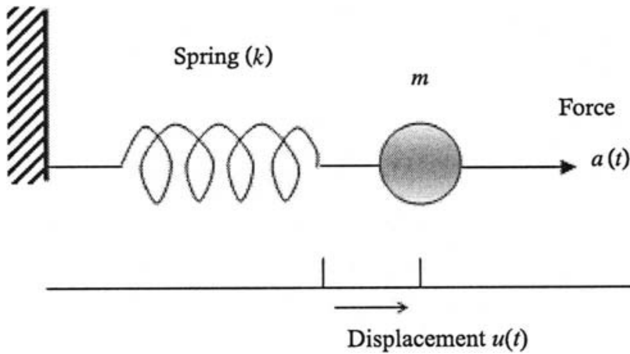


Fig. 7.1. A simple dynamic system.

problem of this chapter is to determine a force $a()$ such that the body is taken to a state $(u_f, 0)$, where u_f is also an arbitrarily given set point.

The dynamic behavior of this problem can be described by the following differential equation:

$$m\frac{d^2u}{dt^2} = a - ku.$$

The above equation can be transformed into a state space representation in the usual way. That is, the state is a vector (displacement, velocity). The representation is

$$\begin{aligned} \frac{du}{dt} &= v, \\ \frac{dv}{dt} &= -\frac{k}{m}u + \frac{1}{m}a. \end{aligned} \tag{7.1}$$

Two control engineering problems can be defined for the above problem.

(i) Feedback Law Problem

Let a desirable set point be $(u_f, 0)$. Find a feedback law: $\{(u, v)\} \to \{a\}$ that takes the system to the state $(u_f, 0)$ from an initial state (u_0, v_0) as soon as possible. The situation is that initially the system was at $(u_f, 0)$ but due to a disturbance input it was taken to the state (u_0, v_0).

For the feedback law problem, if the transition time is required to be a minimum, the problem becomes an optimization problem. We will treat the problem not as an optimization problem but as an efficient solution problem in this chapter. The efficient solution concept is introduced in Section 5.4. According to the classical control theory, a feedback law is determined as an algorithm of a PID (proportial, integral, differential) controller. A feedback law will be derived using the model theory appraoch and will be found to be different from that of the PID controller.

The following problem is a simple version of the feedback law problem.

(i)′ Regulation Problem

Let (u_0, v_0) be an initial state. Let $(u_f, 0)$ be a desirable state (u_f is a set point). Find a time function $a()$ that takes the system from (u_0, v_0) to $(u_f, 0)$.

This chapter discusses the regulation problem first and then the feedback law problem. The optimization problem is investigated in Chapter 8.

7.2 User Model of Regulation Problem for PD Goal-Seeker

Let us consider the regulation problem. Because the output is a sequence of actions $a(t)$, the problem belongs to the explicit solving-action class. Because evaluation of an output is not given by the problem, the problem will be of the open-goal type. We will show that the state space representation of (7.1) can be used as the process of the extSLV representation, and hence a target state is explicitly identified. Then, it is a closed-target problem. Consequently, the regulation problem belongs to the E-O-C class.

The E-O-C class is a large class, and many game-type problems belong to this class. For example, the Chinese checkers problem presented in Appendix 7.2 is an example. Although the regulation problem belongs to the E-O-C class, it is different from many game-type E-O-C problems. First of all, the regulation problem is numerical, while many game-type problems are symbolic. The biggest difference is that the state set of the automaton formulation for the regulation problem is an infinite set, while the usual game-type problems including the Chinese checkers problem have finite state sets. Because the state set of the regulation problem is an infinite set, we cannot apply the brute force method of checking every state or a simple backtracking algorithm to find a solution to the problem.

7.2.1 Drawing Input–Output Block Diagram

Figure 7.2 shows the first stage of the regulation problem.

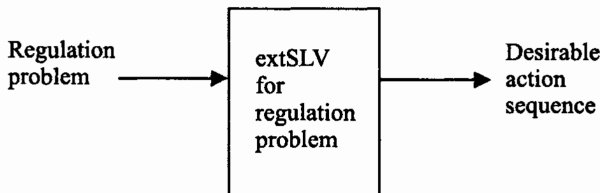


Fig. 7.2. Input–output diagram of regulation problem.

7.2.2 Input–Output Specification in Set Theory

At this stage the input and the output are described in set-theoretic terms.

(i) Input Specification

Because an extSLV of the regulation problem must be implemented on a computer, its input, the regulation problem of the differential equation model, must be transformed into a difference equation model. A difference equation model will be generated by the mean value theorem as follows:

$$\begin{aligned} du/dt &= v \to (u(t+h) - u(t))/h = v(t+\theta h)\ (0 \leq \theta \leq 1), \\ dv/dt &= -ku/m + a/m \to (v(t+h) - v(t))/h \\ &= -ku(t+\theta h)/m + a(t+\theta h)/m\ (0 \leq \theta \leq 1). \end{aligned}$$

Then, the dynamic system can be described by the following difference equation:

$$\begin{aligned} v(k+1) &= -hku(k)/m + v(k) + ha(k)/m, \\ u(k+1) &= u(k) + hv(k+1). \end{aligned}$$

Let $U_p \times V_p$ and A_p be the state set and the input set of the dynamic system, respectively, where

$$U_p = V_p = A_p = \text{Re (set of real numbers)}.$$

Let the state transition function $\delta_p : (U_p \times V_p) \times A_p \to U_p \times V_p$ of the system be

$$\delta_p((u, v), a) = (u + hv, -khu/m + v + ha/m).$$

Let an initial state be

$$c_0 = (u_0, v_0).$$

Let a desirable state c_f be

$$c_f = (u_f, 0),$$

where u_f is a set point.

Then the input of the extSLV is described by the following structure:

$$\langle U_p, V_p, A_p, \delta_p, c_0, c_f \rangle.$$

Although no further result can be derived for the input structure from here in the general case, the current case can yield an important fact. That is, we can check whether the current regulation problem has a controllability property, because it is a linear time-invariant problem [Mesarovic and Takahara, 1989]. Let $\delta_p((u, v), a) = K^{-1}(F(u, v) + ha)$, where

$$F = \begin{pmatrix} 1 & h \\ -h/m & 1 \end{pmatrix}, \quad K = \begin{pmatrix} 1 & -h \\ 0 & 1 \end{pmatrix}, \quad h = \begin{pmatrix} 0 \\ h/m \end{pmatrix}.$$

Then, $\text{rank}[K^{-1}h, K^{-1}Fh] = 2$, which states that the regulation structure has controllability, i.e.,

$$(\forall c_0)(\exists \alpha \in A_p{}^*)(\delta_p(c_0, \alpha) = c_f),$$

where $A_p{}^*$ is the free monoid of A_p. This property is naturally essential for solving the current problem.

(ii) Output Specification

Let the output set Y be

$$Y = A_p{}^*.$$

A desirable output is a selected element of Y.

7.2.3 Process Specification as Automaton

At this stage, an automaton description of the solving activity will be formalized. In order to derive an automaton formulation, a state set and an action set must be identified.

(i) Action Set

The action set A is given by the problem specification environment (PSE) as

$$A = A_p = \mathrm{Re}.$$

(ii) State Transition Function

Because $Y = A_p{}^*$, the general result of Chapter 5 states that an automaton formulation of the solving activity can be given as follows:

$$Y : \text{state set}$$

and

$$A : \text{action set}.$$

The state transition function $\delta_g : Y \times A \to Y$ is then given by

$$\delta_g(y, a) = y \cdot a,$$

where $y \cdot a$ is a concatenation of y and a. Note that δ_g is a representation of the activity process of the dynamic solving process. Conceptually, it is completely different from δ_p, the representation of the dynamics of the problem.

The output function $\lambda_g : Y \times A \to A$ is given by

$$\lambda_g(y, a) = a.$$

The above automaton formulation comes from the general arguments in Section 5.1.3. However, we can introduce a state reduction map ω. Let $\omega : Y \to U \times V$ be given by

$$\omega(y) = \delta_p(c_0, y).$$

Then, if $\omega(y) = \omega(y')$ (or $\delta_p(c_0, y) = \delta_p(c_0, y')$), we have

$$\begin{aligned}\omega(ya) &= \delta_p(c_0, ya)\\ &= \delta_p(\delta_p(c_0, y), a)\end{aligned}$$

$$= \delta_p(\delta_p(c_0, y'), a)$$
$$= \delta_p(c_0, y'a)$$
$$= \omega(y'a).$$

That is, ω satisfies the condition of the state reduction map of Chapter 5 and the left side of the commutative diagram of Fig. 7.3 is satisfied.

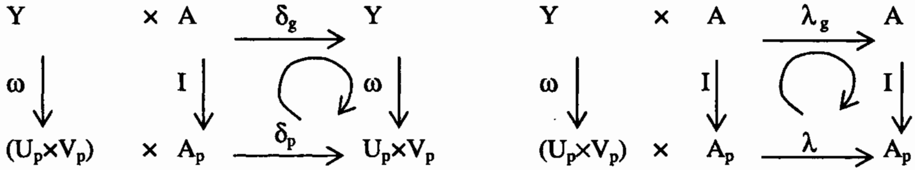


Fig. 7.3. Automaton homomorphism by reduction mapping ω.

Because the output function $\lambda_g : Y \times A \to A$ satisfies $\lambda_g(y, a) = a$, the right side of Fig. 7.3 trivially holds, where $\lambda : (U_p \times V_p) \times A_p \to A_p$ is $\lambda((u, v), a) = a$. Then we have a reduced automaton representation $\langle A, U_p \times V_p, \delta_p, \lambda \rangle$, which is equal to the usual state space representation of the control engineering problem. We will use the representation as the process representation, where δ_p, U_p, and V_p will be denoted by δ, U, and V, respectively, in the subsequent discussions.

Because a state transition function of the extSLV is, in general, a partial function, two functions, genA : $(U \times V) \to \wp(A)$ and constraint : $(U \times V) \to$ {true, false}, are introduced in (iv) and (v) below to secure its proper behavior. Here δ satisfies for $a \in$ genA(c),

$$\delta(c, a) = c' \to \text{constraint}(c') = \text{true}.$$

(iii) Output Function

The output function $\lambda : C \times A \to A$ is naturally

$$\lambda(c, a) = a.$$

(iv) genA: $C \to \wp(A)$

In the current case,

$$\text{genA}(c) = A.$$

(v) constraint: $C \to$ {true, false}

The current problem has no effective constraint. That is,

$$\text{constraint}(u, v) = \text{true}.$$

Then, δ of the regulation problem is a total function. Certainly, a constraint can be imposed on the behavior of the system if it is necessary. Chapter 8 illustrates that the constraint can be used as an effective means to solve a control engineering problem.

(vi) Initial State

The initial state is the same as that given in Section 7.2.2, i.e.,

$$c_0 = (u_0, v_0).$$

(vii) Final State

The final state is the same as the one given in Section 7.2.2, i.e.,

$$c_\mathrm{f} = (u_\mathrm{f}, 0).$$

(viii) Stopping Condition

Let st: $C \to \{\text{true, false}\}$ be given by

$$\mathrm{st}(c) = \text{true} \leftrightarrow c = c_\mathrm{f}.$$

7.2.4 PD Optimization Formulation

Because the regulation problem does not have a goal specified by the PSE, it can be used as a design parameter. Let goal: $C \to \mathrm{Re}$ (Re is the set of real numbers) be given by

$$\mathrm{goal}((u, v)) = 30 * (u - u_\mathrm{f})^2 + v^2.$$

This goal is used in Appendix 7.1. The goal is defined from a heuristic consideration that the state must ultimately reach c_f. Selection of the goal is important because the convergence time depends heavily on the selection. The weight (30, 1) is determined by trial and error.

Finally, the dynamic optimization formulation or a user model is given by

$$\textbf{user model} = \langle A, U \times V, Y, \delta, \lambda, \mathrm{goal}, \mathrm{genA}, \mathrm{constraint}, \mathrm{st}, c_0, c_\mathrm{f} \rangle.$$

7.3 Implementation in extProlog

The goal-seeker of the hill-climbing method with push-down stack is used in this chapter. When a user model is implemented in computer-acceptable set theory and is compiled into extProlog, a workable solver is realized. Appendix 7.1 presents an entire user model in set theory for the regulation problem.

It must be noted that although genA() can be theoretically defined as $\mathrm{genA}((u, v)) = A (= \mathrm{Re})$, in practice $\mathrm{genA}((u, v))$ must be a finite set. In Appendix 7.1, it is heuristically specified as

$$\mathrm{genA}((u, v)) = [-20, -10, -5, -3, -1, 0, 1, 3, 5, 10, 20].$$

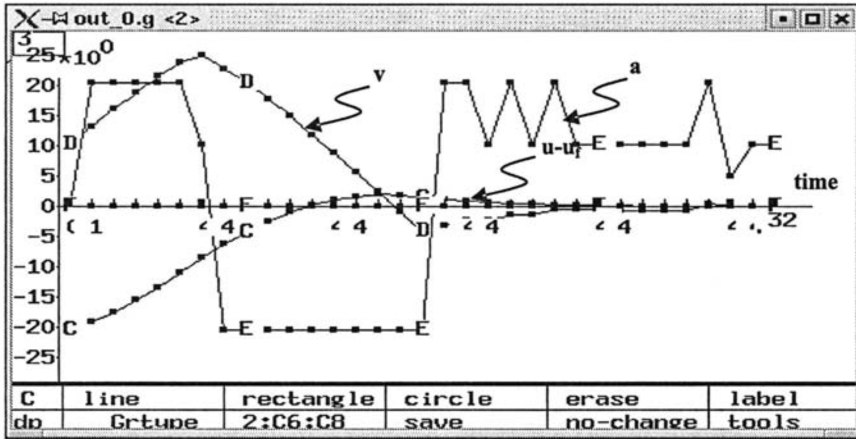


Fig. 7.4. Output of extSLV and behavior of dynamic system.

It should also be noted that δ does not represent a real-time system but is a model representation to plan a solution. The goal-seeker generates a plan of control of the dynamic system using the model and yields it as a solution.

Figure 7.4 illustrates the dynamic behavior of (u, v, a) for a solution $a()$.

The square curve in Fig. 7.4 that moves in a nonlinear fashion shows control input $a()$. The triangular curve corresponds to velocity $v()$, while the smooth-wave curve corresponds to displacement $u() - u_f$. The curves show that the behavior converges to $c_f = (u_f, 0)$.

The behavior of the control input $a()$ is quite different from that of the conventional PID controller, the output of which usually shows a smooth behavior. Although the PD method does not yield a real optimum solution (see Section 5.4), its behavior is similar to that of an optimum "bang-bang" solution. This comes from the fact that the solution is derived as a local optimum specified by the hill-climbing method, although it may not be a global optimum. This optimality is well illustrated by Fig. 7.5, which shows trajectories of (u, v).

Figure 7.5 presents four trajectories with initial states $(10, 10)$, $(0, 10)$, $(0, -10)$, and $(10, -10)$. The trajectories cover the whole domain of the state space. The final state (equilibrium point) is $(5, 0)$ which is selected in an arbitrary way. The trajectories show highly nonlinear but well-known optimal behaviors [Tu, 1994]. If the controller is linear, a trajectory approaches the final state drawing a circle, while the trajectories of Fig. 7.5 go to the equilibrium state almost directly. It should be noted that the basic behavior is restricted by the fact that if $v() > 0$, then $u()(u'() = v())$ must increase with time and if $v() < 0, u()$ must decrease. This restriction indicates that the goal-seeker by the PD method provides a fairly optimal nonlinear controller with the help of proper selection of the goal.

Figure 7.6 shows the behavior of the goal value for the case of the initial state $(10, 10)$.

It should be noted that goal() exhibits monotonically decreasing behavior.

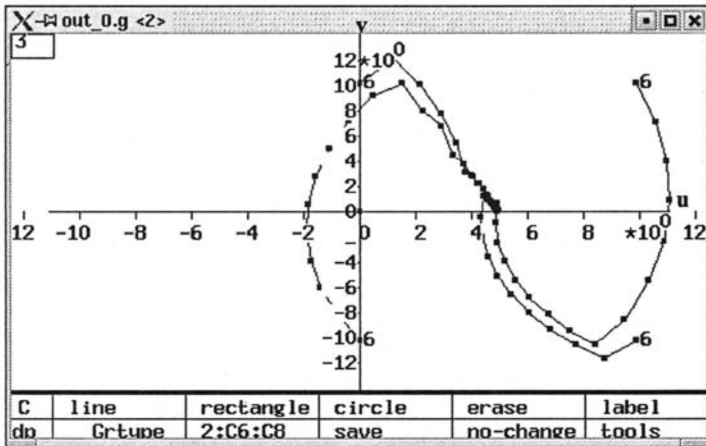


Fig. 7.5. Trajectories of (u, v).

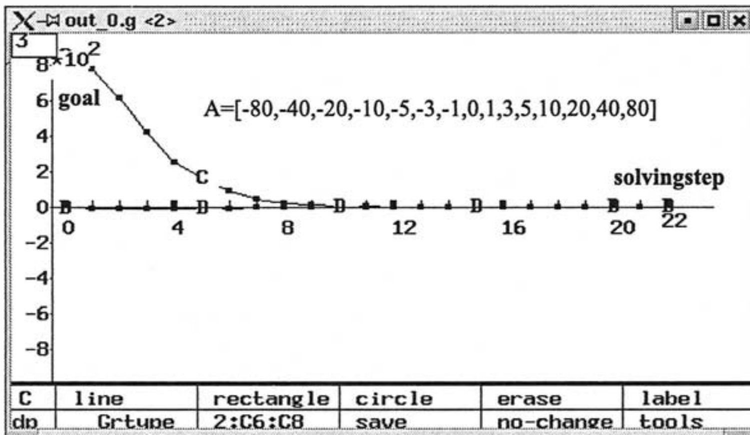


Fig. 7.6. Behavior of goal() for (10, 10).

7.4 Validity of PD Method for Regulation Problem

The fundamental difference between a usual problem of MIS and the regulation problem is, as mentioned in Section 7.2, that the state set of the former is finite, while that of the latter is infinite. If the state set is finite, a solution can be obtained by the PD method (in principle) if c_f is reachable from c_0 or the regulation problem is controllable. The PD method can cover the brute force behavior.

Because the state set is infinite, different circumstances exist, although the PD method is still valid [La Salle and Lefschetz, 1961]. This validity is retained from the following properties of the regulation problem:

1. The goal : $C \to$ Re is a continuous function and satisfies the following property:

$$\text{goal}(c) \geq 0 \text{ and } \text{goal}(c) = 0 \leftrightarrow c = c_f,$$

   where $C = U \times V$ is a state set.
2. Suppose $a_0, a_1, a_2, \ldots$ is an action sequence given by the PD method. Let

$$c_i = \delta(c_0, a_0 \cdots a_{i-1}) \quad (i = 1, 2, \ldots).$$

   Then the sequence goal(c_0), goal(c_1), ... decreases monotonically, i.e.,

$$\text{goal}(c_0) \geq \text{goal}(c_1) \geq \cdots,$$

   if goal(c_i) is small. (Refer to Fig. 7.6.) This property comes from the facts that a_i is given by the hill-climbing method with respect to goal(c) and that no backtracking can occur. (No backtracking occurs because the constraint is free.)
3. If goal(c) > 0, there is $a \in As$ such that goal($\delta(c, a)$) < goal(c) holds. This is due to the fact that the function goal() is of quadratic form and δ is a linear function.

Then, validity of the PD method can be argued in the following way: 2 implies that there is r such that $\lim_{i\to\infty} \text{goal}(c_i) = r \geq 0$. Actually, 3 implies $r = 0$. Then, 1 implies that $\lim_{i\to\infty} c_i = c_f$.

7.5 Feedback Law Problem and Case-Based Reasoning

Using the above results, let us investigate the feedback law problem. The input–output diagram of this problem is illustrated in Fig. 7.7.

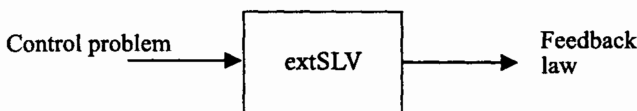


Fig. 7.7. Feedback law problem.

In general, a feedback law is represented by the following function:

$$fb : U \times V \to A.$$

Let

$$c_i = \delta(c_{i-1}, fb(c_{i-1})) \quad (i = 1, 2, \ldots),$$

where $c_i \in U \times V$. Then the following relation is required of a valid feedback law: for any $c_0 \in U \times V$,

$$\lim_{i\to\infty} c_i = c_f \tag{7.2}$$

holds.

Let the output set FB be given by

$$\mathrm{FB} = \{fb | fb : C' \to A, C' \subset U \times V, fb \text{ satisfies (7.2) with respect to } C'\}.$$

A construction of the extSLV of the feedback law problem is given by the extSLV of the regulation problem.

Let $\sigma : C \to A^*$ be given by

$$\sigma(c) = \text{a solution of the regulation problem with respect to the initial state } c,$$

where $C \subset U \times V$ and $c \in C \to$ the regulation problem has a solution with respect to c. In general, $C \neq U \times V$, and C depends on the specification of A. Let $fb : C \to A$ be such that

$$fb(c) = \sigma(c)(0),$$

where $\sigma(c)(0)$ is the first element of $\sigma(c)$. If $\sigma(c) = a_0 a_1 \cdots a_n \cdots$, then $fb(c_{i-1}) = a_i$ and $c_i = \delta(c_{i-1}, a_{i-1})$ hold, that is, (7.2) holds or $fb \in \mathrm{FB}$.

In the real implementation fb is given by the case-based reasoning method [Russel and Norvig, 1995]. Let

$$fb(\sigma, c_0) = \{(c_i, a_i) | \sigma(c_0) = a_0 a_1 \cdots a_n \cdots, c_i = \delta(c_{i-1}, a_{i-1})\}.$$

Let

$$\text{Case} \subset C : \text{set of cases.}$$

Then a feedback law fb is given by Case as

$$fb = \cup\{fb(\sigma, c_0) | c_0 \in \text{Case}\}.$$

Figure 7.4 presents the relation $\{((u, v), a)\}$, which corresponds to $fb(\sigma, c_0)$ or a subset of the feedback law.

The biggest advantage of the model theory approach is that the functions genA, goal, and constraint can be flexibly defined so that a nonlinear controller can be designed. (See Chapter 8.) This is certainly beyond the scope of traditional control theory.

Appendix 7.1 User Model for Regulation Problem

```
/*regulation.set*/
/*regulator problem*/
/*mechanical system*/
/*E-O-C problem*/
/*mu"=a-ku*/
/*u'=v*/
/*v'=-ku/m+a/m*/

.func([delta,genA,initialstate,setPoint,_Solf]);
```

```
delta([U,V],A)=[U2,V2] <->
        V2:=-0.1*U+V+0.1*A,
        U2:=U+0.1*V2,
        constraint([U2,V2]),!;

genA(C)=As <->
        As:=[-20,-10,-5,-3,-1,0,1,3,5,10,20,40];
constraint(C) <->
        C=C;

initialstate()=C0 <->
        C0:=[0,0];

setPoint()=S <->
        S:=10;

goal([U,V])=Re <->
        S:=setPoint(),
        Re:=30*(U-S)*(U-S);

st([U,V]) <->
        S=setPoint(),
        abs(S-U)+abs(V)<0.1;

postprocess() <->
        C0:=initialstate(),
        Solf:=_Solf(),
        genControlLaw(C0,Solf,CLaw),
        xwriteln(0,"CL=",CLaw);

genControlLaw(C,[],[]) <->!;
genControlLaw(C,[A|As],[[E,A]|Ls]) <->
        S:=setPoint(),
        E:=C-[S,0],
        CC:=delta(C,A),
        genControlLaw(CC,As,Ls);
```

In Appendix 7.1 the parameters are as follows:

$$h = 0.1,$$
$$k/m = 1,$$
$$1/m = 1.$$

The predicate postprocess() generates a control law using a regulation problem solution Solf given by _Solf().

Appendix 7.2 Chinese Checkers Problem

This appendix introduces the Chinese checkers problem. Although the problem itself would not be the target of a management information system, it is interesting as a nontrivial E-O-C problem and is a typical example of a game problem, for which

the model theory approach is quite effective. However, this is the only game problem presented in this book.

The Chinese checkers problem describes a board game with the initial state as shown in Fig. 7.8, where 32 tokens (black disks) are arranged on grid points of the board.

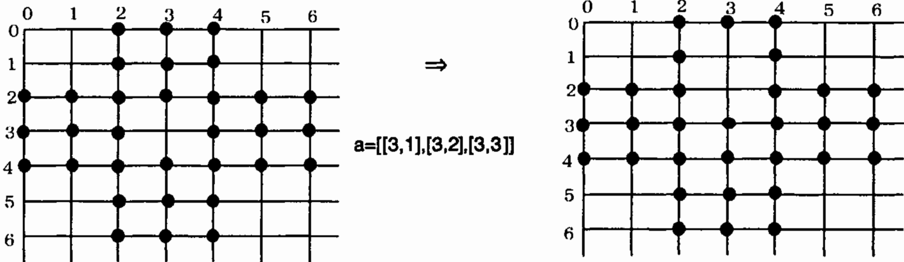


Fig. 7.8. Initial state for Chinese checkers problem.

A token can be moved horizontally or vertically to an empty grid point by skipping one token, and the skipped token is removed. Tokens may be placed only on the 32 grid points occupied in the initial state and the central grid point. Let the grid points be represented by their coordinates. In the initial state, four tokens on [3, 1], [1, 3], [3, 5], and [5, 3] can be moved. Suppose the token on [3, 1] is moved. The initial arrangement of tokens is then transformed into that shown on the right figure of Fig. 7.8. The token on [3, 2] is removed and the token on [3, 1] is moved to [3, 3]. Let the action be denoted by [[3, 1], [3, 2], [3, 3]].

The problem is to find the sequence of actions that transforms the initial state into a state that leaves only one token on the board. Let the user model of the Chinese checkers problem be formalized as follows.

Let

$$I_6 = \{0, 1, 2, 3, 4, 5, 6\}.$$

The action set A is then given by

$$A \subset (I_6 \times I_6)^3,$$

where, for example, $a = [[3, 1], [3, 2], [3, 3]] \in A$ produces the state transition shown in Fig. 7.8. Let $A0$ be defined as

$$\begin{aligned} A0 = \{&[[[0, 3], [1, 3], [2, 3]], [[0, 4], [1, 4], [2, 4]], [[4, 0], [4, 1], [4, 2]],\\ &[[3, 0], [3, 1], [3, 2]], [[0, 2], [0, 3], [0, 4]], [[2, 0], [2, 1], [2, 2]],\\ &[[2, 6], [3, 6], [4, 6]], [[2, 0], [3, 0], [4, 0]], [[6, 2], [6, 3], [6, 4]],\\ &[[2, 1], [3, 1], [4, 1]], [[0, 2], [1, 2], [2, 2]], [[1, 2], [2, 2], [3, 2]],\\ &[[2, 2], [3, 2], [4, 2]], [[3, 2], [4, 2], [5, 2]], [[4, 2], [5, 2], [6, 2]],\\ &[[1, 3], [2, 3], [3, 3]], [[2, 3], [3, 3], [4, 3]], [[1, 2], [1, 3], [1, 4]],\end{aligned}$$

$$[[2,1],[2,2],[2,3]],[[2,2],[2,3],[2,4]],[[2,3],[2,4],[2,5]],$$
$$[[3,4],[3,5],[3,6]],[[2,4],[2,5],[2,6]],[[3,1],[3,2],[3,3]],$$
$$[[3,2],[3,3],[3,4]],[[3,3],[4,3],[5,3]],[[4,3],[5,3],[6,3]],$$
$$[[1,4],[2,4],[3,4]],[[2,4],[3,4],[4,4]],[[3,4],[4,4],[5,4]],$$
$$[[4,4],[5,4],[6,4]],[[2,5],[3,5],[4,5]],[[3,3],[3,4],[3,5]],$$
$$[[4,1],[4,2],[4,3]],[[4,2],[4,3],[4,4]],[[4,3],[4,4],[4,5]],$$
$$[[4,4],[4,5],[4,6]],[[5,2],[5,3],[5,4]]]\}.$$

The set $A0$ represents a subset of feasible actions, where each element is an action $[[X1, Y1], [X2, Y2], [X3, Y3]]$ that satisfies the condition $X1 = X2 = X3$ and $Y1 < Y2 < Y3$, or $X1 < X2 < X3$ and $Y1 = Y2 = Y3$. The other feasible actions are derived from $A0$ (see definition of As below).

Let

$$C \subset (I_6 \times I_6)^*.$$

Although an element of C can be used as a representation of a state of the puzzle, an extension of C is used for the state set of the user model. The extension facilitates computation.

Let vari : $C \to \text{Re}$ be defined as follows: When $c = [[x_1, y_1], \ldots, [x_k, y_k]]$, let

$$x_0 = \Sigma x_i / k,$$
$$y_0 = \Sigma y_i / k.$$

Then, let

$$\text{vari}(c) = \Sigma(|x_i - x_0|^3 + |y_i - y_0|^3)/k.$$

Using the vari function, the state set $\underline{C}$ of the user model is defined as

$$\underline{C} \subset C \times \text{Re},$$

where

$$[c, v] \in \underline{C} \leftrightarrow v = \text{vari}(c).$$

Let the state transition function delta: $\underline{C} \times A \to \underline{C}$ be given by

$$\text{delta}([c, v], [[x1, y1], [x2, y2], [x3, y3]]) = [c2, v2] \leftrightarrow$$
$$c2 = (c - \{[x1, y1], [x2, y2]\}) \cup \{[x3, y3]\}\&$$
$$v2 = \text{vari}(c2)\&$$
$$\text{constraint}([c2, v2]) = \text{true}.$$

The function vari is a measure of the divergence of the token distribution:

Let genA : $\underline{C} \to \wp((I_6 \times I_6)^3)$ be given by

$$\text{genA}([c, v]) = As = \{[[x1, y1], [x2, y2], [x3, y3]] | [x1, y1] \in c\&$$
$$([[x1, y1], [x2, y2], [x3, y3]] \in A0 \vee$$
$$[[x3, y3], [x2, y2], [x1, y1]] \in A0)\&[x2, y2] \in c\&[x3, y3] \notin c\}.$$

Let constraint : $C \to$ {true,false} be given by

$$\text{constraint}([c, v]) = \text{true} \leftrightarrow v < |c|/2.$$

A combination of a constraint condition and the backtracking function is effective for controlling the problem-solving activity (See Chapter 8). In the Chinese checkers problem, the constraint condition is designed heuristically to prevent the token arrangement from diverging. The condition requires that the arrangement converge to a final state as the action sequence expands:

Let $c0$ be given by

$$\begin{array}{l} c0 = [\\ \qquad\qquad [2,0],[3,0],[4,0], \\ \qquad\qquad [2,1],[3,1],[4,1], \\ [0,2],[1,2],[2,2],[3,2],[4,2],[5,2],[6,2], \\ [0,3],[1,3],[2,3], \qquad [4,3],[5,3],[6,3], \\ [0,4],[1,4],[2,4],[3,4],[4,4],[5,4],[6,4], \\ \qquad\qquad [2,5],[3,5],[4,5], \\ \qquad\qquad [2,6],[3,6],[4,6]]. \end{array}$$

Then,

$$\text{initialstate} = [c0, \text{vari}(c0)].$$

Let

$$\text{finalstate}([c, v]) = \text{true} \leftrightarrow |c| = 1.$$

Let

$$\text{goal}([c, v]) = v.$$

Let

$$\text{st}([c, v]) = \text{true} \leftrightarrow |c| = 1.$$

The above definitions in set theory allows the user model of the Chinese checkers problem to be derived directly in set theory as follows:

```
/*cchecker.set*/
.func([delta,vari,genA,initialstate,initialstate0,goal]);
actionSet.g=[[[0,3],[1,3],[2,3]],
[[0,4],[1,4],[2,4]],
[[4,0],[4,1],[4,2]],
[[3,0],[3,1],[3,2]],
[[0,2],[0,3],[0,4]],
[[2,0],[2,1],[2,2]],
[[2,6],[3,6],[4,6]],
[[2,0],[3,0],[4,0]],
[[6,2],[6,3],[6,4]],
[[2,1],[3,1],[4,1]],
[[0,2],[1,2],[2,2]],
[[1,2],[2,2],[3,2]],
[[2,2],[3,2],[4,2]],
[[3,2],[4,2],[5,2]],
[[4,2],[5,2],[6,2]],
[[1,3],[2,3],[3,3]],
```

```
[[2,3],[3,3],[4,3]],
[[1,2],[1,3],[1,4]],
[[2,1],[2,2],[2,3]],
[[2,2],[2,3],[2,4]],
[[2,3],[2,4],[2,5]],
[[3,4],[3,5],[3,6]],
[[2,4],[2,5],[2,6]],
[[3,1],[3,2],[3,3]],
[[3,2],[3,3],[3,4]],
[[3,3],[4,3],[5,3]],
[[4,3],[5,3],[6,3]],
[[1,4],[2,4],[3,4]],
[[2,4],[3,4],[4,4]],
[[3,4],[4,4],[5,4]],
[[4,4],[5,4],[6,4]],
[[2,5],[3,5],[4,5]],
[[3,3],[3,4],[3,5]],
[[4,1],[4,2],[4,3]],
[[4,2],[4,3],[4,4]],
[[4,3],[4,4],[4,5]],
[[4,4],[4,5],[4,6]],
[[5,2],[5,3],[5,4]]];

delta([C,V],[[X1,Y1],[X2,Y2],[X3,Y3]]=[C2,V2] <->
           C20:=minus(C,[[X1,Y1],[X2,Y2]]),
           C2:=append(C20,[[X3,Y3]]),
           V2:=vari(C2),
           constraint([C2,V2]);

vari(C)=Re <->
           [Xs,Ys]:=transpose(C),
           L:=cardinality(C),
           X0:=sum(Xs)/L,
           Y0:=sum(Ys)/L,
           U:=abs(Xs-X0),
           V:=abs(Ys-Y0),
           Re:=(sum(U*U*U)+sum(V*V*V))/L;

genA([C,V])=As <->
          regAs.g:=[],
          As00:=defSet(pgenA(W,U,[C]),[U,C]),
          As:=regAs.g;

pgenA(S,[X,Y],[C]) <->
          A:=actionSet.g,
          (member([[X-2,Y],[X-1,Y],[X,Y]],A) and
               member([X-1,Y],C) and notmember([X-2,Y],C)) ->
                     (
          regAs.g:=append(regAs.g,[[[X,Y],[X-1,Y],[X-2,Y]]])),
          (member([[X,Y],[X+1,Y],[X+2,Y]],A) and
               member([X+1,Y],C) and notmember([X+2,Y],C)) ->
                     (
          regAs.g:=append(regAs.g,[[[X,Y],[X+1,Y],[X+2,Y]]])
                     ),
          (member([[X,Y-2],[X,Y-1],[X,Y]],A) and
               member([X,Y-1],C) and notmember([X,Y-2],C)) ->
                     (
```

```
            regAs.g:=append(regAs.g,[[[X,Y],[X,Y-1],[X,Y-2]]])
                      ),
            (member([[X,Y],[X,Y+1],[X,Y+2]],A) and
                 member([X,Y+1],C) and notmember([X,Y+2],C)) ->
                      (
            regAs.g:=append(regAs.g,[[[X,Y],[X,Y+1],[X,Y+2]]])
                      ),
            S:="OK";

constraint([C,V]) <->
            L:=cardinality(C),
            V<L/2;

initialstate()=[C,V] <->
               C:=initialstate0(),
               V:=vari(C);
initialstate0()=C <->

               C:=[
                              [2,0],[3,0],[4,0],
                              [2,1],[3,1],[4,1],
               [0,2],[1,2],[2,2],[3,2],[4,2],[5,2],[6,2],
               [0,3],[1,3],[2,3],      [4,3],[5,3],[6,3],
               [0,4],[1,4],[2,4],[3,4],[4,4],[5,4],[6,4],
                              [2,5],[3,5],[4,5],
                              [2,6],[3,6],[4,6]];
finalstate([C,V]) <->
              cardinality(C)=1;
goal([C,Re])=Re;
st([C,V]) <->
             lambda([C,V]),
             cardinality(C)=1;
lambda([C,V]) <->
            (puzzleWp.g=Wp0) ->
                     (
                     Wp:=Wp0
                     )
            .otherwise
                     (
                     puzzleWp.g:=[]
                     ),
            makewindowSS(Wp,"puzzzle",X1,Y1,8,8),
            clearsheet(Wp),
            puzzleWp.g:=Wp,
            regC.g:=C,
            Success:=defSet(plambda(Y,X,[Wp]),[X,C]);
plambda(Y,[I,J],[Wp]) <->
                        #(Wp,I,J):="X",
                        Y:="s";
```

A solution of the problem is given as "Solf" in Fig. 7.9.

In "Solf," the starting action is [[1, 3], [2, 3], [3, 3]] and the final action is [[5, 3], [4, 3], [3, 3]]. The solution strategy for this problem depends heavily on the combination of the constraint condition and the backtracking function. Figure 7.10 shows the dynamic behavior of the goal.

The zigzag behavior clearly shows that backtracking due to the constraint provides a strict control on the goal-seeking activity.

```
DIALOG
"Solf="[[[1,3],[2,3],[3,3]],[[2,1],[2,2],[2,3]],[[0,2]
,[1,2],[2,2]],[[0,4],[0,3],[0,2]],[[3,2],[2,2],[1,2]],
[[0,2],[1,2],[2,2]],[[3,0],[3,1],[3,2]],[[2,3],[2,2],[
2,1]],[[2,0],[2,1],[2,2]],[[2,5],[2,4],[2,3]],[[4,5],[
3,5],[2,5]],[[2,6],[2,5],[2,4]],[[4,6],[3,6],[2,6]],[[
2,3],[2,4],[2,5]],[[2,6],[2,5],[2,4]],[[3,3],[3,4],[3,
5]],[[1,4],[2,4],[3,4]],[[3,5],[3,4],[3,3]],[[3,2],[3,
3],[3,4]],[[5,2],[4,2],[3,2]],[[4,0],[4,1],[4,2]],[[4,
3],[4,2],[4,1]],[[2,2],[3,2],[4,2]],[[4,1],[4,2],[4,3]
],[[4,3],[4,4],[4,5]],[[6,4],[5,4],[4,4]],[[6,2],[6,3]
,[6,4]],[[3,4],[4,4],[5,4]],[[6,4],[5,4],[4,4]],[[4,5]
,[4,4],[4,3]],[[5,3],[4,3],[3,3]]]
"Yf="[[[5,3],[4,3]],2.5000e-01]
WARNING:predicate 'postprocess()' is not defined!!!
ccheckerPD2.p" ends"
"normal reset state"
"May I help you?"
X>
```

Fig. 7.9. Solution for Chinese checkers problem.

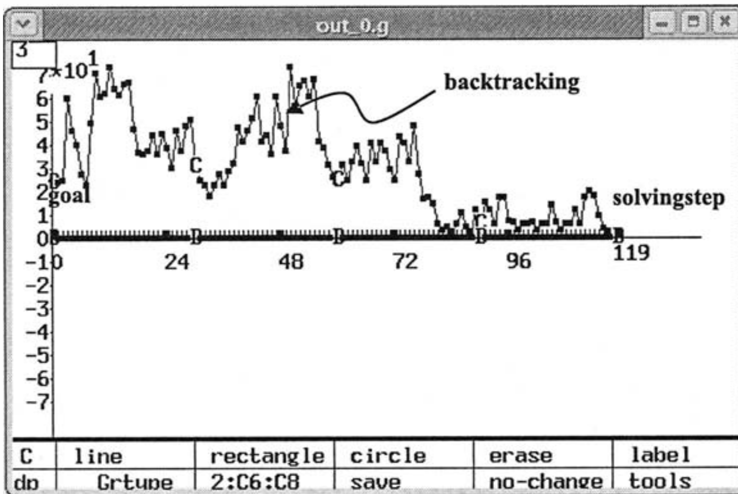


Fig. 7.10. Dynamic behavior of goal.

References

La Salle, J. and Lefschetz, S. (1961) *Stability by Liapunov's Direct Method*, Academic Press.

Mesarovic, M. D. and Takahara, Y. (1989) *Abstract Systems Theory*, Springer.

Stuart, R. and Norvig, P. (1995) *ARTIFICIAL INTELLIGENCE*, Prentice Hall.

Tu, P. (1994) *Dynamical Systems*, Springer.

8

Linear Quadratic Optimization Problem: E-C-O and E-O-O Problems

Chapter 7 introduced a simple dynamic system as a control engineering problem and investigated it as an E-O-C problem. This chapter formalizes an optimization problem for a general second-order dynamic system and examines it as an E-C-O problem. The goal is defined as a traditional quadratic problem. Because the goal of the optimization problem has parameters that can be modified by a user, the problem can also be considered as an E-O-O problem. This chapter introduces a constraint condition to investigate the significance of backtracking, which is the most basic characteristic for distinguishing the goal-seeker from conventional controllers.

The stdPDsolver will be used for the present system as the goal-seeker.

8.1 Linear Quadratic Optimization Problem

Let us introduce an optimization problem that is given in the following way:

Optimization Problem

Let a general second-order dynamic system be as follows:

$$u' = v,$$
$$v' = -\beta u - \alpha v + a,$$

where $\alpha \in \mathrm{Re}$ and $\beta \in \mathrm{Re}$.

Let an initial state be (u_0, v_0). Let a desirable state be $(u_f, 0)$. Find a control function $a()$ that minimizes the following performance function:

$$g((u,v),a) = \int_0^{t_f} ((u,v)-(s,0))^T Q((u,v)-(s,0)) + ra^2 dt \qquad (8.1)$$

subject to

$$|u - s| \geq |v|,$$

where $Q > 0$ (positive symmetric matrix), $r > 0$, and $[0, t_f]$ is the time span for the optimization operation. The variables $u()$, $v()$, and $a()$ are time functions over $[0, t_f]$.

The above constraint is introduced arbitrarily to produce a situation in which an analytical solution could not be found. (The condition can be understood to require a "soft landing.")

This optimization problem is usually called a linear quadratic problem (LQP), which is the most basic and typical dynamic optimization problem [Tu, 1994].

The optimization problem can be attacked in two ways, depending on whether the target state is fixed as c_f of Section 7.2.3. If so, the problem belongs to the E-C-C class, and if not it belongs to the E-C-O class. If the problem is recognized as an E-C-C problem, a (modified) dynamic programming (DP) method can be used. This chapter investigates the problem as an E-C-O problem, so that the hill-climbing method with a push-down stack (PD method) is used for the goal-seeker. As the subsequent discussion shows, it will be demonstrated that the same goal-seeker as used for the regulation problem of Chapter 7 can be used for the present case, although the representation of the solving activity or the user model is completely different.

8.2 User Model of Linear Quadratic Optimization Problem for PD Goal-Seeker

A user model of the optimization problem is derived following the development procedure of Fig. 4.6.

8.2.1 Drawing Input–Output Block Diagram

The input–output block diagram is given in Fig. 8.1.

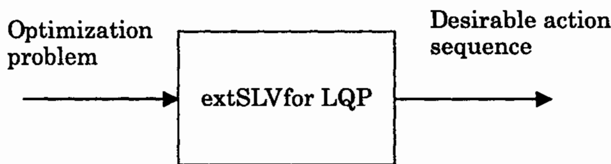


Fig. 8.1. Input–output block diagram of a linear quadratic problem.

8.2.2 Input–Output Specification in Set Theory

(i) Input Specification

The problem specification environment is given by

$$du/dt = v,$$

$$dv/dt = -\beta u - \alpha v + a,$$

$$g((u,v),a) = \int_0^{t_f} ((u,v)-(s,0))^T Q((u,v)-(s,0)) + ra^2 dt, \tag{8.2}$$

$$|u-s| \geq |v|.$$

(ii) Output Specification

$$Y = \{y | y : [0, t_f] \to A\},$$

where $A = \text{Re}$. A desirable sequence of actions will be selected from Y.

8.2.3 Process Specification as Automaton

(i) Action Set

The action set A is given by the problem specification environment (PSE) as

$$A = \text{Re}.$$

(ii) State Transition Function

It should be noted that if $U \times V$ of Chapter 7 is used as a state set, the goal is not a function on the state set but is a function on its trajectory set. We must extend the state set so that the goal can be a function of the state.

The extension is done by introducing the following relations:

$$dg/dt = ((u,v)-(s,0))^T Q((u,v)-(s,0)) + ra^2,$$
$$dt/dt = 1.$$

The last relation is introduced to represent the fact that the solving activity is terminated by the upper time limit t_f of the integral.

Then, the state is

$$c = (u, v, g, t) \in C = U \times V \times G \times T,$$

where $u \in U = \text{Re}$, $v \in V = \text{Re}$, $g \in G = \text{Re}$, and $t \in T = \text{Re}$.

Although the original problem is a continuous one, it must be discretized so that it can be dealt with on a computer. The above differential equations are transformed into difference equations as done in Section 7.2.2. Then, the state transition function $\delta : U \times V \times G \times T \times A \to U \times V \times G \times T$ is given by the following equations:

$$v_2 = -h\beta u + (1-h\alpha)v + ha,$$
$$u_2 = u + hv_2,$$
$$g_2 = g + h(q_1(u_2-s)^2 + q_2 v_2^2 + ra^2),$$
$$t_2 = t + h,$$

where $\delta((u,v,g,t),a) = (u_2, v_2, g_2, t_2)$.

(iii) Output Function

The output function $\lambda : C \times A \to A$ is

$$\lambda(c, a) = a.$$

(iv) genA: $C \to \wp(A)$

The function genA() is the same as that given in Section 7.2.3. That is,

$$\text{genA}(c) = A.$$

(v) constraint: $C \to$ {true, false}

$$\text{constraint}([u, v, g, t]) = \text{true} \leftrightarrow |u - s| \geq |v|.$$

(vi) Initial State

The initial state c_0 is given by modifying the initial state given in Chapter 7. That is,

$$\text{initialstate}(c_0) \leftrightarrow c_0 = (u_0, v_0, 0, 0).$$

(vii) Final State

The final state is not defined in the current case.

$$c_f : \text{undefined}$$

(viii) Stopping Condition

The stopping condition is given by the last component of the state:

$$\text{st}((u, v, g, t)) = \text{true} \leftrightarrow t = t_f$$

8.2.4 PD Optimization Formulation

goal : $C \to$ Re is naturally given as

$$\text{goal}((u, v, g, t)) = g.$$

Then, a user model of the current problem is given by

$$\textbf{user model} = \langle A, C, Y, \delta, \lambda, \text{goal}, \text{genA}, \text{constraint}, \text{st}, c_0, c_f \rangle.$$

8.3 Implementation in extProlog

Because the optimization of the control engineering problem is formalized in the standard form, an extended solver (extSLV) for the problem is derived by expressing the user model in computer-acceptable set theory and compiling it. The entire user model is given in Appendix 8.1.

In the implementation, $\alpha = -2, \beta = 2, A = [-80, -40, -20, -10, -5, -3, -1, 0, 1, 3, 5, 10, 20, 40, 80]$, $(u_0, v_0) = (10, 4)$, and Q and r of the output evaluation $((u, v) - (s, 0))^T Q((u, v) - (s, 0)) + ra^2 = q_1(u - s)^2 + q_2 v^2 + ra^2$ are specified as

$$q_1 = 30, q_2 = 1, \text{ and } r = 0.01.$$

Here $(\alpha, \beta) = (-2, 2)$ indicates that the dynamic system is set as a typical unstable (unstable focus) system.

Figure 8.2 shows dynamic (u, v)-behaviors of extSLV, where the control $a()$ is determined by the stdPDsolver.

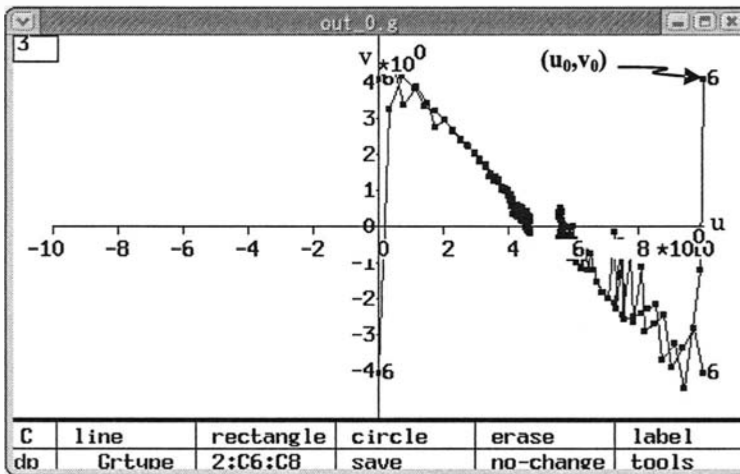


Fig. 8.2. Behaviors with respect to initial states.

The trajectories derived from initial states $(10, 4)$, $(0, 4)$, $(0, -4)$, and $(10, -4)$ display nonlinear but stable behaviors. As such, the desired set point $(5, 0)$ is reached. It should be noted that the constraint $|u - s| \geq |v|$ is naturally satisfied for every trajectory due to the backtracking function, which distinguishes extSLV from conventional solvers. If the goal-seeker were a linear controller, the constraint could not be satisfied.

Figure 8.3 shows the dynamic behavior of (u, v, a), which corresponds to the trajectory in Fig. 8.2 starting from $(u_0, v_0) = (10, 4)$. The curve whose value shows abrupt change corresponds to $a()$, control input. This is a solution produced by extSLV for the LQP. The curve that approaches 0 corresponds to $v()$, velocity, while

the smooth curve corresponds to $u()$, displacement, which shows that the behavior converges: $c_f = (5, 0)$. The highly nonlinear movement of $a()$ comes from the constraint $|u - s| \geq |v|$. If the constraint is ignored, $a()$ displays a regular bang-bang control behavior.

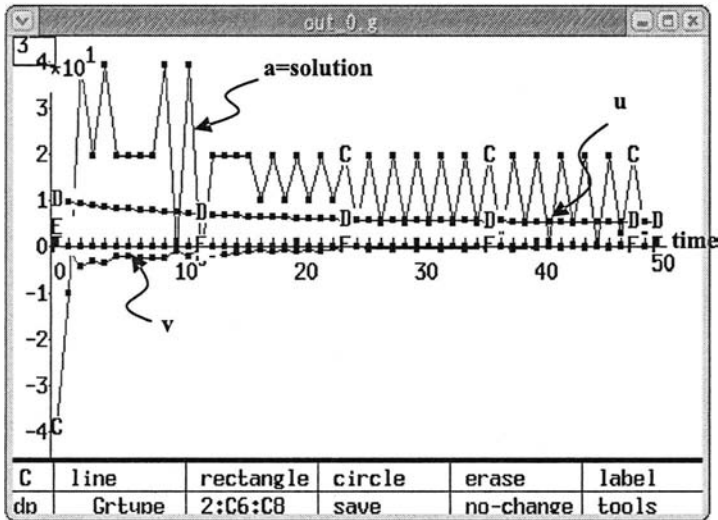


Fig. 8.3. Behavior of (u, v, a) from (10,4) in Fig. 8.2.

The results of this chapter show that the behavior of extSLV is quite different from that of a controller designed by the conventional control engineering technique. In particular, a problem whose control is not feasible for a conventional controller can be made a feasible problem for extSLV with the help of a proper constraint (see Fig. 8.2). This fact indicates that a constraint can be used in a positive way for extSLV, although a constraint is usually treated as a negative condition in conventional optimization theory. This fact owes to the backtracking function.

Appendix 8.1 User Model for Linear Quadratic Optimization Problem

```
/*lqp.set*/
/*optimization problem*/
/*mechanical system*/
/*E-O-C problem*/
/*mu"=a-ku*/
/*u'=v*/
/*v'=-ku/m+a/m*/

.func([goalElement,setPoint,initialstate]);
```

```
delta([U,V,GoalV,T],A)=[U2,V2,GoalV2,T2] <->
        V2:=-0.1*U+V+0.1*A,
        U2:=U+0.1*V2,
        R:=goalElement(U2,V2,A),
        GoalV2:=GoalV+0.1*R,
        T2:=T+0.1,
        constraint([U2,V2,GoalV2,T2]),!;
goalElement(U,V,A)=R <->
        S:=setPoint(),
        R:=(U-S)*(U-S)*30+V*V*1+A*A*0.01;
genA(C)=As <->
       As:=[-80,-40,-20,-10,-5,-3,-1,0,1,3,5,10,20,40];
constraint([U,V,GoalV,T]) <->
       V>= -4;
initialstate()=[10,10,0,0];
setPoint()=5;
goal([U,V,GoalV,T])=Re <->
        Re:=GoalV;
st([U,V,GoalV,T]) <->
        abs(T-5)<0.01;
```

The model description should be clear, where $h = 0.1$ and $t_f = 5$ are assumed. It faithfully represents the user model formulation of the problem.

References

Tu, P. (1994) *Dynamical Systems*, Springer.

9

Cube Root Problem: I-C-C Problem

This chapter discusses a cube root problem as a simple case of an I-C-C problem. As shown below, an extended solver (extSLV) for this problem can be easily developed when the model theory approach is applied because the problem is simple. It is, however, puzzling to see that we cannot find a nontrivial problem for the case of I-C-C, although there are many tough problems for its neighboring cases of E-O-C, I-O-C, E-C-O, and I-C-O. For example, the regulation problem of Chapter 7, an E-O-C problem, can become a tough problem if the number of bodies, which must be controlled, increases. The magic square problem of I-O-C is quite difficult if its size becomes large. The conventional dynamic optimization problem in Chapter 8 falls into the E-C-O class. The famous knapsack problem described in Chapter 10 is an I-C-O problem.

The stdPDsolver will be used for the present system as the goal-seeker.

9.1 Cube Root Problem

The cube root problem is to find a cube root b of a given (positive) real number a, i.e., $b = \sqrt[3]{a}$. That is, the problem is to solve the following equation:

$$f(b) = b^3 - a = 0.$$

The term "cube" has no particular significance and is selected because "cube" is more complicated than "square." Usually, this problem is used to illustrate Newton's method or its convergence problem. This chapter deals with the problem from a different point of view from the conventional ones.

9.2 User Model of Cube Root Problem for PD Goal-Seeker

A user model of the cube root problem is built following Fig. 4.6.

9.2.1 Drawing Input–Output Block Diagram

At this stage, the input and the output of the desired extSLV are presented conceptually. Figure 9.1 shows a block diagram representation of an extSLV of the cube root problem.

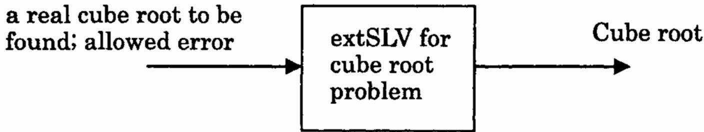


Fig. 9.1. Block diagram description for cube root problem.

9.2.2 Input–Output Specification in Set Theory

(i) Input Specification

Let $a \in \mathrm{Re}$ be a positive real number whose cube root is to be found and $p \in \mathrm{Re}$ an allowed error. (See the final state defined below.) Then, the problem specification environment (PSE) of the problem is given by

$$\mathrm{PSE} = \langle \mathrm{Re}, a, p \rangle.$$

(ii) Output Specification

The output set Y is

$$Y = \mathrm{Re}.$$

9.2.3 Process Specification as Automaton

(i) Action Set

Section 5.2 showed that because the problem is a case of I-C-C, a general form of an action is given by a function $a^{\mathrm{f}} : \mathrm{Re} \to \mathrm{Re}$. However, because a^{f} is characterized by a real number, let

$$A = \mathrm{Re}\text{: set of actions.}$$

(ii) State Transition Function

Following the general procedure of Chapter 5, let

$$Y = \mathrm{Re}\text{: set of state.}$$

Then, let the state transition function $\delta : Y \times A \to Y$ be given as follows:

$$\delta(y, a) = y + a.$$

(iii) Output Function

The output function $\lambda : Y \times A \to Y$ is given by the general procedure

$$\lambda(y, a) = y.$$

(iv) genA

Let genA : $Y \to \wp(\text{Re})$ be

$$\text{genA}(y) = \{-|y^3 - a| * \varepsilon, |y^3 - a| * \varepsilon\},$$

where $\varepsilon \in \text{Re}$ is a design parameter.

(v) constraint

Let constraint : $Y \to \{\text{true, false}\}$ be

$$\text{constraint}(y) = \text{true} \leftrightarrow y \geq 0.$$

(vi) Initial State

Let the initial state be

$$c_0 = 1.$$

(vii) Final State

The final state set C_f is supposed to be given by the problem as

$$C_\text{f} = \{y| \quad |y^3 - a| < p\}.$$

(viii) Stopping Condition

Let the stopping condition st : $Y \to \{\text{true, false}\}$ be

$$\text{st}(y) = \text{true} \leftrightarrow y \in C_\text{f}.$$

In the above formulation, genA is the only significant design factor.

9.2.4 PD Optimization Formulation

Suppose that goal : $Y \to \text{Re}$ is given by the problem as

$$\text{goal}(y) = |y^3 - a|.$$

The user model of the cube root problem is

$$\textbf{user model} = \langle A, Y, \delta, \lambda, \text{goal}, \text{genA}, \text{constraint}, \text{st}, c_0, C_\text{f}\rangle.$$

9.3 Implementation in extProlog

If the above formulation $\langle A, Y, \delta, \lambda,$ goal, genA, constraint, st, c_0, $C_f\rangle$ is described in computer-acceptable set theory and is compiled, we have a desired extSLV for the problem. Appendix 9.1 presents an entire user model of the extSLV for the cube root problem, where $a = 80$, $p = 0.01$, and $\varepsilon = 0.01$.

Figure 9.2 shows the dynamic behavior of the goal. It shows that the desired solution is obtained within ten steps.

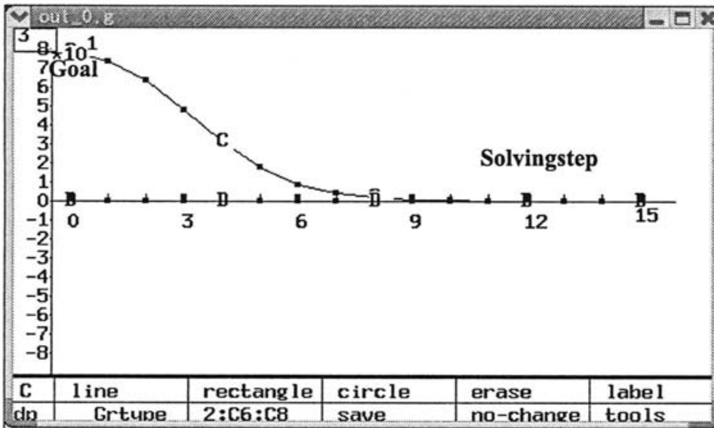


Fig. 9.2. Goal behavior of solver.

9.4 Tuning of PD Goal-Seeker for Cube Root Problem

Because the problem is trivial, no tuning is required except for the adaptation of ε in genA. ε is related to the convergence speed and its stability.

Appendix 9.1 User Model for Cube Root Problem

```
/*cuberoot7.set*/
.func([delta,genA,initialstate,goal]);
delta(y,a) = y2 <->
        y2 := y+a,
        constraint(y2);
genA(y) = As <->
      e := 0.01*abs(y*y*y-x.g),
      As := [-e,e];
constraint(y) <->
        y >= 0;
```

```
initialstate() = c <->
          c := 1;
finalstate(y) <->
        r := goal(y),
        r < p.g;
st(y) <->
          finalstate(y);
goal(y) = r <->
          r := abs(y*y*y-x.g);
p.g = 0.01;
x.g = 80;
```

The above program is a straightforward representation of the user model formulation.

10

Knapsack Problem: I-C-O Problem

This chapter discusses the well-known knapsack problem. This problem is another classical problem and is quoted as often in the problem-solving literature as the traveling salesman problem of Chapter 6. According to the problem classification of the model theory approach, the knapsack problem is more difficult than the traveling salesman problem because the former is less structured than the latter. Since the target state is not well specified except that it should be an optimal state, the stopping condition of the knapsack problem requires an involved expression to secure optimality.

The stdPDsolver will be used for the present system as the goal-seeker.

10.1 Knapsack Problem

The problem is given as follows: Suppose we have a knapsack and n (many) stones that may contain precious gems. The ith stone weighs w_i and its price (value) is p_i. We want to pack as many stones as possible into the knapsack. There is, however, an upper limit maxL on the load for the knapsack. The problem is to select stones so that the total value of stones in the knapsack is maximized.

This problem is usually investigated as an integer programming problem. Let $x_i \in \{0, 1\} (i = 1, \ldots, n)$. Then the integer programming formulation is to find $\{x_i\}$ such that

$$\sum x_i p_i \to \max$$

subject to

$$\sum x_i w_i \leq \text{maxL}.$$

According to our classification, the problem belongs to the I-C-O class because a solution is not a sequence of any solving activity, but rather an evaluation of a solution candidate is given. Because the output set (a set of solution candidates) will be used as the state set (because of the implicit solving activity), the target state is unknown. If a target state were known, it would be a solution.

10.2 User Model of Knapsack Problem for PD Goal-Seeker

10.2.1 Drawing Input–Output Block Diagram

At this stage, the input (problem specification environment, PSE) and the output (solution) of the extended solver (extSLV) of the problem should be conceptually described. Figure 10.1 shows an input–output block diagram description of the extSLV.

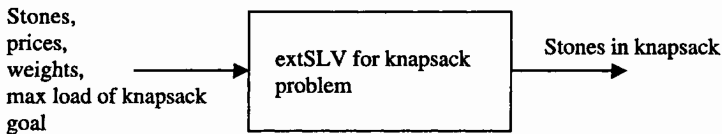


Fig. 10.1. Block diagram of knapsack problem.

The problem is characterized by stones, their weights, their prices, the maximum load of the knapsack, and a goal description. The PSE is specified by a structure on the parameters. The output set is a set of solution candidates: a class of stone sets.

10.2.2 Input–Output Specification in Set Theory

At this stage, the input–output description of Section 10.2.1 is formalized in set-theoretic terms.

(i) Input Specification

Let the identities of the stones be given by

$$I = \{1, \dots, n\} : \text{index of stones.}$$

Let the prices of the stones be represented by

$$P0 = \{p_1, \dots, p_n\} : \text{list of prices of stones,}$$

where

$$p_i = \text{price of the } i\text{th stone.}$$

Let the weights of the stones be represented by

$$W0 = \{w_1, \dots, w_n\} : \text{list of weights of stones,}$$

where

$$w_i = \text{weight of the } i\text{th stone.}$$

Let a function weight: $\wp(I) \to \mathrm{Re}$ be given by

$$\text{weight}(I') = \sum \{w_i | i \in I'\},$$

where $I' \subset I$. The function weight() will be used to obtain the total weight of stones in the knapsack.

Let goal: $\wp(I) \rightarrow \mathrm{Re}$ be given by

$$\mathrm{goal}(I') = -\sum\{p_i | i \in I'\},$$

where $I' \subset I$. Then goal() will be used to evaluate the total value of the stones in the knapsack. Because stdPDsolver is designed to minimize goal(), goal() is defined as $-\sum\{p_i | i \in I'\}$.

Let the maximum load = maxL $\in$ Re.

Let the constraint relation be the function constraint: $\wp(I) \rightarrow \{\text{true, false}\}$, where

$$\mathrm{constraint}(I') = \mathrm{true} \leftrightarrow \mathrm{weight}(I') \leq \mathrm{maxL}.$$

The predicate constraint() checks whether the total weight of the stones in the knapsack is within its limit.

Then, the PSE is given by the following structure:

$$\mathrm{PSE} = \langle I, P0, W0, \ \mathrm{weight, constraint, goal, maxL} \rangle.$$

(ii) Output Specification

The output set is the family of solution candidates that is given by

$$Y = \wp(I).$$

10.2.3 Process Specification as Automaton

At this stage, the solving activity of the extSLV is represented by an automaton. We introduce several auxiliary variables. Let

$$E0 = \{p_1/w_1, \ldots, p_n/w_n\} : \text{list of efficiency factors.}$$

As p_i/w_i becomes higher, it is more preferable for selection.

Let

$$E = \{e_1, \ldots, e_n\} : \text{sorted list of } E0,$$

where

$$e_1 \geq e_2 \geq \cdots \geq e_n.$$

That is, E is a reordered list of $E0$ according to the efficiency values.

Let

$$\mathrm{iOrder} : I \rightarrow J : \text{correspondence from } E \text{ to } E0,$$

where

$$J = \{1, \ldots, n\},$$

$$\mathrm{iOrder}(i) = j \leftrightarrow (e_i = p_j/w_j).$$

The function iOrder() shows the correspondence from E to $E0$. In subsequent discussions, E will be used instead of $E0$ or a stone with the highest efficiency factor is given 1 as its identity number. When the result is reinterpreted in the original identifiers, iOrder() is used for the interpretation.

Let

$$P = \{p'_1, \ldots, p'_n\} : \text{sorted list of } P0 \text{ with respect to iOrder},$$

where

$$\text{iOrder}(i) = j \leftrightarrow p'_i = p_j.$$

That is, P is a reordered list of $P0$ so as to be compatible with the order of E.

Let

$$W = \{w'_1, \ldots, w'_n\} : \text{sorted list of } W0 \text{ with respect to iOrder},$$

where

$$\text{iOrder}(i) = j \leftrightarrow w'_i = w_j.$$

That is, W is a reordered list of $W0$ so as to be compatible with the order of E.

(i) Action Set

The action set A is, in general, equal to $\{a|Y \rightarrow Y\}$. However, let us parameterize $a : Y \rightarrow Y$ by $(i, j) \in J \times J$, where an action (i, j) implies that a stone i in the knapsack is replaced by another stone j outside of the knapsack, i.e.,

$$A = J \times J.$$

(ii) State Transition Function

Because the problem belongs to the I-C-O class, the state set is specified by Y, i.e.,

$$C = Y : \text{state set}.$$

The state transition function $\delta : Y \times A \rightarrow Y$ is then

$$\delta(J', (i, j)) = \begin{cases} J' \cup \{j\} & \text{if constraint}(J' \cup \{j\}) = \text{true}, \\ (J' - \{i\}) \cup \{j\} & \text{if constraint}((J' - \{i\}) \cup \{j\}) = \text{true}, \\ \text{fail} & \text{otherwise}. \end{cases}$$

The first condition constraint$(J' \cup \{j\})$ = true indicates the situation that even if the jth stone is put into the knapsack, the total weight does not exceed maxL.

(iii) Output Function

The output function $\lambda : Y \times A \rightarrow Y$ is simply given by

$$\lambda(y, a) = y.$$

(iv) genA

In the present case, genA : $Y \to \wp(A)$ is slightly complicated. Let the current state be J'. Let

$$J'' = J - J'.$$

Let

$$\text{sortMinJ}' = \text{sortmin}(\text{J}').$$

sortmin(J′) sorts J′ in the order in which a smaller number has a higher priority.

$$\text{sortMaxJ}'' = \text{sortmax}(\text{J}'');$$

sortmax(J′) sorts in the order, in which a larger number has a higher priority.

Let

$$\text{project(sortMinJ}',\ [[1,\ 3]],\ \text{MinJ}'),$$

where MinJ′ is the list of the first three elements of sortMinJ′. Let

$$\text{project(sortMaxJ}'',\ [[1,\ 3]],\ \text{MaxJ}'').$$

Then, let

$$\text{genA(J}') = \text{MinJ}' \times \text{MaxJ}''.$$

That is, MinJ′ is the set of the three worst elements in J′ (or in the knapsack) and MaxJ″ is the set of the three best elements in J″. The function genA() seeks to replace the three worst stones in the knapsack by the best three stones outside of it. The number of stones (three in this case) is heuristically determined and can be tuned to a given situation.

(v) Constraint

The constraint is the same as given in Section 10.2.2, i.e.,

$$\text{constraint}(J') = \text{true} \leftrightarrow \text{weight}(J') \le \text{maxL}.$$

(vi) Initial State

Let the initial state be

$$c_0 = [1],$$

where $w'_1 \le$ maxL is assumed.

(vii) Final State

As mentioned above, a final state of the knapsack problem cannot be specified.

(viii) Stopping Condition

The stopping condition st : $Y \to$ {true, false} is the most difficult in the formulation of the knapsack problem. Informally, let st() be defined as

$$\text{st}(y) = \text{true} \leftrightarrow y \text{ is not surpassed by the last } D \text{ trials},$$

where D is also heuristically decided. In the implementation $D = 30$ is used. Note that precisely speaking, because the above st() depends on Y^* as well as on Y, it cannot be a regular stopping condition although it can be made a regular one if the state definition is modified.

10.2.4 PD Optimization Formulation

The goal : $C \to$ Re is given in Section 10.2.2, i.e.,

$$\text{goal}(J') = -\sum\{p_j | j \in J'\}.$$

Finally, a user model for the PD goal-seeker is

$$\textbf{user model} = \langle A, Y, \delta, \lambda, \ \text{goal, genA, constraint, st,} \ c_0\rangle.$$

10.3 Implementation in extProlog

If the user model specified in Section 10.2 is described in computer-acceptable set theory and compiled, a workable extSLV for the knapsack problem is realized. Appendix 10.1 presents a entire user model of the knapsack problem where the number of stones is assumed to be 500. The model generates randomly a problem of 500 stones by the predicate preprocess(). Although the PD goal-seeker is not designed to generate a true optimum solution, it actually yields a solution that is almost an optimum. This fact is checked by the branch and bound method of the knapsack problem.

Figure 10.2 shows a solution (output of the solver) produced by the solver.

"stones in knapsack" $=$ $[139, \ldots, 486]$ presents the final solution and [Total Weight, Total Value] = [70.0, 1508.0] shows the weight and the total value of it. Because maxL is set to be 70, the solution makes the best use of the constraint.

Figure 10.3 displays the dynamic behavior of the goal.

The horizontal coordinate corresponds to solving steps, while the vertical coordinates corresponds to the value of goal(). The figure says that the constraint is not effective during the first 48 steps and after that, exchange is performed between stones inside and outside of the knapsack. The stopping condition checks optimality during the exchange operation.

```
,[38,39],[39,28],[39,29],[39,40],[40,41],[41,42],[42,4
5],[45,46],[46,47],[47,48],[48,49],[49,50],[50,51],[51
,52],[50,51],[49,50],[52,43],[51,44],[51,49],[50,51],[
51,52],[52,53],[49,50],[53,52],[50,51],[52,53],[51,52]
,[48,49],[53,50],[52,51],[51,53],[50,51],[53,52],[49,5
0],[50,53],[51,50],[52,51]]
"Yf="[53,52,50,47,46,45,44,43,42,41,40,39,38,37,36,35,
34,33,32,31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,
16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
"stones in knapsack="[139,168,318,477,6,28,173,195,236
,270,374,19,25,82,115,145,291,307,356,491,23,52,58,60,
154,219,282,315,410,466,481,388,417,21,57,348,389,418,
478,153,248,396,430,49,143,266,381,163,186,486]
"[Total Weight,Total Value]="[70.00000,1508.000]
knapsackPD3.p" ends"
"normal reset state"
"May I help you?"
X>
```

Fig. 10.2. Solution by the solver.

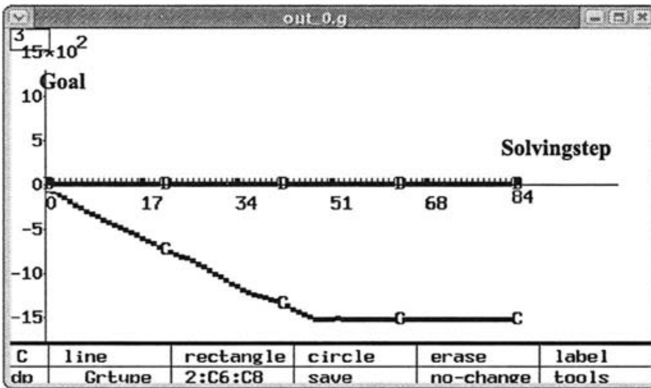


Fig. 10.3. Dynamic behavior of goal.

10.4 Tuning of PD Goal-Seeker for Knapsack Problem

The knapsack problem has a simple structure except for the specification of the stopping condition. However, as the number of stones increases, it becomes more difficult to get a solution because required processing time grows exponentially. Tuning of the stack of the PD goal-seeker is used to overcome the problem. An element of the stack consists of the four components state, possible action set, past history of states, and action history.

1. State: Because a state is an element of $\wp(J)$ or a list of the indices, its representation size increases proportionally as the number of stones increases. However, it is clear that efficient stones are less likely to be removed once they are selected. They are supposed to be fixed elements that can be ignored in the "working" state representation. If this is so, a smaller working state representation is obtained.

2. Action history: The action history is necessary if it is used to compute an output. In the current problem, however, it is not required for computation of an output and hence can be completely ignored in a stack element.
3. History of states: The history of states is used to check the repetition of states. It can easily become a huge list as the number of stones increases. In order to avoid a list that is too long it can be a good heuristic procedure to save only the most recent states. However, we have another problem: how many states are to be saved? It is also answered heuristically. Although there is a risk that state repetition can occur if such heuristics are used, this procedure can work reasonably well.

Appendix 10.1 User Model for Knapsack Problem

```
/*knapsack4.set*/
/*minimization*/
/*Ei=Pi/Wi*/
/*state={I1,...,Ik}*/
/*action=[Ii,Ij]*/
func([sum,goal,sortmax,getWP,genProb]);
delta(C,[[],I2])=CC <->
         C2:=append(C,[I2]),
         (constraint(C2)) ->
                   (
                   CC:=sortmax(C2)
                   )
         .otherwise
                   (
                   CC:=[]
                   );

delta(C,[I,I2])=CC <->
         C2:=append(C,[I2]),
         (constraint(C2)) ->
                   (
                   CC:=sortmax(C2)
                   )
         .otherwise
                   (
                   SubC:=minus(C,[I]),
                   SubC2:=append(SubC,[I2]),
                   (constraint(SubC2)) ->
                            (
                            CC:=sortmax(SubC2)
                            )
                   .otherwise
                            (
                            CC:=[]
                            )
                   );
getWP(C2)=[R,S] <->
         C1:=getC1.g,
         C:=append(C2,C1),
         W:=regW.g,
         P:=regP.g,
```

```
          W2:=project(W,C),
          P2:=project(P,C),
          R:=sum(W2),
          S:=sum(P2);
getC1.g=[];

genA(C)=As <->
     (C=[]) ->
              (
              As:=[]
              )
     .otherwise
              (
              MinC:=C,
              I:=regI.g,
              MaxR:=minus(I,C),
              (MinC=[]) ->
                          (
                          MinC2:=[[]]
                          )
              .otherwise
                          (
    MinC2:=project(MinC,[[1,3]])
                          ),
              (MaxR=[]) ->
                          (
                          MaxR2:=[]
                          )
              .otherwise
                          (
    MaxR2:=project(MaxR,[[1,3]])
                          ),
              As:= product(MinC2,MaxR2)
              );

constraint(C) <->
          [W,P]:=getWP(C),
          MaxL:=regMaxL.g,
          W<=MaxL;

initialstate()=C <->
                    C:=[1];

st(C) <->
          R:=goal(C),
          [MaxP,MaxC,D]:=regMaxP.g,
          (R<MaxP) ->
                    (
                    regMaxP.g:=[R,C,0],
                    .fail
                    )
          .otherwise
                    (
                    (D<30) ->
                               (
          regMaxP.g:=[MaxP,MaxC,D+1],
```

```
                              .fail
                              )
                    );


goal(C)=R <->
          (C=[]) ->
                    (
                    R:=0
                    )
          .otherwise
                    (
                    [R2,P2]:=getWP(C),
                    MaxL:=regMaxL.g,
                    (R2<=MaxL) ->
                              (
                              R:=0 -P2
                              )
                    .otherwise
                              (
                              R:=1000
                              )
                    );


preprocess() <->
          Count:=500,
          [W0,P0]:=genProb(Count),
          E0:=P0/W0,
          Ind:=procC("createindex",[1,Count]),
          EI:=transpose([E0,Ind]),
          EIMax:=sortmax(EI),
          Iorder:=project(transpose(EIMax),2),
          regI.g:=Ind,
          W:=project(W0,Iorder),
          P:=project(P0,Iorder),
          regW.g:=W,
          regP.g:=P,
          C0:=initialstate(),
          regMaxL.g:=70,
          regMaxP.g:=[0,C0,0];
genProb(Count)=[W,P] <->
   W:=procC("randomgen",[Count,[1,2,3,4,5,6,7,8,9,10],
      [0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1]]),
   L2:=procC("randomgen",[Count,[1,2,3,4,5,6,7,8,9,10],
       [0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1]]),
   P:=L2*4;


postprocess() <->
           [MaxP,MaxC,DD]:=regMaxP.g,
           [MaxW,MaxPP]:=getWP(MaxC),
           xwriteln(0,"[MaxW,MaxP]=",[MaxW,MaxP]);
```

The predicate preprocess() generates a problem of 500 stones (Count = 500) and sets maxL = 70 (regMaxL.g = 70). getWP() calculates the total weight and value of stones in the knapsack.

11

Class Schedule Problem: I-O-C Problem

This chapter discusses a class schedule problem as an I-O-C problem. The problem relates to the design of a class schedule for a college and includes assignment of instructors, subjects, and classrooms to slots of a weekly timetable. In order to simplify the problem, we will consider the class schedule for one grade of a specific department.

It is clear that the class schedule problem belongs to the implicit solving activity problem class. The output, a class schedule, is not a sequence of solving actions. Furthermore, it is also clear that the problem does not have an explicit evaluation function of an output. It is, however, not clear whether the problem has an explicit target state. We assume that an output of the current solver is a tentative schedule that avoids conflict as much as logically possible. The schedule produced may not be the final one. The final schedule can be determined only after negotiation over intrinsic conflicts, and this phase is assumed to be out of the scope of the current system. As such, the final state is a situation in which no further logical assignment is possible under the given constraint. The class schedule problem is handled as a closed-target problem in this sense.

The stdPDsolver will be used for the present system as the goal-seeker.

11.1 Class Schedule Problem

The class schedule problem can have many variations. This chapter investigates two problems. The first problem is that because an instructor is allowed to have preferences for particular timetable slots, there may be conflicts among instructors. We assume some order of priority among them, which may be related to seniority or other factors. The second problem is that a classroom must be able to accommodate the number of registered students.

11.2 User Model of Class Schedule Problem for PD Goal-Seeker

A user model for the class schedule problem is constructed following Fig. 4.6.

11.2.1 Drawing Input–Output Block Diagram

At this stage, a target solver is described as an input–output system specifying its input as the problem specification environment (PSE) and its output as the solution. Figure 11.1 illustrates the input–output of a solver for the class schedule problem.

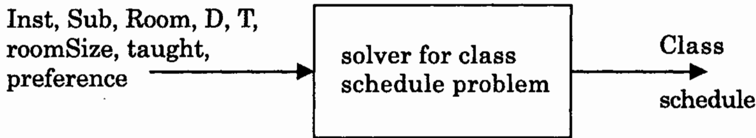


Fig. 11.1. Input–output block diagram of class schedule problem.

The class schedule problem is specified by five sets and four relations: Inst, Sub, Room, D, T, roomSize(), taught(), preference(), and constraint(). The sets Inst, Sub, and Room represent instructors, subjects, and classrooms, respectively. The set D is a set of dates from Monday to Saturday. The set T is a set of lecture times from first to fourth. The function roomSize() indicates a relation between a room identification and its capacity. The function taught() is a correspondence between instructors and their teaching subjects. The function preference() shows when an instructor prefers, to have classes. The output of the class schedule problem is a class schedule candidate, which is not necessarily a final one.

11.2.2 Input–Output Specification in Set Theory

At this stage, the input and output given in Section 11.2.1 are described in set-theoretic terms.

(i) Input Specification

Let

$$\text{Inst} = \{p_1, \ldots, p_n\} : \text{set of instructors},$$
$$\text{Sub} = \{s_1, \ldots, s_m\} : \text{set of subjects},$$
$$\text{Room} = \{r_1, \ldots, r_l\} : \text{set of classrooms},$$
$$D = \{\text{mon, tue, wed, thu, fri, sat}\} : \text{set of class days},$$
$$T = \{1, 2, 3, 4\} : \text{set of lecture times}.$$

Let taught: Sub $\rightarrow$ Inst $\times$ Int be given by

taught(s_i) $= (p_j, n)$ iff the subject s_i is lectured by the instructor p_j whose registered student number is n.

Let roomSize: Room $\rightarrow$ Int be given by

$$\text{roomSize}(r_k) = \text{number of seats in the room } r_k.$$

Let preference: Inst $\rightarrow$ (Sub $\times$ D $\times$ T)* be given by

preference(p_i) $= (s, d, t) \ldots (s', d', t')$ iff the instructor p_i wants to have a class of the subject s at (time) t of (day) $d, \ldots,$ and a class of the subject s' at t' of d', respectively.

Then the input is given by the following structure:

$$\text{PSE} = \langle \text{Inst}, \text{Sub}, \text{Room}, D, T, \text{taught}, \text{roomSize}, \text{preference} \rangle.$$

(ii) Output Specification

The output set Y is a class of time schedule candidates. It is given by

$$Y = \{y | y : D \rightarrow (T \times \text{Sub} \times \text{Inst} \times \text{Room})^*\}.$$

That is, $y(\text{mon}) = (t, s, p, r) \ldots (t', s', p', r')$ iff on Monday at time t the subject s is taught by the instructor p in the room r, and at time t' the subject s' is taught by the instructor p' in the room r'. In general, y = nill (empty list) is possible. An element $y \in Y$ is a partially scheduled timetable.

11.2.3 Process Specification as Automaton

At this stage, the solution-finding activity is formalized as an automaton.

(i) Action Set

Because the problem belongs to the implicit solving activity class, the action set A is formalized in the following way: let

$$A = \{a | a : Y \rightarrow Y\}.$$

In the above formulation, an action $a()$ is represented as a mapping. However, because a set of mappings is too general to be processed effectively, it is normal for the implicit solving case that a mapping be replaced by a parameter that characterizes the mapping. For the class schedule problem, the following set is used as the set of parameters for actions:

$$A = (D \times T) \times (\text{Sub} \times \text{Inst} \times \text{Room}).$$

For example, a parameter

$$((\text{mon}, 1), (s_1, p_1, r_1)) \in A$$

implies an action to assign a class, where a subject s_1 is taught by an instructor p_1 in a room r_1, to the first class (1) on Monday.

(ii) State Transition Function

Let Y be the state set following the general formulation of Chapter 5. Then, the state transition function $\delta : Y \times A \to Y$ is defined as follows:

$$\delta(y, (d, t, s, p, r)) = \begin{cases} \text{nil} & \text{if } (\exists(ss, pp, rr))((t, ss, pp, rr) \in y(d)), \\ y' & \text{otherwise,} \end{cases}$$

where

$$y'(d') = \begin{cases} y(d') & \text{if } d' \neq d, \\ y(d) \cdot (t, s, p, r) & \text{if } d' = d. \end{cases}$$

In the above definition $y(d) \cdot (t, s, p, r)$ is the concatenation of $y(d)$ and (t, s, p, r). At the same time, since $y(d)$ is a list, it is treated as a set and hence $(t, ss, pp, rr) \in y(d)$ is a valid expression. The expression $\delta(y, (d, t, s, p, r)) = \text{nil}$ indicates that there is a conflict.

Here δ is a partial function. As usual, it is restricted by two functions, genA : $Y \to \wp(A)$ and constraint : $Y \to \{\text{true, false}\}$, i.e.,

$$\delta(y, a) = y' \to a \in \text{genA}(y) \text{ and constraint } (\delta(y, a)) = \text{true}.$$

The function genA and constraint are defined below.

(iii) Output Function

The output function $\lambda : Y \times A \to Y$ is defined by

$$\lambda(y, a) = y.$$

(iv) genA: $C \to \wp(A)$

For the class schedule problem, genA is somewhat complicated. Its definition requires the following auxiliary functions:

Let subject : $Y \to \wp(\text{Sub})$ be given by

subject$(y) = \text{Sub} - \{s | (\exists(d, t, p, r))((t, s, p, r) \in y(d))\}$, the set of subjects that have not been assigned in the schedule table y.

Let av : $Y \to \wp(D \times T)$ be given by

av$(y) = D \times T - \{(d, t) | (\exists(s, p, r))((t, s, p, r) \in y(d))\}$, the set of empty slots in the schedule table.

Suppose $S' \subset \text{Sub}$ is ordered according to the priority of instructors. Then, let selectS : $\wp(\text{Sub}) \times \text{Int} \to \wp(\text{Sub})$ be given by

selectS$(S', n) = S'' \leftrightarrow S'' \subset S'$, and the set S'' is the set of n elements in S' that have the highest priorities; the cardinality of S'' is n.

Suppose $DT' \subset D \times T$ is ordered according to preference over $D \times T$. In general, the earliest morning time slot is less preferred than the noontime slot. Then, let selectDT : $\wp(D \times T) \times \text{Int} \to \wp(D \times T)$ be given by

selectDT(DT', n) $= DT''$, where DT'' is the set of n elements in DT' that have the highest priorities; the cardinality of DT'' is n.

Let selectR : Sub $\to$ Room be given by

selectR(s) $= r$, where r is an optimal room for the subject s with respect to its size.

Let taughtP : Sub $\to$ Inst be given by

$$\text{taughtP}(s) = p \leftrightarrow (\exists n)(\text{taught}(s, p, n)).$$

Let

PriorP $= \{p | (\exists \alpha \in (\text{Sub} \times D \times T)^*)\ (\text{preference}(p) = \alpha)\} \subset$ Inst, the set of instructors who have preferences.

Let

PriorS $= \{s | (\exists p)(\exists n)(p \in \text{PriorP} \,\&\, \text{taught}(s) = (p, n))\} \subset$ Sub, the set of subjects that can have priorities due to priorities of the instructors.

Let desirabilities: PriorS $\to D \times T$ be given by

desirabilities(s) $= (d, t) \leftrightarrow p = \text{taughtP}(s) \,\&\, (s, d, t) \in \text{preference}(p)$; the class of the subject s is desired to be held on (d, t).

Then, genA : $Y \to \wp(A)$ is defined as follows: Suppose n is a fixed integer.

$$\begin{aligned}
&\text{genA}(y) = As \leftrightarrow \\
&\qquad S1 = \text{selectS}(\text{subject}(y), n) - \text{PriorS} \ \& \\
&\qquad S2 = \text{selectS}(\text{subject}(y), n) \cap \text{PriorS} \ \& \\
&\quad SPR1 = \{(s, \text{taughtP}(s), \text{selectR}(s)) | s \in S1\} \subset \text{Sub} \times \text{Inst} \times \text{Room} \ \& \\
&\qquad Av = \text{selectDT}(av(y), n) \ \& \\
&\qquad As1 = Av \times SPR1 \ \& \\
&\qquad As2 = \{((d, t), (s, \text{taughtP}(s), \text{selectR}(s))) | s \in S2 \,\&\, (d, t) = \text{desirabilities}(s)\} \ \& \\
&\qquad As = As2 \cup As1.
\end{aligned}$$

(v) constraint : $C \to$ {true, false}

The constraint of the class schedule problem is simple: constraint : $Y \to$ {true, false} is defined by

$$\text{constraint}(y) = \text{true for all } y \in Y.$$

In other words, there is no constraint. This comes from the strategy that conflicts among slot assignments are dealt with by δ.

(vi) Initial State

The initial state $y_0 \in Y$ is

$$y_0(d) = \text{nil for all } d \in D.$$

In other words, y_0 is the empty timetable.

(vii) Final State

The class schedule problem has meta–final states given by the PSE. Because the function of a final state is to specify a stopping condition of the solution-finding activity, let the set of final states $Y_f \subset Y$ be

$$y \in Y_f \leftrightarrow av(y) = \emptyset \vee \delta(y, a) \text{ is not valid for any } a \in \text{genA}(y).$$

In other words, a final state is a situation in which every slot is filled or no assignment is possible due to a conflict.

(vii) Stopping Condition

The stopping condition $\text{st} : Y \rightarrow \{\text{true}, \text{false}\}$ is given by the PSE as follows:

$$\text{st}(y) = \text{true} \leftrightarrow y \in Y_f.$$

11.2.4 PD Optimization Formulation

At this stage, we have to introduce an evaluation function for the output such as

$$\text{goal} : Y \rightarrow \text{Re}.$$

The current problem does not have a goal given by the PSE. When a goal is not given by the problem, one way to determine it is to use a heuristic estimation of the "distance" from the current situation y to Y_f. In particular, the current problem will use the function goal: $Y \rightarrow$ Re, defined by

$$\text{goal}(y) = |\text{subject}(y)|; \text{ number of subjects that have not yet been assigned.}$$

Then, we have a user model for the class schedule problem as follows:

$$\textbf{user model} = \langle A, Y, Y, \delta, \lambda, \text{genA}, \text{constraint}, \text{goal}, \text{st}, y_0, Y_f \rangle.$$

11.3 Implementation in extProlog

The entire user model of the class schedule problem is listed in Appendix 11.1. The class schedule is implemented in a frame structure of extProlog [Takahara et al. 2003]. Figure 11.2 shows an output or a solution of the current solver. According to the solution, on Monday the first class is [s13,p13,[r2,100]], that is, the subject is s13, whose instructor is p13 and the room r2 is assigned to it. The number of registered students for the class is 100. The solution illustrates the situation in which the scheduling cannot be completed due to conflicts. Two time slots, [thu,4] and [fri,4], remain unassigned.

Figure 11.3 shows a dialog output when the computation finishes. It states that the assignment for the slots is unsuccessful because there are conflicts for assignments of {s9,s10}. Here s9 is a subject given by instructor p9, who wants to have the class in

X timetable

4	A	B	C	D	E
0	mon	s13	s2	s4	s19
1		p13	p2	p4	p4
2		[r2,100]	[r1,50]	[r1,50]	[r1,50]
3	**				
4	tue	s14	s5	s6	s15
5		p14	p5	p6	p15
6		[r2,100]	[r2,100]	[r2,100]	[r2,100]
7	**				
8	wed	s17	s3	s18	s20
9		p2	p3	p3	p5
10		[r1,50]	[r2,100]	[r1,50]	[r1,50]
11	**				
12	thu	s21	s8	s11	
13		p6	p8	p11	
14		[r1,50]	[r2,100]	[r2,100]	
15	**				
16	fri	s1	s16	s12	
17		p1	p1	p12	
18		[r1,50]	[r2,100]	[r2,100]	
19	**				
20	sat	s22	s7		
21		p7	p7		
22		[r2,100]	[r2,100]		
23	**				
24					

Fig. 11.2. Class schedule.

X DIALOG

```
1,50]]],[[fri,1],[s1,p1,[r1,50]]],[[fri,2],[s16,p1,[r2
,100]]],[[mon,3],[s4,p4,[r1,50]]],[[mon,4],[s19,p4,[r1
,50]]],[[sat,1],[s22,p7,[r2,100]]],[[sat,2],[s7,p7,[r2
,100]]],[[mon,2],[s2,p2,[r1,50]]],[[tue,2],[s5,p5,[r2,
100]]],[[tue,3],[s6,p6,[r2,100]]],[[thu,2],[s8,p8,[r2,
100]]],[[thu,3],[s11,p11,[r2,100]]],[[fri,3],[s12,p12,
[r2,100]]],[[mon,1],[s13,p13,[r2,100]]],[[tue,1],[s14,
p14,[r2,100]]],[[tue,4],[s15,p15,[r2,100]]],[[wed,1],[
s17,p2,[r1,50]]],[[wed,4],[s20,p5,[r1,50]]],[[thu,1],[
s21,p6,[r1,50]]]]
"Yf=""state_frame.75"
WARNING:predicate 'timetableUp()' is not defined!!!
"assignment is not successful"
"may be there are conflicts about "[s9,s10]
classschedulePD.p" ends"
"normal reset state"
"May I help you?"
X>
```

Fig. 11.3. Dialog output of class schedule problem.

the fourth session on Monday. This time slot is already occupied by subject s15 of instructor p4, who has a higher priority than p9. Similarly, s10 is a subject given by instructor p10, who wants to have the class in the first session on Friday. This time slot is already occupied by subject s1 of instructor p1, who has a higher priority than p10.

Figure 11.4 shows a behavior of goal(). It decreases linearly, which implies that the value of goal() decreases by 1 for each action and there is no backtracking. The goal, however, cannot reach 0, and when it reaches 2, the computation is over. This implies that there are two slots that remain unfilled.

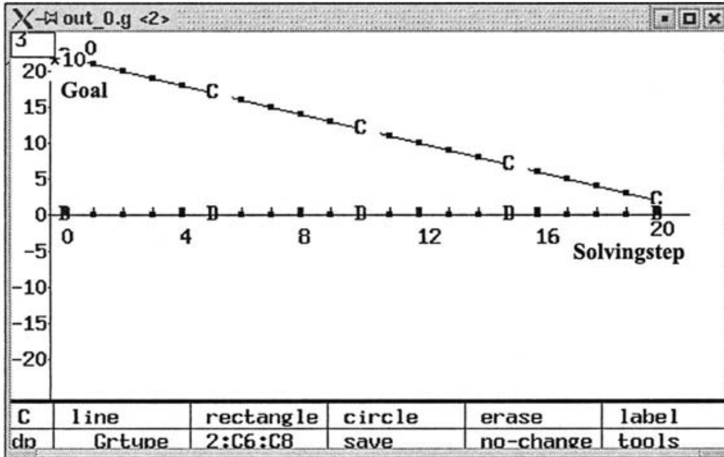


Fig. 11.4. Behavior of goal of class schedule problem.

11.4 Tuning of PD Goal-Seeker for Class Schedule Problem

The current problem has the following properties:

1. There is no backtracking.
2. There is no repetition of states.

The first property comes from the peculiar stopping condition of the problem. In general, the backtrack operation occurs when δ becomes invalid. However, as the above definition implies, this situation indicates that the solving activity has reached the stopping state for the current stopping condition.

The second property comes from the fact that $|\mathrm{av}(y)| > |\mathrm{av}(\delta(y, a))|$ holds for any y and a. Because of this property, H, the history of states, can be neglected in the stack. Because H consumes a large amount of memory, the discard of H is quite helpful for computation speed as well as saving memory.

Appendix 11.1 User Model for Class Schedule Problem

```
/*classschedule2.set*/
func([taught,roomSize,getFrameId,prev_goal,goal,min,selectR]);
frame(state_frame,[
          [.mon,[]],
```

```
            [.tue,[]],
            [.wed,[]],
            [.thu,[]],
            [.fri,[]],
            [.sat,[]],
            [.av,[]],
            [.subject,[]]]);
/*priority order*/
prof.g=[.p1,.p2,.p3,.p4,.p5,.p6,.p7,.p8,.p9,.p10,.p11,.p12,.p13,.p14,
        .p15];
/*order is specified in genPriority*/
sub.g=[.s1,.s2,.s3,.s4,.s5,.s6,.s7,.s8,.s9,.s10,.s11,.s12,.s13,.s14,.s15,
       .s16,.s17,.s18,.s19,.s20,.s21,.s22];
/*subject-prof-audience#*/
taught(s)=y <->
            L:=listPred("taughtD"),
            member([s,p,n],L),
            y:=[p,n];

taughtD(.s1,.p1,31) ;
taughtD(.s2,.p2,41) ;
taughtD(.s3,.p3,59) ;
taughtD(.s4,.p4,26) ;
taughtD(.s5,.p5,53) ;
taughtD(.s6,.p6,58) ;
taughtD(.s7,.p7,97) ;
taughtD(.s8,.p8,93) ;
taughtD(.s9,.p9,23) ;
taughtD(.s10,.p10,82) ;
taughtD(.s11,.p11,71) ;
taughtD(.s12,.p12,82) ;
taughtD(.s13,.p13,81) ;
taughtD(.s14,.p14,82) ;
taughtD(.s15,.p15,81) ;
taughtD(.s16,.p1,73) ;
taughtD(.s17,.p2,21) ;
taughtD(.s18,.p3,41) ;
taughtD(.s19,.p4,42) ;
taughtD(.s20,.p5,22) ;
taughtD(.s21,.p6,36) ;
taughtD(.s22,.p7,67) ;
/*room data*/
room.g=[.r1,.r2];
roomSize(r)=n <->
           l:=listPred("roomSizeD"),
           member([r,n],L);
roomSizeD(.r1,50) ;
roomSizeD(.r2,100);

/*preference of prof*/
preference(.p3,[[.s3,[.wed,2]],[.s18,[.wed,3]]]);
preference(.p1,[[.s1,[.fri,1]],[.s16,[.fri,2]]]);
preference(.p4,[[.s4,[.mon,3]],[.s19,[.mon,4]]]);
preference(.p9,[[.s9,[.mon,4]],[]]);
preference(.p10,[[.s10,[.fri,1]],[]]);
preference(.p7,[[.s22,[.sat,1]],[.s7,[.sat,2]]]);
delta(C,[[D,T],[S,P,R]])=CC <->
```

```
            Clss:=C->D,
            (member([T,SS,PP,RR],Clss)) ->
                    (
                    xwriteln(0,"conflict:[D,T]=",[D,T]),
                    CC:=[]
            )
            .otherwise
                    (
                    N:=getFrameId(),
                    defFrame(.state_frame,N,[]),
                    concat([.state_frame,".",N],CC),
                    CC->.mon:=C->.mon,
                    CC->.tue:=C->.tue,
                    CC->.wed:=C->.wed,
                    CC->.thu:=C->.thu,
                    CC->.fri:=C->.fri,
                    CC->.sat:=C->.sat,
                    Clss2:=append(Clss,[[T,S,P,R]]),
                    CC->D:=Clss2,
                    Nav:=minus(C->.av,[[D,T]]),
                    CC->.av:=Nav,
                      Nsubject:=minus(C->.subject,[S]),
                      CC->.subject:=Nsubject
                      );

/*generate alternative*/
genA(C)=As <->
            Ss0:=C->.subject,
            Av0:=C->.av,
           (Ss0=[] or Av0=[]) ->
                      (
                      As:=[]
                      )
           .otherwise
           (
           /*select 3 subjects*/
           project(Ss0,[[1,3]],Ss1),
           S1:=minus(Ss1,priorS.g),
           (S1<>[]) ->
                      (
                      //SPR=[[S,P,R],...]
SPR1:=defSet(pSPR(Y,S,[]),[S,S1]),
                      //*select 3 pos
                      Av:=project(Av0,[[1,3]]),
                      As1:=product(Av,SPR1)
                      )
           .otherwise
                      (
                      As1:=[]
                      ),
           //subject with priority
           S2:=minus(Ss1,S1),
           (S2=[]) ->
                      (
                      As:=As1
                      )
           /*append desirabilities*/
```

```
        .otherwise
                        (
As2:=defSet(pAs(Y2,X2,[]),[X2,S2]),
                        As:=append(As1,As2)
                        )
        );

pAs(Y2,PS,[]) <->
                        Ds:=desirabilities.g,
                        member([PS,[PD,PT]],Ds),
                        [PP,PSize]:=taught(PS),
                        PR:=selectR(PS),
                        Y2:=[[PD,PT],[PS,PP,PR]];

pSPR(Y,S,[]) <->
                        [P,Size]:=taught(S),
                        R:=selectR(S),
    Y:=[S,P,R];

selectR(S)=R <->
    [P,Size]:=taught(S),
    Rs:=room.g,
    minR0.g:=[-1,100000],
    Ys:=defSet(pRoom(Y,X,[Size]),[X,Rs]),
    R:=minR0.g;
pRoom(Y,R,[Size]) <->
    Rsize:=roomSize(R),
    (Rsize>=Size) ->
        (
        Y:=[R,Rsize],
        [R0,Rsize0]:=minR0.g,
        (Rsize<Rsize0) ->
                    (
                    minR0.g:=Y
                    )
        );


constraint(C) <->
    member(C,[]);

/*set up initial state*/
initialstate()=Fr <->
    N:=getFrameId(),
    defFrame(.state_frame,N,[]),
    concat([.state_frame,".",N],Fr),
    Sub0:=sub.g,
    PSub:=priorS.g,
    Sub:=union(PSub,Sub0),
    Fr->.subject:=Sub,
    /*favorable time order*/
    Fr->.av:=[[.mon,2],[.mon,3],
             [.tue,2],[.tue,3],
             [.wed,2],[.wed,3],
             [.thu,2],[.thu,3],
             [.fri,2],[.fri,3],
             [.mon,1],[.mon,4],
```

```
                [.tue,1],[.tue,4],
                [.wed,1],[.wed,4],
                [.thu,1],[.thu,4],
                [.fri,1],[.fri,4],
                [.sat,1],[.sat,2]];
/*generate frame #*/
getFrameId(N) <->
    .frameId(N),
    assign(.frameId,N+1);
.frameId(0);

st(C) <->
    R:=goal(C),
    V:=prev_goal.g,
    (V>R) ->
                (
                prev_goal.g:=R,
                .fail
                );
prev_goal.g=1000;

goal(C)=R <->
    R:=cardinality(C->.subject);

preprocess() <->
    genPrior(),
    assign(.frameId,0),
    timetableWp.g:=[];

/*generate priority*/
genPrior() <->
    priorS.g:=[],
    desirabilities.g:=[],
    L:=listPred("preference"),
    Ps:=defSet(pgenPrior(P,X,[]),[X,L]),
    assign(.priorP,Ps),
    Sub0:=sub.g,
    PSub:=priorS.g,
    Sub1:=union(PSub,Sub0),
    sub.g:=Sub1;
pgenPrior(P,[P,Y],[]) <->
    desirabilities.g:=append(desirabilities.g,Y),
    genPriorS(P,Y);

genPriorS(P,Y) <->
    Ss:=defSet(pPriorS(S,Z,[P]),[Z,Y]),
    priorS.g:=append(priorS.g,Ss);
pPriorS(S,[S,[D,T]],[P]);

preprocess() <->
    _Yf(Cf),
    lambda(Cf),
    (Cf->.subject<>[]) ->
        (
        xwriteln(0,"assignment is not successful"),
        xwriteln(0,"maybe there are conflicts about",Cf->.subject)
        )
```

```
    .otherwise
        (
        xwriteln(0,"assignment is successful")
        );

/*display time table*/
lambda(Fr) <->
    Wp0:=timetableWp.g,
    (Wp0<>[]) ->
        (
        Wp:=Wp0
        ),
    makewindowSS(Wp,"timetable",X1,Y1,5,25),
    clearsheet(Wp),
    timetableWp.g:=Wp,
    DY.g:=[[.mon,0],[.tue,4],[.wed,8],[.thu,12],[.fri,16],[.sat,20]],
    dummy2:=defList(pclass2(dum2,x2,[Fr,Wp]),[x2,DY.g]);
pclass2(dum2,[D,Y],[Fr,Wp]) <->
    Ls:=Fr->D,
    sheet(Wp,0,Y):=D,
    sheet(Wp,0,Y+3):="**************************************************",
    (Ls<>[]) ->
        (
        dummy:=defList(pclass(dum,x,[Wp,Y]),[x,Ls])
        ),
    dum2:="OK2";

pclass(dum,[T,S,P,R],[Wp,Y]) <->
    sheet(Wp,T,Y):=S,
    sheet(Wp,T,Y+1):=P,
    sheet(Wp,T,Y+2):=R,
    dum:="OK";
```

The user model of the time schedule problem is longer than those of the previous problems because the present problem requires more data than the others. The predicate preprocess() rearranges data according to the priorities. The predicate postprocess() displays Fig. 11.2.

References

Takahara, Y. et al. (2003) *User's Manual for the 4-th Generation Systems Development Methodology*, internal report, Dept. of Management Information Science, Chiba Institute of Technology, Chiba, Japan.

12

Data Mining Problem: I-O-O Problem

This chapter discusses a data mining system as a nontrivial example of the model theory approach. Needless to say, the system has a special significance for the model theory approach due to its role in the intelligent management information system of Fig. 1.2. Because a data mining system is not usually considered as a problem-solving system, this trial may seem strange. This trial has three objectives. First, from a theoretical viewpoint, this trial demonstrates how the model theory approach is applicable to a more realistic system than a simple academic example. It is shown that although a data mining problem is the least structured, the model theory approach is applicable to it in constructing a solver. Second, from a practical viewpoint, because the model theory approach is strong in exploration of the structure of a target system, good insight into a data mining system can be obtained throughout the trial. This is beneficial in practice because a data mining system has become crucial for sofisticated management. Third, this chapter shows how tuning of stage 6 of Fig. 4.6 can be performed using the insights. Because the goal-seeker of the model theory approach is developed on a general level, it is desirable in practice that a generated solver be tuned according to specific properties of the target.

The stdPDsolver will be used for the present system as the goal-seeker.

12.1 Data Mining Problem

A data mining system of this chapter is modeled as an input and output system, in which the input is a data set from a relational database and the output is a decision tree for the set. The tree is assumed to reveal semantic implications of the data set. The tree form is selected as the output form, because among the output forms of a data mining system, a decision tree form may be the most complicated. The most crucial point for this type of data mining system is how to define the evaluation criterion for a decision tree. This chapter adopts the idea of entropy commonly used for data mining systems.

12.2 User Model of Data Mining Problem for PD Goal-Seeker

A user model for the data mining system is constructed following Fig. 4.6.

12.2.1 Drawing Input–Output Block Diagram

This stage presents a conceptual description of a target extended solver (extSLV) specifying the problem specification environment (PSE) and the solution (output). Figure 12.1 shows an input–output block diagram of a solver for the data mining problem.

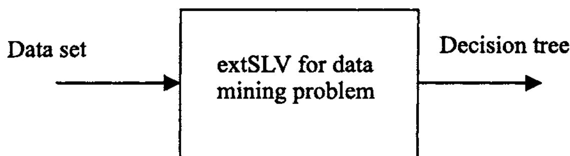


Fig. 12.1. Data mining solver problem.

The data mining solver of this chapter is supposed to derive a decision tree from a given data set. Figure 12.2 shows an example of a data set that is used for illustration in this chapter [Witten and Frank, 2000]. It shows the advice of an optician to a customer.

The data set shown in Fig. 12.2 has five attributes: lenses, age, prescription, astigmatism, and tear. It states that if a customer has values, e.g., young, myopic, no, and normal, for the attributes of age, prescription, astigmatism, and tear, respectively, then the advice of the optician will be that the lenses should be soft. The attribute "lenses" is then called an output attribute, while the other attributes are called input attributes.

Figure 12.3 illustrates an example of a decision tree that corresponds to the above data set. In fact, it is the actual output of the data mining solver of this chapter. The presentation form of the tree is slightly different from the usual one because it is displayed in the spreadsheet of extProlog. A node is denoted by a pair of a branch name and a node number. For example, node 1 is denoted by [reduced, 1]. It states that the node 1 is linked (with node 0) by the branch "reduced." In the same way, node 0 is linked with node 2 by the branch "normal," while node 2 is connected with nodes 3 and 9 by branches "yes" and "no," respectively.

Some nodes represent attributes and are called attribute nodes. Node 0, which represents the attribute "tear," is an attribute node. Node 1 represents a terminal that does not represent an attribute, and is called a terminal node. It will be shown in subsequent sections that an attribute node is a node that requires more detailed analysis by tree expansion, while a terminal node is one that does not.

Nodes 0, 2, 3, 4, 9, and 10 are attribute nodes, while the remaining nodes are terminal nodes. The relation between a node number and an attribute is shown at the bottom of Fig. 12.3. For example, node 2 corresponds to the attribute "astigmatism."

```
ALst=[lenses,age,prescription,astigmatism,tear],
D0:=[
        [soft,young,myope,no,normal],
        [none,young,myope,no,reduced],
        [none,young,myope,yes,reduced],
        [hard,young,myope,yes,normal],
        [none,young,hypermetrope,no,reduced],
        [soft,young,hypermetrope,no,normal],
        [none,young,hypermetrope,yes,reduced],
        [hard,young,hypermetrope,yes,normal],
        [none,pre_presbyopic,myope,no,reduced],
        [soft,pre_presbyopic,myope,no,normal],
        [none,pre_presbyopic,myope,yes,reduced],
        [hard,pre_presbyopic,myope,yes,normal],
        [none,pre_presbyopic,hypermetrope,no,reduced],
        [soft,pre_presbyopic,hypermetrope,no,normal],
        [none,pre_presbyopic,hypermetrope,yes,reduced],
        [none,pre_presbyopic,hypermetrope,yes,normal],
        [none,presbyopic,myope,no,reduced],
        [none,presbyopic,myope,no,normal],
        [none,presbyopic,myope,yes,reduced],
        [hard,presbyopic,myope,yes,normal],
        [none,presbyopic,hypermetrope,no,reduced],
        [soft,presbyopic,hypermetrope,no,normal],
        [none,presbyopic,hypermetrope,yes,reduced],
        [none,presbyopic,hypermetrope,yes,normal]].
```

Fig. 12.2. Example of input: data set.

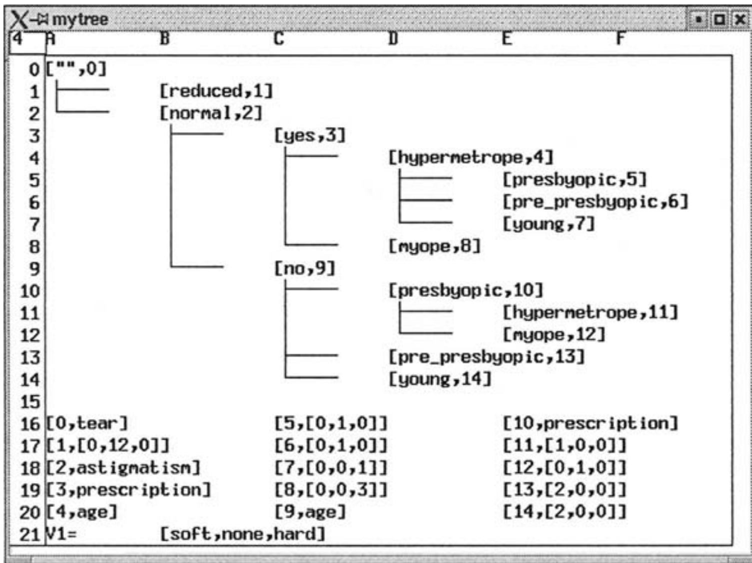


Fig. 12.3. Example of output: decision tree corresponding to Fig. 12.2.

Because node 1 is a terminal node, it does not have an attribute name but has a frequency distribution [0, 12, 0] over the attribute value list V_1 = [soft, none, hard] of the output attribute "lenses" ("lenses" takes a value from {soft, none, hard}). The list V_1 is also presented at the bottom of Fig. 12.3. The distribution at node 1 indicates that the data set of Fig. 12.2 has exactly $12(= 0 + 12 + 0)$ data elements with the value "reduced" for the attribute "tear," which is the attribute of node 0, and all of these data elements have "none" as the value of the output attribute "lenses." This fact can be confirmed with reference to Fig. 12.2.

The decision tree shows that if a customer's tear is "normal" (the attribute of the node 0 = tear), astigmatism is "yes" (the attribute of node 2 = "astigmatism"), the prescription is hypermetrope (the attribute of node 3 = "prescription"), and the customer age is presbyopic (the attribute of node 4 = "age"), then the advice will be "lenses" = none with 100% certainty because the value distribution of the node 5 is [0, 1, 0]. However, because the supporting number is $1 = 0 + 1 + 0$, this is not strong advice.

12.2.2 Input–Output Specification in Set Theory

At this stage, the input PSE and the output solution are formalized in set-theoretic terms. It is usual that the PSE is represented by a structure rather than by simple sets. The output set is a set of solution candidates. Figure 12.4 illustrates stage 2 of the data mining solver.

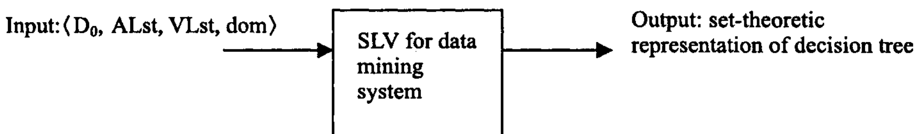


Fig. 12.4. Input–output specification of the data mining solver in set theory.

(i) Input Specification

The input is given by the structure

$$\langle D_0, \text{ALst}, \text{VLst}, \text{dom} \rangle,$$

the components of which are specified as follows:

a. Let the attribute set of the given data set be

$$\text{ALst} = \{a_i | I \in I_a\} \quad (I_a = \{1, 2, \ldots, n\}),$$

where a_1 is the output attribute and $a_2, \ldots, a_n$ are input attributes.

In Fig. 12.2, $n = 5$, $I_a = \{1, 2, 3, 4, 5\}$, and ALst = {lenses, age, prescription, astigmatism, tear}. That is, a_1 = lenses, a_2 = age, a_3 = prescription, a_4 = astigmatism, and a_5 = tear.

Let

$$I_a^+ = I_a - \{1\}$$

and

$$\text{ALst}^{+} = \{a_i | i \in I_a^{+}\}.$$

b. Let dom : ALst → VLst be given by

$$V_i = \text{dom}(a_i) : \text{domain of } a_i \text{ or value set of } a_i.$$

Let

$$\text{VLst} = \{V_1, \ldots, V_n\}.$$

In Fig. 12.2, we have

$$\begin{aligned} V_1 &= \{\text{soft, none, hard}\}, \\ V_2 &= \{\text{young, pre_presbyopic, presbyopic}\}, \\ V_3 &= \{\text{myope, hypermetrope}\}, \\ V_4 &= \{\text{no, yes}\}, \\ V_5 &= \{\text{reduced, normal}\}. \end{aligned}$$

Then, for example, dom $(a_1) = V_1 = \{\text{none, soft, hard}\}$.

Let

$$V = \cup \text{VLst}$$

and

$$V^{+} = V_2 \cup \cdots \cup V_n.$$

c. Let

$$X = \wp(V_1 \times \cdots \times V_n) \quad (\text{the power set of } V_1 \times \cdots \times V_n).$$

Then, a target data set D_0 of the data mining solver is an element of X, i.e.,

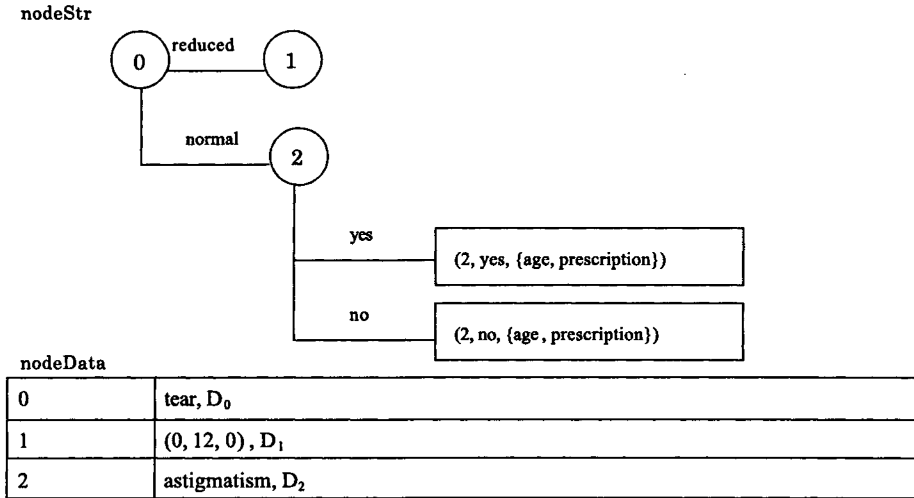
$$D_0 \in X : \text{input data set of data mining.}$$

Figure 12.2 shows D_0 of the example which is used in this chapter.

(ii) Output Specification

Figure 12.3 shows a final decision tree. The output Y is a set of intermediate decision trees including final ones. An intermediate decision tree is essentially a subtree of a final decision tree. Let us consider the intermediate tree shown in Fig. 12.5:

An intermediate tree consists of two components: nodeStr and nodeData. The component nodeStr describes the tree structure, while nodeData represents the node properties. The node (2, yes, {age, prescription}) of nodeStr of Fig. 12.5 is a special node that exists only in an intermediate tree. This node is a temporary node that is either expanded as another subtree using an attribute in {age, prescription} or transformed into a terminal node like node 1. It is called a limbo node. According to the final tree of Fig. 12.3, the limbo node (2, yes, {age, prescription}) is transformed into node 3 using the attribute "prescription," while the other limbo node (2, no, {age, prescription}) is transformed into node 9 using the attribute "age" on its expansion. The principal



0	tear, D_0
1	(0, 12, 0), D_1
2	astigmatism, D_2

Fig. 12.5. Intermediate tree.

element of a limbo node is the remaining attributes of ALst$^+$ that are not used on the path from node 0 to the limbo node. For instance, {age, prescription} is the principal element of (2, no, {age, prescription}). It should be clear that

$$\{\text{age, prescription}\} = \text{ALst}^+ - \{\text{attribute of node 0, attribute of node 2}\}.$$

Consequently, there are three kinds of node in nodeStr: attribute node, terminal node, and limbo node. A limbo node does not have a node number. The value of nodeStr of Fig. 12.5 can be represented as follows:

$$\text{nodeStr} = \{(0, \text{reduced}, 1), (0, \text{normal}, 2), (2, \text{yes}, \{\text{age, prescription}\}),$$
$$(2, \text{no}, \{\text{age, prescription}\})\}.$$

That is, an element of nodeStr takes the form (n, v, n') or $(n, v, \text{set of attributes})$, where $n, n' \in N$ and $v \in V^+$.

Properties of attribute nodes and terminal nodes are given by nodeData, which in Fig. 12.5 is represented as

$$\text{nodeData} = \{(0, \text{tear}, D_0), (1, (0, 12, 0), D_1), (2, \text{astigmatism}, D_2)\}.$$

It shows, for example, that node 2 represents the attribute "astigmatism" and data D_2, which is a subset of D_0 and is given by

$$D_2 = \{d | d \in D_0 \text{ and } d(\text{tear}) = \text{normal}\},$$

where $d(\text{tear})$ is the value of data element d for the attribute "tear." In other words, D_2 is the set of data elements in D_0 that satisfy the branch values on the path from node 0 to node 2.

Similarly, an element $(1, (0, 12, 0), D_1)$ of nodeData indicates that node 1 is a terminal node (because the second component is not an attribute name) and D_1 is given by

$$D_1 = \{d | d \in D_0 \text{ and } d(\text{tear}) = \text{reduced}\}.$$

As mentioned above, (0, 12, 0) of node 1 indicates that at the node the attribute a_1 = lenses takes the value soft 0 times, none 12 times, and hard 0 times in D_1.

Formally, an element of nodeData takes the form (n, attr, D) or $(n, \text{list of integers}, D)$, where $n \in N$, attr $\in$ ALst$^+$, and $D \in X$.

The output y corresponding to Fig. 12.5 is given by

$$y = (\text{nodeStr}, \text{nodeData}).$$

Now, let us formalize Y. Let

$$N = \{0, 1, 2, \ldots\}.$$

Then, because $(n, v, n') \in N \times V^+ \times N$ and $(n, v, \text{set of attributes}) \in N \times V^+ \times \wp(\text{ALst}^+)$, we have

$$\text{nodeStr} \subset \wp(N \times V^+ \times (N \cup \wp(\text{ALst}^+))),$$

or

$$\text{nodeStr} \in \wp(\wp(N \times V^+ \times (N \cup \wp(\text{ALst}^+)))).$$

Similarly, we have

$$\text{nodeData} \subset \wp(N \times (N^* \cup \text{ALst}^+) \times X),$$

or

$$\text{nodeData} \in \wp(\wp(N \times (N^* \cup \text{ALst}^+) \times X)).$$

Here N^* is the free monoid of N, or the set of strings on N. In this book, a string and its vector expression are used interchangeably.

It should be noted that nodeData is a functional relation from N into $(N^* \cup \text{ALst}^+) \times X$, i.e., nodeData : $N \to (N^* \cup \text{ALst}^+) \times X$ is a partial function.

Finally, y is given as

$$y \in \wp(\wp(N \times V^+ \times (N \cup \wp(\text{ALst}^+))) \times \wp(\wp(N \times (N^* \cup \text{ALst}^+) \times X)),$$

or

$$Y \subset \wp(\wp(N \times V^+ \times (N \cup \wp(\text{ALst}^+))) \times \wp(\wp(N \times (N^* \cup \text{ALst}^+) \times X)).$$

Although Y has a complicated structure, it is not important whether Y can be easily understood, but it is important that Y be formally described so that we can proceed to the next step of the solver development.

12.2.3 Process Specification as Automaton

At this stage, the process δ of the goal-seeking system is determined as an automaton. In particular, delta(), genA(), constraint(), initialstate(), finalstate(), and st() are specified.

(i) Action Set

Because the data mining solver is a case of implicit solving activity, the general result of Chapter 5 shows that an action set A is defined in the following way:

$$A = \{a | a : Y \to Y\}.$$

Of course it is not easy to manipulate an action specified as a mapping. Following the usual procedure used in the previous chapters, let us replace a mapping by a parameter that characterizes it.

Considering Fig. 12.5, suppose an action is applied to the limbo node (2, yes, {age, prescription}) to expand the intermediate tree as shown in Fig. 12.6.

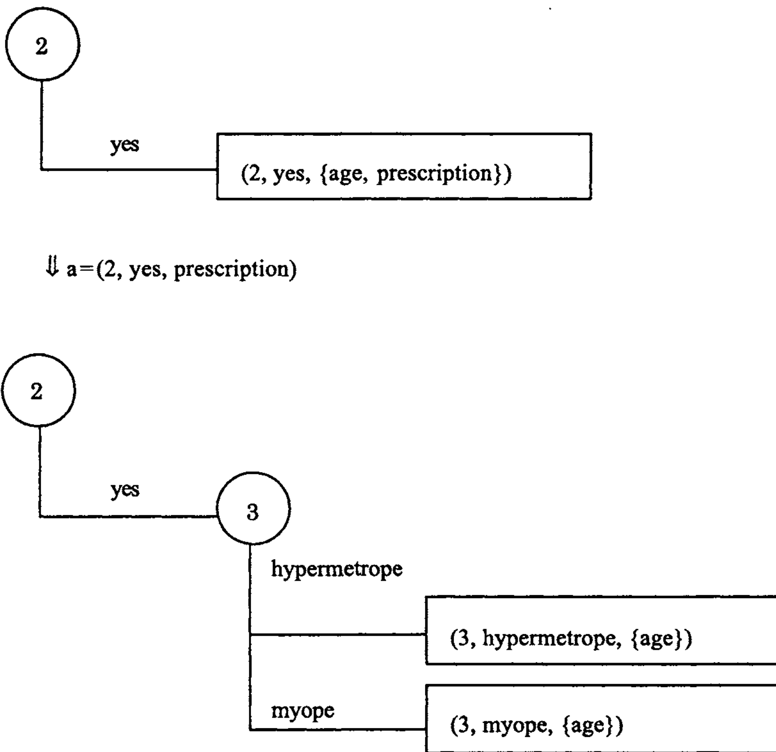


Fig. 12.6. Expansion of intermediate tree.

The action of Fig. 12.6 transforms the limbo node into an attribute node "prescription" with a node number of 3. Consequently, the action can be characterized by a parameter a = (2, yes, prescription), where (2, yes) specifies to which limbo node the action is applied, and prescription shows what attribute is assigned to the node. In fact, according to Fig. 12.3 the limbo node becomes a new node 3, which represents the attribute "prescription," and two limbo nodes, (3, hypermetrope, {age})

and (3, myope, {age}), are generated. It must be clear that {age} is specified by {age} = {age, prescription} − {prescription}. Let us formalize the above idea.

Because an action that is applied to a limbo node (n, v, As), $\text{As} \subset \text{ALst}^+$, is specified by (n, v, attr) for some attr $\in$ As, let

$$A = N \times V^+ \times (\text{ALst}^+ \cup \{\text{dummy}\}) : \text{set of actions,}$$

where dummy is a special symbol used when a limbo node is transformed into a terminal node.

(ii) State Transition Function

Let leaf : $\{\text{nodeStr}\} \to \wp(N \times V \times \wp(\text{ALst}^+))$ be such that

$$\text{leaf(nodeStr)} = \{(n, v, \text{As}) | (n, v, \text{As}) \in \text{nodeStr and As} \subset \text{ALst}^+\}.$$

Then leaf(nodeStr) specifies the set of limbo nodes in nodeStr. Let getNdId: $\{\text{nodeData}\} \to N$ be such that

$$\text{getNdId(nodeData)} = n' \leftrightarrow n' = \max\{n | (\exists\, \text{attr}, D)((n, \text{attr}, D) \in \text{nodeData})\} + 1.$$

Then getNdId(nodeData) yields a node number for the next newly generated node.

A state transition function $\delta : Y \times A \to Y$ is specified using the above definitions. Let (nodeStr, nodeData) $\in Y$ and $(n, v, \text{attr}') \in A$ be arbitrary. Note that if an action (n, v, attr') is applicable, there is a unique limbo node (n, v, As) such that $(n, v, \text{attr}') \in (n, v, \text{As})$ in nodeStr. Let n' = getNdId(nodeData), $(n, \text{attr}, D) \in$ nodeData, and $D' = \{d | d \in D, d(\text{attr}) = v\}$. Then, $\delta((\text{nodeStr, nodData}), (n, v, \text{attr}'))$ = (nodeStr′, nodeData′) is given as follows:

a. If attr′ $\neq$ dummy,

$$\begin{aligned} \text{nodeStr}' &= (\text{nodeStr} - \{(n, v, \text{As})\}) \cup \{(n, v, n')\} \cup \{(n', v', \text{As} - \{\text{attr}'\}) | \\ &\qquad v' \in \text{dom}(\text{attr}')\}, \\ \text{nodeData}' &= \text{nodeData} \cup \{(n', \text{attr}', D')\}. \end{aligned}$$

b. If attr′ = dummy,

$$\begin{aligned} \text{nodeStr}' &= (\text{nodeStr} - \{(n, v, \text{As})\}) \cup \{(n, v, n')\}, \\ \text{nodeData}' &= \text{nodeData} \cup \{(n', \text{elementN}(D, \text{attr}, v, V_1), D')\}, \end{aligned}$$

where elementN(D, attr, v, V_1) is defined in Section 12.2.4.

In general, δ is a partial function. As usual, in order to ensure that δ behaves properly, two functions, genA : $Y \to \wp(A)$ and constraint : $Y \to \{\text{true, false}\}$, are used. Then, δ must satisfy the following relation: for a legitimate state y,

$$\delta(y, a) = y' \to a \in \text{genA}(y) \text{ and } \text{constraint}(y') = \text{true}.$$

(iii) Output Function

The output function $\lambda : Y \times A \to Y$ is naturally

$$\lambda(y, a) = y.$$

(iv) genA: $Y \to \wp(A)$

Let (nodeStr, nodeData) be arbitrary. Let (n, v, As) be the first element of leaf(nodeStr) if leaf(nodeStr) $\neq \varnothing$. (Because a set is implemented as a list, leaf(nodeStr) is linearly ordered). Let $(n, \text{attr}, D_n) \in$ nodeData. Then,

$$\text{genA}((\text{nodeStr}, \text{nodeData})) = \begin{cases} \varnothing & \text{if leaf(nodeStr)} = \varnothing, \\ \{((n, v), \text{dummy})\} & \text{if inf(elementN}(D_n, \text{attr}, v, V_1)) \\ & = 0 \text{ or As} = \varnothing, \\ \{(n, v)\} \times \text{As} & \text{otherwise,} \end{cases}$$

where inf(elementN$(D_n, \text{attr}, v, V_1)$), a measure of information, is defined in Section 12.2.4.

(v) Constraint

The function constraint is trivial for the current case, i.e.,

$$\text{constraint}(y) = \text{true}.$$

There is no constraint.

(vi) Initial State

The initial state $y_0 \in Y$ is given as a trivial situation:

$$y_0 = (\{(-1, [], \text{ALst}^+)\}, \{(-1, [], D_0)\}).$$

That is, nodeStr of y_0 consists of one limbo node $(-1, [], \text{ALst}^+)$, where -1 and [] indicate a virtual node number and a virtual branch value, respectively.

(vii) Final State

The set of final states $Y_f \subset Y$ is specified as follows:

$$Y_f = \{y \mid y = (\text{nodeStr}, \text{nodeData}) \ \& \ \text{leaf(nodeStr)} = \varnothing\}.$$

That is, $y \in Y_f$ is a situation in which no action is feasible.

(viii) Stopping Condition

Let the stopping condition st : $Y \to$ {true, false} be given by

$$\mathrm{st}(y) = \mathrm{true} \leftrightarrow y \in Y_{\mathrm{f}}.$$

The solving activity is suspended when no action is feasible.

Let us consider Fig. 12.5. In this case,

$$\begin{aligned}\mathrm{leaf}(\mathrm{nodeStr}) &= \{(2, \mathrm{yes}, \{\mathrm{age}, \mathrm{prescription}\}), (2, \mathrm{no}, \{\mathrm{age},\\ &\quad \mathrm{prescription}\})\},\\ \mathrm{genA}((\mathrm{nodeStr}, \ \mathrm{nodeData})) &= \{((2, \mathrm{yes}), \mathrm{age}), ((2, \mathrm{yes}), \mathrm{prescription})\}.\end{aligned}$$

Let

$$a = ((2, \mathrm{yes}), \mathrm{prescription}) \in \mathrm{genA}((\mathrm{nodeStr}, \mathrm{nodeData})).$$

Then

$$n' = 3 = \mathrm{getNdId}(\mathrm{nodeData}).$$

Because prescription $\neq$ dummy,

$$\begin{aligned}\mathrm{nodeStr}' &= \{(0, \mathrm{reduced}, 1), (0, \mathrm{normal}, 2), (2, \mathrm{no}, \{\mathrm{age}, \mathrm{prescription}\}),\\ &\quad (2, \mathrm{yes}, 3), (3, \mathrm{hypermetrope}, \{\mathrm{age}\}), (3, \mathrm{myope}, \{\mathrm{age}\})\},\\ \mathrm{nodeData}' &= \mathrm{nodeData} \cup \{(3, \mathrm{prescription}, D_3)\},\end{aligned}$$

where $D_3 = \{d | d \in D_2 \ \& \ d(\mathrm{astigmatism}) = \mathrm{yes}\}$.

The automaton specification of the process is then given as

$$\text{automaton specification of process} = \langle A, Y, \delta, \lambda, \mathrm{genA}, \mathrm{constraint}, \mathrm{st}, y_0, Y_{\mathrm{f}} \rangle.$$

12.2.4 PD Optimization Formulation

Because an automaton formulation is a discrete state space representation, if an evaluation function goal : $Y \to$ Re is combined with it, we have a dynamic optimization formulation or a user model.

Because the current problem is of the I-O-O type, there is no legitimate goal for the problem. We can design an appropriate goal. In fact, we adopt a heuristic method that evaluates an intermediate tree by its uncertainty using the entropy concept [Witten and Frank, 2000]. The tree is expanded in the direction that minimizes its uncertainty.

Let us define an uncertainty concept of a tree. Suppose $y =$ (nodeStr, nodeData) is fixed.

a. Let

$$\mathrm{leaf}(\mathrm{nodeStr}) = \{(n_1, v_1, \ell_1), \ldots, (n_k, v_k, \ell_k)\}.$$

b. Let

$$(n_i, \mathrm{attr}_i, D_i) \in \mathrm{nodeData}. (i = 1, \ldots, k) \quad (\text{Note that } n_i \text{ is given in leaf(nodeStr)}).$$

c. Let

$$n_{\text{it}} = |\{d|d \in D_i \ \& \ d(\text{attr}_i) = v_i \ \& \ d(\text{attr}_1) = v_{1t}\}| \quad (i = 1, \ldots, k),$$

where |Set| is the cardinality of the set Set and $\text{dom}(\text{attr}_1) = V_1 = \{v_{11}, \ldots, v_{1s}\}$. Let

$$\text{elementN}(D_i, \text{attr}_i, v_i, V_1) = (n_{i1}, \ldots, n_{\text{is}}).$$

d. Let

$$p_{\text{it}} = n_{\text{it}} / \sum_t n_{\text{it}}$$

and

$$\inf(\text{elementN}(D_i, \text{attr}_i, v_i, V_1)) = -\sum_t p_{\text{it}} \log(p_{\text{it}}) = \inf(n_{i1}, \ldots, n_{\text{is}}).$$

Here p_{it} is assumed to give the probability for the event that the attribute a_1 takes the value v_{1t} at the limbo node (n_i, v_i, ℓ_i). The function inf() gives an entropy measure of $\text{elementN}(D_i, \text{attr}_i, v_i, V_1)$.

e. Let $\text{goal0} : (N \times V \times \wp(\text{ALst}^+)) \times \{\text{nodeData}\} \to \text{Re}$ be given by

$$\text{goal0}(n_i, v_i, \ell_i, \text{nodeData}) = (|D_i'|/|D_0|)\inf(\text{elementN}(D_i, \text{attr}_i, v_i, V_1)),$$

where $(n_i, \text{attr}_i, D_i) \in \text{nodeData}$ and $D_i' = \{d|d \in D_i \& d(\text{attr}_i) = v_i\}$. Here $\text{goal0}(n_i, v_i, \ell_i, \text{nodeData})$ is an uncertainty measure at the limbo node (n_i, v_i, ℓ_i).

f. Finally, let $\text{goal} : Y \to \text{Re}$ be given by

$$\text{goal}(y) = \sum \{\text{goal0}(n_i, v_i, \ell_i, \text{nodeData}) | (n_i, v_i, \ell_i) \in \text{leaf}(\text{nodeStr})\}.$$

It should be noted that goal(y) neglects uncertainties associated with terminal nodes because goal() is used for evaluation of actions, while actions are related only to limbo nodes and uncertainties of terminal nodes are not related to the evaluation.

Let us illustrate goal() using Fig. 12.6.

Let $a = (2, \text{yes}, \text{prescription}) \in \text{genA}((\text{nodeStr}, \text{nodeData}))$.

Let $\text{leaf}(\text{nodeStr}) = \{(2, \text{yes}, \{\text{age}, \text{prescription}\})\}$, where $n_2 = 2, v_2 = \text{yes}$, and $\ell_2 = \{\text{age}, \text{prescription}\}$.

Let

$$\begin{aligned} D_2 = \{&(\text{soft, young, myope, no, normal}),\\ &(\text{hard, young, myope, yes, normal}),\\ &(\text{soft, young, hypermetrope, no, normal}),\\ &(\text{hard, young, hypermetrope, yes, normal}),\\ &(\text{soft, pre_presbyopic, myope, no, normal}),\\ &(\text{hard, pre_presbyopic, myope, yes, normal}),\\ &(\text{soft, pre_presbyopic, hypermetrope, no, normal}),\\ &(\text{none, pre_presbyopic, hypermetrope, yes, normal}),\\ &(\text{none, presbyopic, myope, no, normal}),\end{aligned}$$

(hard, presbyopic, myope, yes, normal),
(soft, presbyopic, hypermetrope, no, normal),
(none, presbyopic, hypermetrope, yes, normal)}.

Then, $(n_2, \text{attr}_2, D_2) = (2, \text{astigmatism}, D_2) \in$ nodeData,

$$
\begin{aligned}
D'_2 &= \{d \mid d \in D \ \&\ d(\text{astigmatism}) = \text{yes}\} \\
&= \{(\text{hard, young, myope, yes, normal}), \\
&\quad (\text{hard, young, hypermetrope, yes, normal}), \\
&\quad (\text{hard, pre_presbyopic, myope, yes, normal}), \\
&\quad (\text{none, pre_presbyopic, hypermetrope, yes, normal}), \\
&\quad (\text{hard, presbyopic, myope, yes, normal}), \\
&\quad (\text{none, presbyopic, hypermetrope, yes, normal})\},
\end{aligned}
$$

$$
\begin{aligned}
n_{21} &= |D''_{\text{soft}}| = |\{d \mid d \in D_2 \ \&\ d(\text{astigmatism}) = \text{yes} \ \&\ d(\text{lenses}) = \text{soft}\}| = 0, \\
n_{22} &= |D''_{\text{none}}| = |\{d \mid d \in D_2 \ \&\ d(\text{astigmatism}) = \text{yes} \ \&\ d(\text{lenses}) = \text{none}\}| = 2, \\
n_{23} &= |D''_{\text{hard}}| = |\{d \mid d \in D_2 \ \&\ d(\text{astigmatism}) = \text{yes} \ \&\ d(\text{lenses}) = \text{hard}\}| = 4,
\end{aligned}
$$

$\text{elementN}(D_2, \text{astigmatism}, \text{yes}, \{\text{soft, none, hard}\}) = (0, 2, 4)$,

$$
\begin{aligned}
p_{21} &= n_{21}/\sum_t n_{2t} = 0/(0+2+4) = 0, \\
p_{22} &= n_{22}/\sum_t n_{2t} = 2/(0+2+4) = 1/3 = 0.333333, \\
p_{23} &= n_{23}/\sum_t n_{2t} = 4/(0+2+4) = 2/3 = 0.666667,
\end{aligned}
$$

$$
-\sum_t p_{2t} \log(p_{2t}) = \inf(n_{21}, n_{22}, n_{23}) = \inf(0, 2, 4) = 0.918296,
$$

$$
\begin{aligned}
\text{goal0}(2, \text{yes}, \{\text{age, prescription}\}, \text{nodeData}) &= (|D'_2|/|D_0|) \inf(n_{21}, n_{22}, n_{23}) \\
&= (6/24) * 0.918296 = 0.229574,
\end{aligned}
$$

where because leaf(nodeStr) is a singleton, $\text{goal}(y) = \text{goal0}(2, \text{yes}, \{\text{age, prescription}\}, \text{nodeData}) = 0.229574$.

A user model for the data mining problem for the PD goal-seeker is as follows:

$$
\textbf{user model} = \langle A, Y, \delta, \lambda, \text{goal}, \text{genA}, \text{constraint}, \text{st}, y_0, Y_f \rangle.
$$

12.3 Implementation in extProlog

At this stage, the user model is implemented in set theory and is compiled. On compilation the standardized goal-seeker, stdPDsolver122.p, is attached to the user model.

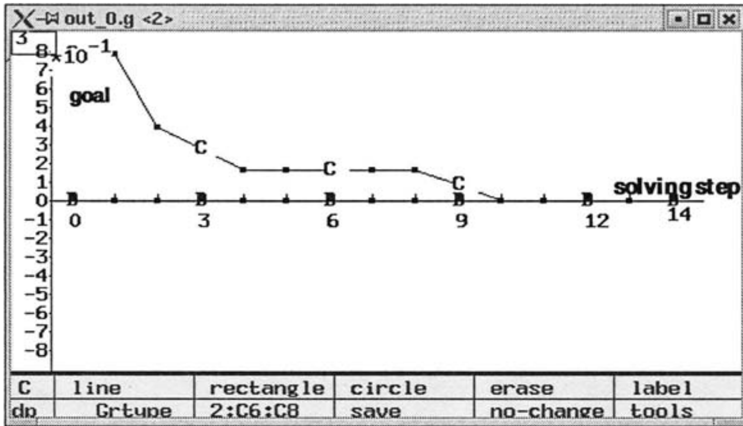


Fig. 12.7. Dynamic behavior of data mining system.

We then have a working data mining solver. Appendix 12.1 presents the entire user model, and Fig. 12.3 shows a decision tree generated by the solver.

Figure 12.7 shows the dynamic behavior of the goal for the example of Fig. 12.2.

The horizontal coordinate of Fig. 12.7 shows actions or corresponds to node creations. Fifteen nodes are created. The vertical coordinate shows the values of the goal or the uncertainty of intermediate trees. Because a value of the horizontal coordinate corresponds exactly to a node number of Fig. 12.3, Figure 12.7 shows that the extSLV decreases the uncertainty of the tree without backtracking as the tree expands. The flat parts of Fig. 12.7 correspond to transformation of limbo nodes into terminal nodes. The optimum goal value = 0 is attained at the tenth node creation, although the final state is not reached. When the fourteenth node is created, Fig. 12.3 is given as a final output.

12.4 Tuning of PD Goal-Seeker for Data Mining Problem*

At stage 5 of Fig. 4.6, a workable data mining solver has been generated. However, because the data mining solver has some convenient properties, the solver can be improved by tuning. We will show that the user model and the goal-seeker can be simplified by tuning. The real implementation is done based on the simplified form. This simplification saves memory and improves computation speed.

We use the following properties of the data mining solver for tuning:

(i) An action, genA(), and goal() are essentially related only to the limbo nodes, or elements of leaf(nodeStr).
(ii) There is no backtracking.
(iii) There is no repetition of state.

Because of property (i), we divide nodeStr into the parts of Unlimbo and Limbo. Unlimbo represents nonlimbo nodes and Limbo represents the limbo nodes in nodeStr,

i.e., nodeStr = (Unlimbo, Limbo), where an element of Unlimbo takes the form of (node number, branch name, node number) and an element of Limbo takes the form of (node number, branch name, set of attributes). Limbo is actually leaf(nodeStr) mentioned above.

Consequently, the set Limbo is a subset of $\wp(N \times V \times \wp(\mathrm{ALst}^+))$. Let a subset of $N \times \mathrm{ALst}^+ \times X$ be symbolically denoted by nodeData. Let the current state be y = (nodeStr, nodeData) where nodeStr = (Unlimbo, Limbo). Suppose n' = getNdId(nodeData), $(n, v, \mathrm{attr'})$ is applicable to (n, v, As), $(n, \mathrm{attr}, D) \in$ nodeData, and $D' = \{d | d \in D, d(\mathrm{attr}) = v\}$. Here D' is denoted by element$(D, \mathrm{attr}, v, *)$. Then, let $\delta : Y \times A \to Y$ be refined as

$$\delta((\mathrm{nodeStr}, \mathrm{nodeData}), (n, v, \mathrm{attr}')) = (\mathrm{nodeStr}', \mathrm{nodeData}'),$$

where nodeStr = (Unlimbo, Limbo) and nodeStr′ = (Unlimbo′, Limbo′) and the following hold:

a. If $\mathrm{attr}' \neq$ dummy

$$n' = \mathrm{getNdId}(\mathrm{nodeData}),$$

$$\mathrm{Unlimbo}' = \mathrm{Unlimbo} \cup \{(n, v, n')\},$$

$$\mathrm{Limbo}' = (\mathrm{Limbo} - \{(n, v, \mathrm{As})\}) \cup \mathrm{genLimboNd}(n', \mathrm{attr}', D', \mathrm{As}),$$

where $\mathrm{genLimboNd}(n', \mathrm{attr}', D', \mathrm{As}) = \{(n', v', \mathrm{As} - \{\mathrm{attr}'\}) | v' \in \mathrm{dom}(\mathrm{attr}')\}$,

$$\mathrm{nodeData}' = \mathrm{nodeData} \cup \{(n', \mathrm{attr}', D')\},$$

where $D' = \{d | d \in D, d(\mathrm{attr}) = v\}$ and $(n, \mathrm{attr}, D) \in$ nodeData.

b. If $\mathrm{attr}' =$ dummy,

$$n' = \mathrm{getNdId}(\mathrm{nodeData}),$$

$$\mathrm{Unlimbo}' = \mathrm{Unlimbo} \cup \{(n, v, n')\},$$

$$\mathrm{Limbo}' = \mathrm{Limbo} - \{(n, v, \mathrm{As})\},$$

$$\mathrm{nodeData}' = \mathrm{nodeData} \cup \{(n', \mathrm{elementN}(D, \mathrm{attr}, v, V_1), D')\},$$

where $(n, \mathrm{attr}, D) \in$ nodeData and $D' = \{d | d \in D, d(\mathrm{attr}) = v\}$. Note that elementN$(D, \mathrm{attr}, v, V_1)$ was defined in Section 12.2.4.

Let $\mathrm{genA}^{\ell} : (\mathrm{Unlimbo} \times \mathrm{Limbo}) \times \{\mathrm{nodeData}\} \to \wp(A)$ be given by

$$\mathrm{genA}^{\ell}((\mathrm{Unlimbo}, \mathrm{Limbo}), \mathrm{nodeData}) = \begin{cases} \varnothing & \text{if Limbo} = \varnothing, \\ \{((n, v), \mathrm{dummy})\} & \text{if } \inf(\mathrm{elementN}(D, \mathrm{attr}, v, V1) = 0 \text{ or } \mathrm{As} = \varnothing, \\ \{(n, v)\} \times \mathrm{As} & \text{otherwise.} \end{cases}$$

Let $\mathrm{goal}^{\ell} : (\mathrm{Unlimbo} \times \mathrm{Limbo}) \times \{\mathrm{nodeData}\} \to \mathrm{Re}$ be given by

$$\mathrm{goal}^{\ell}((\mathrm{Unlimbo} \times \mathrm{Limbo}), \mathrm{nodeData}) = \Sigma\{\mathrm{goal0}(n_i, v_i, \ell_i) | (n_i, v_i, \ell_i) \in \mathrm{Limbo}\}.$$

Let

$$y_0 = ((\{[]\}, \{(-1, [], \text{ALst}^+)\}), \{(-1, [], D_0)\}).$$

Let

$$\begin{aligned} Y_f^r &= ((N \times V \times N) \times \{\emptyset\}) \times (N \times V^+ \times X) \\ &= \text{target states in (Unlimbo} \times \text{Limbo)} \times \{\text{nodeData}\}. \end{aligned}$$

Then, we have a reduced user model

$$\textbf{reduced user model} = \langle A, Y, \delta, \text{goal}^\ell, \text{genA}^\ell, y_0, Y_f^r \rangle.$$

For property (ii), it should be noted that backtracking in the goal-seeker can occur in three ways. When the constraint is active, we may have a situation in which any available action cannot yield a legitimate state. Fortunately, the data mining solver does not have an active constraint. The second is when an action yields a repeated state. Apart from the backtracking operation, the goal-seeker is not permitted to return to a past state. Theoretically, the goal-seeker does not return to a past state in the data mining solver because Unlimbo of δ is a monotonically increasing set as shown above, i.e., Unlimbo increases to Unlimbo $\cup$ $\{(n, v, \text{attr}')\}$ if an action is applied.

The third is that genA() yields an empty set. In the data mining case, that situation is a final state. (Refer to the definition of st().) Consequently, a backtracking situation does not occur for the data mining solver. If there is no backtracking, the stack of the stdPDsolver can be simplified.

Appendix 12.1 User Model for Data Mining Problem

```
/*datamining.set*/
/*state=tree*/
/*.nodeStr(nd,v,nd')*/
/*.nodeStr(nd,v,[])*/
/*limbo node=[[nd,v,As],...,[nd',v',As']]*/
/*nodeData(nd,attrN,data)*/
/*.nodeI(nd,v,inf)*/
/*action=[Nd,V,Attr]*/

func([genLimboNd,inf,dom,infNd,getnodeData]);
    /*As=empty;entropy=0;terminal node*/
    getnodeData(Nd)=Y <->
     L:=listPred(.nodeData),
     member([Nd,Y],L);

delta([Leaf,[NdIdMax,[Attr1,Data1]]],[[Nd,V],Attr2])
      =[Leaf2,[Nd2,[Attr3,D2]]] <->
              (Attr2= .dummy) ->
                      (
                      [[Nd,V,As]|Leaf2]:=Leaf,
                      Nd2:=NdIdMax+1,
              //expand tree
              (.nodeStr(Nd,V,[Nd2])) ->
```

```
                                    (
                                    A:=1
                                    )
                             .otherwise
                                    (
              assign("no cut",[.nodeStr,Nd,V],[Nd2])
                                    ),
                         [Attr,Data]:=getnodeData(Nd),
ALst:=aLst.g,
I:=invproject(ALst,Attr),
              V1:=v1.g,
     N:=procC("getelementN",[Data,I,V,V1]),
([[],N]=getnodeData(Nd2)) ->
                         (
                         AA:=1
                         )
               .otherwise
                         (
               assign("no cut",[.nodeData,Nd2],[[],N])
                                   ),
                        Attr3:=[],
                        D2:=N
                        )
          .otherwise
                   (
                   Attr3:=Attr2,
                   ALst:=aLst.g,
                   I2:=invproject(ALst,Attr2),
                   //get node Id
                   Nd2:=NdIdMax+1,
                   //*update .nodeData

                  [Attr,D]:=getnodeData(Nd),
            I:=invproject(ALst,Attr),
            D2:=procC("getelement",[D,I,V,"*"]),
      [Attr2,D2]=getnodeData(Nd2)) ->
                           (
                           A:=1
                           )
                .otherwise
                           (
                assign("no cut",[.nodeData,Nd2],[Attr2,D2])
                                 ),
            //expand tree
         (.nodeStr(Nd,V,[Nd2])) ->
                                   (
                                   A2:=1
                                   )
                            .otherwise
                                   (
        assign("no cut",[.nodeStr,Nd,V],[Nd2])
                                   ),
                              //update Leaf
         LLeaf:=minus(Leaf,[[Nd,V,As]]),
         Limbo:=genLimboNd(As,[Nd2,[Attr2,D2]]),
         Leaf2:=append(Limbo,LLeaf)
                  );
```

```
genA([X,[NdIdMax,[Attr1,Data1]]])=As <->
        (X=[]) ->
                (
                As:=[]
                )
        .otherwise
                (
                [[Nd,V,Attrs]|Ls]:=X,
    ALst:=aLst.g,
    V1:=v1.g,
         [Attr,Data]:=getnodeData(Nd),
    I:=invproject(ALst,Attr),
                 V1:=v1.g,
         N:=procC("getelementN",[Data,I,V,V1]),
    Inf:=inf(N),
    (Inf=0) ->
                            (
         As:=[[[Nd,V],.dummy]]
                            )
                  .otherwise
                            (
                            (Attrs=[]) ->
                                       (
         As:=[[[Nd,V],.dummy]]
                                       )
                          .otherwise
                                      (
                                      As:=
     product([[Nd,V]],Attrs)
                                      )
                            )
                   );
/*generate limbo node*/
genLimboNd(As,[Nd2,[Attr2,D2]])=Limbo <->
          Attr2V:=dom(Attr2),
          As2:=minus(As,[Attr2]),
          Limbo:=inverse(defList(plimbo(y,x,[Nd2,As2]),[x,Attr2V]));
plimbo(y,V2,[Nd2,As2])<->
         y:=[Nd2,V2,As2];

goal([Leaf,[NdIdMax,[Attr1,Data1]]])=Re <->
         (Leaf=[]) ->
                  (
                  Re:=0
                  )
          .otherwise
                 (
                 V1:=v1.g,
                 D0Size:=d0Size.g,

        infs:=defList(pgoal(y,x,[V1,D0Size]),[x,Leaf]),
        Re:=sum(infs)
        );

pgoal(y,[Nd,V,Attrs],[V1,D0Size]) <->
            L:=listPred(.nodeI),
            (member([Nd,V,Inf],L)) ->
```

```
                        (
                        A:=1
                        )
                  .otherwise
                        (
        [Attr,Data]:=getnodeData(Nd),
                      ALst:=aLst.g,
                      I:=invproject(ALst,Attr),

        N:=procC("getelementN",[Data,I,V,V1]),
                                   Inf0:=inf(N),

       Inf:=Inf0*sum(N)/D0Size,

       assert(.nodeI(Nd,V,Inf) )
                                    ),
                        y:=Inf;

   /* stopping condition */

   st(C)<->
          finalstate(C);

   /*final state*/

   finalstate([Leaf,[Nd,[Attr,Data]]]) <->
          Leaf=[];

   infNd(Data,I)=InfI <->
           LenD:=cardinality(Data),
           VLst:=vLst.g,
           project(VLst,I,VI),
           project(VLst,1,V1),
           LenR:=cardinality(VI),
           Js:=procC("createindex",[1,LenR]),
           InfI:=sum(defList(pInfI(y,x,[Data,I,LenD,VI,V1]),[x,Js]));
   pInfI(y,J,[Data,I,LenD,VI,V1]) <->
                 VIJ:=project(VI,J),

           NIJ:=procC("getelementN",[Data,I,VIJ,V1]),

                 Inf:=inf(NIJ),
                 y:=Inf*sum(NIJ)/LenD;

   .mytree(-1);

   dom(AttrN)=AttrV <->
           ALst:=aLst.g,
           I:=invproject(ALst,AttrN),
           VLst:=vLst.g,
           AttrV:=project(VLst,I);

preprocess   () <->
   //data
           D0:=[
           [.soft,.young,.myope,.no,.normal],
           [.none,.young,.myope,.no,.reduced],
```

```
          [.none,.young,.myope,.yes,.reduced],
          [.hard,.young,.myope,.yes,.normal],
          [.none,.young,.hypermetrope,.no,.reduced],
          [.soft,.young,.hypermetrope,.no,.normal],
          [.none,.young,.hypermetrope,.yes,.reduced],
          [.hard,.young,.hypermetrope,.yes,.normal],
          [.none,.pre_presbyopic,.myope,.no,.reduced],
          [.soft,.pre_presbyopic,.myope,.no,.normal],
          [.none,.pre_presbyopic,.myope,.yes,.reduced],
          [.hard,.pre_presbyopic,.myope,.yes,.normal],
          [.none,.pre_presbyopic,.hypermetrope,.no,.reduced],
          [.soft,.pre_presbyopic,.hypermetrope,.no,.normal],
          [.none,.pre_presbyopic,.hypermetrope,.yes,.reduced],
          [.none,.pre_presbyopic,.hypermetrope,.yes,.normal],
          [.none,.presbyopic,.myope,.no,.reduced],
          [.none,.presbyopic,.myope,.no,.normal],
          [.none,.presbyopic,.myope,.yes,.reduced],
          [.hard,.presbyopic,.myope,.yes,.normal],
          [.none,.presbyopic,.hypermetrope,.no,.reduced],
          [.soft,.presbyopic,.hypermetrope,.no,.normal],
          [.none,.presbyopic,.hypermetrope,.yes,.reduced],
          [.none,.presbyopic,.hypermetrope,.yes,.normal]],
          aLst.g:=[.lenses,.age,.prescription,.astigmatism,.tear],
          D0Size:=cardinality(D0),
          d0Size.g:=D0Size,
          retract([.nodeData,.nodeI,.nodeStr]),
          assign(.nodeI,[]),
          retract([.gamma]),
          ALst:=aLst.g,
          VLst:=procC("getcategory",[D0,ALst]),
          vLst.g:=VLst,
          project(VLst,1,V1),
          v1.g:=V1,
//initialize
          Nd0:=0,
          assign("no cut",[.nodeData,Nd0],[[],D0]),
          getInitialNode(Attr0,I0),
          assign("no cut",[.nodeData,Nd0],[Attr0,D0]);

initialstate()=C0 <->
        Nd0:=0,
        [Attr0,D0]:=getnodeData(Nd0),
        ALst:=aLst.g,
        ALstp:=project(ALst,-1),
        Leaf0:=genLimboNd(ALstp,[Nd0,[Attr0,D0]]),
        C0:=[Leaf0,[Nd0,[Attr0,D0]]];

getInitialNode(Attr0,I0) <->
          [[],Data]:=getnodeData(0),
          ALst:=aLst.g,
          L0:=cardinality(ALst),
          myregInf.g:=[-1,1110000],
          L:=procC("createindex",[2,L0-1]),
          Ys:=defSet(pinitialNode(Y,I,[Data]),[I,L]),
          [I0,V]:=myregInf.g,
          retract([.myregInf,.regI]),
          Attr0:=project(ALst,I0);
```

```
pinitialNode(Y,I,[Data]) <->
                    InfI:=infNd(Data,I),
                    [II,InfII]:=myregInf.g,
                    (InfII>InfI) ->
                              (
                             myregInf.g:=[I,InfI]
                              );

consNdNL(NdNL) <->
          consNdNL0(),
          NdNL:=regN.g,
          retract([.regN]);
consNdNL0() <->
          L:=listPred(.nodeData),
          Ys:=defSet(pconsNdNL(Y,X,[]),[X,L]),
          regN.g:=inverse(Ys);
pconsNdNL(Y,[Nd,[Attr,D]],[]) <->
          (Attr=[]) ->
                   (
                   Y:=[Nd,D]
                   )

         .otherwise
                  (
                  Y:=[Nd,Attr]
                  );

cconsNode(M2) <->
          cconsNode0(),
          M2:=regM.g,
          retract([.regM,.nodeStr]);
cconsNode0() <->
          L:=listPred(.nodeStr),
          Ys:=defSet(pconsNode(Y,X,[]),[X,L]),
          regM.g:=Ys;
pconsNode(Y,[Nd,V,[Nd2]],[]) <->
          (etype(Nd2,"list")) ->
                   (
                   A:=1
                   )
         .otherwise
                  (
                  Y:=[Nd,V,Nd2]
                  );

inf(D)=InfX <->
          procC("entropy",[D],[InfX]);

dispdtree(D,NdNL) <->
          dummy:=defList(ptree(dum,Dx,[]),[Dx,D]),
          .mytree(Wp0),
          (Wp0<> -1 ) ->
                    (
                    Wp:=Wp0,
                    X1:=0,
                    Y1:=0
                    )
```

```
          .otherwise
                   (
                    makewindowSS(Wp,"mytree",X1,Y1,W,H),
                    assign(.mytree,Wp)
                    ),
          project(NdNL,1,[Nd,AN]),
          drawchild(["",Nd],Wp,X1,Y1,W),
          dispNdN(Wp,W+1,NdNL);
ptree(dum,[N,A,CN],[]) <->
                    (.nodeStr(N,L)) ->
                              (
                              append([[A,CN]],L,L2),
                              assign([.nodeStr,N],L2)
                              )
                   .otherwise
                              (
                              assign([.nodeStr,N],[[A,CN]])
                              ),
               dum:="OK";

dispNdN(Wp,W,NdNL) <->
          Row:=5,
          L:=cardinality(NdNL),
          procC("createindex",[1,L],[Is]),
          dummy:=defList(pNdN(dum,I,[Wp,W,NdNL,Row]),[I,Is]),
          V1:=v1.g,
          retract([.regI]),
          #(Wp,0,W+Row):="V1=",
          #(Wp,1,W+Row):=V1;
pNdN(dum,I,[Wp,W,NdNL,Row]) <->
                   project(NdNL,I,N),
                   J:=(I-1)%Row,
                   K:=floor((I-1)/Row),
                   #(Wp,K*2,W+J):=N;

drawchild([Attr,N],Wp,X,Y,W) <->
          .nodeStr(N,[C|Cs]),!,
          #(Wp,X,Y):=[Attr,N],
          retract(.nodeStr(N,[C|Cs])),
          #(Wp,X,Y+1):= .t_bar,
          drawchild(C,Wp,X+1,Y+1,W1),
         (W1>1) ->
                  (
                   procC("createindex",[Y+2,W1-1],[Is]),

                   dummy:=defSet(pchild(dum,I0,[Wp,X,Y]),[I0,Is])
                  ),
         drawbrother(Cs,Wp,X+1,Y+W1+1,W2),
                  W:=W1+W2+1;
pchild(dum,I0,[Wp,X,Y]) <->
                         #(Wp,X,I0):=.i_bar,
                         dum:="OK";

drawchild([Attr,N],Wp,X,Y,1) <->
         #(Wp,X,Y):=[Attr,N];
drawbrother([],Wp,X,Y,0) <-> !;
drawbrother([B|Bs],Wp,X,Y,W) <->
```

```
        #(Wp,X-1,Y):=.t_bar,
        drawchild(B,Wp,X,Y,W1),
        (Bs<>[] ) ->
                (
                procC("createindex",[Y+1,W1],[IIs]),
                Dummy:=defList(pbrother(dum,I,[Wp,X]),[I,IIs])
                ),
        drawbrother(Bs,Wp,X,Y+W1,W2),
        W:=W1+W2;
pbrother(dum,I,[Wp,X]) <->
                        #(Wp,X-1,I):=.i_bar,
                        dum:="OK";

postprocess() <->
        cconsNode(M2),
        consNdNL(NdN2),
        dispdtree(M2,NdN2);
```

The predicate preprocess() generates the basic sets ALst and VLst. Although the function goal() seems involved, in fact it is a straightforward implementation of the goal of Section 12.2.4. The predicate postprocess transforms an output of the extSLV into the data structure of Appendix 3.1 and displays it in the tree form of extProlog.

References

Witten, I. H., and Frank, E. (2000) *Data Mining*, Morgan Kaufmann Publishers.

13

Task Skeleton Model: Intelligent Data Mining System*

Because the problem classification of Chapter 4 is a general one, any problem can be covered by it. Furthermore, because the methodology of Chapters 4 and 5 is also general, in principle every problem can be attacked by it. It is obvious, however, that there are problems that cannot be solved by it in a practical sense. Extension of the basic extSLV of Fig. 4.5 is necessary to address more difficult problems. In the model theory approach there are two ways for extension: introduction of a hierarchical multilevel system [Takahara and Liu, 2005] and extension to a task skeleton model. This chapter introduces the task skeleton model. An intelligent data mining system is developed as a demonstration of the model.

13.1 General Concept of Problem-Solving System

Let us start from investigation of a general concept of problem-solving including uncertainty and the role of an end user (EU). The principal idea of this book is to view problem-solving as a cooperative task of a computer system and an EU. The user and the computer are in a "reflexive" (continuous feedback) relationship. In the course of solving the problem over time, the EU and the computer "walk hand in hand," i.e., neither one can proceed without affecting and simultaneously being affected by the other. In such a "symbiotic" relationship, the user has a goal-seeking (or adaptive) function, while the computer functions as a "process," i.e., performs algorithmic and numerical functions.

To represent the user–computer symbiosis in a problem-solving system, the concept of a generic goal-seeking problem (GSP) of the GST can be used. The context for a GSP is provided by sets of the relationship identities on which a GSP is defined [Mesarovic and Takahara, 1989]. Namely,

M : set of alternative decisions,
U : set of uncertainties which affect the outcome of a decision,
Y : set of outcomes,
V : assessment or evaluation set.

This well-known concept is viewed here in the most general of terms. Given a context, a GSP is defined by three relationships:

$P : M \times U \to Y$	image mapping,
$G : M \times Y \times U \to V$	assessment mapping,
$\text{Const} : U \to \wp(M)$	constraint relationship.

Here P is the model yielding the feasible outcome of the decisions, G is a yardstick used to compare alternative decisions, and Const is the constraint on the domain of actions depending on the anticipated uncertainties. Finally, V has a certain ordering relation for the comparative assessment of decisions. Let the relation be denoted by $\leq$.

Definition 13.1. Goal-Seeking Problem
A goal-seeking problem (GSP) is formally defined by the following structure:

$$\text{GSP} = \langle M, U, Y, V, P, G, \text{Const} \rangle.$$

A key role in characterization of a GSP is played by the uncertainty set, U. Two cases have to be considered: U is a singleton $U = \{u_0\}$ and U has more than one element. In the first case, the function of a GSP is specified by the following statement:

Given $u_0 \in U$ (or $U = \{u_0\}$), find $m^* \in M \cap C(u_0)$ such that the following holds:

$$\begin{aligned}(\forall m \in M \cap C(u_0))(G(m, P(m, u_0), u_0) \leq G(m^*, P(m^*, u_0), u_0)\\ \to G(m, P(m, u_0), u_0)\\ = G(m^*, P(m^*, u_0), u_0)).\end{aligned}$$

This decision problem is well defined as an optimization type as in the previous chapters.

In the second case, when U is not a singleton there is more than one feasible outcome for a given selected decision. If U is not a singleton, which is almost universally the case in practice, there is no way optimization can be used to identify the desired decision. If a decision $m \in M$ is best (optimal) for one outcome of uncertainty $u \in U$, it may be far from that for another $u' \in U$, except in very special, restricted circumstances. This is why the decision problem is ill defined.

In order to provide a basis for a rational procedure to the second case, additional specifications are needed that will transform the problem from ill defined to well defined. One additional specification that has to be made in order to proceed in a systematic way with the process of decision making is referred to as the selection of a decision principle [Mesarovic and Takahara, 1989].

The selection of a decision principle can proceed in two directions:

(i) Leading to a specific $u_0 \in U$ derived from additional (subjective, or risk-aversion-based) specifications made by the decision maker.
(ii) Leading to a subset $U' \subset U$ that contains a set of feasible uncertainties, either one of which can come to pass. This case falls into the category of decision making under true uncertainty.

Examples of the first approach are plentiful: given $U' \subset U$ as the expected set of uncertainties, u_0 can be the most probable element, and it can be derived from U' by a statistical measure based on objective or subjective probabilities, or it can be derived by focusing on special conditions, e.g., worst-case view by taking min (or max) operation on U' to identify u_0.

Examples for the second approach are less common. An example is the satisfaction approach, which is based on the decision maker's selection of a tolerance function on U':

$$T : U' \rightarrow V.$$

Given T and the decision problem, the function of a GSP is then defined by the following statement:

find $m^* \in M$ such that

$$G(m^*, P(m^*, u), u) \leq T(u) \text{ and } m^* \in C(u)$$

for all $u \in U'$ [Mesarivic and Takahara, 1989].

The solution is not unique, and any element of the set $M' \subset M$ that satisfies the above condition is acceptable.

Another aspect of the uncertainty set, which is of special importance for the design of a problem-solving system, refers to the function of uncertainty. A GSP is a target of the problem-solving layer of Fig. 1.2. Because the solving activity of the layer depends on the GSP, the uncertainty can be a parameter that controls the solving layer. The value of the parameter is decided by the higher layers of Fig. 1.2. This chapter treats the uncertainty in this context.

Two categories of uncertainties can be recognized from the control viewpoint: structural U_s and parametric U_p, or $U = U_s \times U_P$. The structural uncertainty U_s refers to alternative categories (or types) of relationships, while the parametric U_p refers to behaviors of the relationships. The former changes the model used in decision making, e.g., from linear to nonlinear, from numeric to symbolic, or changes other relationships in a GSP. The parametric uncertainty refers to the selection of a specific model from a given category by selecting the corresponding parameters.

In general, multilevel, hierarchical concepts of a complex systems theory are used to provide a management model to deal with the two categories of uncertainties [Mesarovic and Takahara, 1989; Takahara and Mesarovic, 2003]. Of particular relevance is the so-called multilayer task structure, which is shown in Fig. 13.1, which corresponds to the upper echelon of Fig. 1.2.

The user layer addresses the structural uncertainty U_s. In the next layer, the adaptive layer, the parametric uncertainty U_p is addressed under the assumption that U_s is dealt with by the upper layer. The task is ultimately to identify $U' \subset U(= U_s \times U_P)$ or a single element $u \in U$ to be used in the decision selection process of the lowest layer. Note that the specification of U_s is a part of the control from the user layer to the adaptation layer. It is also true that specification of U_p is a part of the control from the adaptation layer to the lowest layer.

On the lowest layer, the problem-solving layer, a set of decisions M^* is selected using the relationships selected on the upper layers. This is the activity that has been discussed in the previous chapters.

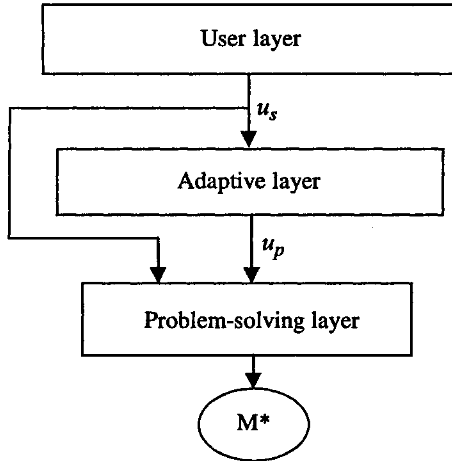


Fig. 13.1. Hierarchical goal-seeking system.

13.2 Task Skeleton Model

The previous chapters have assumed that the problem-solving layer is composed of two components, the problem specification environment (PSE) and the extended solver (extSLV). Usually, these components are not simple ones but are composites of subsystems. These subsystems are identified by functions that are called as tasks.

Figure 13.2 shows a composite model called a task skeleton model [Takahara, Chen, and Shiba, 2003].

In the model, extSLV is further decomposed into two components, a problem formulator (PRF) and a solver (SLV), and a PSE is identified as the output of a component called a data model generator (DMG). It consists of a PSE constructor and a data-transformation component connected to a database.

Although an SLV is still a major part in solving a problem in the task skeleton model, its sole activity cannot determine a solution. A solution can be found as a result of cooperation of the problem-solving layer, the adaptive layer (system), and the user layer.

13.2.1 Problem-Solving Layer

Let a GSP be

$$\mathrm{GSP} = \langle M, U, Y, P, G, \mathrm{Cont} \rangle$$

as specified in Definition 13.1, where $V = \mathrm{Re}$ is assumed.

The function of the problem-solving layer is to find a solution (decision) under the condition that a predicted uncertainty element is given, i.e., $u^* \in U$, which is selected by the higher control layers. We call the decision problem of the problem solver with u^* a particularized decision problem $\mathrm{PP}(u^*)$. That is,

$$\mathrm{PP}(u^*) = \langle M, \{u^*\}, g, \mathrm{Const} \rangle,$$

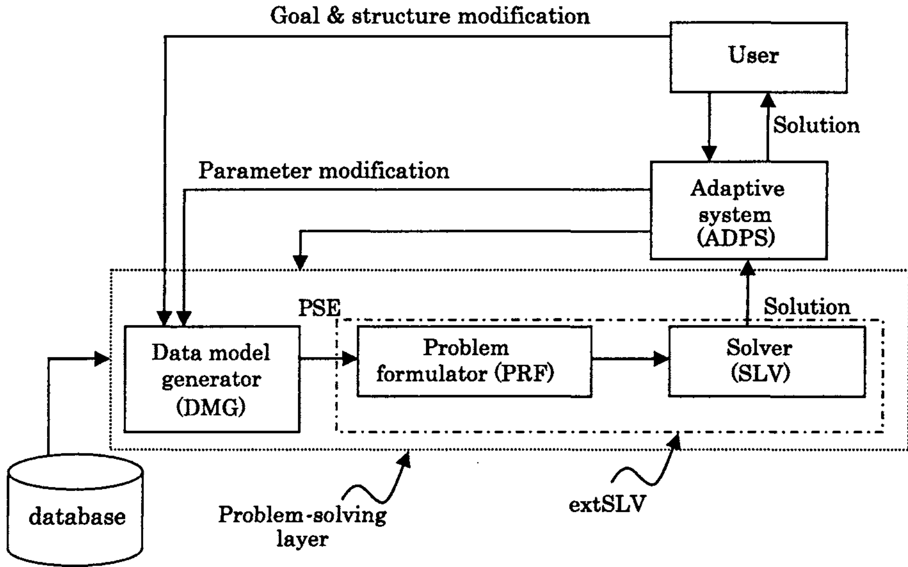


Fig. 13.2. Task skeleton model.

where g is the composition of P and G, i.e.,

$$g : M \times \{u^*\} \to \text{Re}$$

such that

$$g(m, u^*) = G(m, u^*, P(m, u^*)).$$

Then, the task of the problem-solving layer is given by

$$\text{given } u^* \in U, \text{ find } m^* \in C(u^*) \text{ that is optimal for } \text{PP}(u^*).$$

The problem-solving layer specification has three stages: data model generator (DMG) specification, problem formulator (PRF) specification, and SLV specification.

The first-stage DMG specification transforms external data into a PSE. (See the data transformation system and the PSE constructor of Chapter 15.) Because a function of set theory is a type of a relation and a constant is also a type of a function, a structure that specifies a PSE in Section 5.1.2 can be represented by a family of relations. In general, then, a PSE is given in the following way:

$$\text{DM}(u) = \{R_i(u) | i \in I\},$$

where $u \in U$, I is an index set, and $R_i(u)$ is assumed to be given by a relation, table (matrix), or a set of vectors over values of attributes, i.e.,

$$R_i(u) \subset V_{i1} \times \cdots \times V_{iJ},$$

$V_{i1}, \ldots, V_{iJ}$ are sets of values of respective attributes.

It should be noted that because a DM(u) depicts a native figure of the problem or because it is a parameterized representation of the problem structure, it provides good means for the adaptive layer and the user to modify the problem structure. Of course, $u \in U$ is the control variable for that purpose.

The second stage of the problem-solving layer specification, PRF specification, prepares a problem representation PP(u) suitable for the SLV. Then, the PRF is specified as a mapping:

$$\text{PRF} : \text{DM}(u) \mapsto \text{PP}(u),$$

i.e., for each $u \in U$ the problem formulator specifies a problem structure PP(u) for the SLV activity.

Finally, an SLV can be represented as a mapping

$$\text{SLV} : \{\text{PP}(u)\} \rightarrow Y$$

such that if $\text{PRF}(\text{DM}(u^*)) = \text{PP}(u^*)$, $\text{SLV}(\text{PP}(u^*)) = y^*$ is an optimum solution of g, i.e., m^* is an optimum solution of $g(m, u^*)$ and $y^* = P(m^*, u^*)$.

13.2.2 Adaptive Layer

A distinguishing feature of the problem-solving layer is that its task is actual problem-solving using an algorithm. It represents the primary activity of problem-solving. This is in contrast to the task on the adaptive layer and the user layer that control parameters of the given GSP, or, precisely speaking, parameters of the PSE for the problem-solving layer to produce a better solution.

In Section 13.1, the uncertainty set U was decomposed into two sets, $U = U_p \times U_s$, where U_p is a set of operational parameters that is the target of the adaptive layer. The adaptive system of that layer, ADPS, consists of two components: a heuristic rule for adaptation, ADR: $M \times U_p \times U_s \rightarrow U_p$, and a solution evaluation rule ADC $\subset M \times U_p \times U_s$. The adaptive system is then given by

$$\text{ADPS} = \langle M, U, \text{ADR}, \text{ADC} \rangle.$$

ADR is a modification rule of U_p to improve a solution m^* that is given by the problem-solving layer. Suppose u_p is the current selected value of U_p. Let m^* be an optimal solution generated by the problem-solving layer for (u_p, u_s^*), where $u_s^* \in U_s$ is selected by the user layer. Then, if $(m^*, u_p, u_s^*) \notin$ ADC, m^* is considered unsatisfactory by ADPS, which then generates $u'_p = \text{ADR}(m^*, u_p, u_s^*)$ and requests the problem-solving layer to regenerate a new solution for (u'_p, u_s^*). If the new solution is still unsatisfactory, the above cycle is repeated.

ADC is a metaconstraint that cannot be included in Const of the problem-solving layer. There are three typical metaconstraints:

1. A GSP assigned to the SLV must be feasible.
2. The SLV must produce a solution in the allotted decision time frame.
3. There are "soft" constraints that should normally be met but can be ignored in some situations.

Metaconstraint 1 requires that the SLV can generate a solution. Usually Const of the GSP is relaxed to make the GSP feasible. Metaconstraint 2 requires that the solving activity be suspended by the adaptive layer if the period of activity exceeds the time allowed. Metaconstraint 3 indicates an ethical requirement as a typical soft constraint, while an economical one is considered to be a hard constraint.

13.2.3 User Layer

A solution is forwarded to the user from the adaptive layer if it satisfies ADC. The user is supposed to evaluate whether the solution is satisfactory. There is an important special case. Let

$$T : U_s \to \mathrm{Re}$$

be a tolerance function. Suppose the EU can select a finite set $U_s' = \{u_{s1}, \ldots, u_{sp}\} \subset U_s$ as a feasible scenario. In this case m_i is called satisfactory with respect to U_s' if it satisfies the relation

$$(\forall u_s \in U_s')(g(m_i, u_s) \leq T(u_s)).$$

Because U_s' is finite, this condition can be represented as a predicate

$$(g(m_i, u_{s1}) \leq T(u_{s1}))\& \cdots \&(g(m_i, u_{sp}) \leq T(u_{sp})).$$

Then, the satisfaction condition can be embedded into Const or into ADC as an additional condition. In this case, m^*, which is an optimal solution generated by ADPS and extSLV, is automatically a satisfactory solution.

In practice the ultimate task of the user is to select an appropriate scenario, or a subset of U_s.

13.3 Intelligent Data Mining System

13.3.1 Framework for Intelligent Data Mining System

Chapter 12 discussed data mining as an example of the model theory approach. It is a topic of interest in MIS because of its importance, and many software packages are now commercialized. However, at least one deficiency can be identified in current data mining activities. It is said that data mining output is used for prediction or classification using only the class of attributes of a target data set [Weiss and Indurkhya, 1998]. The data mining activity is then not far from conventional statistical analysis. Data mining should be linked to a deep knowledge of management to yield a truly useful result. The linking mechanism is missing. This section will address this problem as problem-solving development.

Development of an intelligent data mining system (IDMS) follows the task skeleton model discussed in Section 13.2. The problem-solving layer performs regular data mining. As Fig. 13.2 shows, the layer consists of the three components: DMG, PRF, and SLV. The DMG is to yield data and parameters; the PRF is to preprocess them,

which includes cleansing, clustering, categorization, and/or transformation of data values to produce a structure model; and the SLV is to carry out data mining of the structure model prepared by PRF. In the IDMS, the mining operation is performed in three ways: reduction of the given data set, generation of a decision tree, and generation of a rule set. The problem-solving layer carries out these primary activities of a data mining system.

The main function of the IDMS is performed by the adaptive layer, which adjusts the solving-layer operation. The adaptive layer first gets a user's goal or a query that is supposed to represent his intention in data mining. It then loads extensive knowledge from a knowledge base (KB), which can address the query. Actually, it should be a mediator between the query of the user and the shallow knowledge produced by the data mining operation. Based on the query of the user and the extensive knowledge, it selects a proper subset of the attribute set using a standard statistical technique and the decision tree analysis of data mining. The rule-generation operation of data mining is then applied to the reduced data set, which is a restriction of the original data set with respect to the selected attributes. Parameters of the rule generation are tuned by analysis of generated rule sets so that hopefully a meaningful rule set can be generated. Finally, by combining the query, the extensive knowledge, and the produced rules, the IDMS generates an expert system and executes it to produce predicative statements about queries given by the user.

The user examines the output of the adaptive layer and changes some parameters of the DMG so as to get a satisfactory solution.

13.3.2 Problem-Solving Layer of IDMS

The problem-solving layer is an input–output system. The inputs are the data set for data mining and the parameters to control it.

The data shown in Fig. 13.3 are used as a typical example of a data set, called the "weather problem" [Witten and Frank, 1998].

In this data set the user is supposedly interested in the weather conditions when a game is played. The output attribute is "play," while the other attributes are the input attributes.

As mentioned above, the problem-solving layer is assumed to perform both the decision tree generation and the rule generation. Decision tree generation is used to check the relevance of input attributes of the data set to its output attribute. The generation produces a decision tree as shown in Fig. 13.4 [Witten and Frank, 1998]. Figure 13.4(a) shows the original display of the tree in extProlog, while Fig. 13.4(b) shows a redrawn tree in a more standard form. The decision tree shows that if the outlook is sunny and the humidity is high, the game will not be played. The confidence of the assertion is 100%.

Figure 13.4 also shows that the attribute of temperature is not included in the tree representation, which means that the adaptive layer can judge that the temperature is not relevant to the output attribute "play" and can reduce the attribute set by deleting temperature from the original attribute set for the rule-generation operation.

ID code	play	outlook	temperature	humidity	windy
D1	no	sunny	hot	high	false
D2	no	sunny	hot	high	true
D3	yes	overcast	hot	high	false
D4	yes	rainy	mild	high	false
D5	yes	rainy	cool	normal	false
D6	no	rainy	cool	normal	true
D7	yes	overcast	cool	normal	true
D8	no	sunny	mild	high	false
D9	yes	sunny	cool	normal	false
D10	yes	rainy	mild	norm	false
D11	yes	sunny	mild	normal	true
D12	yes	overcast	mild	high	true
D13	yes	overcast	hot	normal	false
D14	no	rainy	mild	high	true

Fig. 13.3. Weather problem: data set.

The rule-generation operation is applied to the reduced data set. Figure 13.5 shows an example of the output of the rule-generation operation where the restricted attribute set {play, outlook, humidity, windy} is used.

The rule set generation is controlled by the confidence number (ConL), the support number (SupL), and the attribute set (where irrelevant attributes are ignored). As shown below, the output of the rule set generation is produced as a Prolog rule set.

The operations of attribute selection, tree generation, and rule generation are controlled by commands of the adaptive layer to the problem-solving layer.

13.3.3 Adaptive Layer of IDMS

Figure 13.6 shows the strategy of the adaptive layer. The strategy consists of seven stages.

At the first stage, the adaptive layer obtains a query (dmEV) from the user. It is assumed to be given in predicate form.

Figure 13.7 shows a trivial example of dmEV. The user simply wants to know what kind of satisfaction X can be obtained for the current situation.

At the second stage of Fig. 13.6, extensive knowledge dmSTNDKN is obtained from the KB relevant to the user's query. It is also assumed to be represented in rule form. The knowledge dmSTNDKN must be able to provide a link between dmEV and the rules generated from the data set, that is, factual knowledge about the data

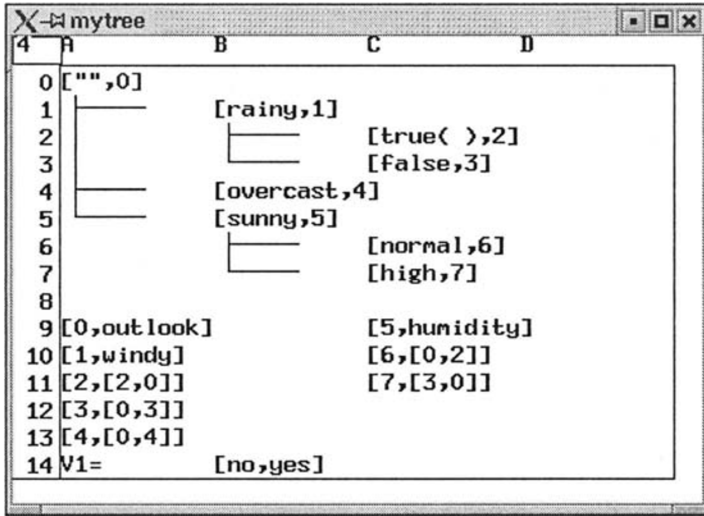


Fig. 13.4(a). Decision tree output of the problem-solving layer.

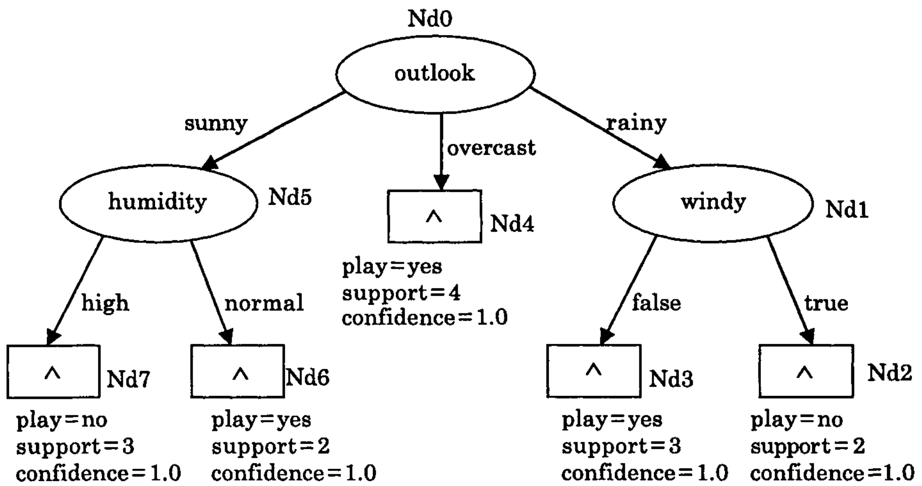


Fig. 13.4(b). Decision tree output of the problem-solving layer.

set. Predicates of dmSTNDKN hopefully connect the predicates used in the query and attributes of the data set. Figure 13.8 shows a trivial example of dmSTNDKN for Figs. 13.3 and 13.7. Determination of dmSTNDKN is the most difficult step of Fig. 13.6. It is done inside the adaptive layer.

At the third stage of Fig. 13.6, a small subset of the data set is extracted. Extraction of the subset can be performed using random sampling, stratified sampling, and equal-distance sampling. In the example of this chapter, the selection is done by simply sampling entries at random.

1. If outlook is overcast, then play is yes with confidence of 100% and support number 4.
2. If humidity is normal, then play is yes with confidence of 85.7% and support number 6.
3. If windy is false, then play is yes with confidence of 75% and support number 6.
4. If humidity is normal and windy is false, then play is yes with confidence of 100% and support number 4.

Fig. 13.5. Rule output of the problem-solving layer.

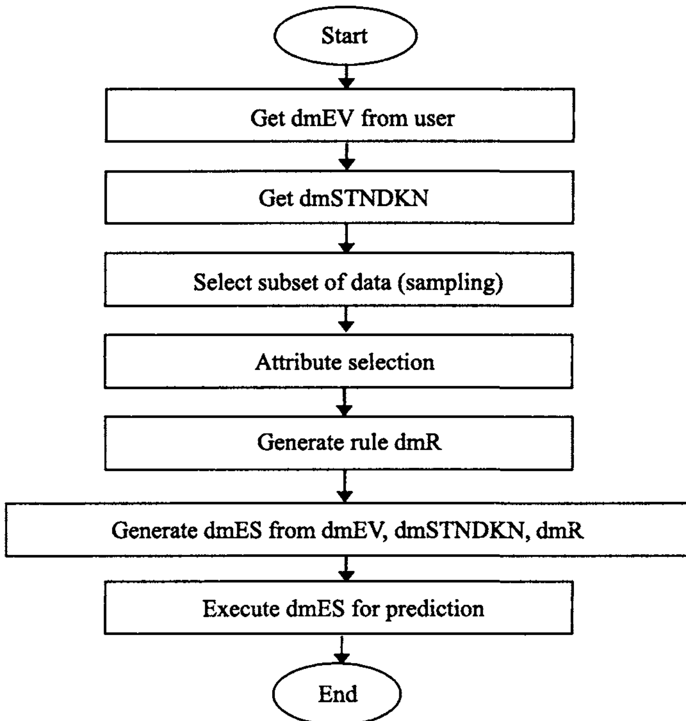


Fig. 13.6. Strategy of dmES.m.

satisfaction(X) ?

Fig. 13.7. Example of dmEV.

If outlook=overcast and play=yes, satisfaction=high.
If outlook=sunny and play=yes, satisfaction=medium.
If outlook=sunny and play=no, satisfaction=low.

Fig. 13.8. Example of dmSTNDKN.

At the fourth stage of Fig. 13.6, attribute selection is performed by finding a relevant attribute subset and removing unimportant or redundant attributes.

At the fifth stage, a reduced data set is determined by deleting the irrelevant attributes from the original data set. For example, the attribute temperature is ignored in the attribute set. A rule-generator algorithm is then applied to the reduced data set to generate a rule set dmR. In order to avoid overfitting of rules to the given data set, the current system uses a simple method to adjust SupL and ConL to cut off unimportant rules. This stage requires repeated operation of rule generation.

At the sixth stage, the system combines dmEV, dmSTNDKN, and dmR to yield an expert system dmES. This can be easily done because dmEV, dmSTNDKN, and dmR are represented in if-then rules and the IDMS is developed in extProlog.

At the final stage, dmES is executed to produce some answers (predictions) relevant to dmEV. Confidence factors of dmR specify the uncertainties of the predictions.

13.3.4 Implementation of IDMS

The model space of extProlog, which is shown in the right-lower window of Fig. 13.9, is used as the platform to implement the IDMS [Takahara et al. 1993]. Figure 13.9 shows the real structure of the system, which consists of four components, that is, two spreadsheets, dmInput.s and dmPara.s, and two models, dmSLV3.m and dmES3.m.

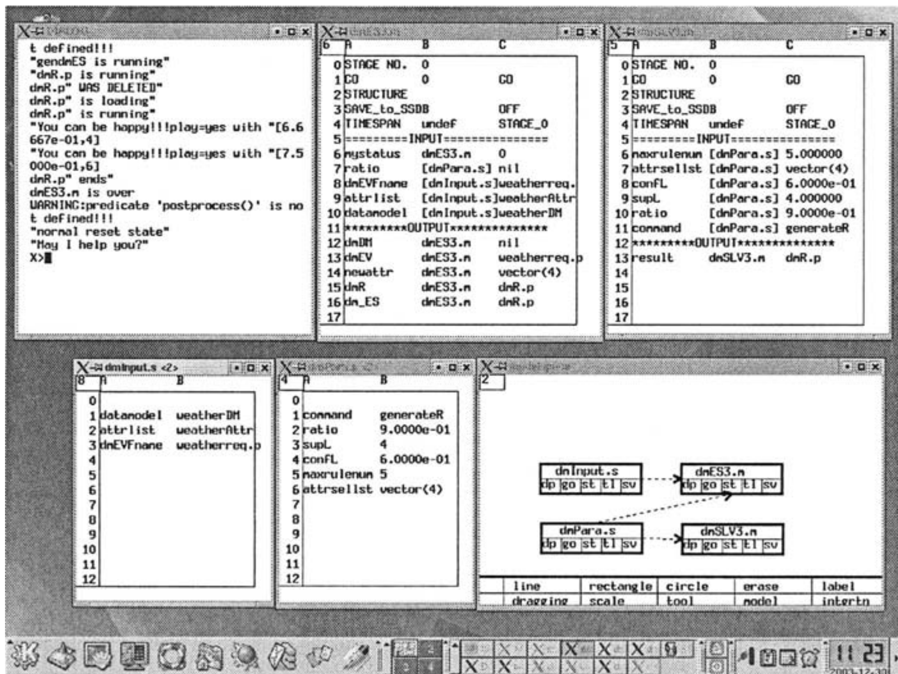


Fig. 13.9. Implementation of the system on the model space in extProlog.

The spreadsheets keep two data, a data set X_1 for data mining and parameters X_2 for control of the mining operation. Naturally, these parameters can be controlled by the user because they are data on the spreadsheets. Let X_1 = [datamodel, attrlist, dmEVFname], in which datamodel specifies the file name of the preprocessed data set, attrlist is the attribute list of the data set, and dmEVFname shows the name of the file that saves a query of the user. Let X_2 = [command, ratio, supL, confL, maxrulenum, attrsellst]. The first element, command, specifies an operation of the problem-solving layer; the second element, ratio, controls a reduction rate of the data set used for attribute selection; the third and fourth elements, supL and confL, specify the values of the supporting level and the confidence level, respectively; the fifth element, maxrulenum, indicates the maximum number of rules that are to be generated at the fifth stage; and the last element, attrsellst, shows the attribute set selected at the fourth stage.

The parameter set X_2 is adjusted to suppress overfitting, and X_1 and X_2 can be controlled by dmES3.m and the user. The dmSLV3.m is a model written in the model description language (MDL) of extProlog [Takahara et al. 1993], which corresponds to the problem-solving layer of Fig. 13.2 and performs the operational activities of Fig. 13.6.

The dmES3.m is the model for the adaptive layer of Fig. 13.2, which implements the strategy of Fig. 13.6. Both dmSLV3.m and dmES3.m are reported in [Takahara et al. 2002].

Because the data preprocessing is already completed in this example, Fig. 13.3 does not show the original data set but rather a preprocessed one. In the original data set, the temperature or humidity data might be numerical. In that case, categorization of values should be done by the PRF to produce a secondary data set.

After deleting the irrelevant attribute "temperature" using Fig. 13.4, rule generation is performed for a given pair (supL, confL). For instance, the case of supL = 2 and confL = 70% produces 14 rules. Because 14 rules are too many when compared with the given data set, the pair (supL, confL) of dmpara.s is modified to generate a more concise rule set. Finally, 4 rules are generated for supL = 4 and confL = 60%. Although the modification can be performed by dmES3.m, it is done manually in the current implementation. The final result is illustrated as Prolog rules in the "dmR.p" part of Fig. 13.10.

The rules of dmR.p are combined with dmEV and dmSTNDKN listed in Fig. 13.10 to produce an expert system, dmES, in extProlog. Figure 13.10 is the entire code of dmES, which is just a combination of dmR.p, dmEV.p, and dm STNDKN.p. It should be noted that dmES collects facts about attributes outside of the data set and predicts the status of satisfaction of the user based on dmR and the facts.

The total structure can be summarized as Fig. 13.11.

It should be noted that the expert system can dynamically change depending on data mining and KB.

Figure 13.12 illustrates a real communication between a user and the system. According to Fig. 13.10, dmES starts with execution of the user's satisfaction query, satisfaction(X). It depends on dmR(), and hence dmR() is next driven: dmR() drives the fact collection program. The fact collection program communicates with the user.

```
/*dmR.p*/
dmR(0,play(yes),1.0000e+00,4):-outlook(overcast);
dmR(1,play(yes),7.5000e-01,6):-windy(false);
dmR(2,play(yes),8.5714e-01,6):-humidity(normal);
dmR(3,play(yes),1.0000e+00,4):-humidity(normal),windy(false);

/*dmSTNDKN.p*/
satisfaction(high):-dmR(Id,play(yes),Conf,Sup),outlook(overcast),
     xwriteln(0,"You must be highly satisfied!!!play=yes with ",
     [Conf,Sup]),assign(resstatus,1);
satisfaction(medium):-dmR(Id,play(yes),Conf,Sup),outlook(sunny),
     xwriteln(0,"You can be happy!!!play=yes with ",[Conf,Sup]),
     assign(resstatus,1);
satisfaction(low):-dmR(Id,play(no),Conf,Sup),outlook(sunny),
     xwriteln(0,"You may be disappointed play=no with ",[Conf,Sup]),
     assign(resstatus,1);
satisfaction(X):-
     resstatus(0), xwriteln(0,"sorry!!!");

/*fact collection*/
ask(A,V):-
     known(yes,A,VV),!,
     V=VV;
ask(A,V):-
     known(X,A,V),!,fail;
ask(A,V):-
     xwriteln(0,"Attr:Value=",[A,V]),
     xwriteln(0,"?(yes/no)"), xread_parse(0,Y),
     assert(known(Y,A,V)), Y=yes;
known();
humidity(V):-ask(humidity,V);
windy(V):-ask(windy,V);
outlook(V):-ask(outlook,V);
temperature(V):-ask(temperature,V);

/*dmEV.p*/
?-assign(resstatus,0),satisfaction(X);
```

Fig. 13.10. Generated expert system: dmES.p.

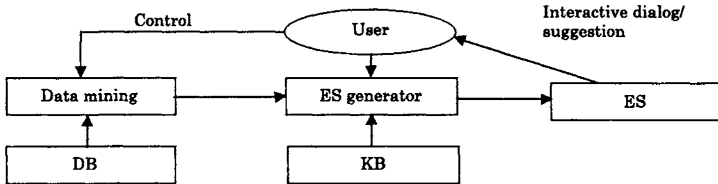


Fig. 13.11. Implemented structure of IDMS.

It first asks whether it is windy. If the user answers yes, it checks whether the outlook is overcast. If the user answers yes again, dmR() yields one predicative conclusion: "You must be highly satisfied!!!play = yes with [0.75, 6]." It states that the game will be played with probability 0.75. The conclusion is derived based on the rules mined from the data.

```
DIALOG
"dmR.p is running"
dmR.p" is loading"
dmR.p" is running"
"Attr:Value="[windy,false]
"?(yes/no)"
X>
Keyin please!!!
X>
Keyin please!!!
X>yes
"Attr:Value="[outlook,overcast]
"?(yes/no)"
X>
Keyin please!!!
X>yes
"You must be highly satisfied!!!play=yes with "[7.5000e
-01,6]
"Attr:Value="[humidity,normal]
"?(yes/no)"
X>
Keyin please!!!
X>yes
"You must be highly satisfied!!!play=yes with "[8.5714e
-01,6]
"You must be highly satisfied!!!play=yes with "[1.0000e
+00,4]
dmR.p" ends"
dmES3.m is over
WARNING:predicate 'postprocess()' is not defined!!!
"normal reset state"
"May I help you?"
X>
```

Fig. 13.12. Execution of the expert system.

References

Takahara, Y., Iijima, J. and Shiba, N. (1993) "A model management system and its implementation," *System Science* 19.

Weiss, S. M. and Indurkhya, N. (1998) *Predicative data mining: a practical guide*, Morgan Kaufmann.

Witten, I. H. and Frank, E. (1998) *Data mining*. Morgan Kaufmann.

Mesarovic, M. D. and Takahara, Y. (1989) "Abstract Systems Theory," *Lecture Notes in Control and Information Sciences*, Springer.

Takahara, Y., Chen, X. and Shiba, N. (2003) *Advances of DSS Design: General Systems Theory Approach*, Economic Science Press.

Takahara, Y. and Mesarovic, M. D. (2003) *Organization Structure: Cybernetic Systems Foundation*, Kluwer Academic Press.

Takahara, Y. and Liu, Y. (2002) *Intelligent Data Mining System*, internal report, Dept. of Management Information Systems, Chiba Institute of Technology, Chiba, Japan.

Takahara, Y. and Liu, Y. (2005) "Multi-level Systems Approach to Solver System Design," Proceedings of 18th International Conference on System Engineering, Las Vegas, Nevada.

Part V

Model Theory Approach to Transaction Processing Systems Development

14

Transaction Processing System on Browser-Based Standardized User Interface

This chapter presents the model theory approach to development of a transaction processing system (TPS), which is the main target of current systems engineering. With this approach, the system developer constructs the specification for the target system based on the relational structure of the TPS (presented in Section 1.2). The specification is described in computer-acceptable set theory (introduced in Chapter 2) and then compiled into an executable TPS in extProlog. The system is executed under the control of a standardized user interface (UI) designed in PHP. The UI has been developed on several levels of sophistication. For the sake of simplicity, this chapter discusses TPS development using the simplest UI.

This chapter reconfirms the following advantages of the model theory approach over the standard approaches:

- It provides a reliable system specification (a general assertion for formal approaches).
- It facilitates reliable implementation and rapid systems development by providing a user interface (for the transaction processing system) and a goal-seeker (for the problem-solving system) as black box components for MIS development coupled with automatic systems generation. Once a system specification has been given in computer-acceptable set theory, an executable transaction processing or problem-solving system can be produced. The executable system is expected to be a correct realization of the specification.
- It may reduce the cost of systems development by providing a means of accelerating development and by the system's being usable in the open-source software LAPP (Linux, Apache, postgreSQL, PHP).
- It can realize an intelligent MIS that accounts for both problem-solving and transaction processing functions on an integrated platform.
- It facilitates end-user maintenance by allowing system construction to be performed using elementary set theory rather than a computer language.

14.1 Model Theory Approach to Transaction Processing System: Canonical Structure

This chapter addresses application of the model theory approach to TPS development. Although the target of this chapter is a TPS, as Section 14.3 shows, a TPS in the model theory approach includes a problem-solving system as a subsystem. The preceding chapters dealt with the problem-solving system, and the definitions referred to in this chapter are drawn from that discussion.

Traditionally, TPSs have been developed without using formal methods. However, there is a school of thought that emphasizes the importance of a formal approach [Fitzgerald et al. 1998], claiming that the formal approach can address the difficulties associated with the traditional approach. In particular, a formal approach can assist in identifying deficiencies in a system specification in an early stage of development. Rigorous validation of a developed system is possible in a formal approach. Providing reliable documentation is also easier by a formal approach because formalized statements can be used as documents. If the formal specification can be automatically translated into an executable system, as proposed here, rapid systems development can be realized. Most importantly, however, a formal approach can establish systems engineering as a solid engineering discipline that is more than just an art.

The most notable example of a formal approach to TPS development is VDM-sl (Vienna development method specification language) [Fitzgerald et al. 1998]. Although VDM-sl relies on set theory and logic, its model description language should be considered as a computer language with strict type requirements rather than set theory. It is certainly better from a practical perspective for an ordinary system developer not to be required to learn another computer language in addition to set theory and logic. Furthermore, as the name indicates, VDM-sl is principally designed for the specification phase of information systems development: the advantages of formalism cannot be fully appreciated if it is relevant only to specification.

The model theory approach presented in this chapter differs from VDM-sl in several points. The fundamental difference is that the model theory approach addresses both SLV development and TPS development. It is not restricted to TPS development. The model theory approach is based on a general formalized structure of an information system. Set theory and logic are used for the model theory approach in the standard form. Moreover, the proposed approach includes the implementation phase for systems development as an integral part, and is supported by automatic system generation. On generation, a standardized UI is attached to the executable system, eliminating the need for the construction of a user interface.

Figure 14.1 shows a simple problem that can be treated by this approach. A detailed discussion of this problem is given in Section 14.6. The figure is part of the data flow diagram (DFD) for a workshop registration system [Koizumi et al. 2003]. A more meaningful example is given in Chapter 15.

A DFD is used as an extended block diagram in this approach. It should be noted that systems engineering usually starts from a block diagram representation of the target problem (see Fig. 4.1). The model theory approach follows that tradition.

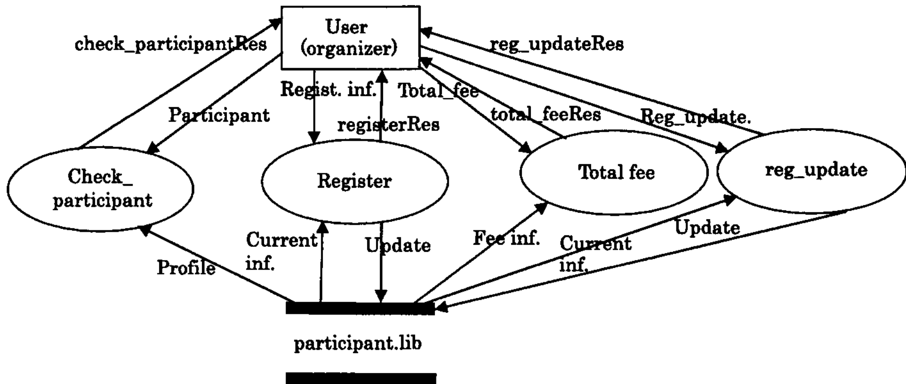


Fig. 14.1. DFD for a workshop registration system.

The DFD shows that when the user (organizer) issues the command "register" with supporting information, the process "register" handles the command using the current data in the file "participant.lib" and returns a response "regisrerRes" to the organizer and at the same time updates the file.

The development procedure of the model theory approach is applied to the DFD to derive a target system (see Section 14.6). Figure 14.2 illustrates a browser-based implementation of the system shown in Fig. 14.1.

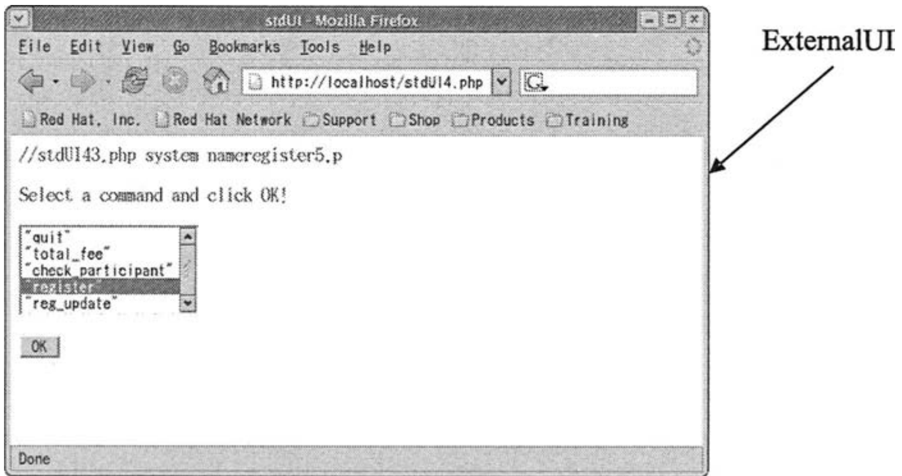


Fig. 14.2. Browser-based implementation.

It is not important to understand this implementation in detail at this point, but it should be understood that Fig. 14.2 can be abstracted into Fig. 14.3, an image of a browser-based MIS, which is the starting point of this chapter.

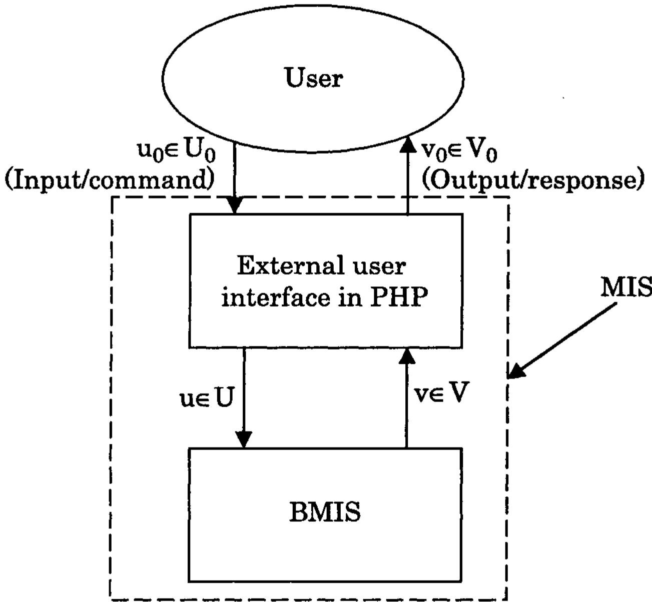


Fig. 14.3. Image of browser-based MIS.

In the abstracted model, an MIS is assumed to consist of two subsystems: a UI and a basic MIS (BMIS). The interface is called an external UI, which is illustrated as an overall structure in Fig. 14.2. Appendix 14.3 discusses the external UI. Since the external UI is standardized and implemented in PHP using a standard browser, and is provided as a black box in MIS development, the development theory for the MIS is concerned primarily with the BMIS. In fact, the theory has been developed on the assumption that the following three subsystems are given as black box components.

Assumption 14.1. The following three components are provided as black box components for MIS development by the model theory approach:

External UI: ξ : UIState $\times U_0 \times V \rightarrow U \cup V_0$,

internalUI: $\cup(\{\text{actionN}_i\} \times \text{Paralist}_i) \rightarrow \{\text{name of } \delta_\lambda_{\text{actionN}_i}\}_{\text{i}} \times \text{Res}$,

Goal-seeker: $\sigma : C \rightarrow A$.

The internalUI is defined in Definition 14.14 in Appendix 14.2, and the goal-seeker was discussed in Sections 5.3 and 5.4.

Following the MIS model shown in Fig. 14.3, let the four sets U_0, V_0, U, and V be defined as follows:

U_0: input alphabet to the external UI,

V_0: output alphabet from the external UI,

U: input alphabet to the BMIS,

V: output alphabet from the BMIS.

The external UI accepts a command $u_0 \in U_0$ from a user and sends it to the BMIS as an input $u \in U$. The BMIS processes the command and generates a proper response $v \in V$, which is returned to the user by the external UI as an output $v_0 \in V_0$. The external UI is thus formally given by

$$\xi(\text{"input"}, u_0, -) = \xi_1(u_0) = u \in U,$$

and

$$\xi(\text{"output"}, -, v) = \xi_2(v) = v_0 \in V_0,$$

where UIState = {"input", "output"} and $\xi_1 : U_0 \to U$ and $\xi_2 : V \to V_0$.

We start here with the formalization of the BMIS.

Definition 14.1. Formal model of the BMIS
Let

$$T = \{0, 1, 2, \ldots\}$$

be the time set to describe the BMIS as a discrete system. The BMIS is a formal system called a time system [Mesarovic and Takahara, 1989], and is expressed as

$$\textbf{BMIS} \subset U^T \times V^T,$$

where $U^T = \{x | x : T \to U\}$.

Starting from the formal model of the BMIS, a canonical model of the BMIS can be derived as presented below. The formal derivation is given in Appendix 14.1.

Since the BMIS is a causal stationary discrete time system [Mesarovic and Takahara, 1989], it can be represented by a reduced automaton model $\langle U, V, C, \delta, \lambda, c_0 \rangle$,

$\delta : C \times U \to C$	(state transition function),
$\lambda : C \times U \to V$	(output function),
$c_0 \in C$	(initial state).

Let δ and λ be extended as $\delta : C \times U^* \to C$ and $\lambda : C \times U^* \to V^*$ where $\delta(c, \Lambda) = c$ and $\delta(c, xu) = \delta(\delta(c, x), u)$ and $\lambda(c, \Lambda) = \Lambda$ and $\lambda(c, xu) = \lambda(c, x) \cdot \lambda(\delta(c, x), u)$. U^* and V^* are the free monoids of U and V respectively. Λ denotes the null string. Then, a congruence relation $\sim_{c0} \subset U^* \times U^*$ can be defined as

$$(x, x') \in \sim_{c0} \leftrightarrow (\forall y)(\lambda(\delta(c_0, x), y) = \lambda(\delta(c_0, x'), y)).$$

Consequently, the BMIS can be represented by the following structure, which is called the canonical model of the BMIS. This is the most basic characterization of the BMIS.

Definition 14.2. Canonical state space model of the BMIS
The canonical state space model of the BMIS is described by

$$\textbf{canonical state space model} = \langle U, V, U^*/\sim_{c0}, \delta_{\text{can}}, \lambda_{\text{can}}, [\Lambda] \rangle,$$

where

$$\delta_{\text{can}} : U^*/\sim_{c0} \times U \to U^*/\sim_{c0} \text{ such that } \delta_{\text{can}}([x], u) = [x \cdot u]$$

and

$$\lambda_{\text{can}} : U^*/\sim_{c0} \times U \to V \text{ such that } \lambda_{\text{can}}([x], u) = \lambda(\delta(c_0, x), u).$$

Figure 14.4 shows a schematic of this structure.

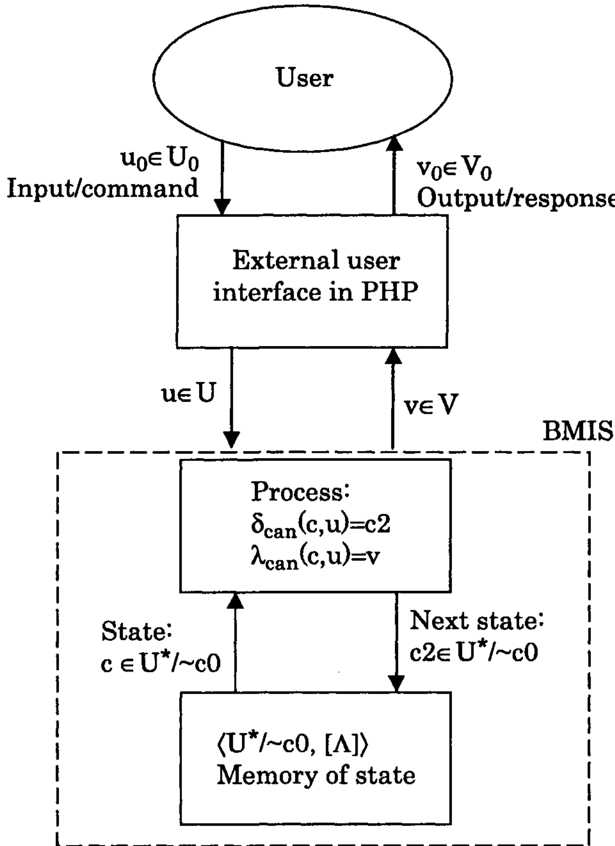


Fig. 14.4. Canonical state space model for BMIS.

14.2 Realization Structure of BMIS: File System

An implementation model for the BMIS is derived from $\langle U, V, U^*/\sim_{c0}, \delta_{\text{can}}, \lambda_{\text{can}}, [\Lambda]\rangle$ using the following structure, called the realization model. In the realization

model, the memory of state is represented by a family of files, called the file system. The notation realization() is used to represent the set-theoretic denotation for a name. For the name n of an attribute or a response let

$$\text{realization}(n) = \text{set of instances indicated by } n.$$

For example, for an attribute name "applicant."

$$\text{realization}(\text{"applicant"}) = \{\text{"banerji"}, \text{"shiba"}, \text{"takahara"}, \text{"saito"}, \ldots\}.$$

Let

$$\text{realization}(\text{name}_1, \ldots, \text{name}_k) = \text{realization}(\text{name}_1) \times \cdots \times \text{realization}(\text{name}_k).$$

The definition of the realization structure is then given as below.

Definition 14.3. Realization structure of the BMIS using a file system

1. ActionName: set of action (command) names $= \{n_1, \ldots, n_s\}$,
2. ResName: set of response names,
3. AttrName: set of attribute names used for actions $= \{\text{attr}_1, \ldots, \text{attr}_t\}$,
4. para: ActionName$\rightarrow \wp$(AttrName): action parameter specification function, where $\wp$(AttrName) is the power set of AttrName.
5. Paralist$_i$ = realization(para(actionN$_i$)), where actionN$_i$ $\in$ ActionName. para (actionN$_i$) will be typically denoted by $P_{i1} \times \cdots \times P_{in_i}$.
6. $\{f_j\}_j$: set of file structures such that f_j: AttrName0$\rightarrow$ {true, false} or $f_j \subset$ AttrName0, where AttrName0 is the set of attribute names used in the file system.

The input and output alphabets U and V are then given by

$$\begin{aligned} U &= \cup\{\{\text{actionN}\} \times \text{realization}(\text{para}(\text{actionN}))|\text{actionN} \in \text{ActionName}\}, \\ V &= \cup\{\text{realization}(\text{resName})|\ \text{resName} \in \text{ResName}\} \\ &\equiv \text{Res}. \end{aligned}$$

That is, $u \in U$ is a pair of an action name and an action parameter.

The sets ActionName and ResName and the para function are given by the developer as external information, where the following should hold:

$$\text{AttrName} = \cup\{\text{para}(\text{actionN}_i)|\ \text{actionN}_i \in \text{ActionName}\}.$$

The file system, which must implement $\langle U/\sim_{c0}, [\Lambda]\rangle$, is not given in the system specification. It can be considered even an independent component when designed as a database. Appendix 14.2 formally discusses relationship between the file system and $\langle U/\sim_{c0}, [\Lambda]\rangle$. The result leads to the realization structure for the BMIS shown in Fig. 14.5.

Figure 14.5 shows that the BMIS is decomposed into two components: an internal UI and a user model. The user model consists of atomic processes. When the BMIS on

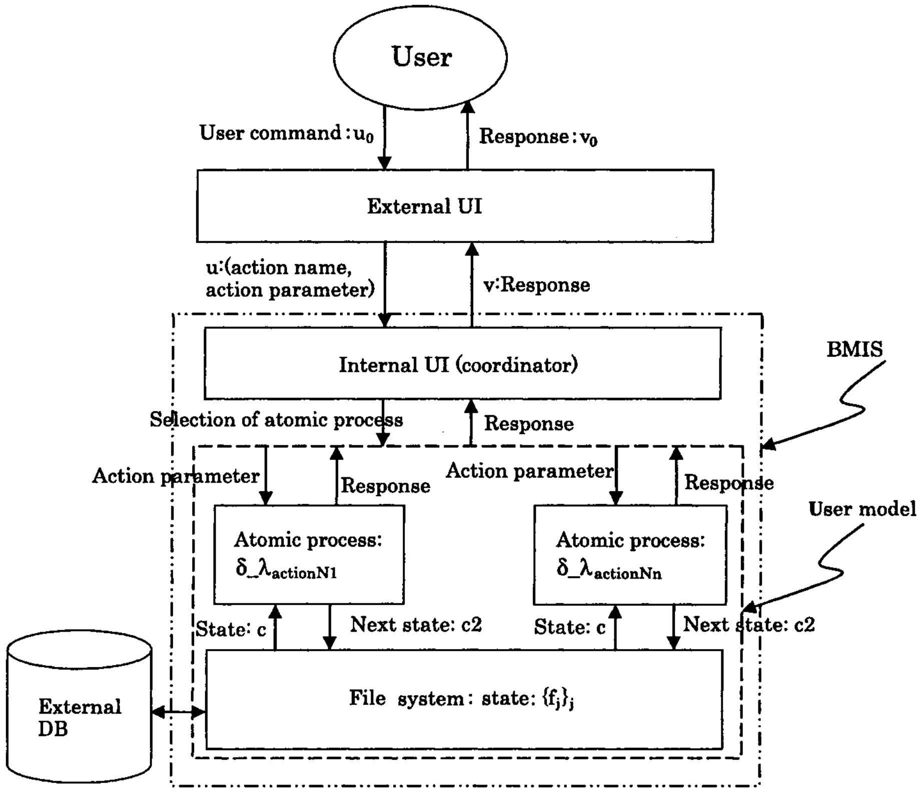


Fig. 14.5. Realization structure of the BMIS.

file system $\{f_j\}_j$ is represented by $\langle U, V, C, \delta, \lambda, c_0 \rangle$, an atomic process $\delta_\lambda_{\text{actionN}_i}$: $C \times P_{i1} \times \cdots \times P_{\text{ini}} \to$ Res $\times\, C$ is defined as follows: for each actionN$_i$ $\in$ ActionName and $(p_{i1}, \ldots, p_{\text{ini}}) \in$ Paralist$_i$,

$$\delta_\lambda_{\text{actionN}_i}(c, p_{i1}, \ldots, p_{\text{ini}}) = (\lambda(c, u), \delta(c, u)),$$

where $u = (\text{actionN}_i, p_{i1}, \ldots, p_{\text{ini}})$. $\delta_\lambda_{\text{actionN}_i}$ represents the unit operation corresponding to actionN$_i$ $\in$ActionName.

The total system shown in Fig. 14.5 starts with execution of the external UI. The interface accepts a user command for the BMIS as a pair of an action name and an action parameter, that is, (actionN, paralist)$\in$ {actionN} $\times$ realization(para(actionN)). In Fig. 14.11(b), action = "register" is selected, and in Fig. 14.11(c) parameter values "takahara" and "cit" are assigned to para("register") = {"name", "institute"}. The external UI sends it to the internal UI. The internal UI, then, selects a proper atomic process according to the action name, and the selected atomic process $\delta_\lambda_{\text{actionN}}$ executes the command using the parameter and information from the file system. On execution, an appropriate response is prepared by the atomic process and is returned to the user via the user interfaces (see Fig. 14.11(d)). At the same time, the file system is

updated by the process. Strictly speaking, the external interface first obtains the name of the target system in extProlog (see Fig. 14.11(a)).

The BMIS is always implemented in the scheme shown in Fig. 14.5, which imitates the scheme of an organizational cybernetic model [Takahara and Mesarovic, 2003]. The total structure is called modTPS (model based TPS).

The internal UI of Fig. 14.5 is an interface between the external UI and the user model, and acts as a coordinator of atomic processes. It is standardized and implemented in extProlog, and is provided as a black box for BMIS development. The internal UI has another function, which is to transmit a response from the user model to the external UI. On transmission, the internal UI acts as an identity mapping.

14.3 Atomic Process and Relational Structure of User Model

According to Fig. 14.5, the BMIS can be realized once the user model has been constructed. For implementation of the user model, the atomic process $\delta_\lambda_{\text{actionN}_i}$ is further decomposed into two components: an interface component and an implementation component.

Definition 14.4. Implementation structure of atomic process
The implementation structure of the ith atomic process is given as follows:

implementation structure of ith atomic process = ⟨ith interface component, ith implementation component⟩,

where

$$\textbf{\textit{i}th interface component} = \langle \delta_\text{lambda}(\text{actionN}_i) : \text{Paralist}_i \rightarrow \text{Res} \rangle,$$

$$\textbf{\textit{i}th implementation component} = \langle \phi_i : C \times \text{Paralist}_i \rightarrow \text{Res} \times C \rangle,$$

and

$$\textbf{delta_lambda}(\text{actionN}_i)(\text{paralist}) = \text{res} \leftrightarrow (\exists c, c2)((\text{res}, c2) := \phi_i(c, \text{paralist})).$$

Here, c and $c2$ are the state values of the file system, and ϕ_i is an implementation of $\delta_\lambda_{\text{actionN}_i}$.

Real operation of the atomic process is performed by ϕ_i. The user model is thus a family of atomic processes, where an atomic process corresponds to an operator of the organizational cybernetic model. Since the function indicated by a command name (action name) is performed by an atomic process, each action name is associated with one atomic process.

According to the implementation structure of an atomic process, the detailed structure of the ith atomic process is shown in Fig. 14.6.

As shown in Fig. 14.6, an atomic process consists of two components: an interface component and an implementation component. After an action name and its

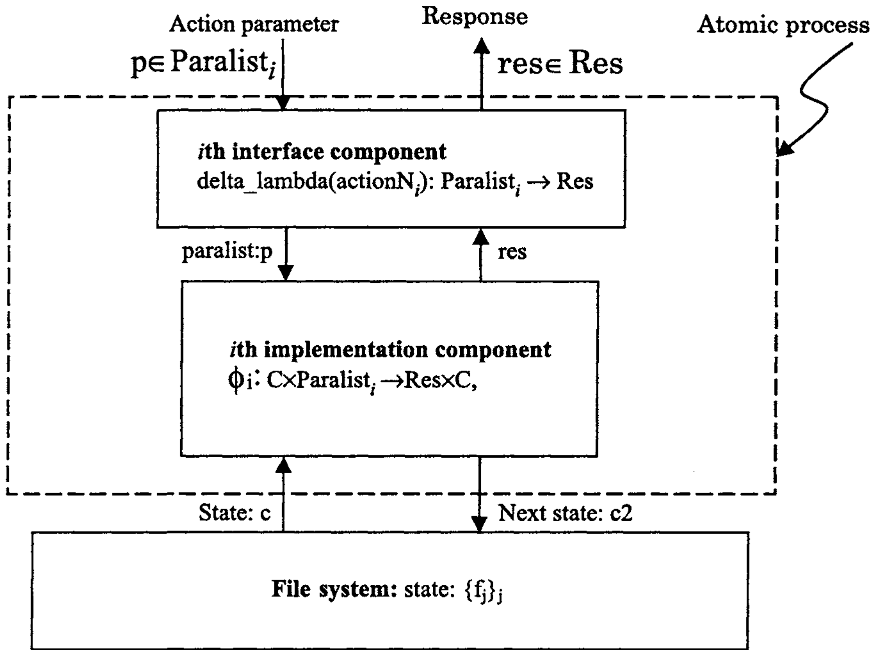


Fig. 14.6. Implementation structure of atomic process.

parameter have been specified by a user as an input (refer to Fig. 14.11(b, c)), and the internal UI has selected an atomic process associated with the action name and triggered the interface component, the selected interface component reads and formats the action parameters as a list and sends the list (paralist) to the implementation component.

Finally, the relational structure representation of the user model, required for implementation, is given below.

Definition 14.5. Relational structure of user model

$$\textbf{relational structure of user model} = \langle \text{ActionName, para}, \{i\text{th interface component, } i\text{th implementation component}\}_i \rangle.$$

The set ActionName and the function para() are used by the external UI to communicate with the user. Figure 14.2 presents ActionName for the registration system as a list. The user selects an action from the list to start the system (see Section 14.4).

There are two types of implementation components: a transaction type and a solver type. A transaction-type implementation component is a simple type that triggers state transition in the file system using information in the paralist. The content of the file system is consequently modified by the transition. In Fig. 14.6, this modification is illustrated as the transition of the current state c to the next state $c2$. The process "register" in Fig. 14.1 is an example of a transaction-type process. An appropriate response is also produced according to the state transition (see Fig. 14.11(d)).

The implementation component for a solver-type atomic process is not a simple mapping because a problem-solving task must be performed. The process "jobarrange" in Chapter 15 is an example of a solver-type process, in which an optimum matching problem is solved.

Since the extSLV discussed in the preceding chapters is a general problem-solving system, the structure of the solver type component can be obtained by replacing the implementation component in Fig. 14.6 with the extSLV. Figure 14.7 shows the implementation structure of a solver-type atomic process.

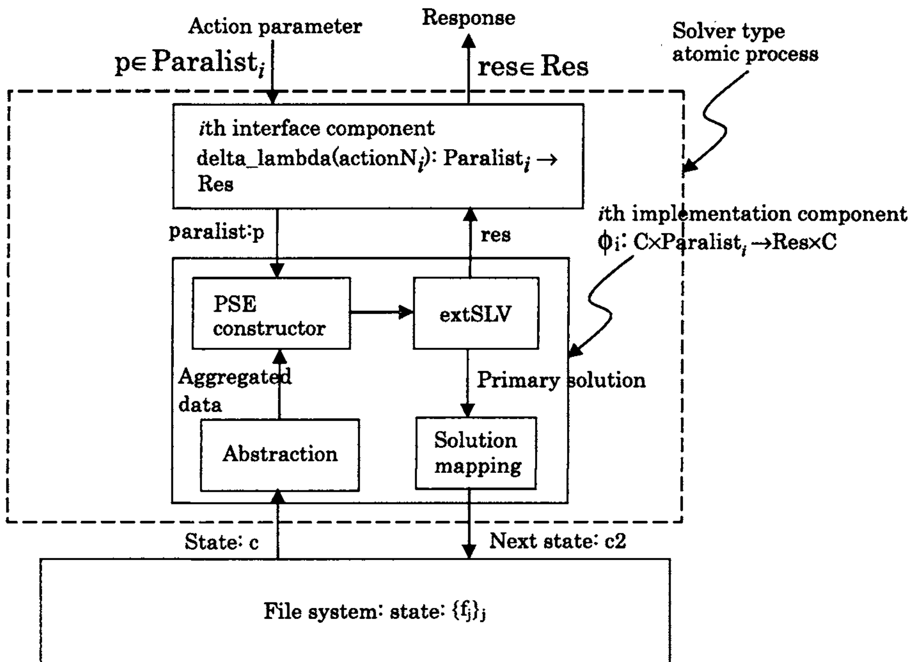


Fig. 14.7. Implementation structure of solver-type atomic process.

Information for the problem specification environment (PSE) is supplied from the user and the file system. Data from the file system is usually transformed into abstracted information suitable for problem-solving. The output from extSLV is called the primary solution, and may be modified as a secondary solution that is more understandable for the user. The solver-type component is discussed in detail in Chapter 15.

14.4 Dynamic Process of modTPS

The dynamic process of modTPS is summarized in Fig. 14.8.

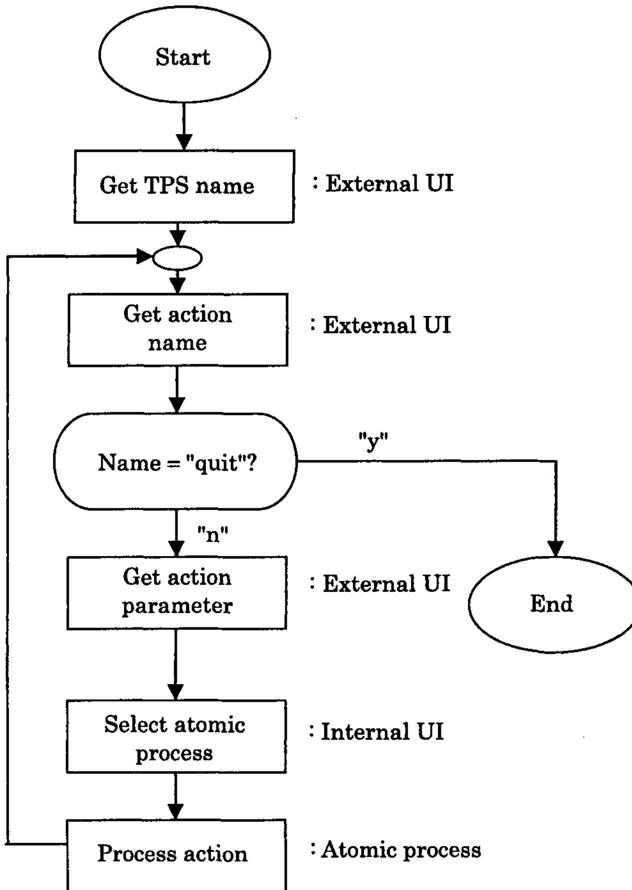


Fig. 14.8. Dynamic process of modTPS.

The activities "get TPS name," "get action name," and "get action parameter" are performed by the external UI. Here, "get TPS name" determines the target TPS. If the action name is "quit," the operation of the TPS is suspended. The activities "select atomic process" and "process action" are performed by the internal UI and the selected atomic process, respectively. After "process action," the TPS waits for a new command from the user.

14.5 Development Procedure for TPS Using the Implementation Structure of the User Model

Figure 14.9, essentially a reproduction of Fig. 1.3, shows the systems development process for a TPS and operation of it.

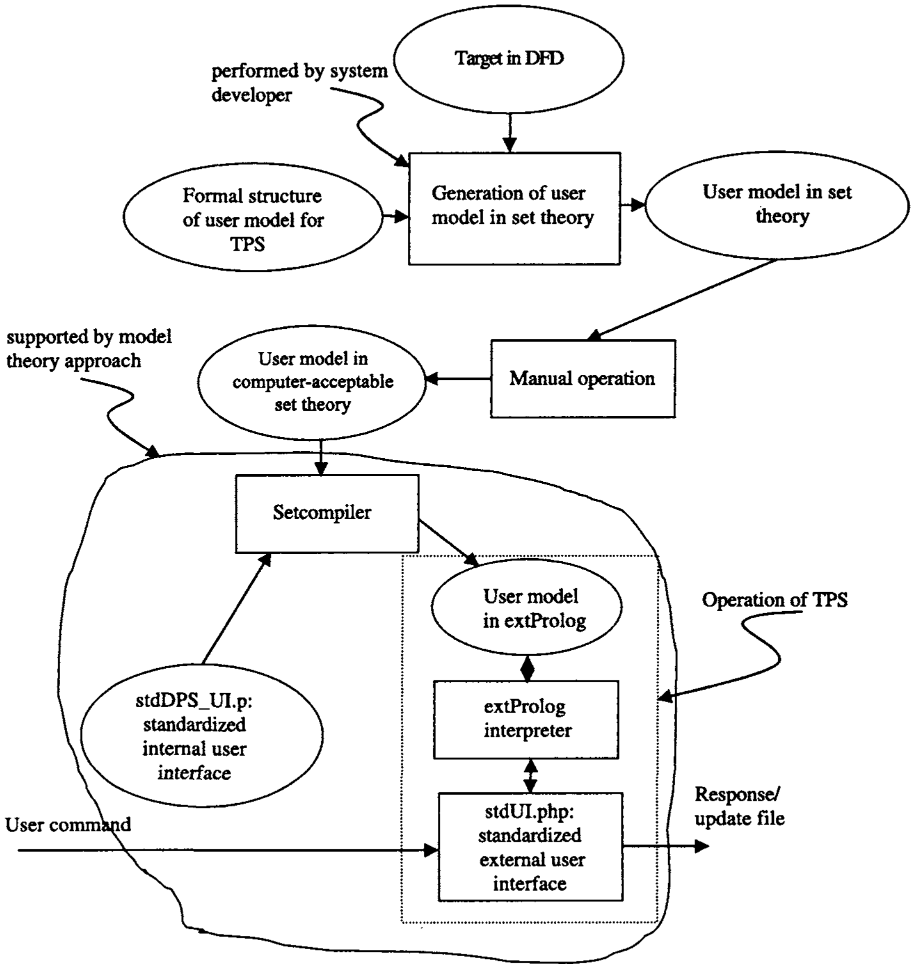


Fig. 14.9. TPS development process by the model theory approach.

A systems developer must provide a user model of the target system in computer-acceptable set theory, and that is the developer's only responsibility. The user model is first constructed from the DFD of the target following the relational structure provided in Section 14.3. Then, the model is transformed into a representation in computer-acceptable set theory. Chapter 1 introduced the representation as an implementation structure. The following is also a reproduction of the implementation structure mentioned in Chapter 1.

Implementation Structure of User Model for TPS in (Computer-Acceptable) Set Theory

```
func(<list of function name>);        /*declaration of function names used
                                         in the user model*/
```

```
ActionName.g=<list of action names>; /*specification of ActionName*/
                                      /*definition of atomic process of
                                        actionNi*/
<actionNi>.g=para(actionNi);          /*specification of parameters for
                                        actionNi*/
                                      /*interface component of actionNi*/
delta_lambda([actionNi],paralist)=    /*implementation component of
  res<->res:=actionNi(paralist);        actionNi*/
actionNi(paralist)=res
  (res,c2):=φ(c,paralist);
```

The set ActionName and the function para are represented as the global variables (see Chapter 2) ActionName.g and ⟨actionN$_i$⟩.g, respectively.

The setcompiler then translates the model into an extProlog user model and attaches the internal UI.

Finally, the derived user model is executed by the extProlog interpreter under the control of the external UI, which accepts a user request and outputs a response prepared by the user model. The user model also modifies the file system if necessary. The three systems, the extProlog user model, the extProlog interpreter and the external UI, construct the target TPS. Section 14.6.5 shows an example of the implementation structure.

The development process involves five components: the formal structure of the user model, the setcompiler, the internal UI, the extProlog interpreter, and the external UI. The setcompiler is presented in Chapter 2, the external UI is discussed in Appendix 14.3, and the extProlog interpreter is discussed in Chapters 17 and 18. The others are introduced in this chapter.

14.6 User Model Construction: Example

Figure 14.10 outlines the user model construction procedure using the implementation structure of a user model.

The construction procedure consists of six stages:

drawing of the data flow diagram (DFD) as a generalized block diagram,
specification of action names and response names in set theory,
design of the file structure system and para function, and realization of names,
design and description of atomic processes in the relational structure,
compilation of the user model into an extProlog system to which the standardized internal UI is attached, and
test and operation under control of the external UI.

The simple example shown in Fig. 14.1 is used to illustrate the TPS development procedure.

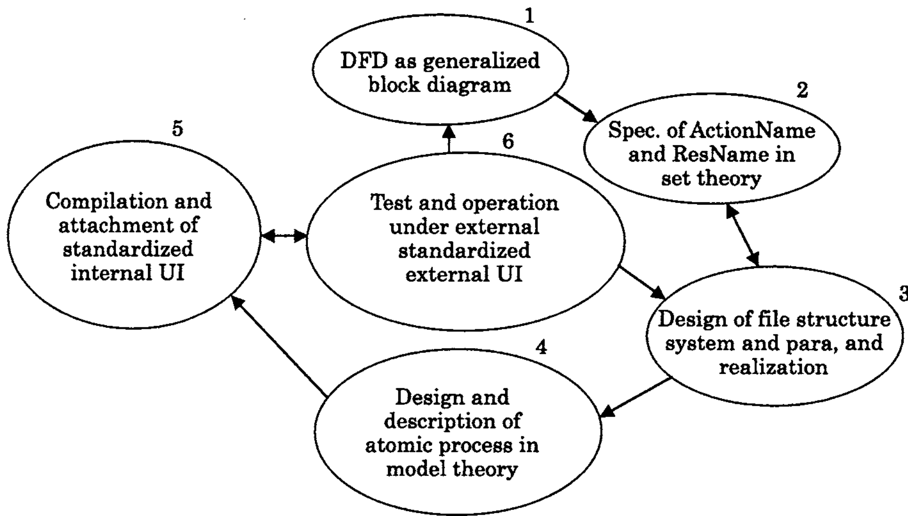


Fig. 14.10. User model construction and compilation for TPS.

14.6.1 DFD (Generalized Input–Output Block Diagram)

The input–output block diagram definition and input–output specification in set theory are critical phases in TPS development. Currently, a DFD is used as a generalized block diagram to determine the input and output sets or the functions of the target system. Figure 14.1 will be used for the registration system.

14.6.2 Specification of ActionName and ResName in Set Theory for Registration System

From the DFD in Fig. 14.1, the following input–output specification can be derived directly.

ActionName set:

ActionName = {"register", "check_participant", "total_fee", "quit",
"reg_update"},

where

register: registration of participant,
reg_update: update of registration,
check_participant: display of participant information,
total_fee: display of total fee paid for registration,
quit: suspension of system.

ResName set:

ResName = {"registerRes", "check_participantRes", "total_feeRes",
"reg_updateRes"}.

14.6.3 Design of File Structure System $\{f_j\}_j$ and Para Function, and Realization of Names

The DFD in Fig. 14.1 specifies one file name for this system. Suppose the para function and hence AttrName are given as follows:

para function:

para("register") = {"name", "institute"},
para("reg_update") = {"name", "institute", "fee", "track"},
para("check_participant") = {"name"},
para("total_fee") = ∅,

where

name: participant name,
institute: affiliation of participant,
fee: paid registration fee,
track: workshop track of participant,
date: date of event.

For example, para("register") = {"name", "institute" } implies that the action "register" requires information of ("name", "institute") for its execution.

AttrName set:

AttrName = {"name", "institute", "fee", "track", "date"}.

Then, Proposition 14.6 of Appendix 14.2 implies the file structure f_1 as below.

File structure system $\{f_j\}_j$:

f_1 = participant.lib = {"name", "institute", "fee", "track", "date"}.

Realization of the attribute names is given below.

Realization of attribute names:

Name = realization("name") = set of participant names (= set of strings),
Institute = realization("institute") = set of institute names (= set of strings),
Fee = realization("fee")= set of integers,
Track = realization("track") = set of track names,
Date = realization("date") = set of event dates (= {(Y, M, D)}).

Using realization of attributes, the file system is realized as follows.

Realization of file system:

$$\text{Participant.lib} = ((\text{Name} \times \text{Institute}) \times (\text{Fee} \times \text{Track} \times \text{Date}))^*.$$

It is assumed that in the model theory approach every file is suffixed with ".lib" (refer to Chapter 2).

An element of ResName is realized in a similar way.

Realization of ResNames:

$$\text{RegisterRes} = \text{realization}(\text{"registerRes"}) = \text{Name} \times \text{Institute} \times \text{RegType} \times \text{RegData},$$

where

$$\begin{aligned}
\text{RegType} &= \{.\text{pre_registration}, .\text{new_registration}, \\
&\quad .\text{incomplete_registration}\}, \\
\text{RegData} &= \{(.\text{fee}, .\text{track}), (.\text{receipt}, .\text{proceeding})\}, \\
\text{Check_participantRes} &= \text{realization}(\text{"check_participantRes"}) \\
&= \text{Institute} \times \text{Track}, \\
\text{Total_feeRes} &= \text{Int}, \\
\text{Reg_updateRes} &= \text{realization}(\text{"reg_updaters"}) \\
&= \{\text{"reg_update is completed"}, \text{"input data is wrong"}\}.
\end{aligned}$$

It should be noted that .pre_registration, .new_registration, .incomplete_registration, .fee, .track, .receipt, and .proceeding are constant symbols (see Chapter 2).

14.6.4 Design of Atomic Process

Following the formulation presented in Section 14.5, the atomic process for "register" is given as follows.

Atomic process for register:

$$\text{delta_lambda}([\text{register}], \text{paralist}) = \text{res} \leftrightarrow \text{res} := \text{register}(\text{paralist});$$

$$\begin{aligned}
\text{register}([\text{name}, \text{institute}]) = \text{res} \leftrightarrow (\text{res}, \text{Participant.lib}) \\
:= \phi\ (\text{Participant.lib}, [\text{name}, \text{institute}]);
\end{aligned}$$

$$\phi(\text{Participant.lib}, [\text{name, institute}]) = \begin{cases} ([\text{name, institute}, .\text{new_registration}, .\text{fee}, .\text{track}], \text{Participant.lib} \cup \{[[\text{name}, \text{institute}], [0, 0, \text{dateV}]]\}) & \text{if } [[\text{name, institute}],*] \notin \text{Participant.lib}, \\ ([\text{name, institute}, \text{pre_registration}, .\text{receipt}, .\text{proceeding}], \text{Participant.lib}) & \text{if } [[\text{name, institute}],*] \in \text{Participant.lib} \ \& \ \text{fee} \neq 0 \& \ \text{track} \neq 0, \\ ([\text{name, institute}, .\text{incomplete_registration}, .\text{fee}, .\text{track}], \text{Participant.lib}) & \text{otherwise} \end{cases}$$

where "*" is used as a wild card symbol.

The meaning of ϕ must be clear. For example, if a participant has not registered yet, i.e., [name, institute,*] $\notin$ Participant.lib, the registration information [[name, institute], [0, 0, dateV]] is appended to Participant.lib and the corresponding response, [name, institute, .new_registration, .fee, .track], is produced, which states that this registration is new and the new registrant is required to pay a registration fee and specify sessions of interest. dateV is the date when the registration operation is performed.

14.6.5 Implementation Description of User Model in Computer-Acceptable Set Theory

A user model of the registration system in computer-acceptable set theory is given below, in which one action "register" is illustrated:

```
/*register53.set*/
func([delta_lambda,register]);
ActionName.g=["quit","register"];
/*atomic process of register*/
register.g=["name","institute"];
delta_lambda([.register],paralist)=res <->
          res := register(paralist);

register([name,institute])=res <->
           (notmember([[name,institute],y],participant.lib)) ->
                  (
                  date:=getDate2(),
                  participant.lib := union(participant.lib,[[[name,
                     institute],[0,0,date]]]),
                  res:=[name,institute,.new_registration,.fee,.track]
                  )
         .otherwise
                  (
```

```
[fee,track,date] := participant.lib([name,institute]),
(fee <> 0 and track <> 0) ->
          (
          res:=[name,institute,.pre_registration,
             .receipt,.proceeding]
          )
.otherwise
          (
          res:=[name,institute,.incomplete_registration,
             .fee,.track]
          )
);
```

The following are correspondences between the implementation structure shown in Section 14.5 and that in the above model:

```
func(<list of function name>);↔func([delta_lambda,register]);
ActionName.g=<list of action names>;↔ActionName.g=["quit","register"];
<actionNi> .g=para (actionNi);↔register.g=["name","institute"];
interface component of <actionNi>;↔delta_lambda(["register"],paralist)=
   res <->
implementation component of <actionNi>;↔register([name,institute])=
   res <->
```

14.7 Operation of Compiled User Model

Figure 14.11 illustrates real operations of the registration system, which is generated from the user model of Section 14.6.5 by the setcompiler. Figure 14.11(a) shows the "get TPS name" function phase of the modTPS shown in Fig. 14.8, where "register5.p"

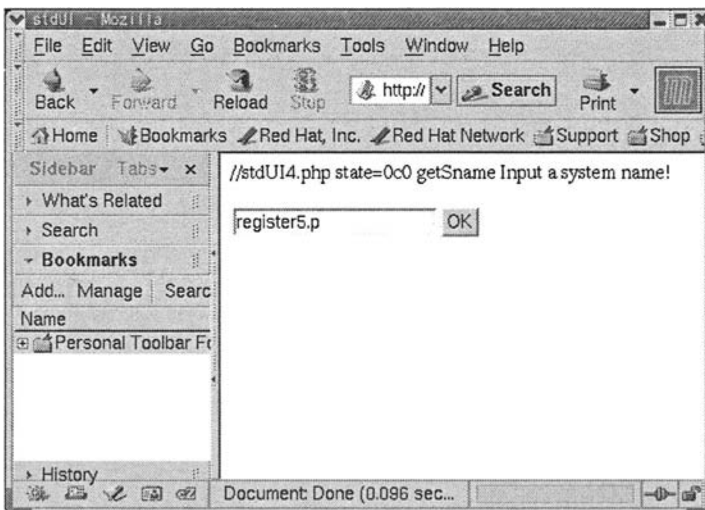


Fig. 14.11(a). Specification of user model.

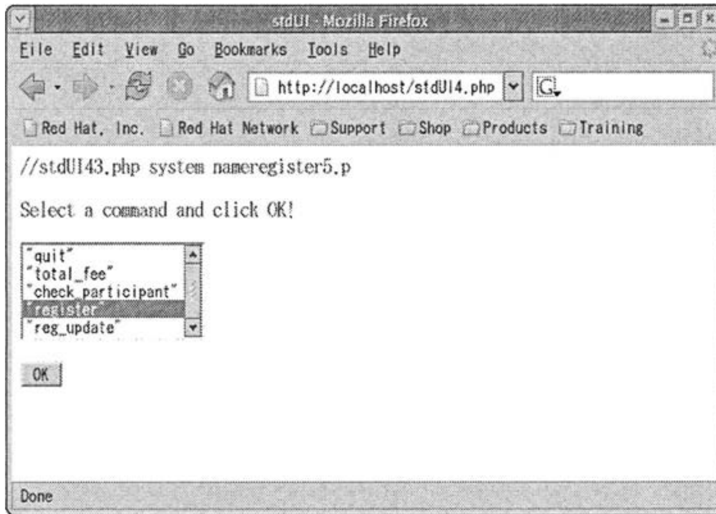


Fig. 14.11(b). Selection of action (command name).

is specified as the target TPS. The system "register5.p" is a superset of "register53.p" of Section 14.6.5.

Figure 14.11(b) shows the "get action name" function phase. Since "register5.set" contains the definition ActionName.g = ["quit", "total_fee", "check_participant", "register", "reg_update"], five actions are displayed as action candidates. The action name "register" is selected in the figure.

Figure 14.11(c) shows the "get action parameter" function phase. The action "register" requires two parameter values for name and institute, or register.g = ["name",

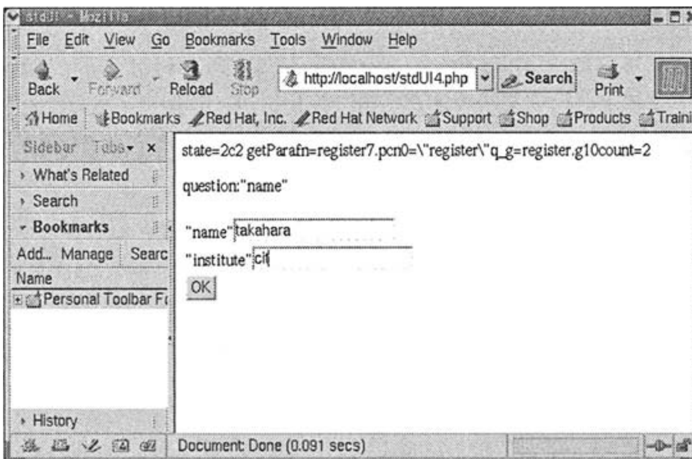


Fig. 14.11(c). Input of parameter.

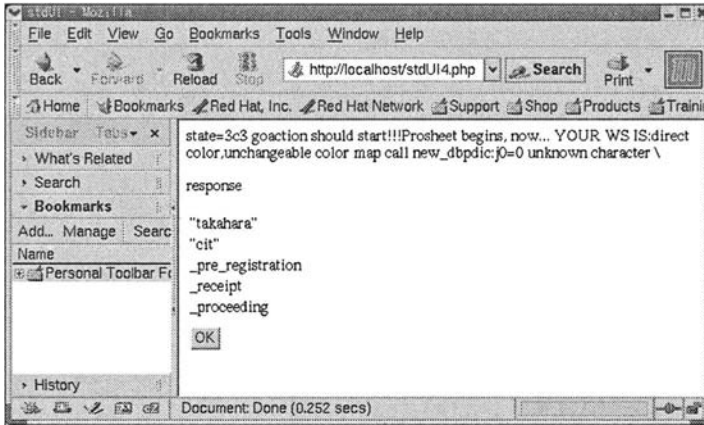


Fig. 14.11(d). Response for action "register".

"institute"] (i.e., para("register")={"name", "institute"}). The values "takahara" and "cit" are inserted in the figure.

Finally, Fig. 14.11(d) illustrates the response of the action "register" or the phase after the "process action" function, showing that "takahara" is already registered and hence should receive a receipt and the proceedings.

More sophisticated external UIs have been designed for the model theory approach, which are illustrated in Chapter 16 [Yano, 2005].

Appendix 14.1 Derivation of Canonical Representation of BMIS*

We introduce several concepts and facts about a time system [Mesarovic and Takahara, 1989]. Let $S \subset X \times Y$ be a time system where

$$X = U^T$$

and

$$Y = V^T.$$

Definition 14.6. Initial response function
Let C_0 be a set. Then, a function $P_0 : C_0 \times X \to Y$ that satisfies

$$(x, y) \in S \leftrightarrow (\exists c)(y = P_0(c, x))$$

is called an initial response function of $S \subset X \times Y$. The set C_0 is called an initial state set.

In order to give descriptive power to the time system model, three operations on time functions are introduced:

(1) Restriction operation

Let

$$T^t = [0,t), T_{ts} = [t,s), T_t = [t,\infty), \underline{T}^t = [0,t], \underline{T}_{ts} = [t,s],$$
$$x^t = x|T^t, x_t = x|T_t, x_{ts} = x|T_{ts}, \underline{x}^t = x|\underline{T}^t, \underline{x}_{ts} = x|\underline{T}_{ts}.$$

(2) Concatenation operation

Let "·" be the concatenation operator, which is specified as follows:

$$x'' = x^t \cdot x'_t,$$

where

$$x''(s) = \begin{cases} x^t(s) & \text{if } s < t, \\ x'_t & \text{if } s \geq t. \end{cases}$$

(3) Shift operation

Let σ^t be a shift operator for a time function that is specified as follows:

$$\sigma^t(x_{uv})(s) = x_{uv}(s-t), \quad \text{where } u+t \leqq s \leqq v+t.$$

That is, σ^t shifts x_{uv} by t parallel to the time axis.

Let

$$\eta^t(-) = \sigma^{-t}(-|T_t).$$

Using the above definitions the fundamental character of the BMIS is given below.

Definition 14.7. Causal system

An initial response function $P_0 : C_0 \times X \to Y$ of $S \subset X \times Y$ is called causal iff for any $c \in C_0$ the following holds: for any x and x' and t,

$$x^t = x'^t \to P_0(c,x)|\underline{T}^t = P_0(c,x')|\underline{T}^t.$$

Then, a time system S is called *causal* iff S has a causal initial response function.

We have the following fact.

Theorem 14.1. *Suppose a time system S is output-complete. Then S is causal iff the following holds:*

$$x^t = x'^t \to S(x)|\underline{T}^t = S(x')|\underline{T}^t,$$

where $S(x)|T^t = \{y^t|(x,y) \in S\}$ [Mesarovic and Takahara, 1989].

Definition 14.8. Stationary system

A time system S is stationary iff for any t the following holds:

$$\eta^t(S) \subset S,$$

where $\eta^t((x,y)) = ((\eta^t(x), \eta^t(y))$.

Definition 14.9. State space representation
For a time system $S \subset X \times Y$ and for a set C let

$$\underline{\delta} = \{\delta_{st} | \delta_{st} : C \times X_{st} \to C \text{ and } s, t \in T \text{ and } s \leq t\}$$

and

$$\underline{\lambda} = \{\lambda_t | \lambda_t : C \times U \to V \text{ and } t \in T\}.$$

Then, $\langle \underline{\delta}, \underline{\lambda} \rangle$ is called a state space representation of S iff the following conditions are satisfied:

(1) $\underline{\delta}$ satisfies

$$(\alpha)\delta_{su}(c, x_{su}) = \delta_{tu}(\delta_{st}(c, x_{st}), x_{tu}) \qquad \text{(composition property)},$$

where $s \leqq t \leqq u$,

$$(\beta)\delta_{tt}(c, x_{tt}) = c \qquad \text{(consistency property)}.$$

(2) $(x, y) \in S \leftrightarrow (\exists c \in C)(\forall t)(y(t) = \lambda_t(\delta_{0t}(c, x^t), x(t)))$.
δ_{st} and λ_t and C are called state transition function, output function and state set, respectively.

Definition 14.10. Time-invariant state space representation
Let a response function at t be defined as

$$P_t(c, x_t)(s) = \lambda_s(\delta_{ts}(c, x_{ts}), x_t(s)), \quad \text{where } t \leqq s.$$

Then, a state space representation $\langle \underline{\delta}, \underline{\lambda} \rangle$ is called time invariant iff the following conditions are satisfied:

(i) $P_t(c, x_t) = \sigma^t[P_0(c, \sigma^{-t}(x_t))]$,
(ii) $\delta_{ts}(c, x_{ts}) = \delta_{0u}(c, \sigma^{-t}(x_{ts}))$, where $u = s - t$,
(iii) $\eta^t(X) = X$.

It should be noted that $\lambda_t(c, u) = \lambda_0(c, u)$ can be proved from $P_t(c, x_t) = \sigma^t[P_0(c, \sigma^{-t}(x_t))]$.

Theorem 14.2. *Every causal stationary system can be modeled as a time-invariant state space representation [Mesarovic and Takahara, 1989].*

Corollary 14.1. MIS $\subset X \times Y$ *is a causal stationary discrete time system. Consequently, it can be represented by a time-invariant state space representation.*

Definition 14.11. State space representation of BMIS
Let

$$\textbf{StateSpaceRepresentationofBMIS} = \langle U, V, C, \delta, \lambda \rangle,$$

where

$$\delta(c, u) = \delta_{01}(c, u),$$

$$\lambda(c, u) = \lambda_0(c, u).$$

Assumption 14.2. We can assume without loss of generality [Mesarovic and Takahara, 1989] that $\langle U, V, C, \delta, \lambda \rangle$ is **reduced**, i.e.,

$$(\forall x)(\lambda(c, x) = \lambda(c', x)) \rightarrow c = c',$$

where λ is extended as $\lambda : C \times U^* \rightarrow V^*$ such that $\lambda(c, []) = []$ and $\lambda(c, xu) = \lambda(c, x) \cdot \lambda(\delta(c, x), u)$. δ is also extended as $\delta : C \times U^* \rightarrow C$ in the same way.

Corollary 14.1 presents a starting model of the BMIS, which is shown in Fig. 14.12.

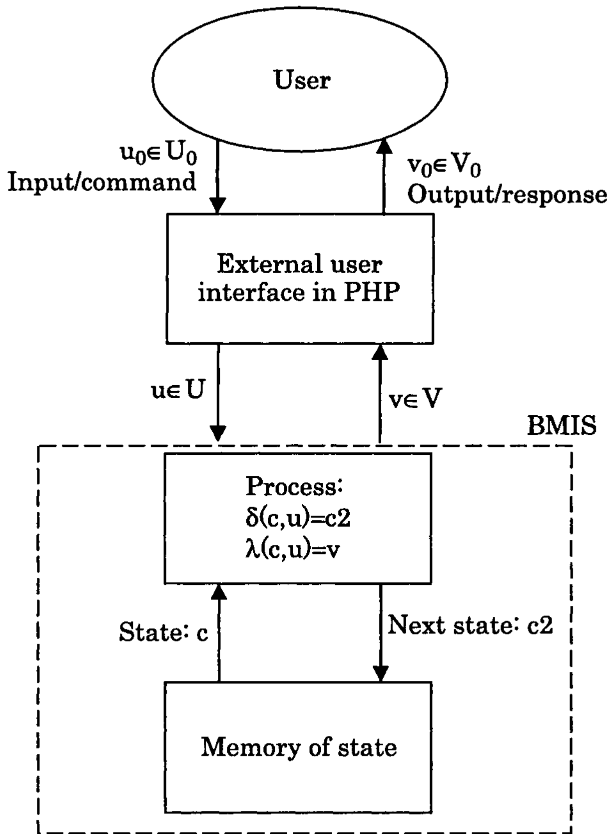


Fig. 14.12. State space model for BMIS.

The structure of the BMIS consists of two components: a memory of state and a process. The process component provides both control of the memory of state and preparation of a response for the input, and the memory of state saves the state of the BMIS. Since the process and file system components can be considered to constitute an automaton model, the model is called an automaton model for transaction processing, and a user command is called an action.

In order to derive a detailed structure for C, we make the following assumption.

Assumption 14.3. The target behavior of system development is described by a subbehavior sub-MIS $\subset U^T \times V^T$ of the BMIS such that there is $c_0 \in C$ and the following holds:

$$(x, y) \in \text{sub-MIS} \leftrightarrow (\forall t)(y(t) = \lambda(c_0, x^t)(t)).$$

Here c_0 is a fixed initial state or a reset state of an MIS. Then we have the following representation of the state set C.

Proposition 14.1. *Let* $\sim_{c0} \subset U^* \times U^*$ *be as follows:*

$$(x, x') \in \sim_{c0} \leftrightarrow (\forall y \in U^*)(\lambda(\delta(c_0, x), y) = \lambda(\delta(c_0, x'), y)).$$

Then, $\sim_{c0}$ *is a congruence relation such that the following functions are well defined:*

$$\delta_{\text{can}} : U^*/\sim_{c0} \times U \to U^*/\sim_{c0} \text{ such that } \delta_{\text{can}}([x], u) = [x \cdot u]$$

and

$$\lambda_{\text{can}} : U^*/\sim_{c0} \times U \to V \text{ such that } \lambda_{\text{can}}([x], u) = \lambda(\delta(c_0, x), u).$$

Proof. δ_{can} is well-defined. Suppose $[x] = [y]$. Let $u \in U$ and $z \in U^*$ be arbitrary. Then, $[x] = [y]$ implies $\lambda(\delta(c_0, x), uz) = \lambda(\delta(c_0, y), uz)$. Then $\lambda(\delta(\delta(c_0, x), u), z) = \lambda(\delta(\delta(c_0, y), u), z)$ holds. It implies $\lambda(\delta(c_0, xu), z) = \lambda(\delta(c_0, yu), z)$. Consequently, $[xu] = [yu]$.

Furthermore, λ_{can} is also well-defined. Suppose $[x] = [y]$. Then, $[x] = [y]$ implies $\lambda(\delta(c_0, x), u) = \lambda(\delta(c_0, y), u)$. Q.E.D.

Moreover, δ_{can} is simple. However, since λ_{can} represents a problem-solving operation as discussed in Section 14.3, it is sometimes complicated.

Proposition 14.2. *Let a function* $\eta : U^*/\sim_{c0} \to C$ *be such that* $\eta([x]) = \delta(c_0, x)$. *Then the following diagrams are commutative.*

$$\begin{array}{ccccc} U^*/\sim_{c0} & \times U & \xrightarrow{\delta_{\text{can}}} & U^*/\sim_{c0} \\ \eta \downarrow & \downarrow I & & \downarrow \eta \\ C & \times U & \xrightarrow{\delta} & C \end{array} \qquad \begin{array}{ccccc} U^*/\sim_{c0} & \times U & \xrightarrow{\lambda_{\text{can}}} & V \\ \eta \downarrow & \downarrow I & & \downarrow I \\ C & \times U & \xrightarrow{\lambda} & V \end{array}$$

Proof. η is well-defined because $[x] = [x']$ implies $(\forall y)(\lambda(\delta(c_0, x), y) = \lambda(\delta(c_0, x'), y))$, which implies $\delta(c_0, x) = \delta(c_0, x')$ ($\langle \delta, \lambda \rangle$ is reduced). Then, $\eta(\delta_{\text{can}}([x], u)) = \eta([xu]) = \delta(c_0, xu) = \delta(\delta(c_0, x), u) = \delta(\eta([x]), u)$ holds. Moreover, $\lambda_{\text{can}}([x], u) = \lambda(\delta(c_0, x), u) = \lambda(\eta([x]), u)$ also holds. Q.E.D.

Definition 14.12. Realization of state space representation

A state space representation $\langle U, V, C', \delta', \lambda', c_0' \rangle$ is called a realization of $\langle U, V, C, \delta, \lambda, c_0 \rangle$ if there is a surjection $h : C' \to C$ such that $h(c_0') = c_0$ and the following diagrams are commutative:

$$\begin{array}{ccc} C' \times U & \xrightarrow{\delta'} & C' \\ h \downarrow \quad \downarrow I & & \downarrow h \\ C \times U & \xrightarrow{\delta} & C \end{array} \qquad \begin{array}{ccc} C \times U & \xrightarrow{\lambda'} & V \\ h \downarrow \quad \downarrow I & & \downarrow I \\ C \times U & \xrightarrow{\lambda} & V \end{array}$$

Corollary 14.2. *$\langle U, V, \eta(U^*/\sim_{c0}), \delta, \lambda, c_0\rangle$ is realized by $\langle U, V, U^*/\sim_{c0}, \delta_{\text{can}}, \lambda_{\text{can}}, [\Lambda]\rangle$. Furthermore, for any $x \in U^*$,*

$$\lambda_{\text{can}}([\Lambda], x) = \lambda(c_0, x).$$

Proof. First, it must be shown that $\delta(\eta(U^*/\sim_{c0}), u) \subset \eta(U^*/\sim_{c0})$ holds for any u. Let $u \in U$ be arbitrary. Then $c \in \eta(U^*/\sim_{c0})$ implies $(\exists x)(c = \eta([x]))$. Then, $\delta(c, u) = \delta(\eta([x]), u) = \eta(\delta_{\text{can}}([x], u))$ (because of Proposition 14.2) $= \eta([xu]) \in \eta(U^*/\sim_{c0})$ is true. Hence, $\delta : \eta(U^*/\sim_{c0}) \times U \to \eta(U^*/\sim_{c0})$ is well-defined or $\langle U, V, \eta(U^*/\sim_{c0}), \delta, \lambda, c_0\rangle$ is realized by $\langle U, V, U^*/\sim_{c0}, \delta_{\text{can}}, \lambda_{\text{can}}, [\Lambda]\rangle$. Next, $\lambda_{\text{can}}([\Lambda], x) = \lambda(c_0, x)$ can be proved by mathematical induction. The commutative diagram of Proposition 14.2 implies that $\lambda_{\text{can}}([\Lambda], u) = \lambda(\delta(c_0, \Lambda), u) = \lambda(c_0, u)$ holds. Suppose $\lambda_{\text{can}}([\Lambda], x) = \lambda(c_0, x)$ is true. Then, $\lambda_{\text{can}}([\Lambda], xu) = \lambda_{\text{can}}([\Lambda], x) \cdot \lambda_{\text{can}}(\delta_{\text{can}}([\Lambda], x), u) = \lambda(c_0, x) \cdot \lambda_{\text{can}}(\delta_{\text{can}}([\Lambda], x), u) = \lambda(c_0, x) \cdot \lambda_{\text{can}}([x], u) = \lambda(c_0, x) \cdot \lambda(\eta([x]), u) = \lambda(c_0, x) \cdot \lambda(\delta(c_0, x), u) = \lambda(c_0, xu)$ holds. Q.E.D.

We have a strong result. Let $C_r = \{c | (\exists x \in U^*)(c = \delta(c_0, x))\}$. Then

Proposition 14.3. *η of Proposition 14.2 is an isomorphism between $\langle U, V, C_r, \delta, \lambda, c_0\rangle$ and $\langle U, V, U^*/\sim_{c0}, \delta_{can}, \lambda_{can}, [\Lambda]\rangle$.*

Proof. Since the relation $c \in \eta(U^*/\sim c_0) \leftrightarrow (\exists x \in U^*)(c = \delta(c_0, x))$ holds, $C_r = \eta(U^*/\sim c_0)$. The mapping η is injective because $\eta([x]) = \eta([x'])$ implies $\delta(c_0, x) = \delta(c_0, x')$. Hence, $(\forall y)(\lambda(\delta(c_0, x), y) = \lambda(\delta(c_0, x'), y))$ holds, which implies $x \sim_{c0} x'$. Therefore, $[x] = [x']$ holds. Q.E.D.

It should be noted that η is essentially an automaton isomorphism, or any reduced state space representation of the BMIS is isomorphic to $\langle U, V, U^*/\sim_{c0}, \delta_{\text{can}}, \lambda_{\text{can}}, [\Lambda]\rangle$ whose isomorphism is given by η.

Appendix 14.2 Derivation of Realization Structure of BMIS on File System*

Let

$$\underline{\text{AttrName0}} = \text{AttrName} \cup \{\text{“actionName”}\},$$

where “actionName” is an attribute to denote the action name of an event.

Let

$$\underline{C_0} = \text{realization}(\underline{\text{AttrName0}})^*$$

and

$$\delta_0 : \underline{C_0} \times U \to \underline{C_0}$$

such that for $u = (\text{actionN}_i, \underline{p})$ $\delta_0(\alpha, u) = \alpha \cdot (q_1, \ldots, q_t, \text{actionN}_i)$ where $\underline{p} \in$ realization(para(actionN$_i$)) and

$$q_j = \begin{cases} \text{project}(\underline{p}, \text{attr}_j) & \text{if attr}_j \in \text{para}(\text{actionN}_i), \\ * & \text{otherwise,} \end{cases}$$

where '$*$' indicates a dummy data.

For example, if para(actionN$_i$) $= \{\text{attr}_2, \text{attr}_3\}$ and $\underline{p} = (p_1, p_2)$ where $p_1 \in$ realization(attr$_2$) and $p_2 \in$ realization(attr$_3$), then project$(\underline{p}, \text{attr}_3) = p_2$.

Let $(q_1, \ldots, q_t) = \underline{q}$ be denoted by ext(actionN$_i$, $\underline{p}$).

Proposition 14.4. *There is an injection $k : U^* \to \underline{C}_0$ and hence a bijection $h : k(U^*) \to U^*$ such that the following diagram is commutative:*

$$\begin{array}{ccccc} k(U^*) & \times & U & \xrightarrow{\delta_0} & k(U^*) \\ h\downarrow & & \downarrow I & & \downarrow h \\ U^* & \times & U & \xrightarrow{\text{append}} & U^* \\ []\downarrow & & \downarrow I & & \downarrow [] \\ U^*/\sim_{c0} & \times & U & \xrightarrow{\delta_{\text{can}}} & U^*/\sim_{c0} \end{array}$$

Consequently, $\langle U, V, U^*/\sim_{c0}, \delta_{\text{can}}, \lambda_{\text{can}}, [\Lambda]\rangle$ can be realized by $\langle U, V, k(U^*), \delta_0, \lambda_0, \Lambda\rangle$, where $\lambda_0 : k(U^*) \times U \to V$ is $\lambda_0(c, u) = \lambda_{\text{can}}([h(c)], u)$.

Proof. Let us prove the above fact using an example. Let

$$u = (\text{actionN}, \underline{p})$$

and

$$u' = (\text{actionN}', \underline{p}').$$

Then, let k be defined as follows:

$$k(u \cdot u') = (\text{ext}(\text{actionN}, \underline{p}), \text{actionN}) \cdot (\text{ext}(\text{actionN}', p'), \text{actionN}').$$

Then, k is an injection. Q.E.D.

Consequently, the memory of state could be realized by one file whose attribute name set is $\underline{\text{AttrName0}} = \text{AttrName} \cup \{\text{"actionName"}\}$. This realization is, however, impractical. For example, it may not satisfy the normal form conditions of database theory.

In order to derive a more practical realization we assume that $\underline{\text{AttrName0}}$ is decomposed into a file system $\{f_j\}_j$ such that the following conditions hold:

(i) $\cup f_j = \text{AttrName} \cup \{\text{"actionName"}, \text{"date"}\}$,
(ii) $f_j \supset \{\text{"actionName"}, \text{"date"}\}$,

where "date" and "actionName" are attribute names to denote the date and the action name of an event, respectively.

Let

$$\underline{C} = (\text{realization}(f_1))^* \times \cdot \times (\text{realization}(f_J))^*$$

and

$$\delta_{\text{rep}} : \underline{C} \times U \to \underline{C}$$

such that for $u = (\text{actionN}_i, \underline{p})$ $\delta_{\text{rep}}((\alpha_1, \ldots, \alpha_J), u) = (\beta_1, \ldots, \beta_J)$, where

$$\beta_j = \begin{cases} \text{append}(\underline{p}_i, \text{actionN}_i, \text{dateV}) \text{ to } \alpha_j & \text{if para}(\text{actionN}_i) \cap f_j \neq \varnothing, \\ \alpha_j & \text{otherwise,} \end{cases}$$

and $\underline{p}_i$ is the sublist of $\underline{p}$ corresponding to para$(\text{actionN}_i) \cap f_j$ and dateV is the date when the transaction event takes place.

Proposition 14.5. *There are two functions $k : U^* \to \underline{C}$ and $h : k(U^*) \to U^*$ such that the following diagram is commutative:*

$$\begin{array}{ccccc} k(U^*) & \times U & \xrightarrow{\delta_{\text{rep}}} & k(U^*) \\ h \downarrow & \downarrow I & & \downarrow h \\ U^* & \times U & \xrightarrow{\text{append}} & U^* \\ [] \downarrow & \downarrow I & & \downarrow [] \\ U^*/\sim_{c0} & \times U & \xrightarrow{\delta_{\text{can}}} & U^*/\sim_{c0} \end{array}$$

Furthermore, h is a surjection.

Proof. Let us prove the above fact using an example. Let

ActionName $= \{a1, a2\}$,

para(a1) = {attr1, attr2, attr3} and para(a2) = {attr2, attr3, attr4},

$f1$ = [attr1, attr2, actionName, date], f2 = [attr2, attr3, actionName, date],

$f3$ = [attr3, attr4, actionName, date].

Suppose we have the following transactions:

$$\begin{aligned} \text{On date}V1\ u1 &= (a1, [p11, p12, p13]), \text{ where} \\ &\quad [p11, p12, p13] \in \text{realization}(\text{para}(a1)), \\ \text{On date}V2\ u2 &= (a2, [p21, p22, p23]), \text{ where} \\ &\quad [p21, p22, p23] \in \text{realization}(\text{para}(a2)), \\ \text{On date}V3\ u3 &= (a1, [p31, p32, p33]), \text{ where} \\ &\quad [p31, p32, p33] \in \text{realization}(\text{para}(a1)). \end{aligned}$$

It is assumed that date$V1$ < date$V2$ < date$V3$.

Then, $(a1, \underline{p1})(a2, \underline{p2})(a1, \underline{p3}) \in U^*$, where $\underline{p1} = [p11, p12, p13], \underline{p2} = [p21, p22, p23], \underline{p3} = [p31, p32, p33]$. Let $k : U^* \rightarrow \underline{C}$ be as follows: $k((a1, \underline{p1})(a2, \underline{p2})(a1, \underline{p3})) = (\alpha 1, \alpha 2, \alpha 3) \leftrightarrow \alpha 1 = d11 \cdot d12 \cdot d13, \alpha 2 = d21 \cdot d22 \cdot d23$ and $\alpha 3 = d31 \cdot d32 \cdot d33$, where

$$d11 = (p11, p12, a1, \text{date}V1), d12 = (*, p21, a2, \text{date}V2),$$
$$d13 = (p31, p32, a1, \text{date}V3),$$
$$d21 = (p12, p13, a1, \text{date}V1), d22 = (p21, p22, a2, \text{date}V2),$$
$$d23 = (p32, p33, a1, \text{date}V3),$$
$$d31 = (p13, *, a1, \text{date}V1), d32 = (p22, p23, a2, \text{date}V2),$$
$$d33 = (p33, *, a1, \text{date}V3).$$

Conversely, let $h : k(U^*) \rightarrow U^*$ be defined as follows. Let

$$\alpha = \alpha 1 \cup \alpha 2 \cup \alpha 3 = \{d11, d12, d13, d21, d22, d23, d31, d32, d33\}.$$

Let $\approx \subset \alpha \times \alpha$ be given by

$$d \approx d' \leftrightarrow \text{date}(d) = \text{date}(d').$$

Then $\approx$ is an equivalence relation. Let

$$[d11] = \{d11, d21, d31\}, \text{ where date}(d11) = \text{date}V1,$$
$$[d12] = \{d12, d22, d32\}, \text{ where date}(d12) = \text{date}V2,$$
$$[d13] = \{d13, d23, d33\}, \text{ where date}(d13) = \text{date}V3.$$

It should be noted that $d \approx d' \rightarrow \text{actionName}(d) = \text{actionName}(d')$ and $\text{actionName}(d11) = a1, \text{actionName}(d12) = a2$ and $\text{actionName}(d13) = a1$. Let

$$\cup[d11] = [p11, p12] \cup [p12, p13] \cup [p13, *] = [p11, p12, p13] = \underline{p1},$$
$$\cup[d12] = [*, p21] \cup [p21, p22] \cup [p22, p23] = [p21, p22, p23] = \underline{p2},$$
$$\cup[d13] = [p31, p32] \cup [p32, p33] \cup [p33, *] = [p31, p32, p33] = \underline{p3}.$$

Then, $(\alpha 1, \alpha 2, \alpha 3)$ indicates the following events.

On $\text{date}(d11) = \text{date}V1$ an action $\text{actionName}(d11) = a1$ is selected with the parameter $\underline{p1}$.

On $\text{date}(d12) = \text{date}V2$ an action $\text{actionName}(d12) = a2$ is selected with the parameter $\underline{p2}$.

On $\text{date}(d13) = \text{date}V1$ an action $\text{actionName}(d13) = a1$ is selected with the parameter $\underline{p3}$.

Therefore, $h(\alpha 1, \alpha 2, \alpha 3) = (\text{actionName}(d11), \underline{p1})(\text{actionName}(d12), \underline{p2})(\text{actionName}(d13), \underline{p3})$. The sequence order is determined by the order of $\text{date}(d11) < \text{date}(d12) < \text{date}(d13)$.

Since $h(\alpha 1, \alpha 2) = (\text{actionName}(d11), \underline{p1})(\text{actionName}(d12), \underline{p2})$, the above diagram is commutative. Q.E.D.

Let

$$\lambda_{\text{rep}}(\underline{c}, u) = \lambda_{\text{can}}([h(\underline{c})], u).$$

Corollary 14.3.

$$\langle U, V, k(U^*), \delta_{\text{rep}}, \lambda_{\text{rep}}, [\Lambda]\rangle \textit{ is a realization of the BMIS.}$$

The above result indicates the following.

Corollary 14.4. *A simple workable file system is given by*

$$f_j = \text{para(actionNj)} \cup \{\text{“actionName”}, \text{“date”}\},$$

where actionNj $\in$ ActionName.

Proof. It is obvious that $\cup f_j$ = AttrName $\cup$ {“actionName”, “date”} and $f_j \supset$ {“date”, “actionName”} hold. Q.E.D.

It can be assumed without loss of generality that actionN $\neq$ actionN′ $\rightarrow$ para (actionN) $\neq$ para(actionN′). However, for (actionN, $\underline{p}$) $\in U$ and (actionN′, $\underline{p}'$) $\in U$ the formula actionN $\neq$ actionN′ $\rightarrow$ $p \neq p'$ may not hold. If the formula holds, we have a strong result.

Proposition 14.6. *Suppose that for any* (actionN, $\underline{p}$) $\in U$ *and* (actionN′, $\underline{p}'$) $\in U$ *the following holds:*

$$\text{actionN} \neq \text{actionN}' \rightarrow p \neq p'.$$

Let $E \subset U \times U$ *be an equivalence relation given by*

$$((\text{actionN},\underline{p}), (\text{actionN}', \underline{p}')) \in E \leftrightarrow \underline{p} = \underline{p}'.$$

Then,

$$(u, u') \in E \rightarrow u \sim_{c0} u'.$$

Proof. Let u = (actionN, $\underline{p}$) $\in U$ and u' = (actionN′, $\underline{p}'$) $\in U$ and $(u, u') \in E$. Then, $p = p'$ and hence actionN = actionN′ due to the assumption. Consequently, $(\forall z)(\lambda(\delta(c_0, u), z) = \lambda(\delta(c_0, u'), z))$ holds and hence we have $u \sim_{c0} u'$. Q.E.D.

In other words, if actionN $\neq$ actionN′ $\rightarrow$ $p \neq p'$ holds, the conditions $\cup f_j$ = AttrName $\cup$ {“actionName”, “date”}, $f_j \supset$ {“actionName”, “date”} can be relaxed as follows:

(i) $\cup f_j$ = AttrName $\cup$ {“date”},
(ii) $f_j \supset$ {“date”}.

In general, $\langle U, V, k(U^*), \delta_{\text{rep}}, \lambda_{\text{rep}}, [\Lambda]\rangle$ is a redundant realization. For example, if f_j represents a “master file” whose transaction history is not important (“static” file), realization(f_j) can be used instead of realization$(f_j)^*$ in $\underline{C}$ because the last element of a string in realization$(f_j)^*$ is essential for it. Or for a file f_j whose transaction history is important (“dynamic file”), if a proper “state” attribute is introduced for it,

the same convenience can be obtained. For the subsequent discussions let realization of the BMIS on a file system $\{f_j\}_j$ be denoted by $\langle U, V, C, \delta, \lambda, c_0 \rangle$.

Let

$$\delta_\lambda : C \times U \to \text{Res} \times C$$

such that

$$\delta_\lambda(c, u) = (\lambda(c, u), \delta(c, u)).$$

Using this function, another function

$$\boldsymbol{\delta_\lambda}_{\text{actionN}_i} : C \times P_{i1} \times \cdots \times P_{\text{ini}} \to \text{Res} \times C$$

is derived: for each $\text{actionN}_i \in \text{ActionName}$ and $(p_{i1}, \ldots, p_{\text{ini}}) \in \text{Paralist}_i$

$$\delta_\lambda_{\text{actionN}_i}(c, p_{i1}, \ldots, p_{\text{ini}}) = (\lambda(c, u), \delta(c, u)),$$

where $u = (\text{actionN}_i, p_{i1}, \ldots, p_{\text{ini}})$. $\delta_\lambda_{\text{actionN}_i}$ represents the unit operation corresponding to $\text{actionN}_i \in \text{ActionName}$.

Definition 14.13. Atomic process
$\delta_\lambda_{\text{actionN}_i}$ will be called an atomic process, i.e.,

$$i\textbf{th atomic process} = \langle \text{Paralist}_i, \text{Res}, C, \delta_\lambda_{\text{actionN}_i} \rangle.$$

Finally, a realization model of the BMIS is given in terms of atomic processes as follows.

Definition 14.14. Realization model of BMIS
The following is called a realization model of the BMIS:

$$\textbf{BMIS} = \langle \text{internalUI, user model} \rangle,$$

where

$$\textbf{internalUI} : \cup(\{\text{actionN}_i\} \times \text{Paralist}_i) \to \{\text{name of } \delta_\lambda_{\text{actionN}_i}\}_i \times \text{Res}$$

such that

$$\begin{aligned} &\text{internalUI}(\text{actionN}_i, p_{i1}, \ldots, p_{\text{ini}}) = (\text{name, res}) \leftrightarrow \\ &\quad \text{res} := \text{delta_lambda(name)}(p_{i1}, \ldots, p_{\text{ini}}) \end{aligned}$$

and

$$\textbf{user model} = \{i\text{th atomic process}\}_i,$$

where delta_lambda()() was given in Definition 14.4. That is, the family of the atomic processes is called a user model.

Appendix 14.3 External User Interface*

The external UI is an interface between the user and the internal UI and is implemented in PHP. Its total function is summarized as ξ : UIState $\times U_0 \times V \rightarrow U \cup V_0$ in Section 14.1. In fact, the external UI consists of four functions: getSname (get system name), getAname (get action name), getPara (get parameter) and go. The relationships between these functions are shown in Fig. 14.13.

When execution of the external UI is started, the function getSname is performed first, generating a request to the user to input the name of the target TPS (see Fig. 14.11(a)). The next function, getAname, then displays a list of action names, taking data from ActionName.g of the implementation structure, and requests the user to specify an action by its name (see Fig. 14.11(b)). If the selected action actionN$_i$ is equal to "quit," the operation of the TPS is suspended. Otherwise, the next function, getPara, is performed, which displays a parameter name list extracted from para(actionN$_i$) in the implementation structure and requests the user to provide necessary parameter information (see Fig. 14.11(c)). The system name,

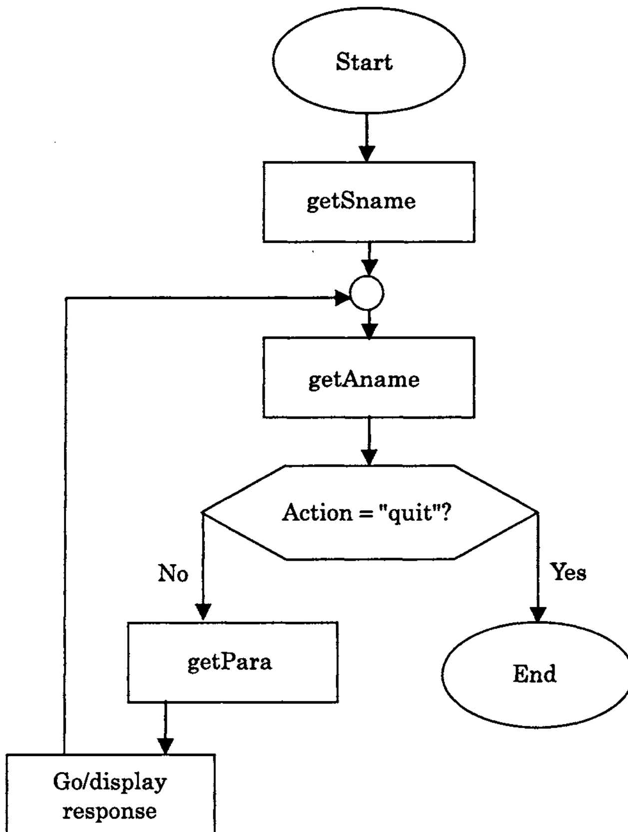


Fig. 14.13. Flow chart for external UI.

action name, and parameter information are saved in special files of the file system, stdUI_model.lib, stdUI_action.lib, and stdUI_para.lib, respectively. Finally, the function go is executed, triggering execution of the target atomic process through the internal UI. The execution result or response produced by the atomic process is saved in another special file stdUI_res.lib. After displaying the response in stdUI_res.lib via the external UI and receiving an acknowledgement from the user (see Fig. 14.11(d)), the function control passes to getAname. The entire program for the external UI is given in Takahara et al. [2003].

The external UI can be defined as an independent component of a user model for the following reasons:

(i) Because the structure of the user model is formalized, the data that must be transferred between the UI and the user model can be represented in a standardized form. Consequently, the UI can access the data in a standardized procedure.
(ii) Because variables of extProlog and PHP are typeless, there is no type conflict among them on model integration.

References

Bridge, J. (1977) *Beginning of Model Theory*, Oxford University Press.

Fitzgerald, J. and Larse, P. G. (1998) *Modeling Systems*, Cambridge University Press.

Koizumi, T., Yoshida, K., and Nakajima, T. (2003) *Software Development*, Ohmusha (in Japanese).

Mesarovic, M. D. and Takahara, Y. (1989) "Abstract Systems Theory," *Lecture Notes in Control and Information Sciences*, Springer.

Takahara, Y. et al. (2003) *Manual for extProlog*, Dept. of Management Information Science, Chiba Institute of Technology, Chiba, Japan.

Takahara, Y. and Mesarovic, M. D. (2003) *Organization Structure: Cybernetic Systems Foundation*, Kluwer Academic Publishers.

Wymore, A. W. (1993) *Model-Based Systems Engineering*, CRC Press.

Yano, Y. (2005) "External UI for the model theory approach," *Internal report*, Dept. of Management Information Science, Chiba Institute of Technology, Chiba, Japan.

15

Browser-Based Intelligent Management Information System: Temporary Staff Recruitment System

This chapter presents an example of systems development involving a combination of the methodologies for developing a transaction processing system and a solver system treated separately in previous chapters. The user model includes a solver-type atomic process as well as transaction-type processes. The target system is a temporary staff recruitment system (employment system) that must perform optimum matching between job-seeking applicants and job-offering clients as a typical problem-solving activity. The development leads to a browser-based intelligent management information system (MIS), which is a modification of Fig. 1.2 in Chapter 1.

15.1 Systems Specification

The informal specification of the target system is given below [Avgerov and Cornford, 1998].

South East Employment is a recruiting agency that specializes in industrial and secretarial recruitment. The company operates in a largely decentralized manner through branch offices. Operating procedures in the company are largely standardized, and each branch has a similar structure, although the branch managers are given adequate authority to run their own branches. A common type of data is reported from each branch, and accounting is performed centrally at the head office.

Branch managers are responsible for the operation of their own branches with the assistance of three to eight consultants, one clerical assistant, and one location administrator. The consultants deal with client companies ("clients"), and applicants for employment ("applicants") endeavor to meet their requirements and match personnel to job vacancies. There are two types of consultants: recruitment consultants, dealing with the recruitment of permanent personnel; and temporary controllers, dealing with the placement of temporary personnel.

The agency has decided to take action to improve the way it handles the task of temporary personnel placement. In the existing procedure, the temporary personnel agent looks through the client file, collects relevant information, and telephones clients to canvass for business every morning. If a client indicates that temporary staff are required, the controller looks through the file containing the details of all applicants and

selects those available with the skills required by the client. The agent then telephones the applicants and asks whether they are interested in working for the day, week, or entire length of the booking, and arranges a placement if an agreement is reached. The agent also compiles reports on temporary staff based on feedback received from clients.

At the end of the week, the temporary personnel return their time sheets to the branch office (signed by the client company) and collect another time sheet for the coming week of their booking along with remuneration for the previous week's work. All the submitted time sheets are dispatched on Saturday mornings by courier to the accounts department at the head office, and the preparation of wages begins the following Monday. In this way, temporary personnel are paid on Fridays, one week in arrears.

There is no formal method of documenting the temporary personnel's performance in the various placements made by the agency.

On Fridays, a large number of temporary personnel come to the office at the same time to collect wages and time sheets, putting the agent under pressure and causing the office to fall into confusion. Although other staff attempt to provide assistance, their involvement is very limited because the temporary personnel agent is the only staff member with full knowledge of the required arrangements.

The analyst providing information for development of the MIS has identified three main units of operation related to the business of temporary personnel placement:

(i) Registration and maintenance of applicant files.
(ii) Canvassing of clients, negotiation with applicants, and matching of applicants to jobs.
(iii) Administration of payments.

Three recommendations are proposed by the analyst:

(i) All members of staff should be able to perform all tasks involved in the three units of operation.
(ii) All members of staff should be able to (1) deal with the registration of new applicants according to the availability of time, (2) update applicant files with performance information for the various placements arranged, and (3) deal with payment claims.
(iii) A new computer-based information system should be provided to facilitate this work structure, providing a common formal information basis and appropriate structures for information utilization and maintenance.

In parallel with the design of a computer-based information system, the analyst will design new organizational procedures.

15.2 DFD for Specification and Browser-Based Intelligent MIS

Figure 15.1 shows the data flow diagram (DFD) for the employment system in accordance with the analysis above.

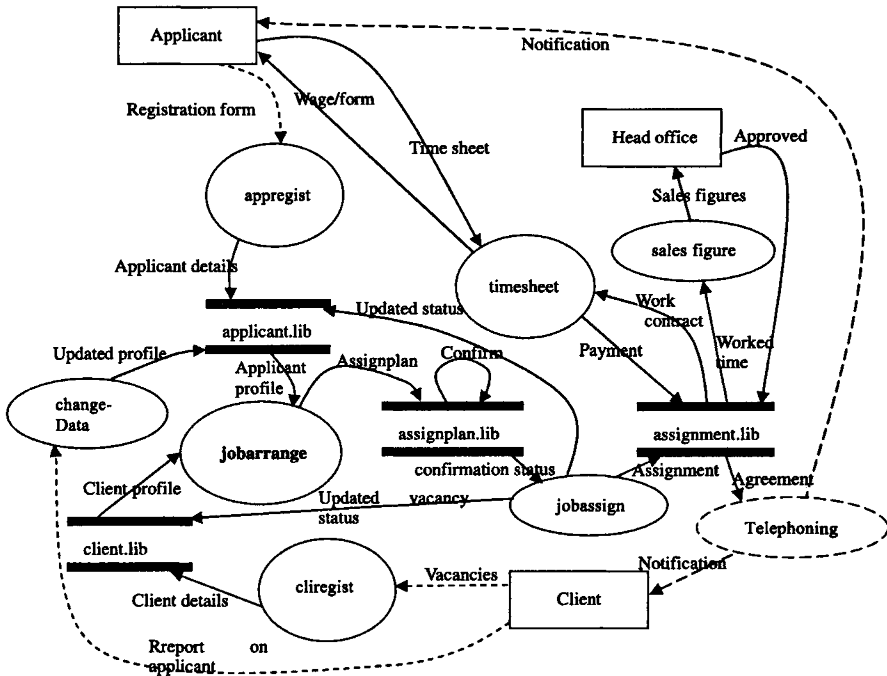


Fig. 15.1. DFD for temporary staff recruitment system.

Four files are used for the system: applicant.lib, client.lib, assignplan.lib, and assignment.lib. These files store information on applicants, clients, placement plans, and real placement, respectively.

According to the DFD the required organizational procedure is designed as follows: when an applicant registers, the applicant's details are inserted into applicant.lib by the action "appregist." Similarly, client details are registered into client.lib by "cliregist." The function "jobarrange" is then performed using the profiles of applicants and clients. The matching results are stored in assignplan.lib, and telephoned to relevant applicants and clients seeking agreements. Responses are recorded in assignplan.lib by the functions "confirmApp" and "confirmCli." When the matching is accepted by both sides, the matchingplan is recorded as a real placement in assignment.lib by "jobassign," applicant.lib and client.lib are updated accordingly, and the placement results are telephoned to the relevant parties. When an applicant comes to the offices on Friday with a time sheet, the action "timesheet" is performed, which checks past wages, calculates new payment, and provides a new time sheet. The action "salesfigure" sends sales figures to the accounts department at the head office. According to a feedback report from a client, the profile of an applicant is modified by "changeData."

Since the action "jobarrange" is a typical problem-solving activity, it is implemented as a solver-type atomic process. The total process of the DFD is therefore implemented in a browser-based intelligent MIS consisting of three layers: a

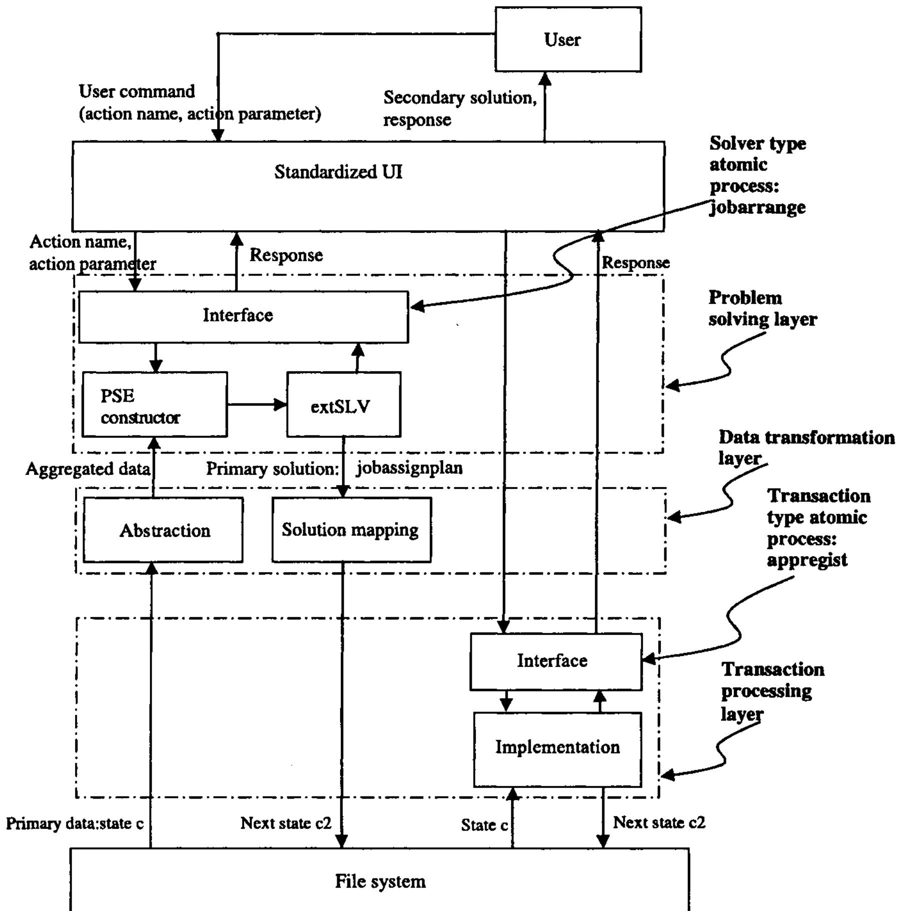


Fig. 15.2. Browser-based intelligent MIS.

transaction-processing layer, a data-transformation layer and a problem-solving layer. Figure 15.2 shows the intelligent MIS scheme that is a modification of Fig. 14.5.

The construction of each of the components of Fig. 15.2 is detailed in the sections below.

15.3 TPS Development for Employment System

15.3.1 Input–Output Specification in Set Theory

From the DFD in Fig. 15.1, which outlines the basic behavior of the system, the following input–output specification can be derived directly for the temporary staff recruitment system.

ActionName set:

ActionName = {"appregist", "cliregist", "jobarrange", "appStatus",
"cliStatus", "assignplanStatus", "confirmApp",
"confirmCli", "jobassign", "assignStatus",
"timesheet", "salesfigure", "changeData"},

where

appregist: registration of applicant,
appStatus: display of applicant.lib,
cliregist: registration of client,
cliStatus: display of client.lib,
jobarrange: matching between applicant and client,
assignplanStatus: display of assignplan.lib,
assignconfirmApp: confirmation of applicant assignment to assignplan,
assignconfirmCli: confirmation of client assignment to assignplan,
jobassign: assignment of job,
assignStatus: display of assignment.lib,
timesheet: handling of time sheet and payment,
salesfigure: reporting of sales status to head office,
changeData: changing data in files.

ResName:

ResName = {"appregistRes", "appStatusRes", "cliregistRes",
"cliStatusRes", "jobarrangeRes", "assignplanStatusRes",
"assignconfirmAppRes", "assignconfirmCliRes",
"jobassignRes", "timesheetRes", "salesfigureRes",
"changeDataRes"}.

15.3.2 Design of File Structure System $\{f_j\}_j$ and Para Function, and Realization of Names

According to the DFD in Fig. 15.1, four files are specified for this system. The detailed structures of them are determined by the function para: ActionName $\rightarrow \wp$(AttrName) with some heuristic considerations. The function para is defined as follows.

Para function:

para("cliStatus") = {"cName", "cAddress"},
para("appStatus") = {"aName", "aAddress"},
para("jobassign") = $\varnothing$,
para("assignconfirmCli") = {"cName", "cAddress", "aName", "aAddress",
"cAccept"},

para("assignconfirmApp") = {"cName", "cAddress", "aAccept"},
para("assignplanStatus") = ∅,
para("jobarrange") = ∅,
para("salesfigures") = ∅,
para("timesheet") = {"aName", "aAddress", "cName", "hours"},
para("cliregist") = {"cName", "cAddress", "cJtype", "cLevel", "cWtime", "cPlevel", "number"},
para("appregist") = {"aName", "aAddress", "aJtype", "aLevel", "aWtime", "aPlevel"},
para("changeData") = {"fNname"},
para("assignStatus") = {"aName", "aAddress", "cName", "cAddress"}.

AttrName:

AttrName = {"aId", "aName", "aAddress", "aJtype", "aLevel", "aWtime", "aPlevel", "cId", "date", "canme", "cAddress", "cJtype", "Level", "cWtime", "cPlevel", "num", "aAccept", "cAccept", "pay", "fName"}.

Consequently, let the file structure system $\{f_j\}_j$ be as follows.

File structure system $\{f_j\}_j$:

applicant.lib = {"aId", "aName", "aAddress", "aJtype", "aLevel", "aWtime", "aPlevel", "cId", "hours", "date"},
client.lib = {"cId", "cName", "cAddress", "cJtype", "cLevel", "cWtime", "cPlevel", "num", "date"},
assignplan.lib = {"aId", "cId", "aAccept", "cAccept", "date"},
assignment.lib = {"aId", "cId", "cPlevel", "paymenthis"},

where

aId: applicant ID,
aName: applicant name,
aAddress: applicant address,
aJtype: applicant job type,
aLevel: applicant skill level,
aWtime: applicant workable time,
aPlevel: applicant desired payment level,
cId: client ID,
hours: worked hours,
date: date of event,

cName: client name,
cAddress: client address,
cJtype: job type offered by client,
cLevel: skill level requested by client,
cWtime: client available working time,
cPlevel: payment level offered by client,
num: number of available positions,
aAccept: acceptance by applicant,
cAccept: acceptance by client,
paymenthis: payment history,
fName: file name.
The set-theoretic realization of attribute names is naturally defined as follows:

Realization of attribute names

AId: set of applicant IDs (= Int),
AName: set of applicant names (= set of strings),
AAddress: set of applicant addresses (= set of strings),
AJtype: set of applicant job types (= {"$j1$", "$j2$", "$j3$", "$j4$"}),
ALevel: set of applicant skill levels (= {"low", "mid", "high"}),
AWtime: set of applicant-workable times (= {"mor", "eve", "all"}),
APlevel: set of applicant-desired payment levels (= {"high", "mid", "low"}),
CId: set of client IDs (= Int),
Hours: set of worked hours (= Int),
Date: set of event dates (= $\{(Y, M, D)\}$),
CName: set of client names (= set of strings),
CAddress: set of client addresses (= set of strings),
CJtype: set of client-desired job types (= {"$j1$", "$j2$", "$j3$", "$j4$"}),
CLevel: set of client-requested skill levels (= {"high", "mid", "low"}),
CWtime: set of client-available working times (= {"mor", "eve", "all"}),
CPlevel: set of client-offered payment levels = {"high", "mid", "low"}),
Num: set of numbers of available positions (= Int),
AAccept: set of acceptance replies by applicant (= {"yes", "no"}),
CAccept: set of acceptance replies by client (= {"yes", "no"}),
Paymenthis: set of payment history (= (Pay × Date)*)
FName: set of file names (= set of strings).
(Pay × Date)* is the free monoid of (Pay × Date).

Using these realizations of attributes, the file set structure system implies the file system implementation as follows.

Implementation of file system

$$\begin{aligned}\text{Applicant.lib} = {} & \text{AId} \times \text{AName} \times \text{AAddress} \times \text{AJtype} \times \text{ALevel} \\ & \times \text{AWtimeA} \times \text{APlevel} \times \text{CId} \times \text{Hours} \times \text{Date},\end{aligned}$$

$$\begin{aligned}
\text{Client.lib} &= \text{CId} \times \text{CName} \times \text{CAddress} \times \text{CJtype} \times \text{CLevel} \\
&\quad \times \text{CWtime} \times \text{CPlevel} \times \text{Num} \times \text{Date}, \\
\text{Assignplan.lib} &= \text{AId} \times \text{CId} \times \text{AAccept} \times \text{CAccept} \times \text{Date}, \\
\text{Assignment.lib} &= \text{AId} \times \text{CId} \times \text{CPlevel} \times (\text{Pay} \times \text{Date})^{*}.
\end{aligned}$$

Realization of ResNames:

$$\text{AppregistRes} = \text{set of messages},$$

• • • •

15.3.3 Design of Atomic Process: {delta_lambda(actionN$_i$), actionN$_i$}$_i$

Atomic processes are defined on the above definitions of state (file system). This section discusses only two atomic processes: appregist and jobarrange. They have transaction-type and solver-type implementation components, respectively. The other atomic processes have transaction-type implementation components, which are presented in employment4.set in Appendix 15.1.

(1) appregist

This process has a typical transaction-type implementation component, whose representation is given by the formal structure described in Section 14.5:

```
delta_lambda ([_appregist],paralist)=res <->
        res := appregist(paralist);

appregist ([name,add,type,level,wtime,pay])=res <->
        [res,applicant.lib] := φ(applicant.lib, [name,add,type,level,
                                wtime,pay]);
```

where for Date = getDate2() and ID = getAId(),

ϕ(applicant.lib, [name, add, type, level, wtime, pay])

$$= \begin{cases} [\text{“appregist is completed”}, \text{applicant.lib}\cdot & \text{if } \neg(\exists\, \text{AId}, P, D)([\text{AId}, \text{name}, \\ \quad [\text{ID}, \text{name}, \text{add}, \text{type}, \text{level}, & \quad \text{add}, \text{type}, \text{level}, \text{wtime}, \text{pay}, \\ \quad \text{wtime}, \text{pay}, -1, \text{Date}]] & \quad P, D] \in \text{Applicant.lib}), \\ [\text{“already registered”}, \text{applicant.lib}] & \text{otherwise}. \end{cases}$$

The meanings of ϕ should be clear. For example, if [AId, name, add, type, level, wtime, pay, P, D] is not in Applicant.lib for any AId, P, and D, Applicant.lib is

extended by appending a new element [ID, name, add, type, level, wtime, pay, −1, Date]. Here ID is the identification number of the applicant and Date is the date when this operation is performed. The Cid part of the new element is −1 because the applicant has yet to be assigned to a client.

(2) jobarrange

The atomic process "jobarrange" has a solver-type implementation component. It is implemented by the following form (see Fig. 14.7):

```
delta_lambda ([_jobarrange])=res <->
        res:=jobarrange();

jobarrange()=res <->
        load_go("jobarrange42.p"),
        res:="job assignment is completed";
```

Here, load_go() is a system-defined predicate that loads and executes an extProlog program. The real implementation component of the current atomic process is separately prepared as "jobarrange42.set," which is shown in Appendix 15.2. Section 15.5 discusses the model construction. Its compiled model is "jobarrange42.p," which is called and executed by load_go.

15.3.4 Implementation Description of User Model in Computer-Acceptable Set Theory

The computer-acceptable atomic process is induced from the design mentioned in Section 15.3.3. The final implementation form is produced by adding three statements—the predicate "func," the set declaration "ActionName," and the function "para"—to the atomic process representations. The entire description is given in Appendix 15.1.

15.4 Data Transformation System

The solver for the employment system requires modified data from the file system. The modification is performed by the data transformation system shown in Fig. 15.3.

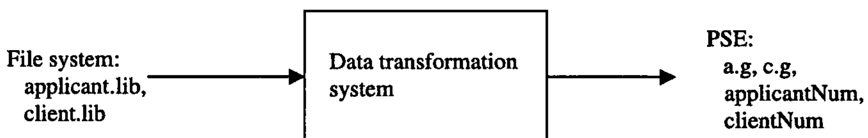


Fig. 15.3. Data transformation system.

Two modifications are performed on the data:

(i) Quantification of job type, skill level, and working time. Quantification is realized by three functions: transJType, transLevel, and transWTime, respectively, which are given below.
(ii) Abstraction of Applicant.lib and Client.lib. Abstracted objects are represented by a.g and c.g as defined below.

- **Quantification**

transJType:AJtype $\rightarrow JT(= \wp(\{1, 2, 3, 4\}))$ such that

$$\begin{aligned}\text{transJType}(j1) &= [1, 2, 3, 4],\\ \text{transJType}(j2) &= [2, 3, 4],\\ \text{transJType}(j3) &= [3, 4],\\ \text{transJType}(j4) &= [4],\end{aligned}$$

where 1, 2, 3, and 4 represent detailed job types.

transLevel:ALevel $\rightarrow L(= \{1, 2, 3\})$ such that

$$\begin{aligned}\text{transLevel}(\text{“low”}) &= 1,\\ \text{transLevel}(\text{“mid”}) &= 2,\\ \text{transLevel}(\text{“high”}) &= 3.\end{aligned}$$

transWTime:AWtime $\rightarrow WT(= \{[\text{“eve”}], [\text{“mor”}], [\text{“eve”}, \text{“mor”}]\})$ such that

$$\begin{aligned}\text{transWTime}(\text{“eve”}) &= [\text{“eve”}],\\ \text{transWTime}(\text{“mor”}) &= [\text{“mor”}],\\ \text{transWTime}(\text{“all”}) &= [\text{“eve”}, \text{“mor”}].\end{aligned}$$

- **Abstraction**

Let

$$\text{a.g} \subset \text{Aid} \times JT \times L \times WT \times PL$$

such that

$$\begin{aligned}&(\text{aid}, jt, l, wt, pl) \in \text{a.g} \leftrightarrow (\exists\ \text{name, add, jtype, level, wtime, plevel, cid, date})\\ &\quad((\text{aid, name, add, jtype, level, wtime, plevel, cid, date}) \in \text{Applicant.lib}\ \&\\ &\quad jt = \text{transJType(jtype)}\ \&\ l = \text{transLevel(level)}\ \&\\ &\quad wt = \text{transWT(wtime)}\ \&\ pl = \text{transLevel(plevel)}).\end{aligned}$$

Let

$$\text{c.g} \subset \text{Cid} \times JT \times L \times WT \times PL \times \text{Num}$$

such that

$$(\text{cid}, jt, l, wt, pl, \text{num}) \in \text{c.g} \leftrightarrow (\exists\ \text{name, add, jtype, level, wtime, plevel, date})$$
$$((\text{cid, name, add, jtype, level, wtime, plevel, num, date}) \in \text{Client.lib}\ \&$$
$$jt = \text{transJType(jtype)}\ \&\ l = \text{transLevel(level)}\ \&$$
$$wt = \text{transWT(wtime)}\ \&\ pl = \text{transLevel(plevel)}).$$

Two parameters, applicantNum and clientNum, are also afforded by the data transformation system. The parameters applicantNum and the clientNum represent the number of applicants and the list of available positions of clients, respectively. Both are defined formally as follows:

$$\text{applicantNum} = |\text{a.g}|,$$
$$\text{clientNum} = [\text{num} | (\exists\ \text{cid}, jt, l, wt, pl)$$
$$((\text{cid}, jt, l, wt, pl, \text{num}) \in \text{c.g})].$$

The data-transformation system is embedded in the preprocess of the solver system, jobarrange. The entire model in set theory is listed as the predicate data_trans() in Appendix 15.2.

15.5 Solver Development: Jobarrange

15.5.1 Job Arrangement Problem

This section discusses the job arrangement problem as an I-O-C problem (see Section 4.3). It is clear that the job arrangement problem belongs to the implicit solving activity problem class. An action is not specified a priori. Furthermore, it is also clear that the problem does not have an explicit evaluation function of the output, assignplan. However, because the final state can be given as a situation in which no further assignment can be found under the given constraints, the job arrangement problem is handled as a closed-target problem. This is simplification of the optimization problem. If real optimization must be performed, the stopping condition used for the knapsack problem can be used for the current problem (see Chapter 10). The stdPDsolver will be used for the present system as the goal-seeker.

15.5.2 User Model of Jobarrange Problem for PD Goal-Seeker

The user model for the job arrangement problem is constructed following Fig. 4.6.

a. Drawing Input–Output Block Diagram

At this stage, the target solver is drawn as an input–output system with the problem specification environment (PSE) as the input and the solution as the output. Figure 15.4 illustrates the input–output of the solver for the job arrangement problem.

The job arrangement problem is specified by the two lists, a.g and c.g, and by two parameters, applicantNum and clientNum, given as the outputs of the data transformation system.

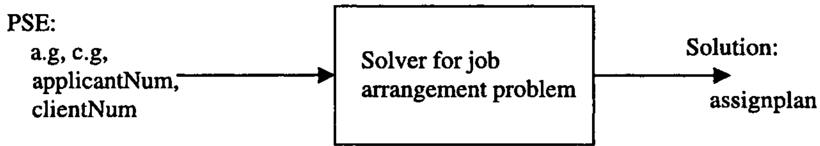


Fig. 15.4. Input–output block diagram of job arrangement solver.

b. Input–Output Specification in Set Theory

The input and output given in Section 15.5.2.a are described in set-theoretic terms as follows.

(i) Input Specification

The PSE for a solver of the job arrangement problem is simply given by

$$\text{PSE} = \langle \text{a.g, c.g, applicantNum, clientNum} \rangle.$$

(ii) Output Specification

The output set Y is a class of assignment plans, which is given by

$$Y = (\text{Aid} \times \text{Cid} \times \{-1\} \times \{-1\})^*.$$

That is, if $(\text{aid}_1, \text{cid}_1, -1, -1) \ldots\ldots (\text{aid}_t, \text{cid}_t, -1, -1) \in Y$, an applicant aid_t is assigned to a client cid_t. The $(-1, -1)$ component of the element indicates that responses for acceptance have yet to be given by either the applicant or client.

c. Process Specification as Automaton

The solution-finding activity is formalized as an automaton as follows.

(i) Action Set

Because the problem belongs to the implicit solving activity class, the action set A is in general formalized in the following way:

$$A = \{a | a : Y \to Y\}.$$

In the above formulation, an action $a \in A$ is represented as a mapping. However, because a set of mappings is too general to be processed effectively, it is usual for the implicit solving case for a mapping to be replaced by a parameter that characterizes the mapping. For the job arrangement problem, the following set is used as the set of parameters for actions:

$$A = \text{Aid} \times \text{Cid}.$$

For example, an element

$$(\text{aid, cid}) \in A$$

implies an action to assign an applicant aid to a client cid.

(ii) State Transition Function

The output set Y can be used as the state set following the general formulation. However, for the sake of computational convenience of a constraint and a goal, two auxiliary elements are appended to the output. The following form is used for the state set: let $G = \text{Re}$ (set of real numbers) and $Vm = (\text{Int})^{\text{CNUM}}$, where $\text{CNUM} = |\text{Client.lib}|$, and cliOrder:Cid $\to$ Int such that $\text{cliOrder(cid)} = i \leftrightarrow$ cid is the identification of the ith element of Client.lib. Note that Client.lib is treated as a list as well as a set. Then,

$$C = (\text{Aid} \times \text{Cid})^* \times G \times Vm.$$

The state transition function $\delta : C \times A \to C$ is then defined as

$$\begin{aligned}&\delta(((\text{aid}_1, \text{cid}_1)\cdots(\text{aid}_t, \text{cid}_t), g_t, vm_t), (\text{aid}, \text{cid}))\\ &\quad = ((\text{aid}_1, \text{cid}_1)\cdots(\text{aid}_t, \text{cid}_t)\cdot(\text{aid}, \text{cid}), g_{t+1}, vm_{t+1}),\end{aligned}$$

where

$$g_{t+1} = g_t + \text{unsatisfaction(aid, cid)},$$

$$vm_{t+1}(i) = \begin{cases} vm_t(i) + 1 & \text{if cliOrder(cid)} = i,\\ vm_t(i) & \text{otherwise.}\end{cases}$$

The function unsatisfaction is provided in the definition of the goal below. A vector $vm \in Vm$ indicates how many applicants have been assigned to each client on planning. As usual, the state transition function δ is a partial function, restricted by two functions genA : $C \to \wp(A)$ and constraint : $C \to \{\text{true, false}\}$, i.e.,

$$\delta(c, a) = c' \to a \in \text{genA}(c) \ \text{ and } \ \text{constraint}(\delta(c, a)) = \text{true}.$$

These functions are given below.

(iii) Output Function

The output function $\lambda : C \times A \to Y$ is given by

$$\begin{aligned}&\lambda((\text{aid}_1, \text{cid}_1)\cdots(\text{aid}_t, \text{cid}_t), g_t, vm_t, a)\\ &\quad = (\text{aid}_1, \text{cid}_1, -1, -1)\cdots(\text{aid}_t, \text{cid}_t, -1, -1).\end{aligned}$$

(iv) genA: $C \to \wp(A)$

Let genA be defined as

$$\begin{aligned}&\text{genA}((\text{aid}_1, \text{cid}_1)\cdots(\text{aid}_t, \text{cid}_t), g_t, vm_t)\\ &\quad = \begin{cases}\{(\text{aid}_{t+1}, \text{cid})|\text{cid} \in \text{Cid}\} & \text{if } t < \text{applicantNum},\\ [] & \text{otherwise,}\end{cases}\end{aligned}$$

where Applicant.lib is treated as a list (aid_{t+1} is the next element of aid_t).

(v) constraint: $C \to$ {true, false}

Let the constraint be defined as follows:

$$\text{constraint}((\text{aid}_1, \text{cid}_1) \cdots (\text{aid}_t, \text{cid}_t), g_t, vm_t) = \text{true} \leftrightarrow (\exists\, aN, aAd, aJt, aL, aWt, aPl, cI, aD)((\text{aid}_t, aN, aAd, aJt, aL, aWt, aPl, cI, aD) \in \text{Applicant.lib})\&(\exists\, cN, cAd, cJt, cL, cWt, cPl, \text{num}, \text{cD})((\text{cid}_t, cN, cAd, cJt, cL, cWt, cPl, \text{num}, \text{cD}) \in \text{Client.lib})\&(\text{clientNum} \geq vm_t\ \&\ aJt \cap cJt \neq \varnothing\ \&aWt \cap cWt \neq \varnothing).$$

The condition clientNum $\geq vm_t$ ensures that an applicant cannot be assigned to a client over the number of the available positions. Since the jobytpe (aJt and cJt) represents the coverage of jobtype, $aJt \cap cJt \neq \varnothing$ indicates that an applicant can meet a jobtype required by a client. A similar requirement for working time implies the condition $aWt \cap cWt \neq \varnothing$.

(vi) Initial State

The initial state $c_0 \in C$ is given by

$$c_0 = (\text{nil}, 0, [0, \ldots, 0]).$$

At the initial state, the value of the goal is 0, and no applicants are assigned to clients.

(vii) Final State

The set of final states $C_f \subset C$ is given by

$$((\text{aid}_1, \text{cid}_1) \cdots (\text{aid}_t, \text{cid}_t), g, vm) \in C_f \leftrightarrow$$
$$t = \text{number of applicants or no assignment is possible due to conflicts.}$$

(viii) Stopping Condition

Let the stopping condition be defined by

$$\text{st}(c) \leftrightarrow c \in C_f.$$

d. Goal for PD Solver

(ix) Goal

The following evaluation function must be defined:

$$\text{goal} : C \to \text{Re}.$$

The PSE of the current problem does not provide a goal. When a goal is not given by the problem, an appropriate goal should be designed by a heuristic consideration. For the current problem, the following goal : $C \to$ Re is used:

$$\text{goal}((\text{aid}_1, \text{cid}_1) \cdots (\text{aid}_t, \text{cid}_t), g, vm) = \text{unsatisfaction}((\text{aid}_1, \text{cid}_1)) + \cdots + \text{unsatisfaction}((\text{aid}_t, \text{cid}_t)) = g,$$

where

$$\text{unsatisfaction}((\text{aid}, \text{cid})) = (aPl - cPl) + |cL - aL| - |aJt \cap cJt|$$

for

$$(\text{aid}, aN, aAd, aJt, aL, aWt, aPl, cl, aD) \in \text{Applicant.lib}$$

and

$$(\text{cid}, cN, cAd, cJt, cL, cWt, cPl, \text{num}, \text{cD}) \in \text{Client.lib}.$$

If $aPl > cPl$, the applicant cannot be satisfied. If $cL \neq aL$, the applicant or the client cannot be satisfied. It is assumed that the similarity between an applicant and a client can be measured by $|aJt \cap cJt|$ and that the more similar their relevant jobtypes are, the bigger is the satisfaction obtained. The goal is determined as accumulation of unsatisfaction. Thus, the user model for the job arrangement problem is as follows:

$$\text{user model} = \langle A, C, Y, \delta, \text{genA}, \text{constraint}, \text{st}, \text{goal}, c_0, C_f \rangle.$$

15.5.3 Implementation Structure of User Model of Solver

Appendix 15.2 shows the implementation structure of the user model for the solver.

15.6 Operation of Employment System

Figure 15.5 illustrates operation of the generated system. Figure 15.5(a) shows the start screen, which is displayed when the external UI, stdUI43.php, is executed.

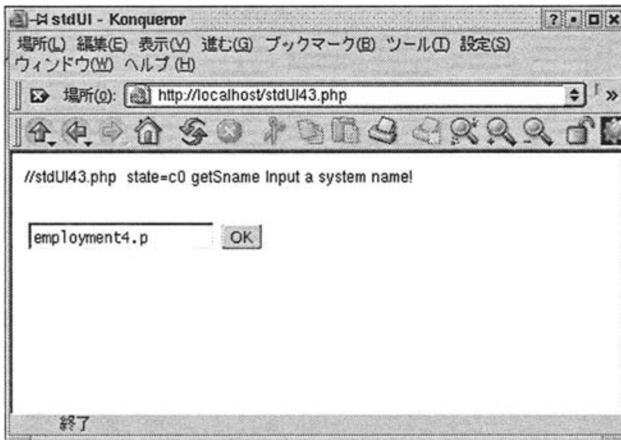


Fig. 15.5(a). Start screen.

The dialog asks the user to input a model name. In the case of Fig. 15.5(a), the input is "employment4.p." After "OK" has been clicked, the action names, given in ActionName.g of the user model, are displayed in Fig. 15.5(b).

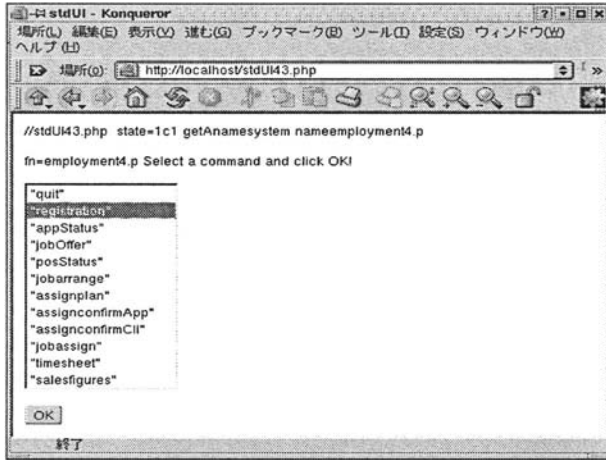


Fig. 15.5(b). Display of action names.

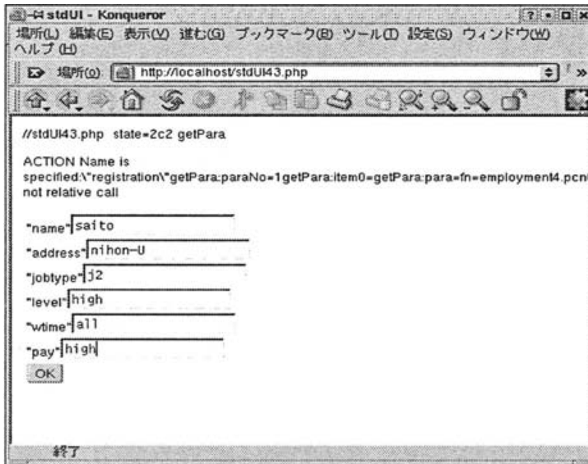


Fig. 15.5(c). Parameter information for an action.

The figure shows the action "appregist" selected. After "OK" has been clicked, the selection is confirmed and execution starts.

As mentioned in Section 14.2, every action actionN_i is associated with parameter information, para(actionN_i). Figure 15.5(c) shows how the parameter information for "appregist" is collected from the user.

Fig. 15.5(d). Selection of action "jobarrange."

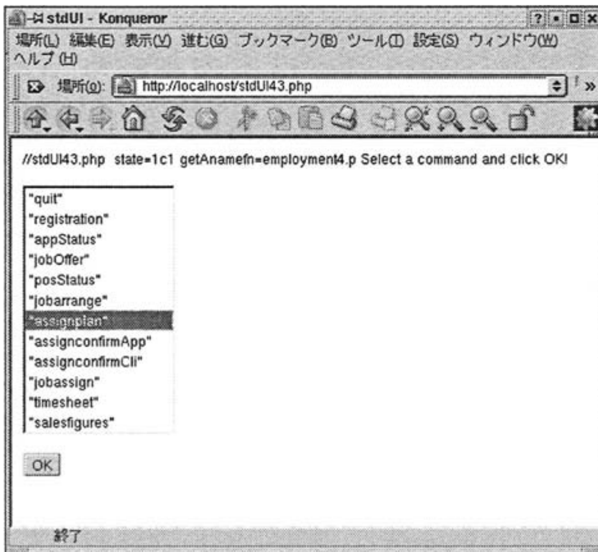


Fig. 15.5(e). Selection of action "assignplanStatus."

The action "appregist" requires information on name, address, jobtype, (skill) level, wtime (working time), and desired pay level. In the figure, the inputs are "saito," "nihon_U," "j2," "high," "all," and "high." When "OK" is clicked, the information is saved in a special file stdUI_para.lib in the file system, which is used by the atomic process "appregist" to execute the action "appregist." After the action has been performed by the process, the system returns to the waiting state ready for the next action input.

Figure 15.5(d) shows the action "jobarrange" selected, which will make a matching plan between applicants and clients.

Figure 15.5(e) shows that action "assignplanStatus" selected, which displays the job arrangement plan.

Figure 15.5(f) shows the result of jobarrange42.p, where "saito" is assigned to "maruzen."

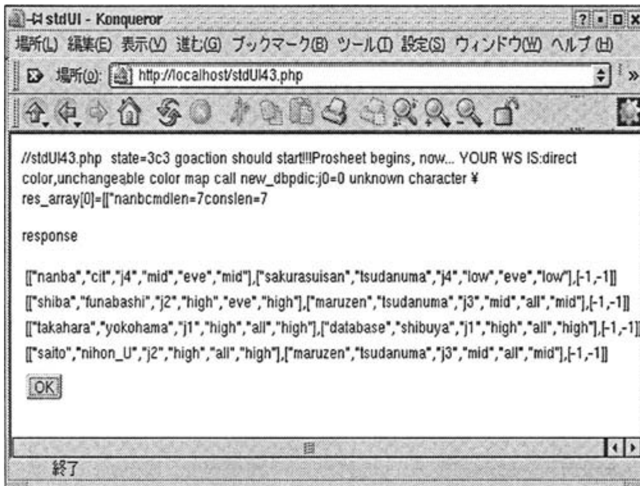


Fig. 15.5(f). Result of problem-solving by action "jobarrange."

Appendix 15.1 Transaction Processing Part of Employment System in Computer-Acceptable Set Theory

```
    /*employment4.set*/
.func ([getDate2,getAId,getCId,appregist,appStatus,jobregist,jobStatus,
       jobarrange,assignplanStatus,assignconfirmApp,assignconfirmCli,
       jobassign,replaceList,timesheet,assignStatus,transCPl,
       salesfigures,changeData]);
ActionName.g  =["quit","appregist","appStatus","jobregist","jobStatus",
        "jobarrange","assignplanStatus","assignconfirmApp",
        "assignconfirmCli","jobassign","timesheet","assignStatus",
        "salesfigures","chaangeData"];
```

```
changeData.g =["filename"];
changeData2.g = ["newdata"];
delta_lambda ([_changeData])=res <->
        paralist:=stdUI_para.lib,
        res:=changeData(paralist),
        stdUI_res.lib2:=res;
changeData ([fname])=res <->
       res:="not implemented";

assignStatus.g = ["appname","appaddress","cliname","cliaddress"];
delta_lambda([_assignStatus])=res <->
        paralist:=stdUI_para.lib,
        res:=assignStatus(paralist),
        stdUI_res.lib2:=res;
assignStatus ([Aname,Aaddress,Cname,Caddress])=res <->
        (Aname= -1) ->
             (
                 (Cname= -1) ->
                                (

        res:=defSet(paStatus0(A,X,[]),[X,assignment.lib])
                               )
                .otherwise
                              (
        member([Cid,Cname,Caddress|X],client.lib),

        res:=defSet(paStatus1(A,X,[Cid,Cname,Caddress]),
             [X,assignment.lib])
                                       )
                              )
              .otherwise
                             (
        member([Aid,Aname,Aaddress|X],applicant.lib),
                 (Cname= -1) ->
                                (
        res:=defSet(paStatus2(A,X,[Aid,Aname,Aaddress]),
             [X,assignment.lib])
                                       )
                       .otherwise
                                     (
        member([Cid,Cname,Caddress|X],client.lib),
        member([Aid,Cid|X],assignment.lib),
        res:=[Aname,Aaddress,Cname,Caddress|X]
                             )
              );
        paStatus0(A,[Aid,Cid|Y],[]) <->
        member([Aid,Aname,Aaddress|Z],applicant.lib),
        member([Cid,Cname,Caddress|ZZ],client.lib),
        A:=[Aname,Aaddress,Cname,Caddress|Y];
        paStatus1(A,[Aid,Cid|Y],[Cid,Cname,Caddress]) <->
        member([Aid,Aname,Aaddress|Z],applicant.lib),
        A:=[Aname,Aaddress,Cname,Caddress|Y];
        paStatus2(A,[Aid,Cid|Y],[Aid,Aname,Aaddress]) <->
             member([Cid,Cname,Caddress|Z],client.lib),
        A:=[Aname,Aaddress,Cname,Caddress|Y];

jobStatus.g=["name","address"];
```

```
delta_lambda ([_jobStatus])=res <->
          paralist:=stdUI_para.lib,
          res:=jobStatus(paralist),
          stdUI_res.lib2:=res;
jobStatus ([name,address])=res <->
         (name= -1) ->
               (
               res:=client.lib
               )
      .otherwise
               (
       member([Aid,name,address,JT,L,WT,P,Cid,D],applicant.lib),
       res:=[name,address,JT,L,WT,P,Cid,D]
               );
appStatus.g =["name","address"];
delta_lambda ([_appStatus])=res <->
         paralist:=stdUI_para.lib,
         res:=appStatus(paralist),
         stdUI_res.lib2:=res;
appStatus ([name,address])=res <->
         (name= -1) ->
              (
              res:=applicant.lib
              )
         .otherwise
              (
              member([Cid,name,address|X],client.lib),
              res:=[name,address|X]
              );

delta_lambda ([_jobassign])=res <->
         res:=jobassign(),
         stdUI_res.lib:=res;
jobassign ()=res <->
         Date:=getDate2(),
         applicant.g2:=applicant.lib,
         client.g2:=client.lib,
         jobassign0:=defSet(pJassign(A,X,[Date]),[X,assignplan.lib]),
         assignment.lib:=append(assignment.lib,jobassign0),
         applicant.lib:=applicant.g,
         client.lib:=client.g,
         res:=jobassign0;
   pJassign(A,[[Aid,Cid],[X,Y]],[Date]) <->
         (X="yes" and Y="yes") ->
            (
            project(applicant.g,AI,[Aid,Aname,Aadd,AJType,
                 ALevel,AWTime,APLevel,CId0,ADate]),
            project(client.g,CI,[Cid,Cname,Cadd,CJType,
                 CLevel,CWTime,CPLevel,Num,CDate]),
            A:=[Aid,Cid,CPLevel,Date,[]],
            applicant.g2:=replaceList(applicant.g,AI,[Aid,
                 Aname,Aadd,AJType,ALevel,AWTime,APLevel,Cid,Date]),
            client.g2:=replaceList(client.g,CI,[Cid,Cname,
                 Cadd,CJType,CLevel,CWTime,CPLevel,Num-1,Date])
            );

assignconfirmCli.g=["name","address","appname","appaddress","accept"];
```

```
delta_lambda ([_assignconfirmCli])=res <->
        paralist:=stdUI_para.lib,
        res:=assignconfirmCli(paralist),
        stdUI_res.lib:=res;
assignconfirmCli ([Cname,Cadd,Aname,Aadd,accept])=res <->
       (member([Aid,Aname,Aadd,AJType,ALevel,AWTime,
            APLevel,CId,ADate],applicant.lib) and
        member([Cid,Cname,Cadd,CJType,CLevel,CWTime,
            CPLevel,Num,CDate],client.lib)) ->
            (
              (project(assignplan.lib,PI,[[Aid,Cid],[X,Y]])) ->
                (
                assignplan.lib:=replaceList(assignplan.
                     lib,PI,[[Aid,Cid],[X,accept]]),
                res:="assignplan.lib is updated"
                )
            .otherwise
                (
                res:="Applicant or client name is wrong"
                )
            )
        .otherwise
            (
            res:="Applicant or client name is wrong"
            );

assignconfirmApp.g  =["name","address","accept"];
delta_lambda ([_assignconfirmApp])=res <->
        paralist:=stdUI_para.lib,
        res:=assignconfirmApp(paralist),
        stdUI_res.lib:=res;
assignconfirmApp  ([Aname,Aadd,accept])=res <->
        (member([Aid,Aname,Aadd,AJType,ALevel,AWTime,APLevel,CId,ADate],
            applicant.lib)) ->
               (
        (project(assignplan.lib,PI,[[Aid,Cid],[X,Y]])) ->
               (
               assignplan.lib:=replaceList(assignplan.lib,
                    PI,[[Aid,Cid],[accept,Y]]),
               res:="assignplan.lib is updated"
               )
              .otherwise
                   (
                   res:="Applicant name is wrong"
                   )
               )
             .otherwise
                  (
                  res:="Applicant name is wrong"
               );

delta_lambda ([_assignplanStatus])=res <->
       res:=assignplanStatus(),
       stdUI_res.lib:=res;
```

```
assignplanStatus ()=res <->
        res:=defSet(pPlan(A,X,[]),[X,assignplan.lib]);
             pPlan(A,[[Aid,Cid],[X,Y]],[]) <->
        member([Aid,AName,AAdd,AJType,ALevel,
        AWTime,APLevel,CId,ADate],applicant.lib),
        member([Cid,CName,CAdd,CJType,CLevel,
        CWTime,CPLevel,Num,CDate],client.lib),
        A:=[[AName,AAdd,AJType,ALevel,AWTime,APLevel],[CName,CAdd,
             CJType,CLevel,CWTime,CPLevel],[X,Y]];

delta_lambda ([_jobarrange])=res <->
         res:=jobarrange(),
         stdUI_res.lib:=res;
jobarrange ()=res <->
         load_go("jobarrange42.p"),
         res:="job assignment is completed";

salesfigures.g =["cname","caddress"];
delta_lambda ([_salesfigures])=res <->
         paralist:=stdUI_para.lib,
         res:=salesfigures(paralist),
         stdUI_res.lib2:=res;
salesfigures ([Cname,Caddress])=res <->
         (cname= -1) ->
            (
            Pays:=defSet(pPays(A,X,[-1]),
                   [X,assignment.lib]),
            )
        .otherwise
            (
            member([Cid,Cname,Caddress|X],client.lib),
            Pays:=defSet(pPays(A,X,[Cid]),
                   [X,assignment.lib])
            ),
        TPays:=transpose(Pays),
        Sales:=sum(project(TPays,3)),
        res:=[TPays,["sales=",Sales]];
    pPays(A,[Aid,Cid,CPl,D,Phis],[CCid]) <->
        (CCid= -1 or CCid=Cid) ->
            (
            project(Phis,0,[P,-1]),
            A:=[Aid,Cid,P]
            );

timesheet.g =["name","address","company","hours"];

delta_lambda ([_timesheet])=res <->
         paralist:=stdUI_para.lib,
         res:=timesheet(paralist),
         stdUI_res.lib2:=res;
timesheet ([Aname,Aadd,Cname,hour])=res <->
         project(applicant.lib,AI,[Aid,Aname,Aadd|X]),
        (project(assignment.lib,AsI,[Aid,Cid,Cpl,D,Phis])) ->
                (
                Pay:=transCPl(Cpl)*hour,
                (Phis<>[])->
                             (
            project(Phis,0,[P,Pdate]),
```

```
            (Pdate = -1) ->
                    (
            Date:=getDate2(),
            project(Phis,"head",Phis0),
            Phis2:=append(Phis0,[[P,Date],[Pay,-1]]),
            res:=["time sheet is accepted",P,.next_sheet]
                    )
                 .otherwise
                    (
             Phis2:=append(Phis,[[Pay,-1]]),
             res:=["time sheet is accepted","no payment",.next_sheet]
                  )
              )
            .otherwise
                 (
                 Phis2:=[[Pay,-1]],
                 res:=["no payment","time sheet is accepted",.next_sheet]
                 ),
            assignment.lib2:=replaceList(assignment.lib,
                 AsI,[Aid,Cid,Cpl,D,Phis2])
             )
           .otherwise
             (
             res:="maybe names are wrong"
             );
    transCPl(CPl)=P <->
        (CPl="high") ->(P:=1500)
        .otherwise
            (
            (CPl="mid") ->(P:=1000)
            .otherwise
                (
                P:=900
                )
            );
```

```
jobregist.g =["name","address","jobtype","joblevel",
                "wtime","paylevel","number"];
delta_lambda([_jobregist])=success <->
      [name,add,type,level,wtime,pay,num]:=stdUI_para.lib,
      success := jobregist(name,add,type,level,wtime,pay,num),
      (success = 1) ->
          (res:="jobregist is successfully completed"),
      stdUI_res.lib2:=res;
jobregist (name,add,type,level,wtime,pay,num)=success <->
        date:=getDate2(),
        Id:=getCId(),
        client.lib2:=append(client.lib,[[Id,name,add,type,
                                level,wtime,pay,num,date]]),
        success:=1;
```

```
appregist.g =["name","address","jobtype","level","wtime","pay"];
delta_lambda ([_appregist])=res <->
        [name,add,type,level,wtime,pay]:=stdUI_para.lib,
        res := appregist(name,add,type,level,wtime,pay),
        stdUI_res.lib:=res;
```

```
appregist (name,add,type,level,wtime,pay)=res <->
       (member([AId,name,add,T,L,WT,PL,P,D],
            applicant.lib)) ->
                (
                res:="already registered"
                )
          .otherwise
              (
              date:=getDate2(),
              Id:=getAId(),
          applicant.lib2:=append(applicant.lib,[[Id,
               name,add,type,level,wtime,pay,-1,date]]),
          res:="appregist is completed"
              );

   getAId()=Id <->
        Id0:=aId.lib,
        Id:=project(Id0,1),
        aId.lib2:=Id+1;

   getCId()=Id <->
        Id0:=cId.lib,
        Id:=project(Id0,1),
        cId.lib2:=Id+1;
```

Appendix 15.2 Data Transformation and Solver Parts of Employment System in Computer-Acceptable Set Theory

```
   /*jobarrange42.set*/
.func ([unsatisfaction,goal,transJType,transLevel,transWTime,_Solf]);
delta ([X,GV,Vs],[A,V])=[X2,GV2,Vs2] <->
        R:=unsatisfaction(A,V),
        X2:=append(X,[[A,V]]),
        GV2:=GV+R,
        Vpos:=invproject(c.g,[V|Dv]),
        Num:=project(Vs,Vpos),
        Vs2:=replaceList(Vs,Vpos,Num+1),
        constraint([X2,GV2,Vs2]);
   unsatisfaction(A,V)=R <->
        member([A,AJT,AL,AWT,APL],A.g),
        member([V,VJT,VL,VWT,VPL,Num],C.g),
        R:=(APL-VPL)+abs(VL-AL)-cardinality(intersection(AJT,VJT));
genA ([X,GV,Vs])=As <->
        N:=cardinality(X),
        ANum:=applicantNum.g,
        Nums:=clientNum.g,
        AIds:=project(transpose(A.g),1),
        (N<ANum) ->
            (
            CIds2:=defSet(pCIds(A,X,[]),[X,C.g]),
            As:=product([project(AIds,N+1)],CIds2)
            )
```

```
          .otherwise
              (
              As:=[]
              );
     pCIds(A, [V,VJT,VL,VWT,VPL,Num], []) <->
          Num>0,
          A:=V;
constraint ([X,GV,Vs]) <->
        clientNum.g>=Vs,
        [A,V]:=project(X,0),
        member([A,AJT,AL,AWT,APL],A.g),
        member([V,VJT,VL,VWT,VPL,Num],C.g),
        AL>=VL,
        intersection(VWT,AWT)<>[],
        intersection(AJT,VJT)<>[];
initialstate ()=C <->
       C:=[[],0,constantlist(0,cardinality(clientNum.g))];
finalstate ([X,GV,Vs]) <->
         ANum:=applicantNum.g,
         N:=cardinality(X),
         N=ANum;
st(C)   <->
         finalstate(C);
goal ([X,GV,Vs])=Re <->
        Re:=GV;

preprocess () <->

data_trans ();
data_trans () <->
        getApplicant(),
        applicantNum.g:=cardinality(A.g),
        getClient(),
        TCs:=transpose(C.g),
        Nums:=project(TCs,0),
        clientNum.g:=Nums;
    getApplicant() <->
        As:=defSet(pAg(A,X,[]),[X,applicant.lib]),
        a.g:=As;
    pAg(A,X,[]) <->
        X<>"[]" and X<>"",
        [AId,Name,Add,JType,Level,WTime,PLevel,CId,Date]:=X,
        CId= -1,
        JT:=transJType(JType),
        L:=transLevel(Level),
        WT:=transWTime(WTime),
        PL:=transLevel(PLevel),
        A:=[AId,JT,L,WT,PL];
    getClient() <->
        Cs:=defSet(pCs(A,X,[]),[X,client.lib]),
        c.g:=Cs;
    pCs(A,X,[]) <->
        X<>"[]",
        [CId,Name,Add,JType,Level,WTime,PLevel,Num,Date]:=X,
        JT:=transJType(JType),
        L:=transLevel(Level),
        WT:=transWTime(WTime),
```

```
      PL:=transLevel(PLevel),
      A:=[CId,JT,L,WT,PL,Num];

  transJType(JType)=JT <->
      (JType="j1") ->
          (
          JT:=[1,2,3,4]
          )
      .otherwise
          (
          (JType="j2") ->
              (
                  JT:=[2,3,4]
              )
          .otherwise
              (
              (JType="j3") ->
                  (
                  JT:=[3,4]
                  )
              .otherwise
                  (
                  (JType="j4") ->
                      (
                      JT:=[4]
                      )
                  )
              )
          );
  transLevel(Level)=L <->
      (Level="high") ->
          (
          L:=3
          )
      .otherwise
          (
          (Level="mid") ->
              (
              L:=2
              )
          .otherwise
              (
              (Level="low") ->
                  (
                  L:=1
                  )
              )
          );

  transWTime(WTime)=WT <->
      (WTime="all") ->
          (
          WT:=[1,2]
          )
      .otherwise
          (
          (WTime="mor") ->
```

```
            (
            WT:=[1]
            )
        .otherwise
            (
            (WTime="eve") ->
                (
                WT:=[2]
                )
            )
        );
postprocess () <->
        Sol:=_Solf(),
        assignplan.lib2:=defSet(pGenP(P,X,[]),[X,Sol]);
    pGenP(P,[Aid,Cid],[]) <->
        P:=[Aid,Cid,[-1,-1]];
```

The data transformation is implemented as a preprocess of the jobarrange. Note that a.g and c.g are produced by two predicates, getApplicant() and getClient(), respectively, using defSet().

References

Avgerov, C. and Cornford, T. (1998) *Developing Information Systems*, Macmillan Press.

16

Database Connectivity for the Model Theory Approach*

The target of the model theory approach is development of an intelligent MIS, as illustrated in Fig. 1.2 and in Fig. 15.2. The lower part of the system, a transaction processing system (TPS), constitutes the infrastructure of an MIS. Its main task is management of data. Although a simple file system is used for TPS development in Chapters 14 and 15, the model theory approach requires implementation of database connectivity because the file system is, in fact, supported by a database system. The language extProlog and hence the model theory approach are extended to implement that function.

The extension of extProlog is realized in two ways. First, extProlog is extended to handle a database as a relational database system does. This is certainly needed, because there are applications in which a regular database system rather than a simple file system is employed in the model theory approach. Second, it is extended to manipulate a relational database in an object-oriented way. This is necessary for handling complicated data structures for the model theory approach. As Chapters 14 and 15 show, a data structure for the approach may not satisfy even the first normal form requirement of a relational database. In fact, the approach allows an attribute to take a list structure as its value. Needless to say a complicated data structure is also inevitable for problem-solving.

This chapter first discusses the latter extension of extProlog and then, the former extension as an application of the latter. Finally, it is presented how the extended scheme is embedded into the model theory approach.

16.1 Database Connectivity in extProlog

The language extProlog uses postgreSQL as its database system. Since it is a relational database manipulated by SQL (standard query language), a special object-oriented database language (OODB) is designed for extProlog. Then, a subsystem is developed for extProlog that interprets the OODB language to SQL and vice versa. Because OODB and extProlog are seamlessly connected, extProlog itself can function as an object-oriented language.

16.2 OODB Language

We first introduce the syntax of OODB and explain its semantics using examples.

16.2.1 Syntax of OODB Language

This subsection introduces the syntax of OODB. In the following definitions, ⟨constant symbol⟩, which denotes a Prolog constant, ⟨integer⟩, ⟨Prolog predicate list⟩, ⟨Prolog variable⟩, ⟨Prolog variable list⟩, and ⟨Prolog value⟩, which denotes a value for a Prolog variable, are used as well-known terms.

The syntax consists of four definitions: definition of class, definition of object, definition of method, and definition of data processing.

1. Definition of class

```
<class definition>::= class(<class name>,<attribute list>);

<class name>::= <constant symbol>

<attribute list>::= [<attribute list0>]

<attribute list0>::= <attribute list1>|<attribute list1>,
                     inherits(<class name list>)

<attribute list1>::= <atomic attribute declaration>|<attribute list1>,
                     <atomic attribute declaration>

<class name list>::= <class name>|<class name list>,<class name>

<atomic attribute declaration>::= "<attribute name><attribute type>"

<attribute name>::= <constant symbol>

<attribute type>::= int|float|<string>|<pointer>|<vecpointer>

<string>::= char(<integer>)

<pointer>::= <class name>

<vecpointer>::= <pointer>(<integer>,<integer>)
```

2. Definition of object

```
<object definition>::= <object name>: = newdbob(<db name>,<class name>)

<object name>::= <constant symbol>|this

<db name>::= <constant symbol>
```

3. Definition of method

```
<method definition>::= <method declaration><method implementation>

<method declaration>::= methodclass(<method name>,<method input>,
                        <method output>):-!;
```

```
<method name>::= <constant symbol>

<method input>::= <class name>

<method output>::= <class name>|undef

<method implementation>::= <method name>(<method argument list>):-
                           <Prolog predicate list>,
                           return(<return object>);

<method argument list>::= <Prolog variable list>

<return object>::= <object name>|<Prolog variable>|void
```

The cut operation is used in the method declaration in order to avoid unnecessary execution of unification.

4. Definition of data processing

```
<data assignment>::= <variable form>: = <value form>

<variable form>::= <Prolog variable>|<attribute form>

<attribute form>::= <attribute head> -> <attribute name>
                    |<attribute head> -> []|<attribute head> ->
                    [<attribute name list>]|<attribute head> -> <integer>

<attribute head>::= <object name>|<attribute head> -> <attribute name>

<attribute name list>::= <attribute name>|<attribute name list>,
                         <attribute name>

<value form>::= <data form>|<action form>

<data form>::= <Prolog value>|<attribute form>

<action form>::= <object name>*<method form>|<action form>*<method form>

<method form>::= <method name>(<method argument list>)
```

16.2.2 Example of Program in OODB Language

A program that deals with a database consists of the two parts, a program head and a program body. Classes and methods are defined at the program head, and objects and data processing are defined in the program body. Real execution is specified by rules in the program body. Let us explain the above definition using an example. The example is a part of a personnel database as illustrated in Fig. 16.1 [Ishizuka, 1996]. Figure 16.2 shows its program in the OODB language.

There are three classes: company, divisions, and division. The three objects compA, diva, and divb are objects of the company, division, and division, respectively. In Fig. 16.1, the object compA has two divisions, diva and divb. The object divb is a member of the lower divisions of the object diva or the object diva, is the upper division of the object divb.

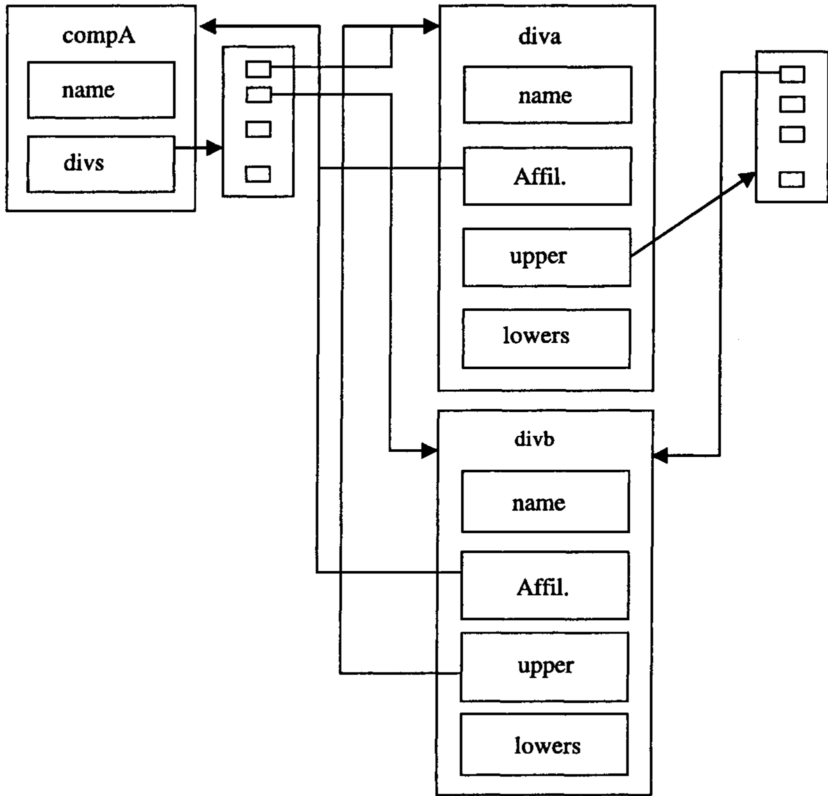


Fig. 16.1. Personnel database.

First, let us define classes, a method, objects, and a program body, and then explain the syntax using the program.

Let us examine the syntax using the above example.

(i) Definition of Class

The program in Fig. 16.2 has three class definitions: company, division, and divisions. The class company has the two attributes of name and divs, the types of which are char(40) and divisions, respectively. Here char(40) represents a string with a maximum length of 40, while divisions represents a class. Because a class is allowed to be used as a type, a complicated data structure can be defined in OODB, which is difficult for a regular relational database because it does not allow a class to be an attribute type. Here "company, ["name char(40)", "divs divisions"]" corresponds to ⟨class name⟩, ⟨attribute list⟩ while "divs divisions" corresponds to ⟨atomic attribute declaration⟩ of the syntax. "divisions" is a ⟨pointer⟩, that is, a ⟨class name⟩.

The class division has the attributes of name, affiliations, upper, and lower. The types of affiliation, upper, and lower are also classes.

```
/**personnel data base**/
/**program head **/
/* class definition */
class(company,[
        "name char(40)",
        "divs divisions"]);
class(division,[
        "name char(40)",
        "affiliation company",
        "upper division",
        "lower divisions"]);
class(divisions,[
        "vecdiv division(0,10)"];

/* method definition */
methodclass(cre_division,company,company):-!;
cre_division(D,N):-
        D:=newdbob(mydb,division),
        N:=newdbob(mydb,division),
        this->divs->veddiv->X:=N,
        N->upper:=D,
        return(this);

/**program body**/
personnelmain():-
/* object definition */
        compA:=newdbob(mydb,company),
        diva:=newdbob(mydb,division),
        divb:=newdbob(mydb,division),
        divas:=newdbob(mydb,divisions),

/* object data assignment */
        compA->[]:=["CompA",divas],
        diva->affiliation:=compA,
        divb->[name,affiliation,upper]:=["Divb",compA,diva],
        divas->vecdiv->0:=diva,

/* usage of method */
        X:=compA*cre_division(divb,divc),
        Xwriteln(0,"current division=",diva->affiliation->divs);
?-personnelmain();
```

Fig. 16.2. OODB representation of Fig. 16.1.

The type of the attribute vecdiv of the class divisions is a vector of the class division. A value of vecdiv is then a list of objects of the class division, whose length is 11. Internally, ["vecdiv division(0, 10)"] is expanded as ["vecdiv0 division",...., "vecdiv10 division"].

(ii) Definition of Object

Four objects, compA, diva, divb, and divas, are defined in the program. The object compA is created by the predicate newdbob(mydb, company), which indicates that compA is an object of the class company and is stored in a database mydb of PostgreSQL. In the same way, the object diva is defined as an object of the class division

in mydb, the object divb is defined as an object of the class division in mydb, and the object divas is classed as an object of the class divisions in mydb. The object divas is a vector of objects of the class division.

The terms compA, mydb, and company correspond to ⟨object name⟩, ⟨db name⟩, and ⟨class name⟩, respectively.

(iii) Definition of Method

One method,

cre_division,

is defined in the program. The predicate methodclass(cre_division, company, company) indicates that cre_division is a name of a method that is to be applied to the class company and that yields an object of the class company as its output.

The argument N of the method cre_division specifies an object that is appended to the attribute divs of the object "this" to which the method is applied. The other argument D specifies an object that is an upper division of the object N. The object D and argument N are created in the method just in case they are not yet defined. If they are already defined, the two newdbob objects are ignored. Finally, the predicate

return(this)

yields "this" as an output of this method.

The terms cre_division, company, company, and (D, N) correspond to ⟨method name⟩, ⟨method input⟩, ⟨method output⟩, and ⟨method argument list⟩ of the language specification, respectively.

(iv) Data Processing

In the program, four data assignments of objects are illustrated. The whole class of attributes [name, divs] of the object compA is indicated by []. Hence, the assignment command

compA -> [] := ["CompA", divas]

assigns "CompA" and divas to the attributes name and divs of the object compA, respectively.

In order to assign a value to a specific attribute, the operator -> is also used. The object compA is assigned to the attribute affiliation of the object diva by the statement

diva -> affiliation := compA,

which means that diva is a division belonging to compA.

The statement

divb -> [name, affiliation, upper] := ["Divb", compA, diva]

implies that values "Divb", compA, diva are assigned to attributes, name, affiliation, and upper of divb, respectively.

The statement

divas -> vecdiv -> 0 := diva

implies that the value diva is assigned to divas as its 0th element.

The operations comp -> [], divas -> vecdiv -> 0, this -> divs -> vecdiv -> X, diva -> affiliation, and divb -> [name, affiliation, upper] correspond to ⟨attribute head⟩ -> [], ⟨attribute head⟩ -> ⟨integer⟩, ⟨attribute head⟩ -> ⟨Prolog variable⟩, ⟨attribute head⟩ -> ⟨attribute name⟩, and ⟨attribute head⟩ -> [⟨attribute name list⟩] of the language specification, respectively. Furthermore, ["CompA", divas] corresponds to ⟨data form⟩.

(v) Execution of Method

The data assignment expression X := compA ∗ cre_division(divb, divc) illustrates how a method is used. It implies that the method cre_division is applied to the object compA and the output of cre_division is assigned to X. Because the object divc in the argument list of the method cre_division is not yet defined, it is created in the method.

16.3 Implementation of OODB Language

The above language specification is implemented using a relational database. Actually, an interpreter is designed to connect extProlog with a relational database so that the relational database can be viewed as an object-oriented database from extProlog. The interpreter communicates with the relational database in SQL.

In order to represent the interpreter in set-theoretic terms, let us introduce the following three structures and three arrays of them. The structures are actually functions.

16.3.1 Structure A

Structure A is defined as representing an attribute of a class. It has the following attributes:

name (string),
type (string),
start (int),
end (int),
link (int),

where string and int represent types of attribute. The type string is a general name of char(n) where n is a natural number, while the type int naturally indicates integers. "type," "start," and "end" are used to represent ⟨attribute type⟩, the start index, and the end index of a vector, respectively. The latter two are used only when the attribute is a vector. The attribute "link" is used to link to the next attribute. As Fig. 16.3 shows, each attribute of the class company is realized by structure A, and the family of realizations of the structures are organized as a list, where Attr represents an array of the realizations.

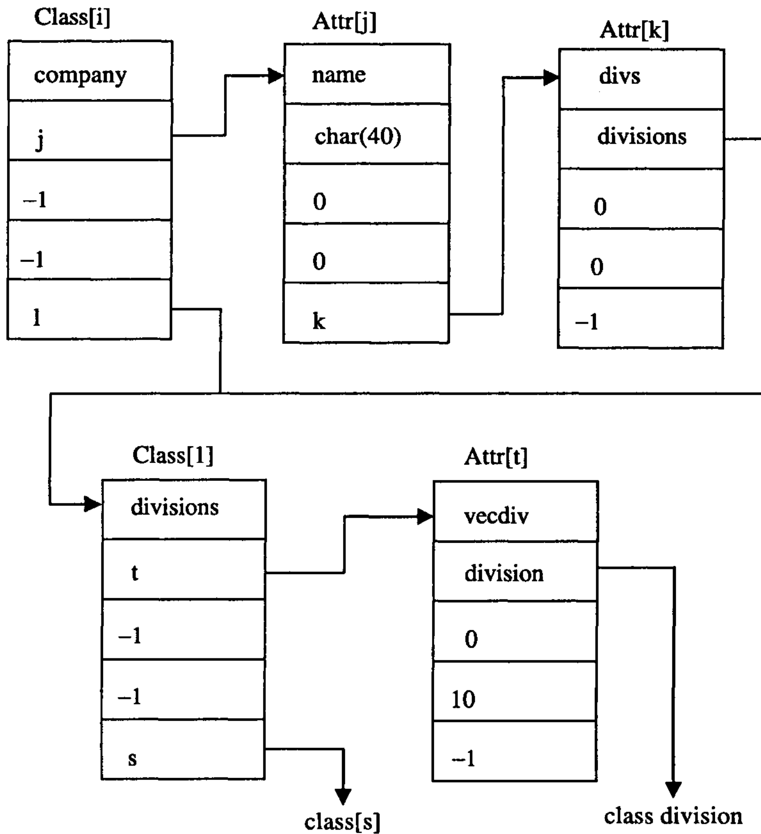


Fig. 16.3. Class representation structure.

Let

String = set of strings,
Int = set of integers,
ALst_a = {name, type, start, end, link},
V_a = String ∪ Int.

Then, a realization of structure A is represented as a function

$$\text{realization of structure A} : \mathrm{ALst}_a \rightarrow V_a.$$

16.3.2 Structure C

Structure C is defined as representing a class. It has the following attributes:

name (string),
attribute (int),
parent (int),
child (int),
link (int).

The attribute "attribute" represents a pointer to the first element of the list of attributes that constitute the class, while "parent" and "child" are pointers to other classes. They are used to implement inheritance. The attribute "link" is also used to represent a pointer to another class. Classes in one database are organized as a list by "link."

Let

$$\text{ALst}_c = \{\text{name, attribute, parent, child, link}\},$$
$$V_c = \text{String} \cup \text{Int}.$$

Then, a realization of structure C is represented by a function

$$\text{realization of structure C} : \text{ALst}_c \rightarrow V_c.$$

16.3.3 Structure O

Structure O is defined as representing an object. It has the following attributes:

name (string),
dbname (int),
classname (int).

The attribute "dbname" and "classname" are pointers to a database in which the object exists and to a class to which the object belongs, respectively.

Let

$$\text{ALst}_{ob} = \{\text{name, dbname, classname}\},$$
$$V_{ob} = \text{String} \cup \text{Int}.$$

Then, a realization of structure O is represented by a function

$$\text{realization of structure O} : \text{ALst}_{ob} \rightarrow V_{ob}.$$

Let arrays of realizations of the above structures be Attr, Class, and Obj. That is,

$$\text{Attr} : N \rightarrow [\text{ALst}_a, V_a],$$
$$\text{Class} : N \rightarrow [\text{ALst}_c, V_c],$$
$$\text{Obj} : N \rightarrow [\text{ALst}_{ob}, V_{ob}],$$

where $N = \{0, 1, 2, \ldots\}$ and $[\text{ALst}_a, V_a] = \{x | x : \text{ALst}_a \rightarrow V_a\}$. The definitions of $[\text{ALst}_c, V_c]$ and $[\text{ALst}_{ob}, V_{ob}]$ have the same meaning. Then, for example, Attr[0] implies a realization of structure A whose index is 0. The index value is used for the link value. If Attr[1] is linked to Attr[0], the value of the "link" attribute of Attr[0] is 1. The convention is applied to Class and Obj. If we use the usual notation of mathematics, the value of link of Attr[0] is denoted by Attr(0)(link). However, following the convention of database literature, we use

$$\text{Attr[0].link}(= \text{Attr(0)(link)}).$$

Figure 16.3 shows the class company. It consists of Class[i], the *i*th element of Class. The class has the two attributes of "name" and "divs." They are represented by a list of attribute structure. The first attribute of company, "name," is represented by Attr[j], the *j*th element of Attr. Therefore, the value of "attribute" of Class[i] is equal to j. The attribute "name" of Attr[j] has "name" as its value. The values of the attributes "start" and "end" are 0 because "name" is not a vector type. The attribute "link" of Attr[j] has k as its value because the next attribute "divs" is represented by Attr[k].

Because the attribute type of "divs" is divisions, the type of Attr[k] has "divisions" as its value. Because Attr[k] is the last attribute of the class company, the value of "link" of Attr[k] is nil (-1). Because no inheritance is assumed for the class company, the "parent" and "child" of a realization of company take -1 as their values. Class[i] is linked to Class[l], where both are supposed to exist in the same database.

The class "divisions" is also represented by Class[l] in Fig. 16.3. The attribute "attribute" of Class[l] is pointed to Attr[t], which represents an object "vecdiv" of the class divisions. Because vecdiv is a vector of the class division, Attr[t] takes division, 0, and 10 as its values of "type," "start," and "end," respectively.

As Fig. 16.4 illustrates, a class is implemented as a table in a relational database, while an object is implemented as an entry of a table.

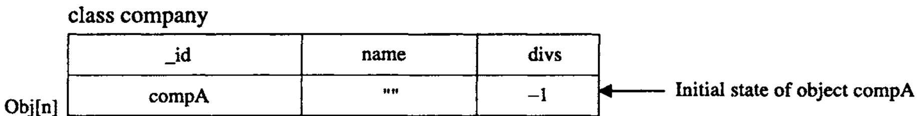


Fig. 16.4. Internal representation of class company.

Pointers are represented by integers.

When the statement

$$\text{compA} := \text{newdbob(mydb, company)}$$

creates an object of the class company, the creation internally generates the table of Fig. 16.4 and Obj[n], where n is an integer. The attribute "name" of Obj[n] naturally takes compA as its value.

It is a basic assumption that there is no duplication in object names. According to Fig. 16.3, the pointer to the class company is i and its pointers to the attributes "name" and "divs" are j and k, respectively. When values are assigned to the attributes of compA, they are inserted into the name part and the divs part of the entry of the object compA, where "" and -1 are used as initial values.

When a table for a class is generated (refer to Fig. 16.4), an extra attribute, the name of which is _id, is appended to the attributes of the class. The attribute _id is used as a key of the table. Due to the current convention that no duplication exists in object names, the name of an object is assigned to the attributes.

Using the arrays introduced above, we can prove the following theorem.

Theorem 16.1. *The specification of OODB introduced in Section 16.2 can be implemented using the arrays* Attr, Class, *and* Obj, *and a relational database that uses SQL as its manipulation language.*

Proof. Refer to Appendix 16.1.

The functions inheritance and polymorphism are also implemented. The inheritance is realized by copying inherited attributes into a class definition. The polymorphism is realized with the restriction that the implementation of a method is placed immediately after its method declaration predicate, methodclass. Under the restriction, the Prolog interpreter can find a correct implementation of the method.

16.4 SQL and OODB Languages

Sometimes, it is necessary to construct an application system that handles tables rather than complicated data structures. A typical case is given when the TPS component of Fig. 1.2 is implemented as discussed in Chapter 14. Because SQL is designed to deal with tables, it is superior to an object-oriented database language in these applications. The language extProlog manipulates table handling in two ways. The first method is to use SQL commands directly. An SQL command can be written in text form and can be executed in extProlog. For instance, the following commands create a table called company:

sqlcom -> cmd : = "create table company (name char(40));",
X : = sqlcom $*$ sqlexec(),

where the object sqlcom and its method sqlexec() are supplied by extProlog. The SQL command "create table company (name char(40));" is executed by sqlexec(), which calls the postgreSQL system. This method essentially does not use the object-oriented scheme.

The second method is to extend OODB to manipulate a table. The second method, which allows the object-oriented approach and the SQL approach simultaneously, will be presented in this section. Tables are assumed not to be huge or comparable in size with the main memory when the second method is used. Because a 64-bit CPU is now available, this restriction is not serious.

16.4.1 Example of Table Handling in OODB Language

In order to illustrate the objective of this section, let us consider the example given by Fig. 16.5 [Date, 1989].

This example consists of the four tables s (supplier of parts), p (parts), j (project), and spj (relationship among supplier, parts, projects). The structures of the tables are as follows:

```
s = (       sno         char(5),
            sname       char(20),
            status      int,
```

```
              city       char(15));
p = (         pno        char(6),
              pname      char(20),
              color      char(6),
              weight     int,
              city       char(15));
j = (         jno        char(4),
              jname      char(10),
              city       char(15));
spj = (       sno        char(5),
              pno        char(6),
              jno        char(4),
              qty        int)).
```

SPJ

<object>	SNO	PNO	JNO	QTY
spj1	S1	P1	J1	200
spj2	S1	P1	J4	700
spj3	S2	P3	J1	400
spj4	S2	P3	J2	200
spj5	S2	P3	J3	200
spj6	S2	P3	J4	500
spj7	S2	P3	J5	600
spj8	S2	P3	J6	400
spj9	S2	P3	J7	800
spj10	S2	P5	J2	100
spj11	S3	P3	J1	200
spj12	S3	P4	J2	500
spj13	S4	P6	J3	300
spj14	S4	P6	J7	300
spj15	S5	P2	J2	200
spj16	S5	P2	J4	100
spj17	S5	P5	J5	500
spj18	S5	P5	J7	100
spj19	S5	P6	J2	200
spj20	S5	P1	J4	100
spj21	S5	P3	J4	200
spj22	S5	P4	J4	800
spj23	S5	P5	J4	400
spj24	S5	P6	J4	500

S

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P

PNO	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

J

JNO	JNAME	CITY
J1	Sorter	Paris
J2	Punch	Rome
J3	Reader	Athens
J4	Console	Athens
J5	Collator	London
J6	Terminal	Oslo
J7	Tape	London

Fig. 16.5. Database for parts.

Suppose the data of Fig. 16.5 are given. When we use the data in the second of the two ways mentioned above, it must be transformed into table forms that are compatible with OODB. For instance, the table s is transformed into a class and objects by the following program:

```
class(s,[ "sno       char(5)",
          "sname     char(20)",
          "status    int",
          "city      char(15)"]);
a0():-
          s1: = newdbob(hisdb,s),
          s2: = newdbob(hisdb,s),
          s3: = newdbob(hisdb,s),
          s4: = newdbob(hisdb,s),
          s5: = newdbob(hisdb,s),
```

```
            s1 -> []: = ["S1","Smith",20,"London"],
            s2 -> []: = ["S2","Jones",10,"Paris"],
            s3 -> []: = ["S3","Blake",30,"Paris"],
            s4 -> []: = ["S4","Clark",20,"London"],
            s5 -> []: = ["S5","Adams",30,"Athens"];
?-a0();
```

The class declaration generates a template s. The literal s1 := newdbob(hisdb, s) in the rule $a0()$ creates an object of the class s in the database hisdb. The literal s1 -> [] := ["S1", "Smith", 20, "London"] then inserts data "S1," "Smith," 20, "London" into the object s1. In the same way the objects s2, s3, s4, and s5 are created and appropriate data are inserted into them. Because the conversion can be done by a Prolog program in extProlog, a user need not write a program like the one listed above.

This section investigates how the tables created as above can be handled in OODB.

Let us consider the following simple SQL command, which is applied to the table s:

select s.sno, s.sname, s.city from s;

This command selects data corresponding to the attributes sno, sname, city from the table s and creates a subtable.

It is true that this operation can be realized in an object-oriented way. For example, the following program aa0() is an object-oriented implementation of the above command:

```
class(s,[ "sno     char(5)",
          "sname   char(20)",
          "status  int",
          "city    char(15)"]);
class(stable,["ss  s(0,10)"]);
methodclass(getelements,stable,undef):-!;
getelements(Attrlist):-
        Ss: = this -> [],
        X: = Ss -> Attrlist,
        return(X);

aa0():-

        ssob: = newdbob(hisdb,stable),
        ssob -> ss: = [s1,s2,s3,s4,s5],
        X: = ssob*getelements([sno,sname,city]);

?-aa0();
```

The table stable has a single entry that is a vector of eleven tables whose templates are given by s. The declaration methodclass says that the rule below it, getelements, is an applicable method to the class stable. The type of an output of the method is undefined (undef).

The method getelements has a parameter, Attrlist. The literal Ss := this -> [] inserts the data list of the current object (this) into Ss. That is, Ss is a list that is equivalent to an entry of the table s. The next literal X := Ss -> Attrlist extracts a sublist of Ss corresponding to Attrlist and inserts it into X. The literal return(X) outputs X.

The execution of the method is carried out in the rule aa0(). An object ssob is created by the literal ssob: = newdbob(hisdb, stable), and the literal ssob -> ss: = [s1, s2, s3, s4, s5] inserts s1, s2, s3, s4, s5 into it. Finally, by applying getelements to ssob, where Attrlist = [sno, sname, city], we have the desired result.

There are two difficulties with the above procedure. First, a user must specify explicitly the rows s1, s2, s3, s4, s5 in the program. If the number of rows changes, the program must be modified, which is obviously undesirable. Second, in the present program, the user must create a new class and write a method for it even though the required processing is trivial, at least from the SQL viewpoint.

Although one line is enough to describe the above operation in SQL, a rather complicated program must be written if we follow the object-oriented approach. A user should not be required to accept this inconvenience.

The above example uses only one table. However, in general, an SQL command uses more than one table and includes condition clauses like group by... having.... It is not feasible to expect a user to be always able to write a program in an object-oriented way that is equivalent to the given SQL command.

This section will extend OODB in a way such that the following conditions can be met:

1. The host language, extProlog, can implement the same functions as those given by SQL commands without the necessity for an SQL command embedded in a host language. Then, one language can process tables as well as objects.
2. A user is not required to create a metatable, the element of which is another table as illustrated above when an SQL command is implemented.
3. When a program is written in an extended language, the ease of its writing, its readability, and its execution speed must be comparable to the counterpart in SQL.

If OODB can be extended while still satisfying the above conditions, it can provide a powerful environment, because OODB has strong descriptive capability for complicated data structures, and conventional database processing can also be performed in it.

16.4.2 Implementation Scheme for SQL Functions

This section summarizes basic facts of OODB, which will be used for the implementation. (Refer to Section 3.4.2.)

(i) Representation of Set

A set is represented as a list:

$$\text{Representation of set} = [a_1, \ldots, a_n].$$

Here a_i represents an atomic element of Prolog. It can be an integer, a real, a constant, a string, a list, or a more complicated element. Hence, naturally, an atom can cover the entire types of SQL. In particular, if $a_i (i = 1, \ldots, n)$ is a list, the expression of a set is a representation of a table. For instance, [[1, 2, 3], [4, 5, 6]] is an expression of a table (or of a matrix). An empty set is represented by [].

(ii) Construction of Set

A set is constructed by directly specifying elements of a list or by extension of a predicate of Prolog. The latter is, of course, important. In extProlog the latter construction is given by the predicate defSet() (Refer to Section 3.4.2).

In this section, defSet() is decomposed into the two predicates: extent_of() and in(). Suppose a predicate $p(X, \langle\text{parameterlist}\rangle)$ is defined by a user. Then, the set Xs of X that satisfies the predicate, i.e., $Xs = \{X \mid p(X, \langle\text{parameterlist}\rangle)\}$, is given by

$$Xs := \text{extent_of}\,(p(X, \langle\text{parameterlist}\rangle)).$$

When the predicate $p()$ denotes a specific row of a table by a variable Y, the notation is expressed by in(). For instance, if we want to say that Y represents a row in a table t, the following expression is used:

$$\text{in}(Y, t).$$

An example of in() is given in the program a1() below.

(iii) Definition of Table

In OODB the template of a table is defined as a class. For instance, in Section 16.4.1 the table s is specified as follows:

```
class(s,[   "sno       char(5)",
            "sname     char(20)",
            "status    int",
            "city      char(15)"]);
```

(iv) Creation of a Row of a Table

A row is defined as an object. For instance, the following literal can create a row in a table s:

$$s1 := \text{newdbob(mydb, s)}.$$

(v) Specification of an Attribute Value

The value of an attribute a in an object s1 (row) is expressed by s1 -> a. A value can be inserted using the same expression. For instance, s1 -> city: = "Tokyo".

(vi) Row Insertion

If we want to insert values $v_1, \ldots, v_n$ into all of the attributes of an object obn, we can write as follows:

$$obn- > [] := [v_1, \ldots, v_n].$$

Each v_i is inserted into the ith attribute.

(vii) Table Insertion

If we want to simultaneously insert a value v into an attribute a of objects $ob1, \ldots, obn$ of the same class, we can write as follows:

$$[ob_1, \ldots, ob_n] \text{ -> } a := v.$$

In general, if we want to simultaneously insert a list of values $v_1, \ldots, v_m$ into a list of objects $ob_1, \ldots, ob_n$ of the same class, we can write as follows:

$$[ob_1, \ldots, ob_n] \text{ -> } [] := [v_1, \ldots, v_m],$$

where m is the number of attributes of the class.

(viii) Deletion of an Object

The following predicate deletes objects $s_1, \ldots, s_n$ from the database mydb:

$$\text{destroydbob("mydb", } [s_1, \ldots, s_n]).$$

(ix) View function in OODB

The view function is implemented in the following procedure:

1. Construct data as a table (list of lists)
2. Display the data by the following predicate:

$$\text{createDM}(\langle\text{name of view}\rangle, \langle\text{data}\rangle).$$

The data will be displayed on a spreadsheet of extProlog.

(x) Operations on Lists

The following are system-defined predicates to facilitate operations on sets.

- **Unary and binary operations on lists (sets)**

Most of the unary and binary arithmetic operations are extended as termwise operations to be applicable to lists. Refer to Section 3.4.2.

- **Predicate in()**

The predicate in() can be used to check whether an element belongs to a list as it is usually interpreted in SQL. For example, suppose X and Xs are an element and a list, respectively. Then in(X, Xs) = true iff X is an element of Xs. In this case, in() is equivalent to member() of Prolog.

• **Special functions for lists**

Many special functions are defined for lists (sets). Refer to Section 3.4.2. Usual functions used for sets in SQL are all implemented in extended OODB.

(xi) Special Functions to Describe SQL Functions

Two functions, group_by() and joint_of(), are introduced to provide easy implementation of some SQL functions. The function group_by is used as follows:

⟨grouped table⟩ : = group_by(⟨target table⟩, ⟨list of attributes for grouping⟩,
⟨retrieval condition⟩).

The function joint_of is used as follows:

⟨jointed table⟩ := joint_of(⟨list of subtables to be jointed⟩).

Their meanings are explained using examples below.

16.4.3 Implementation of SQL Functions in OODB Language

Let us examine SQL functions in the following categories:

(i) Data definition
(ii) Data retrieval
(iii) Updating of table
(iv) Embedding SQL commands

(i) Data Definition

The data definition has two functions, create_table and create_view. The create_table function is implemented by the class definition discussed in Section 16.4.2. The create_view function is also discussed in Section 16.4.2 except for how a subtable is constructed, which is discussed below.

(ii) Data Retrieval

We assume that SQL commands for data retrieval can be expressed in two forms:

select⟨selection list⟩from⟨list of table names⟩,

select⟨retrieval condition⟩group by⟨row list⟩having⟨retrieval condition⟩.

There are other forms including terms such as "order by" and "into ⟨temporary file⟩." However, functions from these terms are simply implemented in Prolog after necessary data is retrieved using the above functions.

a. The case in which "group by" is not used

We investigate this case using the following example:

```
select T1.a1, T2.a2 from t1 T1, t2 T2 where p(T1,T2)
```

where T1 and T2 are variables to represent rows of table t1 and table t2, respectively, and a1 and a2 are names of attributes of t1 and t2, respectively. Here p(T1, T2) is a predicate to specify a retrieval condition. This command is implemented by the following rules:

```
a1():-

        Ans: = extent_of(qa(X));

qa(X):-

in(T1,t1),
in(T2,t2),
pa(T1,T2),
X: = [T1 -> a1,T2 -> a2];
```

where pa() is a predicate that is an implementation of the retrieval condition p() in OODB. The rule qa() requires that the variables T1 and T2 represent rows of tables t1 and t2 by in(T1, t1) and in(T2, t2), respectively, and furthermore, satisfy the predicate pa(). The literal

```
X: = [T1 -> a1,T2 -> a2]
```

extracts values corresponding to the attributes a1 and a2 from T1 and T2.

Let us consider the following SQL command, which consists of a simple retrieval, subretrievals, exists, and SQL functions:

```
select  spjx.jno
from    spj         spjx
where   exists
        (select     *
        from        spj     spjy
        where       spjy.pno ='p2' and spjy.qty >
        (select     avg(spjz.qty)
        from        spj     spjz
        where       spjz.pno = 'p1' and
                    spjz.sno = spjx.sno)).
```

This is a retrieval command regarding the table spj of Fig. 16.5.

The following is a direct translation of the above SQL command into an OODB program. Suppose we are given the following class definition of the table spj:

```
class(spj,[

        "sno char(5)",
        "pno char(6)",
        "jno char(4)",
        "qty int'']);
```

Then, the required program is

```
a2():-

    Xs: = extent_of(qa(X)), /*select spjx.jno*/
    Ans: = Xs;

qa(X):-

    in(SPJX,spj), /*from spj spjx*/
    Ys: = extent_of(qb(Y,SPJX)), /*select * */
    Ys <> [], /*exists*/
    X: = SPJX -> jno; /*spjx.jno*/

qb(Y,SPJX):-

    in(SPJY,spj), /*from spj spjy*/
    SPJY -> pno = 'p2', /*spj.pno = 'p2'*/
    Zs: = extent_of(qc(Z,SPJY,SPJX)),
    SPJY -> qty > average(Zs -> qty),
    Y: = SPJY;

qc(Z,SPJY,SPJX):-

    in(SPJZ,spj), /*from spj spjz*/
    SPJZ -> pno = 'p1', /*spjz.pno = 'p1' and*/
    SPJZ -> sno = SPJX -> sno,/*spjz.sno = spjx.sno*/
    Z: = SPJZ;
```

Because select-from-where is implemented as Xs: = extent_of($p(X, \langle \text{parameter} \rangle)$), the condition "where exists..." is implemented by the predicate qa(). The expression "in(SPJY, spj)" of qb() corresponds to "from spj spjx."

The command "exists()" is implemented as follows: first, find the set Ys that satisfies the condition of exists(); second, check Ys <> []. The condition is expressed by the predicate qb() in the rule qa(). Ys is, therefore, determined by qb(Y,SPJX), where SPJX is a parameter.

The expression "from spj spjy" is implemented as in(SPJY, spj). The part "(select avg(... = spjx.sno))" is implemented by "Zs := extent_of(... > average(Zs - > qty)." Here Zs is the set where the condition "from spj spjz where ... = spjx.sno". The function "average" in the program is obviously a counterpart of "avg" of SQL.

b. The case in which "group by" is used

We will investigate this case using the following SQL command:

```
select    T.a1,T.a2, f(T.a3,T.a4)
from      t          T
where     p1(T)
group by  T.a1,T.a2
having    p2(T),
```

where f(T.a3, T.a4) is a binary function.

The above SQL command is realized by the following program:

```
a3():-
        Ts: = extent_of(q1(T)),
        Ys: = group_by(Ts,[a1,a2],q2(Y)),
        Ans: = joint_of([min(Ys -> a1),min(Ys -> a2),
               f(Ys -> a3,Ys -> a4)]);
q1(T):-
        in(T, t),
        p1a(T);
q2(Y):-
        p2a(Y);
```

where q1() and q2() are rules corresponding to the retrieval conditions p1() and p2(), respectively.

Let us consider the above implementation using the following SQL command regarding the table spj of Fig. 16.5:

```
select SPJ.sno, SPJ.pno, max(SPJ.qty) - min(SPJ.qty)
from        spj        SPJ
group by    SPJ.sno,    SPJ.pno
having sum(SPJ.qty) > 500;
```

The above command is implemented as follows:

```
a4():-
    Ts: = extent_of(q1(SPJ)),
    Ys: = group_by(Ts,[sno,pno],q2(Y)),
    Ans: = joint_of([min(Ys -> sno),min(Ys -> pno),
           max(Ys -> qty)-min(Ys -> qty)]);
q1(SPJ):-
    in(SPJ,spj);
q2(Y):-
    sum(Y -> qty) > 500;
```

The literal "Ts := extent_of(q1(SPJ))" gets a table (list of rows) that satisfies the select-from function and inserts the table into Ts. Because there is no where condition, Ts is equal to the table spj. Then, the function group_by does grouping of the table Ts, where each group consists of rows that have the same value with respect to the attributes [sno, pno]. That is, the result is [[spj1, spj2], [spj3, spj4, spj5, spj6, spj7, spj8, spj9], ..., [spj22]], where, for example, spj1 and spj2 constituting the subgroup [spj1, spj2] have the value [S1, P1] for [sno, pno]. The subgroups are substituted into Y of q2(Y) and finally Ys is determined as a set of subgroups that satisfies the rule q2(). Then, we have

$$
\begin{aligned}
Ys = {} & [[\text{spj1, spj2}], [\text{spj3, spj4, spj5, spj6, spj7, spj8, spj9}], \\
& [\text{spj12}], [\text{spj13, spj14}], [\text{spj17, spj18, spj23}], [\text{spj19, spj24}], [\text{spj22}]].
\end{aligned}
$$

Finally, Ans is given by joint_of(). Note that "min(Ys -> sno)" is used to eliminate duplicated elements in Ys -> sno. As mentioned in Section 16.4.2, Ys -> sno is equal to [[spj1, spj2] -> sno,...] and [spj1, spj2] -> sno is equal to [spj1 -> sno, spj2 ->

sno] = [S1, S1] and hence we have a duplicated result. Also, min([S1, S1]) yields S1. On the other hand,

$$\max(Ys \text{ -> } \mathrm{qty}) - \min(Ys \text{ -> } \mathrm{qty}) = [\max([\mathrm{spj1} \text{ -> } \mathrm{qty}, \mathrm{spj2} \text{ -> } \mathrm{qty}]) \\ - \min([\mathrm{spj1} \text{ -> } \mathrm{qty}, \mathrm{spj2} \text{ -> } \mathrm{qty}]), \ldots] = [700 - 200, \ldots] = [500, \ldots].$$

The command "joint_of" transforms these results into a table. Section 16.5.b uses the above implementation as an example of database connectivity for the model theory approach. Figure 16.7(c) shows the result.

The joint operation in retrieval can be easily implemented. For instance, let us consider the following SQL command:

```
select  s.sno,p.pno
from    s,p
where   s.city = p.city;
```

This is implemented by

```
a5():-
        Ans: = extent_of(p(X));
p(X):-
        in(S,s),
        in(P,p),
        S -> city = P -> city,
        X: = [S -> sno,P -> pno];
```

The union operation in retrieval can also be implemented without difficulty. Let us consider the following SQL command:

```
select s.city from s
union
select p.city from p;
```

This is implemented by

```
a6():-
        Ss: = extent_of(p1(S)),
        Ps: = extent_of(p2(P)),
        union(Ss -> city,Ps -> city,Ans);
p1(S):-
        in(S,s);
p2(P):-
        in(P,p);
```

The operation "union" of a6() is a system-defined predicate of OODB.

(iii) Updating of Table

There are three kinds of updating operations: update, delete, and insert. These are implemented by the extent_of predicate. Let us consider the following example of update:

```
update p
set     color = 'orange'
where   p.color = 'red';
```

This is implemented by

```
a7():-
        Ps: = extent_of(p(P)),
        Ps -> color: = 'orange';
p(P):-
        int(P,p),
        P -> color = 'red';
```

Similarly, let us consider the following SQL command of delete:

```
delete  from    j
where j.jno not in(select spj.jno from spj);
```

This is implemented by

```
a8():-
        Js: = extent_of(p(J)),
        destroydbob(mydb,Js);
p(J):-
        in(J,j),
        SPJs: = extent_of(p2(SPJ)),
        not(member(J -> jno,SPJs -> jno));
p2(SPJ):-
        in(SPJ,spj);
```

It is assumed that the table *j* exists in the database "mydb." It should be noted that if SPJs is obtained before computation of Js and if SPJs is used as a parameter of p(), execution speed of a8() will be optimized.

As an example of "insert," let us consider the following problem:

"Increase the output by 10% of every supplier who supplies a red part."

The above problem is implemented by the following OODB program:

```
a9():-
        Ss: = extent_of(p1(S)),
        Ss -> qty: = (Ss -> qty)*1.1;
p1(S):-
        in(S,spj),
        in(P,p),
        P -> color = 'red',
        P -> pno = S -> pno;
```

According to [Date, 1989], the above problem is solved by the following SQL command:

```
create  table   reds
        (sno    char(5),
        primary key(sno));
insert  into    reds(sno)
        select  spj.sno
        from    spj,p
        where   spj.pno = p.pno and p.color = 'red';
update  spj
        set qty = spj.qty*1.1
        where   spj.sno in (select reds.sno from reds);
```

This example shows that the update operation of SQL needs an involved procedure.

(iv) Embedding SQL Command

Because the OODB is seamlessly connected with its host language extProlog, as demonstrated above, there is no embedding problem in the reference system.

16.4.4 Ease of Description in OODB Language

The previous section shows that every SQL command can be implemented in OODB. This section examines whether OODB is an easy language.

SQL can be used as a restricted and structured natural language for a user when the database manipulation is described. Furthermore, a description can be made in a declarative way. This indicates that SQL should be considered a quite convenient language for a user.

OODB is also a declarative language, but due to the nature of Prolog there is no term to describe a relationship among statements. For instance, because Prolog does not have "where," the meaning of a5() cannot be grasped as easily as its corresponding SQL statement. This indicates that a program written in Prolog is less structured than a command of SQL and is quite similar to a program written in an assembly language, although it may be claimed that because OODB is similar to an assembly language, it can be more flexible than SQL.

Regarding readability, it is true that a language akin to a natural language is superior to one that is unlike a natural language. However, when a complicated command is to be built, the analytical description of OODB can be more readable than a structured description in SQL. Let us consider the following command [Date, 1989]:

```
select distinct spjx.sno
from    spj spjx
where   exists
    (select p.pno
    from    spj spjy
    where   not exists
        (select j.jno
        from    j
        where   not exists
            (select *
            from    spj spjz
            where
                spjz.sno = spjx.sno and
                spjz.pno = spjy.pno
                and spjz.jno = j.jno)));
```

It is doubtful that a user can understand this command correctly and quickly. In order for it to be understood, the user may have to analyze it. The following program is an analyzed representation of the above command.

```
a10():-
    Js: = extent_of(p1(J)),
    Xs: = extent_of(p2(X,Js)),
    distinct(Xs -> sno,Z),
    Ans: = Z;
```

```
p1(J):-
    in(J,j);
p2(X,Js):-
    in(X,spj),
    Cs: = extent_of(p3(C,X)),
    minus(Js -> jno,Cs -> jno,[]);
p3(C,X):-
    in(C,spj),
    X -> sno = C -> sno,
    X -> pno = C -> pno;
```

OODB provides a natural means to describe a command in an analytical way.

16.4.5 Execution Speed in OODB Language

In order to improve execution speed in OODB, the following execution mechanism is introduced:

1. When an object (a row of a table) is requested for execution, it is loaded into the main memory as a predicate;
2. Execution on the object is carried out in the main memory without using the hard disk;
3. When execution of a program is over, the object in the main memory is copied to the hard disk.

These operations are automatically performed by the system. The above on-memory database mechanism naturally enhances execution speed and makes OODB a practical tool for an intelligent MIS development. The on-memory database mechanism has become feasible because a 64 bits microprocessor is available.

16.5 Database Connection to Model Theory Approach

The previous sections presented a database-handling scheme as an extension of extProlog. This section shows how the scheme is embedded into the model theory approach. Two methods are available for integration.

a. Direct Connection Method

As mentioned in Section 16.4, the extProlog extension includes a scheme for handling SQL commands directly. The direct connection method uses this function. The following atomic process "execSQL" accepts an SQL command from the external UI and returns the result of the command to the UI:

```
execSQL.g=["data base name","SQL command"];
delta_lambda([_execSQL],paralist)=res <->
        res:=execSQL(paralist);
execSQL([dbn, cmd])=res <->
        db.g:=dbn,
```

```
.sqlcom ->.cmd:=cmd,
res0:= .sqlcom *sqlexec(),
res:=append(["table",-1],res0);
```

The argument "dbn" in execSQL specifies to which database the SQL command "cmd" is applied. The name ".sqlcom" is the object name in computer-acceptable set theory for the object sqlcom introduced in Section 16.4.

In this atomic process, an SQL command, which is represented by "cmd," is inserted into the object sqlcom and executed by the method sqlexec(). Real execution is carried out by the postgreSQL system. The result is sent to the external UI by res:=append(["table, −1], res0) in table form. Here "−1" in ["table", −1] indicates that no attribute is specified for a column of the table, that is, it is a blank.

Figure 16.6 shows how the direct connection method works. Figure 16.6(a) shows the selection of the action execSQL, which implies selection of direct connection.

Fig. 16.6(a). Selection of direct connection.

The external UI then requests the user to input an SQL command. Figure 16.6(b) shows an input of "select * from *j*," which indicates that the user wishes to view the entire contents of table *J* (Fig. 16.5) in the database "hisdb." The tables in this section are inserted into the database "hisdb" as classes by the database-handling model shown in Appendix 16.2 (see Section 16.4).

The atomic process "execSQL" executes the SQL command, giving the result shown in Fig. 16.6(c). The figure shows the internal form of table *J*. It should be noted that the first column of Fig. 16.6(c) corresponds to the attribute "_id."

b. Indirect Connection Method

The other method implements the connection using the same scheme as that used to connect the solver "jobarrange42.p" to the TPS "employment4.set" in Chapter 15. This

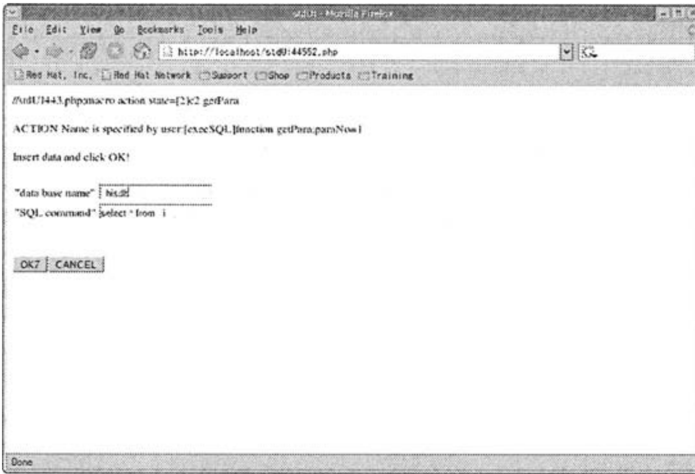


Fig. 16.6(b). Input of SQL command.

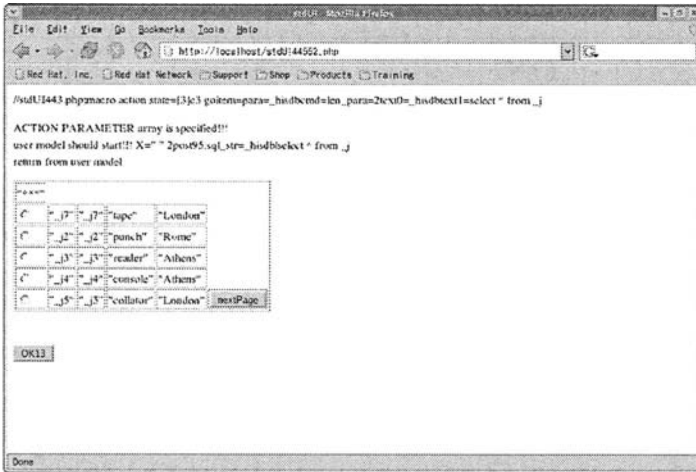


Fig. 16.6(c). Output of SQL command.

method has an advantage over the first in that the object-oriented scheme can be fully exploited. The only inconvenience of the indirect method is that a database must be reconstructed in the object-oriented way.

The embedding scheme consists of two steps:

(1) A database-handling model is prepared in set theory and compiled (see Appendix 16.2) or in extProlog as discussed in the previous sections.
(2) The model in extProlog is called by the following atomic process:

```
execPrologPrg.g=["PrologPrg name"];
delta_lambda([_execPrologPrg],paralist)=res <->
```

```
        res:=execPrologPrg(paralist);
execPrologPrg([prgn])=res <->
        load_go(prgn),
        res:=loaddb_res();
loaddb_res()=res <->
        res:=append(["table",-1],db_res.lib);
```

When the action "execPrologPrg" is selected by the user, the external UI requests the name of the database-handling model as an input. The model is then executed by the atomic process "execPrologPrg" using the system-defined predicate "load_go." The output of execution is saved in the file "db_res.lib" by the database-handling model (see Appendix 16.2). The atomic process then retrieves the result by the loaddb_res() function and sends the data to the external UI.

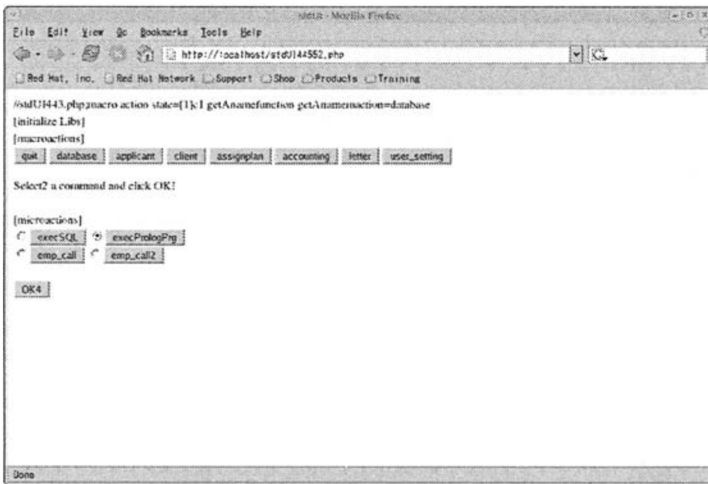


Fig. 16.7(a). Selection of indirect connection.

Figure 16.7 shows the operation of the scheme using an implementation of the group_by function discussed in Section 16.4.3 as an example. Figure 16.7(a) shows the user selection of the action "execPrologPrg," which corresponds to the indirect connection method.

Following the definition of the atomic process, the external UI requests the user to input the name of the model in extProlog. Selection of the model "sqlex9.p" is shown in Fig. 16.7(b). (The model in set theory is listed in Appendix 16.2.)

Figure 16.7(c) shows the final result of model execution, which is presented in a reduced form of the table SPJ by the group_by operation as explained in Section 16.4.3.

Fig. 16.7(b). Input of database-handling model name.

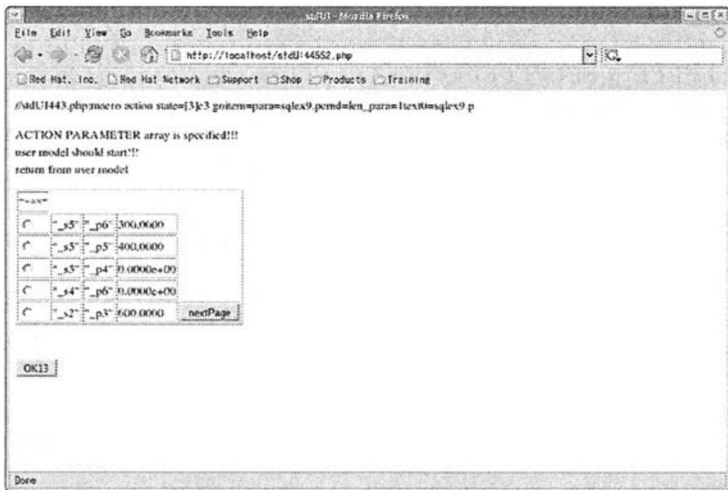


Fig. 16.7(c). Output of database-handling model.

Appendix 16.1 Proof of Theorem 16.1

Let us define the following sets:

$$\begin{aligned}\mathrm{Obj}_n &= \text{set of names of object,}\\ C_{\mathrm{name}} &= \text{set of names of class,}\\ \mathrm{Attr}_n &= \text{set of names of attribute,}\\ \mathrm{Attr}_{\mathrm{type}} &= \text{set of types of attributes} = \{\text{int, float, string}\} \cup C_{\mathrm{name}},\end{aligned}$$

and

$$\mathrm{DataV} = \mathrm{Int} \cup \mathrm{Float} \cup \mathrm{String},$$

where Int, Float, and String are sets of integers, floats, and strings, respectively. DataV is a set of values of attributes of a usual relational database. Let

$$\mathrm{Attr} = \mathrm{Attr}_n \times \mathrm{Attr}_{\mathrm{type}}$$

and

$$\mathrm{ObV} = \mathrm{Int} \cup \mathrm{Float} \cup \mathrm{String} \cup \mathrm{Obj}_n,$$

where ObV is the set of values of attributes of OODB. Let

$$\mathrm{Int}_m = \{0, \ldots, m-1\}.$$

Int_m is an index set of objects, where the number of objects is m.

Based on the above sets and the three arrays, the following functions are defined.

Definition 16.1. Let obj: $\mathrm{Int}_m \to \mathrm{Obj}_n$ be such that

$$\mathrm{obj(i) = Obj[i].name},$$

where Obj[i].name is the value of the attribute "name" of Obj[i]. The same convention is used below.

Definition 16.2. Let obindex: $\mathrm{Obj}_n \to \mathrm{Int}_m$ be such that

$$\mathrm{obindex} = \mathrm{obj}^{-1}.$$

The function obj^{-1} exists because no duplication is allowed for objects.

Definition 16.3. Let cname: $\mathrm{Obj}_n \to C_{\mathrm{name}}$ be such that

$$\mathrm{cname}(\mathrm{obj}_n) = \mathrm{Class[Obj[obindex(obj_n)].classname].name}.$$

The function cname yields the class name of an object.

Definition 16.4. Let attrnamelist: $C_{\mathrm{name}} \to \{_\mathrm{id}\} \times \cup_{k=1}^{\infty} (\mathrm{Attr}_n)^k$ be such that

$$\mathrm{attrnamelist}(c_{\mathrm{name}}) = \text{list of attributes of the class } c_{\mathrm{name}}.$$

For instance, attrnamelist(company) = (_id, name, divs).

Definition 16.5. Let attrtypelist: $C_{\text{name}} \to \{\text{string}\} \times \cup_{k=1}^{\infty}(\text{Attr}_{\text{type}})^k$ be such that

$$\text{attrtypelist}(c_{\text{name}}) = \text{list of types of attributes of } c_{\text{name}}.$$

For instance, attrtypelist(company) = (string, string, divisions).

Definition 16.6. Let attrtype: $(C_{\text{name}} \times \text{Attr}_n) \to \text{Attr}_{\text{type}}$ be such that

if $\text{attr}_n \in \text{attrnamelist}(c_{\text{name}})$,

$$\text{attrtype}(c_{\text{name}}, \text{attr}_n) = \begin{cases} \text{type of attr}_n \text{ of } c_{\text{name}} & \text{if attr}_n \in \text{attrnamelist}(c_{\text{name}}), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

For instance, attrtype(company, name) = string.

Definition 16.7. Let attrlist: $C_{\text{name}} \to \{(_\text{id}, \text{string})\} \times \cup_{k=1}^{\infty}(\text{Attr})^k$ be such that

$$\text{attrlist}(c_{\text{name}}) = ((_\text{id}, \text{string}), (\text{attr}_{n1}, \text{type}_1), \ldots, (\text{attr}_{nk}, \text{type}_k)),$$

where the class c_{name} is assumed to be specified by the attribute list $((\text{attr}_{n1}, \text{type}_1), \ldots, (\text{attr}_{nk}, \text{type}_k))$.

For instance, attrlist(company) = ((_id, string), (name, string), (divs, divisions)).

Definition 16.8. Let objV: $(\text{Obj}_n \times \text{Attr}_n \times \text{DataV}) \to \text{ObV}$ be such that

$$\text{if attrtype}(\text{cname}(\text{obj}_n), \text{attr}_n) \in \{\text{int, float, string}\}$$
$$\text{objV}(\text{obj}_n, \text{attr}_n, d) = d;$$

if $\text{attrtype}(\text{cname}(obj_n), \text{attr}_n) \in (\text{attrtypelist}(\text{cname}(\text{obj}_n)) - \{\text{int,float,string}\})\ \&d \in$ Int

$$\text{objV}(\text{obj}_n, \text{attr}_n, d) = \text{obj}(d);$$

otherwise

$$\text{objV}(\text{obj}_n, \text{attr}_n, d) = \text{undefined}.$$

When $\text{attr}_n \notin \{\text{int, float, string}\}$ holds or the type represents a name of a class, $d \in$ DataV is an integer that represents a pointer to an object. For instance, objV(diva, affiliation, d) = compA, where obindex(compA) = d.

Definition 16.9. Let entryV: $(\text{Obj}_n \times \text{Attr}_n \times \text{ObV}) \to \text{DataV}$ be such that

$$\text{if attrtype}(\text{cname}(\text{obj}_n), \text{attr}_n) \in \{\text{int, float, string}\}$$
$$\text{entryV}(\text{obj}_n, \text{attr}_n, v) = v,$$
$$\text{if attrtype}(\text{cname}(\text{obj}_n), \text{attr}_n) \in (\text{attrtypelist}(\text{cname}(\text{obj}_n))$$
$$- \{\text{int, float, string}\})\&v \in \text{Obj}_n$$
$$\text{entryV}(\text{obj}_n, \text{attr}_n, v) = \text{obindex}(v),$$

otherwise

$$\text{entryV}(\text{obj}_n, \text{attr}_n, v) = \text{undefined}.$$

For instance, entryV(diva, affiliation, compA) = obindex(compA) = $i \in$ Int, where Obj[i].name = compA.

Definition 16.10. Let initialattrV: $\text{Obj}_n \to \text{Obj}_n \times \cup_{k=1}^{\infty}(\text{DataV})^k$ be such that

$$\text{initialattrV}(\text{obj}_n) = \text{list of initial values of obj}_n \text{in the table where obj}_n \text{ is defined.}$$

For instance, initialattrV(compA) = (compA, "", −1).

Initial values of attributes are given in the following way:

int	0,
float	0.0,
string	"",
pointer	−1,
vecpointer	(−1, ..., −1).

Functions from Definitions 16.1 to 16.3 are defined on the arrays Obj : $N \to [\text{ALst}_{ob}, V_{ob}]$ and Class : $N \to [\text{ALst}_c, V_c]$. Functions from Definitions 16.4 to 16.7 are defined on the arrays on Class : $N \to [\text{ALst}_c, V_c]$ and Attr:N $\to [\text{ALst}_a, V_a]$. Functions from Definitions 16.8 to 16.10 are given as compositions of the functions from Definitions 16.1 to 16.7. The desired interpreter is constructed by the above functions.

The predicate

$$\langle\text{object name}\rangle := \text{newdbob}(\langle\text{db name}\rangle, \langle\text{class name}\rangle)$$

is implemented by the following two SQL commands:

sql1:create table ⟨class name⟩(attrlist(⟨class name⟩)),

sql2:insert into ⟨class name⟩(attrnamelist(⟨class name⟩)) values

(initialattrV(⟨object name⟩));

attrlist(⟨class name⟩) of sql1 yields the definition of a class. When these two commands are executed, a table like that in Fig. 16.4 is created in the database that represents the required class.

We will use the notation x = execsql(sql) to denote that x is obtained by execution of an SQL command sql where execsql:{SQL command} → DataV.

Let us consider the data assignment operation using the following typical case:

$$\langle\text{object name1}\rangle \to \langle\text{attr}_n 1\rangle := \langle\text{object name2}\rangle \to \langle\text{attrr}_n 2\rangle.$$

This predicate is implemented by the following two SQL commands:

sql3:select $\langle\text{attr}_n 2\rangle$ from cname(⟨object name2⟩) where_id = ⟨object name2⟩.

Let $x3$ = execsql(sql3). Suppose $\text{obj}_n = \text{objV}(\langle\text{object name2}\rangle, \text{attr}_n 2, x3) \in \text{Obj}_n$. Then, let

$$\text{sql4:update cname}(\langle\text{object name1}\rangle) \text{ set } \langle\text{attr}_n 1\rangle = \text{entryV}(\langle\text{object name1}\rangle, \langle\text{attr}_n 1\rangle, \text{obj}_n),$$

where $_id = \langle\text{object name1}\rangle$.

If $x4 = \text{execsql}(sql4)$ ($x4$ is a dummy in this case), the pointer of $\text{obj}_n = \langle\text{object name2}\rangle \rightarrow \langle\text{attrn2}\rangle$ is assigned to the attribute $\langle\text{attr}_n 1\rangle$ of the object $\langle\text{object name1}\rangle$ of the class cname($\langle$object name1$\rangle$).

According to the syntax, the following form is allowed for the left and right sides of the data assignment:

$$\langle\text{object name}\rangle \rightarrow \langle\text{attr}_n 1\rangle \rightarrow \cdots \rightarrow \langle\text{attr}_n k\rangle.$$

This case is implemented by recursive execution of sql3.

Suppose the assignment uses the following form:

$$\langle\text{object name}\rangle \rightarrow [].$$

In this case, the list of attributes of $\langle$object name$\rangle$ is first obtained as

$$[_\text{id}, \text{attr}_n 1, \ldots, \text{attr}_n k] = \text{attrnamelist}(\text{cname}(\langle\text{object name}\rangle)).$$

Then, $\langle$object name$\rangle \rightarrow$ [] is decomposed as a set of the following predicate:

$$\langle\text{object name}\rangle \rightarrow \text{attr}_n i.$$

Finally, the sql3 type command is repeatedly applied for each of the above predicates.

Let us consider implementation of a method. Let the following be a typical form:

$$X := \langle\text{object name}\rangle * m(-),$$

where m is the name of a method.

There are two problems with the implementation of m. First, because $m()$ is defined as a rule rather than as a function, a special mechanism must be devised to assign a computation result to X. Second, the constant object name "this" may be used in the definition of $m()$. Then, "this" must be properly replaced by $\langle$object name$\rangle$ during computation. In order to solve these problems, two special predicates, _THIS() and _RETURNVAL(), are introduced. The predicate _RETURNVAL() addresses the first problem, while _THIS() handles the second.

The above application of the method $m()$ is internally represented as

$$\text{is}(X, \text{action}(\langle\text{object name}\rangle, m(-))).$$

Then, the above predicate is executed by the following steps:

1. The value of Y of _THIS(Y) is saved onto a stack. The stack is necessary because $m()$ can call another method that may also use "this."

2. ⟨object name⟩ is inserted into _THIS(-) and a fact _THIS(⟨object name⟩) is created. (_THIS is the name of the fact.)
3. m(-) is executed by the interpreter of extProlog. For the execution "this" is replaced by ⟨object name⟩ in _THIS().
4. If m(-) has a predicate return(V), the value of V is inserted into _RETURNVALUE(-) and a fact _RETURNVALUE(V) is created.
5. V in _RETURNVALUE(V) is produced as output of the function action.
6. The value of Y saved at step 1 is returned into _THIS().
7. The predicate is() assigns the value of V to X.

According to the syntax, the following form is also allowed:

$$\langle\text{object name}\rangle * \text{method1}() * \cdots * \text{methodn}().$$

This form is implemented by recursive application of the function action. Q.E.D.

Appendix 16.2 Database-Handling Model in Computer-Acceptable Set Theory

The following illustrates a database-handling model in computer-acceptable set theory that implements the group_by function. When a database-handling model is compiled, a statement "? – ⟨model name⟩ main();" is automatically attached to the compiled output. In the current case ⟨model name⟩ = sqlex9.set.

```
/*sqlex9.set*/
/*definition of classes:see Fig. 16.5*/
class(_spjj,[
         "_sno    char(5)",
         "_pno    char(6)",
         "_jno    char(4)",
         "_qty    int"]);
class(_spj,[
         "_sno    _s",
         "_pno    _p",
         "_jno    _j",
         "_qty    int"]);

class(_s,[
         "_sno    char(10)",
         "_sname      char(20)",
         "_status     int",
         "_city   char(20)"]);
class(_p,[
         "_pno    char(10)",
         "_pname char(20)",
         "_color char(10)",
         "_weight     int",
         "_city   char(20)"]);
```

```
class(_j,[
         "_jno   char(10)",
         "_jname char(20)",
         "_city  char(20)"]);
/*start of model execution*/
sqlex2.setmain() <->
/*creation of objects*/
         db.g:="_hisdb",
         createOb1(),
         createOb2(),
         createOb3(),
/*insertion of data*/
         insertData1(),
         insertData2(),
         insertData3(),
/*implementation of group_by*/
         a.1();

createOb1() <->
         Hisdb:=_hisdb,
         _s1:=newdbob(Hisdb,_s),
         _s2:=newdbob(Hisdb,_s),
                •
                •
                •
         _j6:=newdbob(Hisdb,_j),
         _j7:=newdbob(Hisdb,_j);
createOb2() <->
         Hisdb:=_hisdb,
         _spj1:=newdbob(Hisdb,_spj),
         _spj2:=newdbob(Hisdb,_spj),
                •
                •
                •
         _spj23:=newdbob(Hisdb,_spj),
         _spj24:=newdbob(Hisdb,_spj);
createOb3() <->
         Hisdb:=_hisdb,
         _spjj1:=newdbob(Hisdb,_spjj),
         _spjj2:=newdbob(Hisdb,_spjj),
                •
                •
                •
         _spjj23:=newdbob(Hisdb,_spjj),
         _spjj24:=newdbob(Hisdb,_spjj);
insertData1() <->
         _s1->[]:=['_s1','Smith',20,'London'],
         _s2->[]:=['_s2','Jones',10,'Paris'],
                •
                •
                •
],
         _p5->[]:=['_p5','cam','Blue',12,'Paris'],
         _p6->[]:=['_p6','cog','Red',19,'London'];
insertData2() <->
         _j1->[]:=['_j1','sorter','Paris'],
```

```
                •
                •
                •
          _j7->[]:=['_j7','tape','London'];
insertData3() <->
          _spj1->[]:=[_s1,_p1,_j1,200],
          _spj2->[]:=[_s1,_p1,_j4,700],
                •
                •
                •
          _spj23->[]:=[_s5,_p5,_j4,400],
          _spj24->[]:=[_s5,_p6,_j4,500];
a.1() <->
          Ts:=extent_of(a.1.q1(SPJ)),
          Ys:=group_by(Ts,[_sno,_pno],a.1.q2(Y)),
          Xs:=joint_of([min(Ys->_sno),min(Ys->_pno),
              max(Ys->_qty)-min(Ys->_qty)]),
/*result is saved into "db_res.lib"*/
          appendf("db_res.lib","w",Xs);
a.1.q1(SPJ) <->
          in(SPJ,_spj);
a.1.q2(Y) <->
          sum(Y->_qty)>500;
```

References

Date, C. J. (1989) *A Guide to the SQL Standard: Second Edition*, Addison-Wesley.
Ishizuka, K. (1996) *Object-Oriented Database*, ASCII (in Japanese).

Part VI

Theoretical Basis for extProlog

17

extProlog as Logic Programming Language*

This chapter presents the theoretical base of extProlog. It is discussed as a logic programming language. It would therefore be convenient to start with the definition of logic programming; however, there is no formal definition of a logic programming language. It is commonly understood that a logic programming language is one that has the ultimate goal of providing clarity and declarativeness for programming. In general, a programming activity consists of two parts: design of the algorithm (heuristic) and its implementation in a language. The algorithm part can be further decomposed into a logic aspect and a control aspect. The logic aspect refers to the facts or data and the rules and requirements specifying what the algorithm does. The control aspect refers to how the algorithm can be implemented by arranging the rules and requirements in a particular order. The latter aspect becomes serious when the algorithm is modified.

In a common programming language like C, a user is required to determine the control aspect as well as the logic aspect, and hence that kind of language is called procedural. In a logic programming language, however, a user is required to specify only the logic part, and the control part is dealt with by the language. In other words, if a user declares a structure (in the sense of model theory) or an objective of a program, the logic programming language processor takes responsibility to realize the objective, although the responsibility may not be taken care of successfully.

It should be noted that extProlog can deal with a procedural program as well as a declarative one if required. For instance, the extProlog program for a simultaneous equation of Section 3.2.1 demonstrates this case. This flexibility is a big advantage for extProlog. However, because extProlog is required to behave as a logic programming language, its theoretical base lies in the theory of logic programming languages. This chapter discusses this aspect of extProlog.

17.1 Prolog as Logic Programming Language

Because Prolog is a logic programming language, its programs can represent logical truths about the world using first-order predicate calculus [Bridge, 1977]. Let us consider the following Prolog program, which is used as a working example of this chapter and Chapter 18:

```
/*testgpa.p*/
testgpa(X,Who):-var(X) and var(Who),!,xwriteln(0,"illegal goal form");
testgpa(X,Who):-grandparent(X,Who);                             /*r1*/
/*inference rule to find grandparent*/
/* parents(X,Y,Z) ↔ Y and Z are parents of X*/
/*parent(X,Y) ↔ Y is a parent of X*/
parent(X,Y):-parents(X,Y,Z);                                    /*r2*/
parent(X,Y):-parents(X,Z,Y);                                    /*r3*/
grandparent(X,G):- parent(X,P),parent(P,G);                     /*r4*/
/*database of parent-child relation*/
parents(wil,cha,dia);                                           /*r5*/
parents(hen,cha,dia);                                           /*r6*/
parents(cha,phi,eli);                                           /*r7*/
parents(dia,edw,fra);                                           /*r8*/
?- testgpa(wil,Who), xwriteln(0,Who, "is a grand parent of  ",wil);
```

The sentence grandparent(X, G):-parent(X, P), parent(P, G) presents a true definition of the concept of a grandparent, that is, G is a grandparent of X if some P is a parent of X and G is a parent of P. It must be noted that Prolog can express more than logical truths [Clocksin and Mellish, 1997]. The following are typical examples:

1. (Control information) Prolog can represent control information. Let us consider the following sentence:

   testgpa(X, Who):-var(X) and var(Who), !, xwriteln(0, "illegal goal form");

   var(X) does not say anything about a grandparent but refers to a state of affairs. The cut operation relates only to something about the proving process of a proposition.
2. Prolog can convert symbols to a character string such as a constant abc into "abc", a predicate to a list by the univ operation, and structures to clauses and so on. These operations violate predicate calculus propositions.
3. (Self-organization) Prolog can modify hypotheses using the assert operation. We will be in a position of having a different set of axioms at different points in the proof.
4. (Higher-order calculus) In Prolog, a logical variable is allowed to stand for a proposition appearing in an axiom. For instance, in the program

   ```
   implies(Assum, Concl):-
      asserta(Assum),
      call(Concl),
      retract(Concl);
   ```

   the variables Assum and Concl are supposed to represent propositions or predicates. This feature is reminiscent of what higher-order logic can provide.

In summary, Fig. 17.1 shows the position of Prolog relative to first-order predicate calculus. As Fig. 17.1 indicates, Prolog is a language that deals with a special class of sentences called universal formulae of Horn clauses as a part of logic programming. Although Prolog is not completely covered by predicate calculus, a theoretical basis of Prolog is examined using predicate calculus.

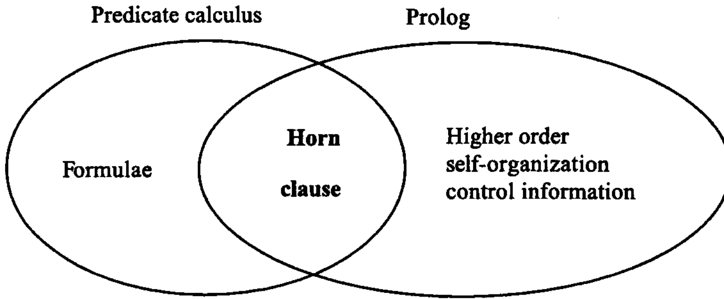


Fig. 17.1. Prolog and predicate calculus.

17.2 Predicate Calculus

In the subsequent discussion, predicate calculus (syntax) and model theory (semantics) are used. The calculus is necessary for a mechanical theorem proof, whereas model theory is used to show the validity of the proving process. As such, the arguments rely on the equivalence between predicate calculus and model theory. The equivalence is usually represented by the following completeness theorem [Bridge, 1977].

Completeness Theorem

A set of sentences $\Sigma \subset \mathrm{Sent}(L)$ is consistent iff Σ has a model.

In the standard definition, the inference rules of calculus are modus ponens and the generalization. However, as the subsequent discussion of Prolog shows, another inference rule, the resolution principle, is used. This chapter is based on the following assertion:

A set of universal formulae of Horn clauses S yields an empty clause by the resolution principle iff S does not have a model.

We will introduce several concepts and definitions to illustrate the above result. First-order predicate calculus was introduced in Chapter 2. This chapter uses a restricted form of the standard one.

Definition 17.1. Relational Structure
A relational structure is an ordered triplet

$$\mathrm{ST} = \langle A, \{r_i | i \in I\}, \{c_k | k \in K\}\rangle$$

with associated function $\lambda : I \rightarrow N^+$ such that

(i) A, the domain of ST, is a nonempty set,
(ii) I is a set such that each $i \in I \lambda(i)$ is a positive integer and r_i is a $\lambda(i)$-arity relation on A.
(iii) K is a set such that for each $k \in K c_k$, a distinguished element is an element of A.

Although functions are included in the standard definition of a relational structure (see Chapter 2), the above definition excludes functions because, in principle, Prolog deals only with predicates (relations).

Definition 17.2. First-Order Language
A first-order language L(ST) for the structure ST $= \langle A, \{r_i | i \in I\}, \{c_k | k \in K\}\rangle$ consists of

(1) individual variables $V_0, V_1, \ldots$;
(2) individual constant symbol c_k for each $k \in K$;
(3) a $\lambda(i)$-arity predicate symbol r_i for each $i \in I$;
(4) logical connectives $\neg$ (not) and & (and);
(5) universal quantifier $\forall$;
(6) parentheses (,).

Formally, variable symbols are $V_0, V_1, \ldots$. However, for the sake of readability, Prolog variable symbols will also be used in addition to the formal ones. This convention is also used for constants.

Definition 17.3. Term
The set of terms of the first-order language L, Term(L), is the smallest set X such that all individual variables $V_0, V_1, \ldots$ and constant symbols c_k are members of X.

Definition 17.4. Atomic Formulae
The set of atomic formulae of L, Atom(L), consists of all elements of the form $r_i(t_1, \ldots, t_{\lambda(i)})$, where $t_1, \ldots, t_{\lambda(i)} \in \text{Term}(L)$.

An atomic formula $r_i(t_1, \ldots, t_{\lambda(i)})$ or its negation $\neg r_i(t_1, \ldots, t_{\lambda(i)})$ is called a literal. The working example has an expression parents(wil, char, dia). Here parents is a ternary predicate symbol and parents(wil, char, dia) is an atomic formula, in which wil, char, and dia are constant symbols.

Definition 17.5. Well-Formed Formula
The set of well-formed formulae (or simply formulae) of L, Form(L), is the smallest set Y such that

(i) Atom$(L) \subset Y$,
(ii) if $\phi, \psi \in Y$, then $\neg\phi, \phi \& \psi, \forall V_i \phi \in Y$.

The following expressions on the left-hand side of the equal sign are also used for those on the right-hand side:

$$A \vee B = \neg(\neg A \& \neg B);$$

$$A \to B = \neg A \vee B;$$

$$A \leftrightarrow B = A \to B \& B \to A;$$

$$\exists V_i \phi = \neg(\forall V_i \neg \phi). \quad (\exists \text{ is called the existential quantifier.})$$

Definition 17.6. Scope
The scope of the quantifier $\forall V_i$ (or $\exists V_i$) is a subformula that extends to the end of the enclosing formula, or to the next unmatched right parenthesis if the latter comes first.

For instance, in the formula

$$\forall V_1(\forall V_2(\text{parent}(V_1, V_2) \vee \neg\exists V_3\text{parents}(V_1, V_2, V_3))),$$

the scope of $\forall V_1$ is $(\forall V_2(\text{parent}(V_1, V_2)\vee\neg\exists V_3\text{parents}(V_1, V_2, V_3)))$, the scope of $\forall V_2$ is $(\text{parent}(V_1, V_2) \vee \neg\exists V_3\text{parents}(V_1, V_2, V_3))$, and the scope of $\exists V_3$ is $\text{parents}(V_1, V_2, V_3)$.

A quantifier binds all occurrences of its associated variable in its scope. They are then called **bound** variables. A variable that is not bound by any quantifier is called a **free** variable.

A formula that contains at least one free variable is called an **open formula**. A formula that is not an open formula is called a **closed formula** or a **sentence**.

In the formula $\neg\exists V_3\text{parents}(V_1, V_2, V_3)$, V_3 is bound while V_1 and V_2 are free. In $\forall V_1(\forall V_2(\text{parent}(V_1, V_2) \vee \neg\exists V_3\text{parents}(V_1, V_2, V_3)))$, V_1, V_2, and V_3 are bound and hence it is a sentence.

Definition 17.7. Denotation
Let $\text{ST} = \langle A, \{r_i | i \in I\}, \{c_k | k \in K\}\rangle$ be given. Let $\mathbf{a} = (a_0, a_1, \ldots)$ be an infinite sequence of elements of A. The denotation of a term t in $L(\text{ST})$ with respect to $\mathbf{a}$, $t[\mathbf{a}]$, is defined as follows:

(i) if t is V_i, then $t[\mathbf{a}] = a_i$;
(ii) if t is c_k, then $t[\mathbf{a}] = c_k$.

Here $\mathbf{a} = (a_0, a_1, \ldots)$ is called an interpretation of the variable symbols. An interpretation is dealt with as a function like $\mathbf{a}(V_i) = a_i$.

Definition 17.8. Satisfaction
That an interpretation $\mathbf{a} = (a_0, a_1, \ldots)$ satisfies a formula ϕ in $L(\text{ST})$, which will be denoted by $\text{ST} \models_a \phi$, is defined recursively:

(i) $\text{ST} \models_a r_i(t_1, t_2, \ldots)$ iff $(t_1[\mathbf{a}], t_2[\mathbf{a}], \ldots) \in r_i$;
(ii) $\text{ST} \models_a \neg\phi$ iff it is not the case that $\text{ST} \models_a \phi$;
(iii) $\text{ST} \models_a \phi\&\varphi$ iff $\text{ST} \models_a \phi$ and $\text{ST} \models_a \varphi$;
(iv) $\text{ST} \models_a \forall V_i\phi$ iff for any $b \in A$ $\text{ST} \models_{a(b/i)} \phi$;

where $\mathbf{a}(b/i) = (a_0, a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots)$.

Definition 17.9. Model
Let ST be a realization for a first-order language L in which every constant symbol and every predicate symbol is defined as a constant and a relation in ST, respectively.

(i) $\phi \in \text{Form}(L)$ is valid (or true) in ST just in case $\text{ST} \models_a \phi$ for all sequences $\mathbf{a}$. When ϕ is valid in ST, we say ST is a model for ϕ.
(ii) $\phi \in \text{Form}(L)$ is universally valid if it is valid for all realizations for L.

(iii) $\phi \in \text{Form}(L)$ can be satisfied if for some realization ST2 for L and some sequence **b** in the domain of ST2, ST2 $\models_b \phi$.
(iv) $\phi \in \text{Form}(L)$ is refutable if $\neg\phi$ can be satisfied.

Note that a sentence σ is valid in a structure ST, or ST is a model of σ, iff it can be satisfied in ST.

Definition 17.10. Logical Implication
If ϕ and φ are sentences in a first-order language L, then ϕ logically implies φ, i.e., $\phi \models \varphi$ if whenever a structure ST is a realization of L such that ST $\models \phi$, then ST $\models \varphi$. If $\phi \models \varphi$ and $\varphi \models \phi$, ϕ and φ are said to be logically equivalent, i.e., $\phi \equiv \varphi$.

The following are important logical equivalence relations: for formulae f and g,

$$\neg\forall x f \equiv \exists x \neg f,$$
$$\neg\exists x f \equiv \forall x \neg f,$$
$$\forall x \forall y f \equiv \forall y \forall x f,$$

and

$$\forall x(f \& g) \equiv \forall x f \& \forall x g.$$

17.3 Special Forms

17.3.1 Prenex Form

As mentioned above, Prolog uses a special formula, the universal formula of Horn clauses. We first introduce a standard form of formula, called conjunctive formula.

A formula is in the prenex form if all the quantifiers appear at the left of the formula and the scope of each is the remainder of the formula. The string of quantifiers is called the prefix, while the remainder of the formula is called the matrix:

$$\forall V_1(\forall V_2(\text{parent}(V_1, V_2) \vee \neg\exists V_3 \text{parents}(V_1, V_2, V_3)))$$

is not in prenex form, while

$$\forall V_1(\forall V_2(\forall V_3(\text{parent}(V_1, V_2) \vee \neg\text{parents}(V_1, V_2, V_3))))$$

is in prenex form. Also, $\forall V_1 \forall V_2 \forall V_3$ is a prefix, whereas $\text{parent}(V_1, V_2) \vee \neg\text{parents}(V_1, V_2, V_3)))$ is a matrix.

17.3.2 Universal Formula and Conjunctive Normal Formula

A universal formula is a formula in the prenex form in which the prefix contains only universal quantifiers, and the matrix is a conjunction of disjunctions of literals. A conjunction of disjunctions of literals is called a conjunctive normal formula. If we are allowed to use function symbols, a formula is converted into a universal formula by the following steps:

1. Removing implications by the definition $A \rightarrow B = \neg A \vee B$.
2. Moving negation inwards by De Morgan's law.
3. Skolemizing.
4. Moving universal quantifiers outward.
5. Distributing & over $\vee$.
6. Putting into clauses, where each disjunction in a conjunctive normal form is called a **clause**.

Let us consider a simple example. Let a formula be $F = \phi \rightarrow \neg(\neg\psi \vee \chi)$. Then,

$$\begin{aligned} F &\Rightarrow \neg\phi \vee \neg(\neg\psi \vee \chi) && \text{(step 1)} \\ &\Rightarrow \neg\phi \vee (\psi \& \neg\chi) && \text{(step 2)} \\ &\Rightarrow (\neg\phi \vee \psi)\&(\neg\phi \vee \neg\chi) && \text{(step 5).} \end{aligned}$$

Step 3 needs an explanation. Skolemizing addresses an existential quantifier. If a formula has an existential quantifier at this step, variables in the scope of the quantifier are replaced by Skolem constant symbols, which are new symbols that are not used elsewhere. If the existential quantifier is in the scope of a universal quantifier, variables associated with the existential quantifier are replaced by function symbols to express how what exists depends on what variables are chosen to stand for. Because the formulation of this chapter does not allow function symbols, we are concerned with a formula that does not require Step 3.

17.3.3 Horn Clauses

A clause with at most one unnegated literal is called a Horn clause. For example, $\neg\phi \vee \psi$ of the example of Section 17.3.2 is a Horn clause. A Prolog program itself is a universal formula of Horn clauses. In fact, as explained below, the working example testgpa.p is a representation of a universal formula:

$$\begin{aligned} &(\forall X, \text{Who}, \ldots, \text{Who9})((\text{testgpa}(X, \text{Who}) \vee \neg\text{var}(X) \vee \neg\text{var}(\text{Who}) \\ &\quad \vee \neg ! \vee \neg\text{xwriteln}(0, \text{“illegal goal form”})) \\ &\quad \&(\text{testgpa}(X2, \text{Who2}) \vee \neg\text{grandparent}(X2, \text{Who2}))\& \cdots \&(\neg\text{testgpa}(\text{wil}, \text{Who9}) \\ &\quad \vee \neg\text{xwriteln}(0, \text{Who9}, \text{“is a grandparent of”}, \text{wil})). \end{aligned}$$

There are three cases for Horn clauses in a Prolog program.

(i) A clause with no unnegated literal:

$$\neg Q1 \vee \neg Q2 \vee \cdots \vee \neg Qn \equiv \neg(Q1 \& Q2 \& \cdots \& Qn).$$

This form will be used for a goal, and in a Prolog program it is expressed as

$$?\text{-}(Q1, Q2, \ldots, Qn),$$

where "&" is replaced by ",".

?-testgpa(wil, Who), xwriteln(0, Who, "is a grandparent of", wil)

is an example of this case.
(ii) A clause with one unnegated literal:

$$P1 \vee \neg Q1 \vee \neg Q2 \vee \cdots \vee \neg Qn \equiv P1 \vee \neg(Q1 \& Q2 \& \cdots \& Qn)$$
$$\equiv P1 \leftarrow (Q1 \& Q2 \& \cdots \& Qn).$$

This form will be used for rules, and in a Prolog program it is expressed as

$$P1\text{:- } (Q1 \& Q2 \& \cdots \& Qn).$$

parent(X, Y):- parents(X, Y, Z) is an example of this case.
(iii) A clause consisting of one unnegated literal:

$$P1.$$

This form will be used for a fact. The predicate parents(wil, cha, dia) is an example of this case.

17.4 Theorem Proving in Prolog

A Prolog program Σ is a universally quantified conjunction of clauses, i.e., $\Sigma = \forall x P(x)$, where $P(x)$ is a conjunction of clauses. The task in Prolog is to decide whether Σ logically implies a goal list $g_1 \& \cdots \& g_k$, where each g_i is an atom, and the list is assumed to be existentially quantified. That is, we want to know whether

$$\forall x P(x) \text{ logically implies } \exists y g_1(y) \& \cdots \& g_k(y),$$

or

$$\forall x P(x) \models \exists y g_1(y) \& \cdots \& g_k(y).$$

This question is transformed into the following form.
Using the relations

$$(\text{ST} \models \forall x P(x) \text{ implies ST } \models \exists y g_1(y) \& \cdots \& g_k(y) \text{ for any structure ST})$$
$$\text{iff ST} \models \neg\forall x P(x) \vee \exists y g_1(y) \& \cdots \& g_k(y) \text{ for any structure ST}$$
$$\text{iff not the case that ST } \models \forall x P(x) \text{ or ST} \models \exists y g_1(y) \& \cdots \& g_k(y)$$
$$\text{for any structure ST,}$$

the above question is transformed into the following form:

$$\forall x P(x) \models \exists y g_1(y) \& \cdots \& g_k(y)$$
$$\text{iff } \models \forall x P(x) \rightarrow \exists y g_1(y) \& \cdots \& g_k(y)$$

$$\text{iff } \neg(\forall x P(x)) \vee \exists y g_1(y) \& \cdots \& g_k(y) \text{ is universally valid}$$

$$\text{iff } \neg(\neg(\forall x P(x)) \vee \exists y g_1(y) \& \cdots \& g_k(y)) \text{ is unsatisfiable}$$

$$\text{iff } (\forall x P(x)) \& \forall y(\neg g_1(y) \vee \cdots \vee \neg g_k(y)) \text{ is unsatisfiable}$$

$$\text{iff } (\forall x \forall y)(P(x) \& (\neg g_1(y) \vee \cdots \vee \neg g_k(y)) \text{ is unsatisfiable.}$$

That is, the implication of a goal list $g_1 \& \cdots \& g_k$ by a Prolog program Σ can be reduced to unsatisfiability of a universal formula. If all structures could be enumerated, we could test unsatisfiability, although this is not practical. Let us consider a practical way.

Suppose f is a matrix of a conjunctive normal form, i.e.,

$$f = C1 \& C2 \& \cdots \& Cn (= P(x) \& (\neg g_1(y) \vee \cdots \vee \neg g_k(y)),$$

where Ci is a clause.

Let

$$S = \{C1, \ldots, Cn\}.$$

Let

$$D = \text{arbitrary domain or universe,}$$

$$\text{Con}_f = \text{the set of constants in } f,$$

$$C_D : \text{Con}_f \to D.$$

Let

$$B_{f,D,C_D} = \text{base of } f$$

be constructed as follows:

Take any atom A from f. Form a new atom by replacing each constant a in the argument list of A by $C_D(a)$, and replacing each variable in the argument list by any member of the domain D.

Let us consider the working example. In the example let

$C_1(V1, V2) =$ testgpa($V1$, $V2$):- var($V1$) and var($V2$), !,
xwriteln(0, "illegal goal form"),

$C_2(V3, V4) =$ testgpa($V3$, $V4$):- grandparent($V3$, $V4$),

$C_3(V5, V6, V7) =$ parent($V5$, $V6$):- parents($V5$, $V6$, $V7$),

$C_4(V8, V9, V10) =$ parent($V8$, $V9$):- parents($V8$, $V10$, $V9$),

$C_5(V11, V12, V13) =$ grandparent($V11$, $V12$):- parent($V11$, $V13$),
parent($V13$, $V12$),

$C_6 =$ parents(wil, cha, dia),

$C_7 =$ parents(hen, cha, dia),

$C_8 =$ parents(cha, phi, eli),

$C_9 =$ parents(dia, edw, fra),

$C_{10}(V14)$ =?-testgpa(wil, $V14$), xwriteln(0, $V14$, "is a grandparent of", wil).

Con_f = {wil, 0, "illegal goal form", "is a grandparent of", cha, dia, hen, phi, eli, edw, fra}.

Let $D = \text{Con}_f$ and let C_D be the identity mapping. Then, testgpa(wil, 0) $\in B_{f,D,C_D}$.

If $D = \text{Con}_f$ and C_D is the identity mapping, Con_f and B_{f,D,C_D} are called the Herbrand universe and Herbrand base, respectively. The predicate testgpa(wil, 0) is an element of the Herbrand base.

Definition 17.11. Herbrand Interpretation
Suppose f is a universal formula such as $\forall x \forall y P(x)\&(\neg g_1(y) \vee \cdots \vee \neg g_k(y))$. Let the matrix of f be $C_1 \& \cdots \& C_k$. Let $S = \{C_1, \ldots, C_k\}$. Let $D = \text{Con}_f$ and $B_{\text{f, Con}_\text{f}, \text{ ID}}$ = Herbrand base.

Then, a Herbrand interpretation for f or S is

$$\text{Herbrand interpretation} = \langle \text{Con}_f, \text{ID}, TA \rangle,$$

where

$$\text{ID} = \text{identify mapping}$$

and

$$\text{TA} = \text{truth assignment for } B_{\text{f, Con}_\text{f}, \text{ ID}}, \text{ i.e., TA} : B_{\text{f, Con}_\text{f}, \text{ ID}} \rightarrow \{\text{true, false}\}.$$

In fact, a Herbrand interpretation is a structure $\langle \text{Con}_f, \{r_i | i \in I\}, \text{Con}_f \rangle$, where $\{r_i | i \in I\}$ is specified by TA. Then, the concepts of satisfaction and model are applicable to a Herbrand interpretation.

Let us consider the working example. Let

$$f = \forall \underline{V}(C_1 \& C_2 \& \cdots \& C_{10}),$$

where $\underline{V} = (V1, \ldots, V14)$.

Because the set of predicates (relations) is

{testgpa(), testgpa(wil,), var(), !, xwriteln(), grandparent(), parent(), parents(), parents(wil, cha, dia), parents(hen, cha, dia), parents(cha, phi, eli), parents(dia, edw, fra)},

$B_{\text{f, Con}_\text{f}, \text{ ID}}$ = {testgpa(wil, wil), ..., parents(fra, fra, fra), ...}.

If TA : $B_{\text{f, Con}_\text{f}, \text{ ID}} \rightarrow$ {true, false} is specified as TA(parents(wil, cha, dia)) = false (or (wil, cha, dia) $\notin$ parents), $\langle \text{Con}_f, ID, TA \rangle$ cannot be a model of f.

Definition 17.12. Ground Instance of Clause in f
A ground instance of a clause in f is a clause that is constructed by replacements of variables from the Herbrand universe. That is, a ground instance of a clause is constructed by denotation of the clause in a Herbrand interpretation of f.

For the above example, $C2(\text{wil}, \text{wil}) = \text{testgpa}(\text{wil}, \text{wil}) \vee \neg\text{grandparent}(\text{wil}, \text{wil})$ is a ground instance.

Theorem 17.1. *For a universal formula f, if there exists a structure $ST = \langle A, \{r_i\}, \{c_k\}\rangle$ that is a model for f, then there is a Herbrand interpretation $H = \langle Con_f, ID, TA_H\rangle$ that is a model for f.*

Proof. Refer to Appendix 17.1.

Theorem 17.2. *Let S be a set of clauses. Let H be a Herbrand interpretation of S. Let G be the set of ground instances of clauses in S. Then*

$$H \models S \text{ iff } H \models G.$$

Proof. Refer to Appendix 17.2.

Theorem 17.3. (Herbrand's Theorem) *A clause set S is unsatisfiable iff G is unsatisfiable.*

Proof. Refer to Appendix 17.3.

17.5 Theorem Proving by Resolution Principle

As mentioned above, in order to prove the relation $\forall x P(x) \models \exists y g_1(y), \ldots, g_k(y)$, which is a task of the Prolog interpreter, unsatisfiability of a universal formula $(\forall x \forall y)(P(x)\&(\neg g_1(y) \vee \cdots \vee \neg g_k(y))$ has only to be shown. Then, Theorem 17.3 states that unsatisfiability of G, the set of ground instances of clauses in $(P(x)\&(\neg g_1(y) \vee \cdots \vee \neg g_k(y))$, has to be shown. This theorem proving must be formally performed by a Prolog machine.

In predicate calculus, formal theorem proving is usually achieved by applying the two inference rules modus ponens and generalization to axioms. Because this method is not efficient for a computer, another inference rule called the resolution principle is used for Prolog [Maier and Warren, 1988].

Suppose that S is a set of clauses of propositions. The resolution principle is stated as follows:

Definition 17.13. Resolution Principle
The resolution rule takes two clauses from S such that one contains a literal and the other contains its complement. It constructs the new clause C, called the resolvent, by taking the union of the two clauses and deleting the complementary pair of the literals.

The most important formal property of the resolution principle is that if (and only if) a set of clauses is unsatisfiable, they will yield an empty clause (refutation) via resolution. This property is called **refutation complete**.

We will illustrate refutation completeness for a conjunctive form of propositions using the working example.

Let

$$a = \text{pa(wil, cha)},$$
$$b = \text{pas(wil, cha, dia)},$$
$$c = \text{gpa(wil, phi)},$$
$$d = \text{pa(cha, phi)},$$
$$e = \text{pas(cha, phi, eli)}.$$

Let $\Sigma' = \{C_1, C_2, C_3, C_4, C_5\}$ be

$$C_1 : a \leftarrow b;$$
$$C_2 : c \leftarrow a, d;$$
$$C_3 : d \leftarrow e;$$
$$C_4 : b;$$
$$C_5 : e;$$

Let the goal be

$$c;$$

In order to show $\Sigma' \models c$,

$$\Sigma = \Sigma' \& \neg c \text{ is unsatisfiable}$$

has to be shown. Let

$$C_6 : \neg c;$$

Then $\underline{\Sigma} = \{C_1, C_2, C_3, C_4, C_5, C_6\}$ is a set of clauses of the proposition. Because $B_{\text{f, Conf, ID}} = \{a, b, c, d, e\}$, Σ has $2^5 = 32$ Herbrand interpretations. Each interpretation is characterized by TA,

$$\text{TA} : \{a, b, c, d, e\} \rightarrow \{\text{true, false}\}.$$

For instance, one interpretation is

$$\text{TA}(a) = \text{true},$$
$$\text{TA}(b) = \text{false},$$
$$\text{TA}(c) = \text{true},$$
$$\text{TA}(d) = \text{false},$$
$$\text{TA}(e) = \text{true}.$$

This interpretation will be expressed as $(a, \neg b, c, \neg d, e)$. Let us represent the set of interpretations in a systematic way using a tree structure. Figure 17.2 shows a binary tree on $\{a, b\}$.

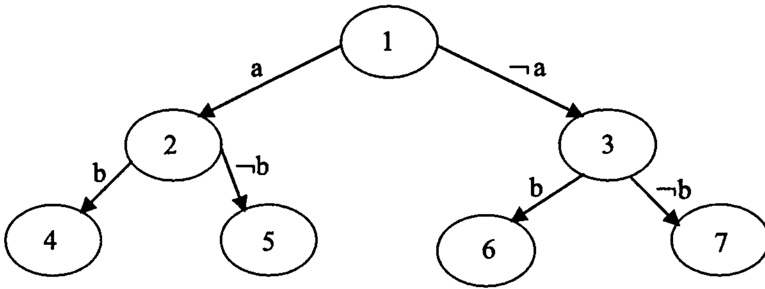


Fig. 17.2. Semantic tree on $\{a, b\}$.

In Fig. 17.2, each path from the root node to a leaf node represents a Herbrand interpretation on $\{a, b\}$. For example, the path to node 5 corresponds to the interpretation $(a, \neg b)$. Because the tree lists all the interpretations, it is called a **semantic tree.**

Let us consider the semantic tree Ts on $\{a, b, c, d, e\}$. The tree shown in Fig. 17.3 is a subtree of Ts.

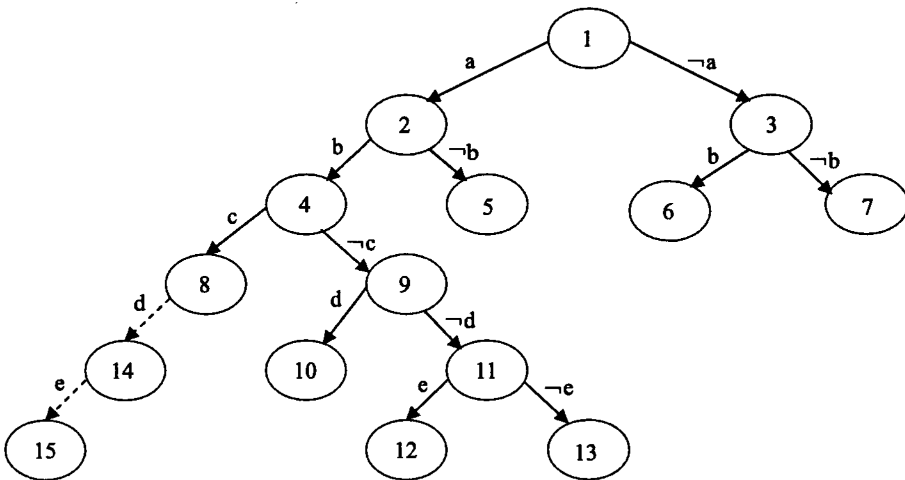


Fig. 17.3. Failure tree.

There are 32 paths in Ts corresponding to the 32 Herbrand interpretations. Suppose Σ is unsatisfiable. We will show that the resolution principle produces an empty clause.

Because no path of Ts can be a model of Σ, one of the clauses $C_1, \ldots, C_6$ should not hold on any path. For example, on the path (a, b, c, d, e) from node 1 to node 15 of Fig. 17.3, C_6 cannot hold because $\neg c$ does not hold on the path.

Let us call a node i in Ts a **failure node** for a clause C if the path from the root node to i contains the complement of every literal in C, and no ancestor of the node

i has this property. For example, node 8 of Fig. 17.3 is a failure node for C_6 because it contains the complement of every literal of C_6, i.e., $\neg c$ but nodes 4, 2, and 1 do not have this property.

A semantic tree is called a failure tree if every path contains a failure node. It is clear that when drawing a failure tree, to check whether a path can be a model in Σ, we can prune off the branches below the failure node so that the failure nodes are actually the leaves. For instance, in Fig. 17.3 we do not have to list nodes 14 and 15, so that node 8 is a leaf node. The tree of Fig. 17.3 is derived from Ts in this way.

Let us call a node i a **resolution node** in Σ if both its children are failure nodes. For instance, nodes 3 and 11 of Fig. 17.3 are resolution nodes. (Node 3 is a resolution node because C_4 and C_1 do not hold for $(\neg a, \neg b)$ and $(\neg a, b)$, respectively.)

Because the tree under consideration has a finite number of nodes, a nonempty tree has a resolution node. If there is no resolution node, there should be an infinite number of nodes.

Let us consider resolution node 11. Because nodes 12 and 13 are failure nodes, there must be two clauses, C and C', that fail at them, respectively. Actually, $C = C_3$ and $C' = C_5$. Because C fails at node 12 and does not fail at node 11, C should contain $\neg e$ as its literal. Similarly, C' contains e as its literal. Let $C = \neg e \vee s_1 \vee \cdots \vee s_p$ and $C' = e \vee t_1 \vee \cdots \vee t_q$. Then, if the resolution principle is applied to C and C' with respect to $\neg e$ and e, the resolvent is $C'' = s_1 \vee \cdots \vee s_p \vee t_1 \vee \cdots \vee t_q$. Actually, $C'' = d$. Because node 12 is a failure node for $C, s_1 \vee \cdots \vee s_p$ must fail on the path from the root to resolution node 11. The same is true for $t_1 \vee \cdots \vee t_q$. Then the resolvent must fail by node 11. Consequently, we have a reduced failure tree for $\Sigma \cup \{C''\}$, in which node 11 or its ancestor is a failure leaf node. Notice that $\Sigma \models C''$. Figure 17.4 shows the real new failure tree.

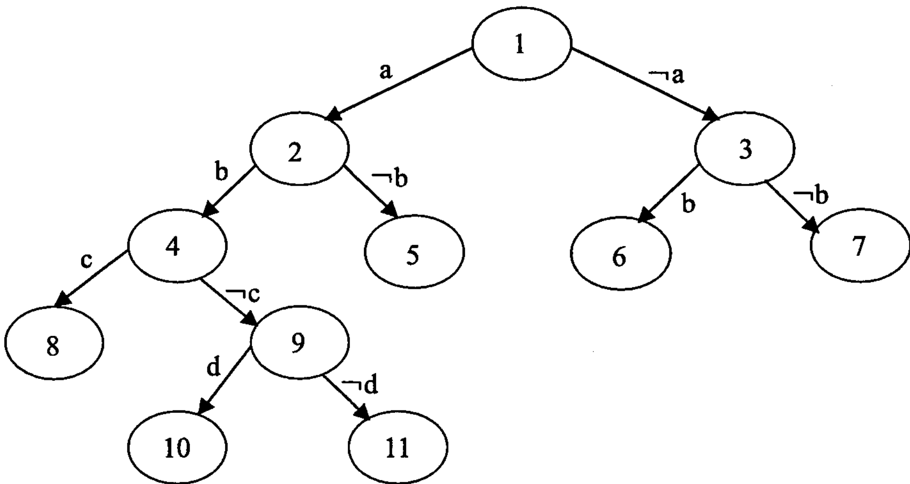


Fig. 17.4. Reduced failure tree.

Nodes 9 and 3 are resolution nodes in the new tree. If the same procedure is repeated, a reduced failure tree (Fig. 17.5) is obtained for the set of clauses $\Sigma \cup \{d\} \cup \{c \vee \neg a\} \cup \{\neg a\} \cup \{a\}$. Then, the final resolution to resolution node 1 produces an empty clause.

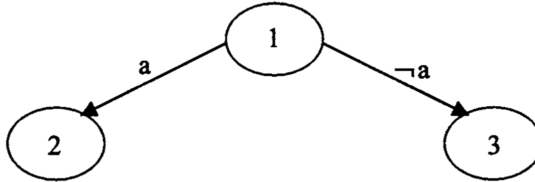


Fig. 17.5. Semifinal reduced failure tree.

Because the set of ground instances of clauses G of $(\forall x \forall y)(P(x)\&(\neg g_1(y) \vee \cdots \vee \neg g_k(y))$ is a conjunctive form of propositions, we have that if the clause set S of G is unsatisfiable, we can obtain the empty clause by resolution on G. If we want to use this result directly, we may have to generate a large set G. Fortunately, we can avoid this situation using the lifting lemma mentioned below [Maier and Warren, 1998].

In order to explain the lemma, we must introduce the concept of a unifier. The unifier is a type of substitution operation that replaces a variable by another variable as well as by a constant. For instance, let us consider the following clause:

$$C_2(V3, V4) = \text{testgpa}(V3, V4) \vee \neg\text{grandparent}(V3, V4).$$

Let a unifier be $\theta = \{V3 = \text{wil}, V4 = \text{Who}\}$, where $V3$ and $V4$ are to be replaced by wil and Who, respectively. (Note that Who is a variable.) Then, it produces

$$C_2\theta = \text{testgpa(wil, Who)} \vee \neg\text{grandparent(wil, Who)}.$$

Let another clause be

$$C_{10}(V4) = \neg\text{testgpa}(\text{wil}, V4) \vee \neg\text{xwriteln}(0, V4, \text{“is a grandparent of”}, \text{wil}).$$

Then,

$$C_{10}\theta = \neg\text{testgpa(wil, Who)} \vee \neg\text{xwriteln(0, Who, “is a grandparent of”, wil)}.$$

Using a unifier, the definition of the resolution principle can be extended. For the current example, ¬grandparent(wil, Who)∨¬xwriteln(0, Who, “is a grandparent of”, wil) is a resolvent of C_2 and C_{10}.

Armed with the above concepts, the lifting lemma can be stated.

Lifting Lemma

If clauses C and D have instances C' and D', respectively, and E' is a resolvent of C' and D', then C and D have a resolvent E such that E' is an instance of E.

The lifting lemma states that we can "lift" those ground relations on predicate clauses and avoid generating ground instances.

Prolog uses the resolution principle for a mechanical theorem-proving inference rule. However, although the resolution principle tells us how to derive a consequence from two clauses, the principle does not tell us how to decide which clauses to look at next or which literals to match. In other words, it is difficult for the principle to find a proper sequence of steps. In fact, as Chapter 18 shows, the Prolog interpreter of this book adopts the depth-first search method, a kind of brute-force method to check every possible resolution.

The Prolog interpreter of this book also adopts the strategy of linear input resolution; that is, at each stage, we resolve the clause last obtained with one of the original hypotheses. In Prolog, the literal to be matched is always the first one in the goal clause, and the new goals are placed at the front of the goal clause.

The Prolog depth-first strategy has a problem in that it can get into a "loop" and hence never follow up some of the alternatives. For example, let us consider the unification of the following clauses [Clocksin and Mellish, 1997]:

$$\text{equal}(X, X);$$

$$?\text{-equal}(\text{foo}(Y), Y);$$

Then, we have

$$Y = \text{foo}(Y) = \text{foo}(\text{foo}(Y)) = \cdots .$$

A value for Y is never obtained.

Appendix 17.1 Proof of Theorem 17.1

We must show that if ST $\models f$, then there is a choice for TA_H such that $H \models f$. Let $r_i(c_1, c_2, \ldots, c_n)$ be any atom in the Herbrand base of f, where each c_i is a constant. Let

$$\text{TA}_H(r_i(c_1, c_2, \ldots, c_n)) = \text{true} \leftrightarrow (C(c_1), C(c_2), \ldots, C(c_n)) \in r_i,$$
$$\text{where } C(c_k) = c_k.$$

The formula f can be written as $\forall \underline{X} g$, where $\forall \underline{X}$ signifies a universal quantifier for each variable in g. Thus, ST $\models f$ iff ST $\models_I g$ for every interpretation I. Similarly, to show that $H \models f$, we must show that for every interpretation I, $H \models_I g$.

Consider any interpretation I that maps variables to the Herbrand universe, Con_f. Consider the function $I' = CI$. I' maps variables to values in A, so I' is an interpretation relative to ST. Thus, ST $\models_{I'} g$. Consider any atom

$$r_i(t_1, \ldots, t_n)$$

in g. We have

$$H \models_I r_i(t_1, \ldots, t_n) \leftrightarrow H \models r_i(c_1, \ldots, c_n),$$

where $c_i = ID(t_i)$ if t_i is a constant, and $c_i = I(t_i)$ if t_i is a variable. Also

$$\mathrm{ST} \models_{I'} r_i(t_1, \ldots, t_n) \leftrightarrow \mathrm{ST} \models r_i(d_1, \ldots, d_n),$$

where $d_i = C(t_i) = C(c_i)$ if t_i is a constant and $d_i = I'(t_i) = C(I(t_i)) = C(c_i)$ if t_i is a variable, because $I' = CI$. We have that $d_i = C(c_i)$ for every i. Therefore

$$\mathrm{ST} \models_{I'} r_i(t_1, \ldots, t_n) \leftrightarrow \mathrm{ST} \models r_i(C(c_1), \ldots, C(c_n))$$
$$\leftrightarrow H \models r_i(c_1, \ldots, c_n) \leftrightarrow \mathrm{ST} \models_{I'} r_i(t_1, \ldots, t_n).$$

Because $\mathrm{ST} \models_{I'} r_i \leftrightarrow H \models_I r_i$ for every atom r_i in g, $H \models_I g$ and hence, because I is arbitrary, $H \models f$. Q.E.D.

Appendix 17.2 Proof of Theorem 17.2

We prove that any Herbrand model for S is a model for G and vice versa. First note that S and G have the same set of Herbrand interpretations. They have the same set of constants and the same Herbrand bases. Let H be a Herbrand model for S. Let C_G be a clause in G. There must be a clause C_S in S such that C_G is a ground instance of C_S. Let I be an interpretation such that $I(C_S) = C_G$. We know that $H \models S$, hence $H \models \forall \underline{X} C_S$, $H \models_I C_S$. However,

$$H \models_I C_S \leftrightarrow H \models I(C_S) \leftrightarrow H \models C_G,$$

so $H \models C_G$. Because C_G is arbitrary, $H \models G$; hence H is a model for G.

Similarly, suppose H is a Herbrand model for G. Let C_S be any clause in S. We want to show that $H \models \forall \underline{X} C_S$, where $\underline{X}$ is the set of all variables in C_S. For any interpretation I, $I(C_S)$ is some clause C_G in G. So

$$H \models_I C_S \leftrightarrow H \models I(C_S) \leftrightarrow H \models C_G.$$

Hence, H makes every clause in S true, and thus is a model for S. Q.E.D.

Appendix 17.3 Proof of Theorem 17.3

Suppose S is unsatisfiable. If G is satisfiable, Theorem 17.1 implies that G has a Herbrand interpretation. Then, Theorem 17.2 implies that S also has a Herbrand interpretation, which is a contradiction. Suppose G is unsatisfiable. If S is satisfiable, S has a Herbrand interpretation, and hence G has a Herbrand interpretation, which is a contradiction. Q.E.D.

References

Clocksin, W. F. and Mellish, C. S. (1997) *Programming in Prolog*, Springer.
Bridge, J. (1977) *Beginning of Model Theory*, Oxford University Press.
Maier, D. and Warren, D. S. (1988) *Computing with Logic*, Benjamin.

18

Implementation of extProlog*

An extProlog program is executed by the extProlog interpreter. This chapter discusses how the interpreter is designed for extProlog. Chapter 17 indicated that the main functions of a Prolog interpreter are unification and backtracking. A Prolog program is executed by these functions. Then, it is natural to guess that a generalization of a push-down automaton (PDA) can be a model for the interpreter, where the push-down stack of a PDA can handle the backtracking while the head can perform unification, provided an appropriate Prolog database for matching is supplied [Takahara and Iijima, 1990]. The Prolog interpreter of this book is constructed as a generalized PDA with a Prolog database.

18.1 Implementation of extProlog: Generalized PDA (Push-Down Automaton) Model

The following program is used in Chapter 17. It is again used as an example for this chapter.

Example 18.1.

```
/*testgpa.p*/
testgpa(X,Who):-gpa(X,Who);                          /*r1*/
/*inference rule to find grandparent*/
pa(X,Y):-pas(X,Y,Z);                                 /*r2*/
pa(X,Y):-pas(X,Z,Y);                                 /*r3*/
gpa(X,G):- pa(X,P),pa(P,G);                          /*r4*/
/*database of parent-child relation*/
pas(wil,cha,dia);                                    /*r5*/
pas(hen,cha,dia);                                    /*r6*/
pas(cha,phi,eli);                                    /*r7*/
pas(dia,edw,fra);                                    /*r8*/
?- testgpa(wil,Who), xwriteln(0, "Who=",Who);
```

The program finds a grandparent (gpa) of "wil." The predicate symbols pa and pas are abbreviations of parent and parents, respectively.

Let a Prolog program be, in general, represented as

$$\text{rlist} = (r_1, \ldots, r_n) \text{ or rlist} = r_1 \ldots r_n,$$

where r_i is a rule of Prolog. The query predicate is not included in rlist.

Let a rule form be

$$p_i()\text{:- } p_{i1}(), \ldots, p_{ij}(),$$

where $p_i()$ and $p_{ij}()$ are predicates and j depends on i. Although $p_{ij}()$ can be a general literal, it is assumed to be a predicate in this chapter. For notational convenience, the rule is denoted by

$$p_i | p_{i1}, \ldots, p_{ij}$$

or simply by

$$p_i | q_i,$$

where q_i is the predicate list $p_{i1}, \ldots, p_{ij}$. For the sake of simplicity, let us assume that rlist is fixed in the subsequent discussions.

Example 18.2. The program rlist of Example 18.1 is given by

$$\text{rlist} = (r_1, \ldots, r_8),$$

where

$$r_1 = \text{testgpa}(X, \text{Who})\text{:- gpa}(X, \text{Who}),$$

$$r_2 = \text{pa}(X, Y)\text{:- pas}(X, Y, Z),$$

$$\bullet$$

$$\bullet$$

$$r_8 = \text{pas(dia,edw,fra)}.$$

Then,

$$p_1() = \text{testgpa}(X, \text{Who}),$$

$$p_{11}() = \text{gpa}(X, \text{Who}).$$

Let Rule be the set of tail segments of rlist, i.e.,

$$\text{Rule} = \{r_1 \ldots r_n, r_2 \ldots r_n, r_3 \ldots r_n, r_{n-1} r_n, r_n\}.$$

Example 18.3. Rule of Example 18.1 is given by

$$\text{Rule} = \{r_1 \ldots r_8, r_2 \ldots r_8, \ldots, r_7 r_8, r_8\}.$$

For a given Prolog program let

$$N = \text{set of nonnegative integers},$$

$$\text{Var} = \text{set of variable symbols in the Prolog program},$$

$$\text{Cons} = \text{set of constant symbols including text type constants and numerals in the program},$$

$$\text{Func} = \text{set of function symbols in the program},$$

$$\text{extVar} = \text{Var} \times N,$$

and

$$\text{Pred} = \text{set of predicate symbols in the program}.$$

Example 18.4. In the program of Example 18.1,

$$\text{Var} = \{X, Y, Z, \text{Who}, G, P\},$$

$$\text{Cons} = \{\text{wil, cha, dia, hen, phi, eli, edw, fra}\},$$

$$\text{Func} = \varnothing(\text{empty set}),$$

$$\text{extVar} = \{(X, 0), (X, 1), \ldots\},$$

$$\text{Pred} = \{\text{testgpa, gpa, pa, pas}\}.$$

For notational convenience we write

$$X.k$$

instead of (X, k). The symbol k is called a unification index.

Let us define a set of terms, Term, as follows:

1. extVar $\cup$ Cons $\subset$ Term;
2. If f is an n-ary function symbol and if $t_1, \ldots, t_l$ are terms, then

$$f(t_1, \ldots, t_l) \in \text{Term};$$

3. Term is the smallest set that satisfies conditions 1 and 2.

Let k be a nonnegative integer. Then, let a unification-indexed term $t[k]$ for a term t be as follows:

(i) If t is a variable,

$$t[k] = t.k.$$

(ii) If t is a constant,

$$t[k] = t.$$

(iii) If $t_1, \ldots, t_n$ are terms and if f is an n-ary function,

$$f(t_1, \ldots, t_n)[k] = f(t_1[k], \ldots, t_n[k]).$$

Let the set of unification indexed terms be called extTerm.

If $p(t, \ldots, t')$ is a predicate and k is an integer, then the unification-indexed predicate $p[k]$ denotes a predicate generated from p by suffixing its terms with k, that is,

$$p[k] = p(t, \ldots, t')[k] = p(t[k], \ldots, t'[k]).$$

Note that each of $t, \ldots, t'$ can be a predicate as well as a term. This definition covers both terms and predicates. Let extPred be the set of unification-indexed predicates. Because functions (including constants) are special relations, extTerm is treated as a subset of extPred.

Example 18.5. For Example 18.1,

$$\text{wil}, X.2 \in \text{extTerm}$$

and

$$\begin{aligned}
&\text{testgpa}(\text{gpa}(X, \text{Who}), \text{pa}(X, \text{cha}))[1] \\
&\quad = \text{testgpa}(\text{gpa}(X, \text{Who})[1], \text{pa}(X, \text{cha})[1]) \\
&\quad = \text{testgpa}(\text{gpa}(X.1, \text{Who}.1), \text{pa}(X.1, \text{cha})) \in \text{extPred}.
\end{aligned}$$

If

$$q = (p_1, \ldots, p_t), \text{ let}$$
$$q[k] = (p_1[k], \ldots, p_t[k]).$$

Example 18.6. For Example 18.1,

$$\begin{aligned}
(pa(X, P), pa(P, G))[2] &= (pa(X, P)[2], pa(P, G)[2]) \\
&= (pa(X.2, P.2), pa(P.2, G.2)).
\end{aligned}$$

Let Q be the set of sequences of unification-indexed predicates:

$$Q = \{q | q = (p()[k], \ldots, p'()[k'])\} \text{ or } Q = \{q | q = p[k] \ldots p'[k']\},$$

where $k, \ldots, k'$ are nonnegative integers.

Example 18.7. For example,

$$(\text{pas}(X, Y, Z)[3], \text{pa}(P, G)[2]) \in Q.$$

Let us define the unification operation. In general, a binary relation $p \subset X \times Y$ is called functional iff the following relation holds:

$$(x, y), (x, y') \in p \rightarrow y = y'.$$

Let

$$D = \text{extVar} \times \text{extPred}.$$

The relation $\varepsilon \in D^*$ is then called a functional relation if $\{d | d \text{ is an element of } \varepsilon\}$ is functional, where D^* is the free monoid of D. For convenience, a functional relation ε is treated as a partial function, i.e., $\varepsilon : (\text{extVar}) \rightarrow \text{extPred}$.

Let

$$\text{pDatabase} = \{\varepsilon | \varepsilon \in D^* \text{ and } \varepsilon \text{ is a functional relation}\} \subset D^*.$$

Then pDatabase is the set of Prolog databases associated with a PDA, or $\varepsilon \in$ pDatabase is a Prolog database.

Example 18.8. The following is a Prolog database ε of Example 18.1:

$$(X.1, \text{wil}) \in D,$$

$$(X.0, \text{Who}.1) \in D,$$

$$(X.2, X.1) \in D,$$

$$(\text{Who}.1, G.2) \in D,$$

$$\varepsilon = (X.1, \text{wil})(X.0, \text{Who}.1)(X.2, X.1)(\text{Who}.1, G.2) \in \text{pDatabase}.$$

Here ε is a functional relation because $\{(X.1, \text{wil}), (X.0, \text{Who}.1), (X.2, X.1), (\text{Who}.1, G.2)\}$ is functional.

Because ε of Example 18.8 is essentially a function from the domain $\{X.1, X.0, X.2, \text{Who}.1\}$ to extPred, standard notation for a function is used for ε, i.e., $\text{dom}(\varepsilon) = \{X.1, X.0, X.2, \text{Who}.1\}$ and $\varepsilon(X.0) = \text{Who}.1$.

The automaton of the PDA performs the unification operation using a Prolog database in pDatabase. Let eval: extPred $\times$ pDatabase $\to$ extPred be given by

$$\text{eval}(c, \varepsilon) = c \qquad \text{if } c \in \text{Cons},$$

$$\text{eval}(X.k, \varepsilon) = \begin{cases} \text{eval}(\varepsilon(X.k), \varepsilon) & \text{if } X.k \in \text{dom}(\varepsilon), \\ X.k & \text{otherwise}, \end{cases}$$

$$\text{eval}(p(t_1, \ldots, t_l), \varepsilon) = p(\text{eval}(t_1, \varepsilon), \ldots, \text{eval}(t_l, \varepsilon)),$$

where p is either a function symbol or a predicate symbol. In particular, if $p(t_1, \ldots, t_l)$ is an arithmetic or logic formula and if $\text{eval}(p(t_1, \ldots, t_l), \varepsilon)$ can be calculated as the conventional operation whose value is v, then

$$\text{eval}(p(t_1, \ldots, t_l), \varepsilon) = v.$$

Example 18.9. In Example 18.1, suppose

$$\varepsilon = (X.1, \text{wil})(X.0, \text{Who}.1)(X.2, X.1)(\text{Who}.1, G.2).$$

Then

$$\begin{aligned}
\text{eval}(\text{wil}, \varepsilon) &= \text{wil}, \\
\text{eval}(X.2, \varepsilon) &= \text{eval}(\varepsilon(X.2), \varepsilon) = \text{eval}(X.1, \varepsilon) = \text{eval}(\varepsilon(X.1), \varepsilon) = \text{wil}, \\
\text{eval}(\text{pa}(X, P)[2], \varepsilon) &= \text{eval}(\text{pa}(X.2, P.2), \varepsilon) \\
&= \text{pa}(\text{eval}(X.2, \varepsilon), \text{eval}(P.2, \varepsilon)) \\
&= \text{pa}(\text{wil}, P.2), \\
\text{eval}((1 + 2 + 3), \varepsilon) &= 6, \\
\text{eval}(1 = 2, \varepsilon) &= \text{false}.
\end{aligned}$$

Let

$$\text{unif} : \text{extPred} \times \text{extPred} \times (\text{pDatabase} \cup \{\text{fail}\}) \to \text{pDatabase} \cup \{\text{fail}\}$$

be defined by

$$\text{unif}(t_1, t_2, \varepsilon) = \text{unif}(\text{eval}(t_1, \varepsilon), \text{eval}(t_2, \varepsilon), \varepsilon),$$

where if $t_1^* = \text{eval}(t_1, \varepsilon)$ and $t_2^* = \text{eval}(t_2, \varepsilon)$, $\text{unif}(t_1^*, t_2^*, \varepsilon)$ is specified by the following ten cases:

Case 0.

$$\text{unif}(t_1^*, t_2^*, \text{fail}) = \text{fail}.$$

Case 1.

$$\text{unif}(c_1, c_2, \varepsilon) = \begin{cases} \varepsilon & \text{if } c_1 = c_2, \\ \text{fail} & \text{if } c_1 \neq c_2, \end{cases}$$

where $c_1, c_2 \in \text{Cons}$.

Case 2.

$$\text{unif}(c_1, X_2.k_2, \varepsilon) = \varepsilon \cdot (X_2.k_2, c_1),$$

where $\varepsilon \cdot (X_2.k_2, c_1)$ is the concatenation of ε and $(X_2.k_2, c_1)$.

Case 3.

$$\text{unif}(c_1, p(), \varepsilon) = \text{fail},$$

where $p()$ is a predicate. Unification between a constant and a predicate must fail.

Case 4.

$$\text{unif}(X_1.k_1, c_2, \varepsilon) = \varepsilon \cdot (X_1.k_1, c_2).$$

Case 5.

$$\text{unif}(X_1.k_1, X_2.k_2, \varepsilon) = \begin{cases} \varepsilon \cdot (X_1.k_1, X_2.k_2) & \text{if } k_1 \leq k_2, \\ \varepsilon \cdot (X_2.k_2, X_1.k_1) & \text{otherwise.} \end{cases}$$

Case 6.

$$\text{unif}(X_1.k_1, p(), \varepsilon) = \varepsilon \cdot (X_1.k_1, p()).$$

Case 7.

$$\text{unif}(p(), c_2, \varepsilon) = \text{fail}.$$

Case 8.

$$\text{unif}(p(), X_2.k_2, \varepsilon) = (X_2.k_2, p()).$$

Case 9.

$$\text{unif}(p_1(X_1, \ldots, X_n), p_2(Y_1, \ldots, Y_m), \varepsilon)$$
$$= \begin{cases} \text{fail} & \text{if} p_1 \neq p_2 \text{ or } n \neq m, \\ \text{unif}([X_1, \ldots, X_n], [Y_1, \ldots, Y_m], \varepsilon) & \text{otherwise,} \end{cases}$$

where

$$\text{unif}([X_1, \ldots, X_n], [Y_1, \ldots, Y_m], \varepsilon)$$
$$= \text{unif}([X_2, \ldots, X_n], [Y_2, \ldots, Y_m], \text{unif}(X_1, Y_1, \varepsilon)).$$

The function unif represents the unification operation. A successful unification expands pDatabase.

Example 18.10. For Example 18.1, suppose

$$\varepsilon = (X.5, \text{cha}) \in \text{pDatabase}.$$

Then

$$\begin{aligned}
&\text{unif}(\text{pas}(X, Y, Z)[5], \text{pas}(\text{wil}, \text{cha}, \text{dia})[6], \varepsilon) \\
&\quad = \text{unif}(\text{pas}(\text{cha}, Y.5, Z.5), \text{pas}(\text{wil}, \text{cha}, \text{dia}), \varepsilon) \\
&\quad = \text{unif}([\text{cha}, Y.5, Z.5], [\text{wil}, \text{cha}, \text{dia}], \varepsilon) \\
&\quad = \text{unif}([Y.5, Z.5], [\text{cha}, \text{dia}], \text{unif}(\text{cha}, \text{wil}, \varepsilon)) \qquad \text{(see Case 1)} \\
&\quad = \text{unif}([Y.5, Z.5], [\text{cha}, \text{dia}], \text{fail}) \qquad \text{(due to Case 1)} \\
&\quad = \text{fail} \qquad \text{(due to Case 0).}
\end{aligned}$$

Using the above notation, the Prolog interpreter can be modeled by a PDA as follows:

$$\text{Prolog interpreter} = \langle A, C, \Gamma, c_0, \gamma_0, \delta \rangle,$$

where

$$\begin{aligned}
&A = \varnothing; \text{ empty input set,} \\
&C = \Gamma = Q \times \text{Rule} \times N \times \text{pDatabase}; \text{ state set,} \\
&c_0 = (q_0[0], \text{rlist}, 1, \Lambda); \text{ initial state of head,} \\
&\gamma_0 = \Lambda; \text{ initial state of stack,} \\
&\delta : C \times \Gamma \to C \times \Gamma^*; \text{ state transition function,}
\end{aligned}$$

where q_0 is the initial query.

For $\delta(c_1, \gamma_1) = (c_2, \gamma_2^*)$, γ_1 is the top element of the push-down stack and γ_2^* is an element of Γ^* that replaces γ_1. The definition of δ is given below.

If $p[k]$ is executable,

$$\delta((p[k]\alpha, \beta, t, \varepsilon), \gamma) = \begin{cases} ((\alpha, \text{rlist}, t, \varepsilon), \gamma) & \text{if } p[k] \neq \text{fail}, \\ (\gamma, \Lambda) & \text{if } p[k] = \text{fail}. \end{cases}$$

If $(p|q)$ is the first rule to which unification is applicable and $\text{unif}(p[k], p[t], \varepsilon) \neq \text{fail}$,

$$\begin{aligned}
&\delta((p[k]\alpha, \beta(p|q)\beta', t, \varepsilon), \gamma) \\
&\quad = ((q[t]\alpha, \text{rlist}, t + 1, \text{unif}(p[k], p[t], \varepsilon)), \gamma \cdot (p[k]\alpha, \beta', t, \varepsilon)).
\end{aligned}$$

If $(p|q)$ is the first rule to which unification is applicable and $\text{unif}(p[k], p[t], \varepsilon) = \text{fail}$,

$$\delta((p[k]\alpha, \beta(p|q)\beta', t, \varepsilon), \gamma) = \begin{cases} ((p[k]\alpha, \beta', t, \varepsilon), \gamma) & \text{if } \beta' \neq \Lambda, \\ (\gamma, \Lambda) & \text{if } \beta' = \Lambda. \end{cases}$$

Here $\varepsilon \in$ pDatabase is an instance of the Prolog database for unification.

Figure 18.1 is a conceptual illustration of the formulation. Here $d \in D$ represents an element generated by unification, and γ is the top element of the stack. The unification index is represented by $t \in N$ in the state set C.

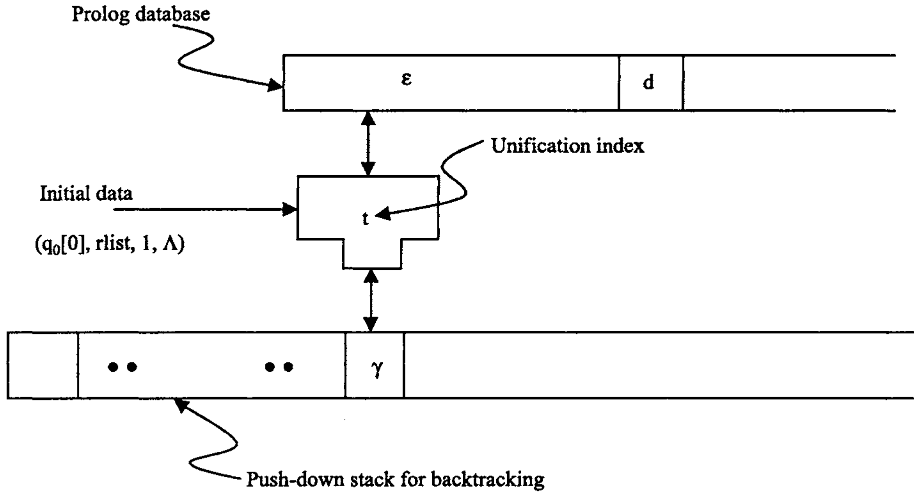


Fig. 18.1. Prolog interpreter as a push-down automaton.

Example 18.11. Let us check the behavior of Example 18.1 using the above formulation.

Let

$$\begin{aligned} c_0 &= (q_0[0], \text{rlist}, 1, \Lambda), \\ q_0 &= (\text{testgpa(wil, Who), xwriteln(“Who=”, Who)}), \\ \text{rlist} &= (r_1, \ldots, r_8), \\ \gamma_0 &= \Lambda. \end{aligned}$$

Let

$$\delta(c_0, \gamma_0) = (c_1, \gamma_1^*),$$

where because unif(testgpa(wil, X)[0], testgpa(X, Who)[1], Λ) $\neq$ fail,

$$\begin{aligned} c_1 &= (q_1, (r_1, \ldots, r_8), 2, \varepsilon_1), \\ q_1 &= (\text{gpa}(X, \text{Who})[1], \text{xwriteln(“Who=”, Who)}[0]), \\ \varepsilon_1 &= \text{unif(testgpa(wil, Who)[0], testgpa}(X, \text{Who})[1], \Lambda) \\ &= \Lambda \cdot (X.1 = \text{wil}, \text{Who}.0 = \text{Who}.1), \\ \gamma_1^* &= \Lambda \cdot (q_0[0], (r_2, \ldots, r_8), 1, \Lambda). \end{aligned}$$

Let

$$\delta(c_1, \gamma_1) = (c_2, \gamma_2^*),$$

where

$$\gamma_1 = \text{top of the stack}(= (q_0[0], (r_2, \ldots, r_8), 1, \Lambda);$$

because unif(gpa(X, Who)[1], gpa(X, G)[2], ε_1) $\neq$ fail,

$$c_2 = (q_2, (r_1, \ldots, r_8), 3, \varepsilon_2),$$
$$q_2 = (\text{pa}(X, P)[2], \text{pa}(P, G)[2], \text{xwriteln(``Who='', Who)}[0]),$$
$$\varepsilon_2 = \varepsilon_1 \cdot \text{unif}(\text{gpa}(X, \text{Who})[1], \text{gpa}(X, G)[2], \varepsilon_1)$$
$$= \varepsilon_1 \cdot (X.1 = X.2, \text{Who}.1 = G.2),$$
$$\gamma_2^* = \gamma_1 \cdot (q_1, (r_5, \ldots, r_8), 2, \varepsilon_1).$$

Let

$$\delta(c_2, \gamma_2) = (c_3, \gamma_3^*),$$

where

$$\gamma_2 = \text{top of the stack}(= (q_1, (r_5, \ldots, r_8), 2, \varepsilon_1));$$

because unif(pa(X, P)[2], pa(X, Y)[3], ε_2) $\neq$ fail,

$$c_3 = (q_3, (r_1, \ldots, r_8), 4, \varepsilon_3),$$
$$q_3 = (\text{pas}(X, Y, Z)[3], \text{pa}(P, G)[2], \text{xwriteln(``Who='', Who)}[0]),$$
$$\varepsilon_3 = \varepsilon_2 \cdot \text{unif}(\text{pa}(X, P)[2], \text{pa}(X, Y)[3], \varepsilon_2)$$
$$= \varepsilon_2 \cdot (X.2 = X.3, P.2 = Y.3),$$
$$\gamma_3^* = \gamma_2 \cdot (q_2, (r_3, \ldots, r_8), 3, \varepsilon_2).$$

Let

$$\delta(c_3, \gamma_3) = (c_4, \gamma_4^*),$$

where

$$\gamma_3 = \text{top of the stack}(= (q_2, (r_3, \ldots, r_8), 3, \varepsilon_2));$$

because unif(pas(X, Y, Z)[3], pas(wil, cha, dia)[4]) $\neq$ fail,

$$c_4 = (q_4, (r_1, \ldots, r_8), 5, \varepsilon_4),$$
$$q_4 = (\text{pa}(P, G)[2], \text{xwriteln(``Who='', Who)}[0]),$$
$$\varepsilon_4 = \varepsilon_3 \cdot (X.3 = \text{wil}, Y.3 = \text{cha}, Z.3 = \text{dia}),$$
$$\gamma_4^* = \gamma_3 \cdot (q_3, (r_6, r_7, r_8), 4, \varepsilon_3).$$

Let

$$\delta(c_4, \gamma_4) = (c_5, \gamma_5^*),$$

where

$$\gamma_4 = \text{top of the stack}(= (q_3, (r_6, \ldots, r_8), 4, \varepsilon_3));$$

because unif(pa(P, G)[2], pa(X, Y)[5]) $\neq$ fail,

$$c_5 = (q_5, (r_1, \ldots, r_8), 6, \varepsilon_5),$$
$$q_5 = (\text{pas}(X, Y, Z)[5], \text{xwriteln(``Who='', Who)}[0]),$$

$$\varepsilon_5 = \varepsilon_4 \cdot (X.5 = \text{cha}, G.2 = Y.5),$$
$$\gamma_5^* = \gamma_4 \cdot (q_4, (r_3, \ldots, r_8), 5, \varepsilon_4).$$

Let

$$\delta(c_5, \gamma_5) = (c_6, \gamma_6^*),$$

where

$$\gamma_5 = \text{top of the stack}(= (q_4, (r_3, \ldots, r_8), 5, \varepsilon_4));$$

because unif(pas(X, Y, Z)[5], pas(wil, cha, dia)[6], ε_5) = fail and $(r_6, \ldots, r_8) \neq \Lambda$,

$$c_6 = (q_5, (r_6, \ldots, r_8), 6, \varepsilon_5),$$
$$\gamma_6^* = \gamma_5.$$

Let

$$\delta(c_6, \gamma_6) = (c_7, \gamma_7^*),$$

where

$$\gamma_6 = \text{top of the stack}(= (q_4, (r_3, \ldots, r_8), 5, \varepsilon_4));$$

because unif(pas(X, Y, Z)[5], pas(hen, cha, dia)[6], ε_5) = fail and $(r_7, r_8) \neq \Lambda$,

$$c_7 = (q_5, (r_7, r_8), 6, \varepsilon_5),$$
$$\gamma_7^* = \gamma_6.$$

Let

$$\delta(c_7, \gamma_7) = (c_8, \gamma_8^*),$$

where

$$\gamma_7 = \text{top of the stack}(= (q_4, (r_3, \ldots, r_8), 5, \varepsilon_4));$$

because unif(pas(X, Y, Z)[5], pas(cha, phi, eli)[6]) $\neq$ fail,

$$c_8 = (q_6, (r_1, \ldots, r_8), 7, \varepsilon_6),$$
$$q_6 = (\text{xwriteln}(0, \text{“Who=”}, \text{Who})[0]),$$
$$\varepsilon_6 = \varepsilon_5 \cdot (X.5 = \text{cha}, Y.5 = \text{phi}, Z.5 = \text{eli}),$$
$$\gamma_8^* = \gamma_7 \cdot (q_5, (r_8), 6, \varepsilon_5).$$

Let

$$\delta(c_8, \gamma_8) = (c_9, \gamma_9^*),$$

where

$$\gamma_8 = \text{top of the stack}(= (q_5, (r_8), 6, \varepsilon_5));$$

because xwriteln(0, “Who=”, Who)[0] is executable,

$$c_9 = (q_7, (r_1, \ldots, r_8), 7, \varepsilon_7),$$
$$q_7 = \Lambda,$$

$$\varepsilon_7 = \varepsilon_6,$$

$$\gamma_9^* = \gamma_8.$$

xwriteln(0, "Who=", Who)[0] $\rightarrow$ evaluation of Who.0;

eval(Who.0, ε_6) = eval(Who.1, ε_6) = eval(G.2, ε_6) = eval(Y.5, ε_6) = phi.

"Who = phi" must be printed out due to execution of xwriteln(0, "Who=", Who)[0]. This shows that the interpreter behaves in a desirable way.

References

Takahara, Y. and Iijima, J. (1990) *Systems Theory*, Kyouritu (in Japanese).

Index

$\vee$, 24, 328, 329
π_σ, 86
$\delta\text{–}\lambda_{\text{actionN}_i}$, 232
σ_{pd}, 94

$\langle \text{actionN}_i \rangle$.g, 238
action, 115, 248
action name, 232
action parameter, 232
action set, 131, 147, 154, 162, 171, 190, 270
actionN$_i$, 7, 238
ActionName, 7, 231, 239
ActionName.g, 238
adaptive layer, 209, 212, 214
ADC, 212
additive, 14
additivity condition, 82
ADPS, 212
ADR, 212
allowable activity function, 116
ALst, 186
append, 39
assert, 39, 58
assign, 39, 58
atomic formulae, 24, 328
atomic process, 232, 241, 255
attribute node, 184
attrlist, 316
AttrName, 7, 231, 240
attrnamelist, 315
attrtype, 316
attrtypelist, 316
automaton, 14
automaton specification of process, 82, 85

backtrack, 94
backtracking, 49, 51, 91, 343
backward layer structure, 87
bar graph, 61
basic MIS, 228
BMIS, 228
bound variable, 329
branch and bound method, 164
branch name, 184
browser-based MIS, 227
brute force behavior, 135
brute-force method, 340

c_0, 116, 133, 148, 155, 163
canonical model of the BMIS, 229
causal, 246
c_f, 116, 133, 148
C_f, 155
Chinese checkers, 138
class schedule problem, 169, 269
classification of problem, 74
clause, 331
clearsheet, 39
closed formula, 329
closed goal, 74
closed target, 74
cname, 315
compatibility condition, 94
completeness theorem, 327
composition, 31
computer-acceptable set theory, 8, 23
concat, 39
confidence number, 215
conjunctive formula, 330

ConL, 215
Cons, 345
conssym, 52
constant symbol, 24, 328
constraint, 72, 116, 132, 155, 163, 173, 192
construction of list, 58
construction of set, 55, 301
control engineering problem, 127
cube root problem, 153

D, 346
D_0, 187
data definition, 303
data flow diagram, 9, 226
data handling, 35
data mining system, 183
data model generator, 210, 211
data processing, 292
data retrieval, 303
data set, 184, 187
database connectivity, 44
database system, 287
decision principle, 208
decision tree, 183
declarativeness, 325
definition of class, 288, 290
definition of data processing, 289
definition of method, 288, 292
definition of object, 288, 291
definition of table, 301
defList, 58
defSet, 39, 55
deleteList, 39
delta_lambda, 7, 233, 238, 241
denotation, 329
depth-first search method, 340
DFD, 9, 226, 239
distinct, 57
dmEV, 215, 219
DMG, 210, 211, 213
dmR, 218
dmSTNDKN, 215
dom, 187
do-while clause, 54
DP method, 86, 89
drawtree, 60
dynamic optimization formulation, 82, 85, 92, 193
dynamic optimization problem, 14
dynamic problem, 127
dynamical mapping, 86

E-C-C, 79, 113
E-C-O, 145
E-O-C, 127, 129
efficient solution path, 93
embedding SQL command, 309
end-user development, 4
entropy, 193
entryV, 316
EUD, 4
eval, 347
execution of method, 293
expert system, 214
explicit solving action, 74
extended solver, 6
external UI, 228, 256
external user interface, 9
extPred, 346
extProlog, 3, 44
extProlog interpreter, 343
extSLV, 6, 98, 235
extTerm, 345
extVar, 345

fact, 50, 332
failure node, 337
feedback information, 70
feedback law problem, 128, 136
file structure, 7, 231, 240
file system, 231, 251
final state, 116, 133, 148, 155, 163, 174, 192
first-order language, 24, 328
first-order predicate calculus, 24, 325
f_j, 231
for clause, 54
formal approach, 11, 226
forward layer structure, 93
free variable, 329
Func, 345
func, 26, 237, 242
function, 30, 55
function declaration, 55
functional relation, 346

genA, 72, 116, 132, 148, 155, 163, 172, 192
general systems theory, 3
general-purpose language, 53

generalization, 327, 335
genGoal, 90
genIndex, 39
genSol, 91
getDate, 39
getDate2, 39
getvalue, 39
global variable, 58
goal, 14, 85, 133, 148, 174, 193, 194, 272, 331
goal-seeker model, 68
goal-seeking model, 69
goal-seeking problem, 207
goalElement, 14
graphical user interface (GUI), 44, 59
ground instance, 334
group by, 305
GSP, 207, 208, 210
GST, 3, 67

hard constraint, 213
head, 49
Herbrand base, 334
Herbrand interpretation, 334, 337
Herbrand universe, 334
hill-climbing method with a push-down stack, 71
Horn clause, 331

I-C-C, 153
I-C-O, 159
I-O-C, 169, 269
I-O-O, 79, 183
ideal goal, 86
ideal goal generation, 86
idGoal family, 89
if clause, 53
implementation component, 7, 233
implementation of OODB, 293
implementation procedure, 238
implementation structure, 8, 23, 237, 243
implementation structure of TPS, 8, 237
implicit solving action, 74
initial response function, 245
initial state, 116, 133, 148, 155, 163, 173, 192, 249, 272
initialattrV, 317
input, 69
input attributes, 184
input specification, 114
input–output specification, 239
integer programming, 159
intelligent MIS, 5
interface component, 7, 233
internal UI, 233
internal user interface, 9
internalUI, 228
invδ, 87, 117
inverse, 39
inverse δ, 87
inverse st, 87
inverse stopping condition, 117
inverse transition, 117
invproject, 39
invst, 87, 117

KB, 214
knapsack problem, 159
knowledge base, 214

legalAs, 93
lifting lemma, 340
limbo node, 187
linear input resolution, 340
linear quadratic problem, 146
list, 54
list structure, 51
listPred, 39
literal, 328
ln, 56
load_go, 39
logic programming, 325
logic programming language, 53
logical connective, 24, 328
logical equivalence, 330
loop free, 94
LQP, 146

makewindowSS, 39
management information system, 3
manipulating variable, 70
matched, 49
matrix, 330
matrix manipulation, 57
max, 56
MDL, 44
mechanical theorem-proving, 340
min, 56

minimization, 32
MIS, 3
model, 329
model description language, 44
model integration approach, 43
model space, 218
model theory approach, 6, 68, 226
modified dynamic programming method, 71
modus ponens, 327, 335
multiple-trajectory graph, 61, 62

N, 344
node number, 184
nodeData, 187
nodeStr, 187
numeric processing, 45

obindex, 315
obj, 315
object-oriented, 287
object-oriented database, 287
objV, 316
OODB, 287
open formula, 329
open goal, 74
open target, 74
operations of the registration system, 243
operations on list, 56
operations on set, 54
output attribute, 184
output function, 247
output specification, 187

para, 7, 231
paralist, 234
Paralist$_i$, 7, 231
parametric uncertainty, 209
participant.lib, 240
particularized decision problem, 210
PD method, 91
PDA, 343
pDatabase, 346
PID controller, 128
pie graph, 61
plot graph, 61
postgreSQL, 287
postprocess, 39
Pred, 345
predicate, 55
predicate symbol, 24, 328
predsym, 52
prefix, 330
prenex form, 330
preprocess, 39
PRF, 74, 210, 211, 213
primitive recursion, 31
problem formulator, 74, 210, 211
problem specification environment, 6, 9
problem-solving layer, 209
problem-solving system, 3, 13
procC, 41
procedural, 325
Process, 69
program structure of an extSLV, 76
project, 39
Prolog, 45, 326, 340
Prolog database, 346
Prolog interpreter, 49, 349
PSE, 6, 9, 235
push-down automaton, 343

Q, 346
query clause, 49

radar graph, 61
Re, 82
realization, 231, 240, 249
realization model, 255
realization structure, 231
reduced, 229, 248, 250
reduced state mapping, 115
reduced-layer model, 87
refutable, 330
refutation complete, 335
register, 241
regulation problem, 129
relation, 29, 55
relational database, 183
relational structure, 24, 327
relational structure representation, 234
replaceList, 39
representation of set, 300
reset state, 249
ResName, 7, 231, 240
resolution node, 338
resolution principle, 327, 335
resolvent, 335
ResStatus, 241

restricted allowable activity function, 88, 93
retract, 41
retract_byhead, 41
Rule, 344
rule, 332
rule head class, 49

satisfaction, 329
satisfaction approach, 209
satisfied, 330
scope, 329
self-executable predicate, 49
semantic tree, 337
semioptimal solution, 89
sentence, 329
set, 27, 54
setcompiler, 8, 36, 238
setvalue, 41
shift operator, 246
show1, 41, 61
skeleton model, 6
Skolemizing, 331
SLV, 13, 210, 211, 213
soft constraint, 213
software engineering approach, 10
solution generation, 86, 91
solution path, 93
solvable state, 93
solver, 13, 210
solver type, 235
solving-process model, 68
sort, 41
sortmax, 41
special functions and predicates for list, 56
spreadsheet, 60
SQL, 287, 297
st, 72, 82, 85, 133, 148, 155, 164, 193
standardized external UI, 238
standardized goal-seeker, 6, 79, 86
standardized internal UI, 238
standardized UI, 226
standardized user interface, 6
state reduction map, 80, 131
state set, 247, 349
state space representation, 245–247
state space representation of BMIS, 247
state transition function, 116, 131, 147, 154, 162, 172, 191, 247, 271, 349
stationary, 246
stdDPS_UI.p, 9, 39
stdDPsolver, 113
stdPDsolver, 127, 145, 159, 169, 183, 269
stdPDsolver.p, 76
stdUI.php, 9
stopping condition, 82, 85
strlen, 56
structural uncertainty, 209
structure A, 293
structure C, 294
structure O, 295
sum, 41, 56
SupL, 215
support number, 215
switch clause, 54
symbolic programming, 48
syntax of extProlog, 52
syntax of OODB, 288
systems approach, 10
systems development, 236

tail, 49
task, 210
Term, 345
term, 24, 328
terminal node, 184
time invariant, 247
time system, 229, 245
tolerance function, 213
TPS, 3, 225
trajectory graph, 61
transaction processing system, 3, 225
transaction type, 234
transpose, 41
traveling salesman problem, 113
tree display, 60
truth assignment, 334
typeless, 51

U, 231
uncertainty, 193, 207
undef, 299
unif, 347
unification, 49, 343, 349
unification index, 345
unifier, 339
universal formula, 330
universal formula of Horn clauses, 331
universal quantifier, 24, 328

universally quantified conjunction of clauses, 332
universally valid, 329
unsatisfiability, 335
unsolvable state, 93
updating of table, 307
user layer, 209, 212
user model, 6, 13, 14, 79, 117, 133, 148, 155, 164, 174, 195, 233, 234, 237, 242, 273
user model construction, 238
user model in computer-acceptable set theory, 8
user model in extProlog, 8
user model in set theory, 8

V, 231
valid, 329
Var, 344
variable, 24, 328
varsym, 52
VDM-sl, 12, 226
Vienna development method, 12
Vienna development method-specification language, 226
view function in OODB, 302
VLst, 187

warp2.p, 15
warp2.set, 15
well-formed formulae, 24, 328
workshop registration system, 226

xread, 41
xwriteln, 41
xyplot graph, 61

y_0, 173, 192, 272
Y_f, 192